



nVIDIA®

Holoscan SDK User Guide

Release 2.6.0

NVIDIA Corporation

Oct 23, 2024

INTRODUCTION

1	Overview	1
2	Relevant Technologies	3
3	Getting Started with Holoscan	5
4	SDK Installation	7
5	Additional Setup	13
6	Third Party Hardware Setup	27
7	Holoscan Core Concepts	37
8	Holoscan by Example	39
9	Creating an Application	81
10	Creating a Distributed Application	109
11	Packaging Holoscan Applications	127
12	Creating Operators	137
13	Logging	181
14	Debugging	185
15	Writing Python bindings for a C++ Operator	201
16	Simplified Python operator creation via the create_op decorator	215
17	Built-in Operators and Extensions	223
18	Visualization	227
19	Inference	245
20	Schedulers	257
21	Conditions	259
22	Resources	263

23 Analytics	267
24 Holoscan Application Package Specification (HAP)	269
25 Holoscan CLI	285
26 Application Runner Configuration	297
27 GXF Core concepts	299
28 Holoscan and GXF	301
29 GXF by Example	303
30 Using Holoscan Operators in GXF Applications	319
31 GXF User Guide	321
32 Data Flow Tracking	397
33 GXF job statistics	403
34 NSight Systems Profiling	405
35 Video Pipeline Latency Tool	407
36 Holoscan SDK FAQs	433

OVERVIEW

NVIDIA Holoscan is the AI sensor processing platform that combines hardware systems for low-latency sensor and network connectivity, optimized libraries for data processing and AI, and core microservices to run streaming, imaging, and other applications, from embedded to edge to cloud. It can be used to build streaming AI pipelines for a variety of domains, including medical devices, high-performance computing at the edge, industrial inspection, and more.

The Holoscan SDK assists developers by providing:

1. Various installation strategies

From containers, to Python wheels, to source, and from development to deployment environments, the Holoscan SDK comes in many packaging flavors to adapt to different needs. Find more information in the [sdk installation](#) section.

2. C++ and Python APIs

These APIs are now the recommended interface for the creation of application pipelines in the Holoscan SDK. See the *Using the SDK* section to learn how to leverage those APIs, or the Doxygen pages (C++/Python) for specific API documentation.

3. Built-in Operators

The units of work in Holoscan applications are implemented within Operators, as described in the [core concepts](#) of the SDK. The operators included in the SDK provide domain-agnostic functionalities such as IO, machine learning inference, processing, and visualization, optimized for AI streaming pipelines, relying on a set of [Core Technologies](#). This guide provides more information on the operators provided within the SDK [here](#).

4. Minimal Examples

The Holoscan SDK provides a list of examples to illustrate specific capabilities of the SDK. Their source code can be found in the [GitHub repository](#). The [Holoscan by Example](#) section provides step-by-step analysis of some of these examples to illustrate the inner workings of the Holoscan SDK.

5. Repository of Operators and Applications

HoloHub is a central repository for users and developers to share reusable operators and sample applications with the Holoscan community. Being open-source, these operators and applications can also be used as reference implementations to complete the built-in operators and examples available in the SDK.

6. Tooling to Package and Deploy Applications

Packaging and deploying applications is a complex problem that can require large amount of efforts. The [Holoscan CLI](#) is a command-line interface included in the Holoscan SDK that provides commands to [package and run applications](#) in OCI-compliant containers that could be used for production.

7. Performance tools

As highlighted in the relevant technologies section, the soul of the Holoscan project is to achieve peak performance by leveraging hardware and software developed at NVIDIA or provided by third parties. To validate this, Holoscan provides performance tools to help users and developers track their application performance. They currently include:

- a *Video Pipeline Latency Measurement Tool* to measure and estimate the total end-to-end latency of a video streaming application including the video capture, processing, and output using various hardware and software components that are supported by the NVIDIA Developer Kits.
- the *Data Flow Tracking* feature to profile your application and analyze the data flow between operators in its graph.

8. Documentation

The Holoscan SDK documentation is composed of:

- This user guide, in a [webpage](#) or [PDF](#) format
- Build and run instructions specific to each *installation strategy*
- [Release notes](#) on Github

Note: In previous releases, the prefix [Clara](#) was used to define Holoscan as a platform designed initially for [medical devices](#). Starting with version 0.4.0, the Holoscan SDK is built to be domain-agnostic and can be used to build sensor AI applications in multiple domains. Domain specific content will be hosted on the [HoloHub](#) repository.

RELEVANT TECHNOLOGIES

Holoscan accelerates streaming AI applications by leveraging both hardware and software. The Holoscan SDK relies on multiple core technologies to achieve low latency and high throughput:

- *Rivermax and GPUDirect RDMA*
- *Graph Execution Framework*
- *TensorRT Optimized Inference*
- *Interoperability between CUDA and rendering frameworks*
- *Accelerated image transformations*
- *Unified Communications X*

2.1 Rivermax and GPUDirect RDMA

The NVIDIA Developer Kits equipped with a [ConnectX network adapter](#) can be used along with the [NVIDIA Rivermax SDK](#) to provide an extremely efficient network connection that is further optimized for GPU workloads by using [GPUDirect](#) for RDMA. This technology avoids unnecessary memory copies and CPU overhead by copying data directly to or from pinned GPU memory, and supports both the integrated GPU or the discrete GPU.

Note: NVIDIA is committed to supporting hardware vendors enabling RDMA within their own drivers, an example of which is provided by the [AJA Video Systems](#), as part of a partnership with NVIDIA for the Holoscan SDK. The [AJASource](#) operator is an example of how the SDK can leverage RDMA.

For more information about GPUDirect RDMA, see the following:

- [GPUDirect RDMA Documentation](#)
- [Minimal GPUDirect RDMA Demonstration](#) source code, which provides a real hardware example of using RDMA and includes both kernel drivers and user space applications for the RHS Research PicoEVB and HiTech Global HTG-K800 FPGA boards.

2.2 Graph Execution Framework

The Graph Execution Framework (GXF) is a core component of the Holoscan SDK that provides features to execute pipelines of various independent tasks with high performance by minimizing or removing the need to copy data across each block of work, and providing ways to optimize memory allocation.

GXF will be mentioned in many places across this user guide, including a *dedicated section* which provides more details.

2.3 TensorRT Optimized Inference

NVIDIA TensorRT is a deep learning inference framework based on CUDA that provided the highest optimizations to run on NVIDIA GPUs, including the NVIDIA Developer Kits.

The *inference module* leverages TensorRT among other backends, and provides the ability to execute multiple inferences in parallel.

2.4 Interoperability between CUDA and rendering frameworks

Vulkan is commonly used for real-time visualization and, like CUDA, is executed on the GPU. This provides an opportunity for efficient sharing of resources between CUDA and this rendering framework.

The *Holoviz* module uses the *external resource interoperability* functions of the low-level CUDA driver application programming interface, the Vulkan *external memory* and *external semaphore* extensions.

2.5 Accelerated image transformations

Streaming image processing often requires common 2D operations like resizing, converting bit widths, and changing color formats. NVIDIA has built the CUDA accelerated NVIDIA Performance Primitive Library (NPP) that can help with many of these common transformations. NPP is extensively showcased in the Format Converter operator of the Holoscan SDK.

2.6 Unified Communications X

The Unified Communications X (UCX) framework is an open-source communication framework developed as a collaboration between industry and academia. It provides high-performance point-to-point communication for data-centric applications. Holoscan SDK uses UCX to send data between fragments in distributed applications. UCX's high level protocols attempt to automatically select an optimal transport layer depending on the hardware available. For example technologies such as TCP, CUDA memory copy, CUDA IPC and GPUDirect RDMA are supported.

GETTING STARTED WITH HOLOSCAN

As described in the [Overview](#), the Holoscan SDK provides many components and capabilities. The goal of this section is to provide a recommended path to getting started with the SDK.

3.1 1. Choose your platform

The Holoscan SDK is optimized and compatible with multiple hardware platforms, including NVIDIA Developer Kits (aarch64) and x86_64 workstations. Learn more on the [developer page](#) to help you decide what hardware you should target.

3.2 2. Setup the SDK and your platform

Start with [installing the SDK](#). If you have a need for it, you can go through additional [recommended setups](#) to achieve peak performance, or [setup additional sensors](#) from NVIDIA's partners.

3.3 3. Learn the framework

1. Start with the [Core Concepts](#) to understand the technical terms used in this guide, and the overall behavior of the framework.
2. Learn how to use the SDK in one of two ways (or both), based on your preference: a. Going through the [Holoscan by Example](#) tutorial which will build your knowledge step-by-step by going over concrete minimal examples in the SDK. You can refer to each example source code and run instructions to inspect them and run them as you go. b. Going through the condensed documentation that should cover all capabilities of the SDK using minimal mock code snippets, including [creating an application](#), [creating a distributed application](#), and [creating operators](#).

3.4 4. Understand the reusable capabilities of the SDK

The Holoscan SDK does not only provide a framework to build and run applications, but also a set of reusable operators to facilitate implementing applications for streaming, AI, and other general domains.

The list of existing operators is available [here](#), which points to the C++ or Python API documentation for more details. Specific documentation is available for the [visualization](#) (codename: HoloViz) and [inference](#) (codename: HoloInfer) operators.

Additionally, [HoloHub](#) is a central repository for users and developers to share reusable operators and sample applications with the Holoscan community, extending the capabilities of the SDK:

- Just like the SDK operators, the HoloHub operators can be used in your own Holoscan applications.
- The HoloHub sample applications can be used as reference implementations to complete the examples available in the SDK.

Take a glance at [HoloHub](#) to find components you might want to leverage in your application, improve upon existing work, or contribute your own additions to the Holoscan platform.

3.5 5. Write and run your own application

The steps above cover what is required to write your own application and run it. For facilitating packaging and distributing, the Holoscan SDK includes utilities to *package and run your Holoscan application* in an OCI-compliant container image.

3.6 6. Master the details

- Expand your understanding of the framework with details on the *logging utility* or the *data flow tracking* benchmarking tool and *job statistics* measurements.
- Learn more details on the configurable components that control the execution of your application, like [Schedulers], [Conditions], and [Resources]. (Advanced) These components are part on the GXF execution backend, hence the **Graph Execution Framework** section at the bottom of this guide if deep understanding of the application execution is needed.

SDK INSTALLATION

The section below refers to the installation of the Holoscan SDK referred to as the **development stack**, designed for NVIDIA Developer Kits (arm64), and for x86_64 Linux compute platforms, ideal for development and testing of the SDK.

Note: or Holoscan Developer Kits such as the [IGX Orin Developer Kit](#), an alternative option is the *deployment stack*, based on [OpenEmbedded](#) (Yocto build system) instead of Ubuntu. This is recommended to limit your stack to the software components strictly required to run your Holoscan application. The runtime Board Support Package (BSP) can be optimized with respect to memory usage, speed, security and power requirements.

4.1 Prerequisites

NVIDIA Developer Kits

Setup your developer kit:

Developer Kit	User Guide	OS	GPU Mode
NVIDIA IGX Orin	Guide	IGX Software 1.0 Production Release	iGPU or* dGPU
NVIDIA Jetson AGX Orin and Orin Nano	Guide	JetPack 6.0	iGPU
NVIDIA Clara AGX Only supporting the NGC container	Guide	HoloPack 1.2 Upgrade to 535+ drivers required	dGPU

* iGPU and dGPU can be used concurrently on a single developer kit in dGPU mode. See [details here](#).

NVIDIA SuperChips

This version of the Holoscan SDK was tested on the Grace-Hopper SuperChip (GH200) with Ubuntu 22.04. Follow setup instructions [here](#).

Attention: Display is not supported on SBSA/superchips. You can however do headless rendering with [HoloViz](#) for example.

x86_64 Workstations

Supported x86_64 distributions:

OS	NGC Container	Debian/RPM package	Python wheel	Build from source
Ubuntu 22.04	Yes	Yes	Yes	Yes
RHEL 9.x	Yes	No	No	No ¹
Other Linux distros	No ²	No	No ³	No ¹

¹ Not formally tested or supported, but expected to work if building bare metal with the adequate dependencies. ² Not formally tested or supported, but expected to work if [supported by the NVIDIA container-toolkit](#). ³ Not formally tested or supported, but expected to work if the glibc version of the distribution is 2.35 or above.

NVIDIA discrete GPU (dGPU) requirements:

- Ampere or above recommended for best performance
- [Quadro/NVIDIA RTX](#) necessary for GPUDirect RDMA support
- Tested with [NVIDIA Quadro RTX 6000](#) and [NVIDIA RTX A6000](#)
- [NVIDIA dGPU drivers](#): 535 or above
- For RDMA Support, follow the instructions in the [Enabling RDMA](#) section.
- Additional software dependencies might be needed based on how you choose to install the SDK (see section below).
- Refer to the [Additional Setup](#) and [Third-Party Hardware Setup](#) sections for additional prerequisites.

4.2 Install the SDK

We provide multiple ways to install and run the Holoscan SDK:

4.2.1 Instructions

NGC Container

- **dGPU** (x86_64, IGX Orin dGPU, Clara AGX dGPU, GH200)

```
docker pull nvcr.io/nvidia/clara-holoscan/holoscan:v2.6.0-dgpu
```

- **iGPU** (Jetson, IGX Orin iGPU, Clara AGX iGPU)

```
docker pull nvcr.io/nvidia/clara-holoscan/holoscan:v2.6.0-igpu
```

See details and usage instructions on [NGC](#).

Debian package

Try the following to install the holoscan SDK:

```
sudo apt update
sudo apt install holoscan
```

Attention: This will not install dependencies needed for the Torch nor ONNXRuntime inference backends. To do so, install transitive dependencies by adding the `--install-suggests` flag to `apt install holoscan`, and refer to the support matrix below for links to install libtorch and onnxruntime.

Troubleshooting

If **holoscan** is not found with **apt**:

```
E: Unable to locate package holoscan
```

Try the following before repeating the installation steps above:

- **IGX Orin:** Ensure the [compute stack is properly installed](#) which should configure the L4T repository source. If you still cannot install the Holoscan SDK, use the [arm64-sbsa installer](#) from the CUDA repository.
- **Jetson:** Ensure [JetPack is properly installed](#) which should configure the L4T repository source. If you still cannot install the Holoscan SDK, use the [aarch64-jetson installer](#) from the CUDA repository.
- **GH200:** Use the [arm64-sbsa installer](#) from the CUDA repository.
- **x86_64:** Use the [x86_64 installer](#) from the CUDA repository.

If you get missing CUDA libraries at runtime like below:

```
ImportError: libcudart.so.12: cannot open shared object file: No such file or directory
```

This could happen if your system has multiple CUDA Toolkit component versions installed. Find the path of the missing CUDA library (`libcudart.so.12` here) using `find /usr/local/cuda* -name libcudart.so.12` and select that path in `sudo update-alternatives --config cuda`. If that library is not found, or other cuda toolkit libraries become missing afterwards, you could try a clean reinstall of the full CUDA Toolkit:

```
sudo apt update && sudo apt install -y cuda-toolkit-12-2
```

If you get missing CUDA headers at compile time like below:

```
the link interface contains: CUDA::nppidei but the target was not found. [...] fatal
↳ error: npp.h: No such file or directory
```

Generally the same issue as above due from mixing CUDA Toolkit component versions in your environment. Confirm the path of the missing CUDA header (`npp.h` here) with `find /usr/local/cuda-* -name npp.h` and follow the same instructions as above.

If you get missing TensorRT libraries at runtime like below:

```
Error: libnvinfer.so.8: cannot open shared object file: No such file or directory
...
Error: libvonnxparser.so.8: cannot open shared object file: No such file or directory
```

This could happen if your system has a different major version installed than version 8. Try to reinstall TensorRT 8 with:

```
sudo apt update && sudo apt install -y libnvinfer-bin="8.6.*"
```

If you cannot import the holoscan Python module:

```
ModuleNotFoundError: No module named 'holoscan'
```

To leverage the python module included in the debian package (instead of installing the python wheel), include the path below to your python path:

```
export PYTHONPATH="/opt/nvidia/holoscan/python/lib"
```

Python wheel

```
pip install holoscan
```

See details and troubleshooting on [PyPI](#).

Note: For x86_64, ensure that the [CUDA Runtime](#) is installed.

4.2.2 Not sure what to choose?

- The **Holoscan container image on NGC** is the safest way to ensure all the dependencies are present with the expected versions (including Torch and ONNX Runtime), and should work on most Linux distributions. It is the simplest way to run the embedded examples, while still allowing you to create your own C++ and Python Holoscan application on top of it. These benefits come at a cost:
 - large image size from the numerous (some of them optional) dependencies. If you need a lean runtime image, see [section below](#).
 - standard inconvenience that exist when using Docker, such as more complex run instructions for proper configuration.
- If you are confident in your ability to manage dependencies on your own in your host environment, the **Holoscan Debian package** should provide all the capabilities needed to use the Holoscan SDK, assuming you are on Ubuntu 22.04.
- If you are not interested in the C++ API but just need to work in Python, or want to use a different version than Python 3.10, you can use the **Holoscan python wheels** on PyPI. While they are the easiest solution to install the SDK, it might require the most work to setup your environment with extra dependencies based on your needs. Finally, they are only formally supported on Ubuntu 22.04, though should support other linux distributions with glibc 2.35 or above.

	NGC dev Container	Debian Package	Python Wheels
Runtime libraries	Included	Included	Included
Python module	3.10	3.10	3.9 to 3.12
C++ headers and C-Make config	Included	Included	N/A
Examples (+ source)	Included	Included	retrieve from GitHub
Sample datasets	Included	retrieve from NGC	retrieve from NGC
CUDA runtime ¹	Included	automatically ² installed	require manual installation
NPP support ³	Included	automatically ^{Page 11, 2} installed	require manual installation
TensorRT support ⁴	Included	automatically ^{Page 11, 2} installed	require manual installation
Vulkan support ⁵	Included	automatically ^{Page 11, 2} installed	require manual installation
V4L2 support ⁶	Included	automatically ^{Page 11, 2} installed	require manual installation
Torch support ⁷	Included	require manual ⁸ installation	require manual ^{Page 11, 8} installation
ONNX Runtime support ⁹	Included	require manual ¹⁰ installation	require manual ^{Page 11, 10} installation
MOFED support ¹¹	User space included Install kernel drivers on the host	require manual installation	require manual installation
CLI support	Included	needs docker w/buildx plugin	needs docker w/buildx plugin

¹ CUDA 12 is required. Already installed on NVIDIA developer kits with IGX Software and JetPack.

² Debian installation on x86_64 requires the latest [cuda-keyring](#) package to automatically install all dependencies.

³ NPP 12 needed for the FormatConverter and BayerDemosaic operators. Already installed on NVIDIA developer kits with IGX Software and JetPack.

⁴ TensorRT 10.3+ needed for the Inference operator. Already installed on NVIDIA developer kits with IGX Software and JetPack.

⁵ Vulkan 1.3.204+ loader needed for the HoloViz operator (+ libegl for headless rendering). Already installed on NVIDIA developer kits with IGX Software and JetPack.

⁶ V4L2 1.22+ needed for the V4L2 operator. Already installed on NVIDIA developer kits with IGX Software and JetPack. V4L2 also requires libjpeg.

⁷ Torch support requires LibTorch 2.5+, TorchVision 0.20+, OpenBLAS 0.3.20+, OpenMPI v4.1.7a1+, UCC 1.4+, MKL 2021.1.1 (x86_64 only), NVIDIA Performance Libraries (aarch64 dGPU only), libpng, and libjpeg. Note that container builds use OpenMPI and UCC originating from the NVIDIA HPC-X package bundle.

⁸ To install LibTorch and TorchVision, either build them from source, download our [pre-built packages](#), or copy them from the holoscan container (in /opt).

⁹ ONNXRuntime 1.18.1+ needed for the Inference operator. Note that ONNX models are also supported through the TensorRT backend of the Inference Operator.

¹⁰ To install ONNXRuntime, either build it from source, download our [pre-built package](#) with CUDA 12 and TensorRT execution provider support, or copy it from the holoscan container (in /opt/onnxruntime).

¹¹ Tested with MOFED 24.07

4.2.3 Need more control over the SDK?

The [Holoscan SDK source repository](#) is **open-source** and provides reference implementations, as well as infrastructure, for building the SDK yourself.

Attention: We only recommend building the SDK from source if you need to build it with debug symbols or other options not used as part of the published packages. If you want to write your own operator or application, you can use the SDK as a dependency (and contribute to [HoloHub](#)). If you need to make other modifications to the SDK, [file a feature or bug request](#).

4.2.4 Looking for a light runtime container image?

The current Holoscan container on NGC has a large size due to including all the dependencies for each of the built-in operators, but also because of the development tools and libraries that are included. Follow the [instructions on GitHub](#) to build a runtime container without these development packages. This page also includes detailed documentation to assist you in only including runtime dependencies your Holoscan application might need.

ADDITIONAL SETUP

In addition to the required steps to *install the Holoscan SDK*, the steps below will help you achieve peak performance:

5.1 Enabling RDMA

Note: Learn more about RDMA in the *technology overview* section.

There are two parts to enabling RDMA for Holoscan:

- *Enabling RDMA on the ConnectX SmartNIC*
- *Enabling GPUDirect RDMA*

5.1.1 Enabling RDMA on the ConnectX SmartNIC

Skip to the next section if you do not plan to leverage a ConnectX SmartNIC.

The NVIDIA IGX Orin developer kit comes with an embedded [ConnectX Ethernet adapter](#) to offer advanced hardware offloads and accelerations. You can also purchase an individual ConnectX adapter and install it on other systems, such as x86_64 workstations.

The following steps are required to ensure your ConnectX can be used for RDMA over Converged Ethernet (RoCE):

1. Install MOFED drivers

Ensure the Mellanox OFED drivers version 23.10 or above are installed:

```
cat /sys/module/mlx5_core/version
```

If not installed, or an older version is installed, you should install the appropriate version from the [MLNX_OFED download page](#), or use the script below:

```
# You can choose different versions/OS or download directly from the
# Download Center in the webpage linked above
MOFED_VERSION="24.07-0.6.1.0"
OS="ubuntu22.04"
MOFED_PACKAGE="MLNX_OFED_LINUX-${MOFED_VERSION}-${OS}-${uname -m}"
wget --progress=dot:giga https://www.mellanox.com/downloads/ofed/MLNX_OFED-${MOFED_
↪VERSION}/${MOFED_PACKAGE}.tgz
```

(continues on next page)

(continued from previous page)

```
tar xf ${MOFED_PACKAGE}.tgz
sudo ./${MOFED_PACKAGE}/mlnxofedinstall
# add the --force flag to force uninstallation if necessary:
# sudo ./${MOFED_PACKAGE}/mlnxofedinstall --force
rm -r ${MOFED_PACKAGE}*
```

2. Load MOFED drivers

Ensure the drivers are loaded:

```
sudo lsmod | grep ib_core
```

If nothing appears, run the following command:

```
sudo /etc/init.d/openibd restart
```

3. Switch the board Link Layer to Ethernet

The ConnectX SmartNIC can function in two separate modes (called link layer):

- Ethernet (ETH)
- Infiniband (IB)

Holoscan does not support IB at this time (as it is not tested), so the ConnectX will need to use the ETH link layer.

To identify the current mode, run `ibstat` or `ibv_devinfo` and look for the Link Layer value. In the example below, the `mlx5_0` interface is in Ethernet mode, while the `mlx5_1` interface is in Infiniband mode. Do not pay attention to the transport value which is always InfiniBand.

```
$ ibstat
CA 'mlx5_0'
  CA type: MT4129
  Number of ports: 1
  Firmware version: 28.37.0190
  Hardware version: 0
  Node GUID: 0x48b02d0300ee7a04
  System image GUID: 0x48b02d0300ee7a04
  Port 1:
    State: Down
    Physical state: Disabled
    Rate: 40
    Base lid: 0
    LMC: 0
    SM lid: 0
    Capability mask: 0x00010000
    Port GUID: 0x4ab02dffffee7a04
    Link layer: Ethernet
CA 'mlx5_1'
  CA type: MT4129
  Number of ports: 1
```

(continues on next page)

(continued from previous page)

```

Firmware version: 28.37.0190
Hardware version: 0
Node GUID: 0x48b02d0300ee7a05
System image GUID: 0x48b02d0300ee7a04
Port 1:
    State: Active
    Physical state: LinkUp
    Rate: 100
    Base lid: 0
    LMC: 0
    SM lid: 0
    Capability mask: 0x00010000
    Port GUID: 0x4ab02dffffee7a05
    Link layer: InfiniBand

```

If no results appear after `ibstat` and `sudo lsmod | grep ib_core` returns a result like this:

```
ib_core          425984  1 ib_uverbs
```

Consider running the following command or rebooting:

```
sudo /etc/init.d/openibd restart
```

To switch the link layer mode, there are two possible options:

1. On IGX Orin developer kits, you can switch that setting through the BIOS: [see IGX Orin documentation](#).
2. On any system with a ConnectX (including IGX Orin developer kits), you can run the command below from a terminal (this will require a reboot). `sudo ibdev2netdev -v` is used to identify the PCI address of the ConnectX (any of the two interfaces is fine to use), and `mlxconfig` is used to apply the changes.

```

mlx_pci=$(sudo ibdev2netdev -v | awk '{print $1}' | head -n1)
sudo mlxconfig -d $mlx_pci set LINK_TYPE_P1=ETH LINK_TYPE_P2=ETH

```

Note: `LINK_TYPE_P1` and `LINK_TYPE_P2` are for `mlx5_0` and `mlx5_1` respectively. You can choose to only set one of them. You can pass `ETH` or `2` for Ethernet mode, and `IB` or `1` for InfiniBand.

This is the output of the command above:

```

Device #1:
-----

Device type:    ConnectX7
Name:           P3740-B0-QSFP_Ax
Description:    NVIDIA Prometheus P3740 ConnectX-7 VPI PCIe Switch Motherboard;
↪400Gb/s; dual-port QSFP; PCIe switch5.0 X8 SLOT0 ;X16 SLOT2; secure boot;
Device:         0005:03:00.0

Configurations:
                Next Boot      New
LINK_TYPE_P1   ETH(2)         ETH(2)
LINK_TYPE_P2   IB(1)          ETH(2)

Apply new Configuration? (y/n) [n] :

```

Next `Boot` is the current value that was expected to be used at the next reboot, while `New` is the value you're about to set to override `Next Boot`.

Apply with `y` and reboot afterwards:

```
Applying... Done!
-I- Please reboot machine to load new configurations.
```

4. Configure the IP addresses of the Ethernet interfaces

First, identify the logical names of your ConnectX interfaces. Connecting a cable in just one of the interfaces on the ConnectX will help you identify which port is which (in the example below, only `mlx5_1` – i.e., `eth3` – is connected):

```
$ sudo ibdev2netdev
mlx5_0 port 1 ==> eth2 (Down)
mlx5_1 port 1 ==> eth3 (Up)
```

Tip: For IGX Orin Developer Kits with no live source to connect to the ConnectX QSFP ports, adding `-v` can show you which logical name is mapped to each specific port:

- `0005:03:00.0` is the QSFP port closer to the PCI slots
- `0005:03:00.1` is the QSFP port closer to the RJ45 ethernet ports

```
$ sudo ibdev2netdev -v
0005:03:00.0 mlx5_0 (MT4129 - P3740-0002 ) NVIDIA IGX, P3740-0002, 2-port QSFP up to
↳ 400G, InfiniBand and Ethernet, PCIe5                               fw 28.37.0190 port 1 (DOWN ) ==>
↳ eth2 (Down)
0005:03:00.1 mlx5_1 (MT4129 - P3740-0002 ) NVIDIA IGX, P3740-0002, 2-port QSFP up to
↳ 400G, InfiniBand and Ethernet, PCIe5                               fw 28.37.0190 port 1 (DOWN ) ==>
↳ eth2 (Down)
```

If you have a cable connected but it does not show Up/Down in the output of `ibdev2netdev`, you can try to parse the output of `dmesg` instead. The example below shows that `0005:03:00.1` is plugged, and that it is associated with `eth3`:

```
$ sudo dmesg | grep -w mlx5_core
...
[ 11.512808] mlx5_core 0005:03:00.0 eth2: Link down
[ 11.640670] mlx5_core 0005:03:00.1 eth3: Link down
...
[ 3712.267103] mlx5_core 0005:03:00.1: Port module event: module 1, Cable plugged
```

The next step is to set a static IP on the interface you'd like to use so you can refer to it in your Holoscan applications (e.g., *Emergent cameras, distributed applications...*).

First, check if you already have an address setup. We'll use the `eth3` interface in this example for `mlx5_1`:

```
ip -f inet addr show eth3
```

If nothing appears, or you'd like to change the address, you can set an IP and MTU (Maximum Transmission Unit) through the Network Manager user interface, CLI (`nmcli`), or other IP configuration tools. In the example below, we use `ip` (`ifconfig` is legacy) to configure the `eth3` interface with an address of `192.168.1.1/24` and a MTU of `9000` (i.e., “jumbo frame”) to send Ethernet frames with a payload greater than the standard size of 1500 bytes:

```
sudo ip link set dev eth3 down
sudo ip addr add 192.168.1.1/24 dev eth3
sudo ip link set dev eth3 mtu 9000
sudo ip link set dev eth3 up
```

Note: If you are connecting the ConnectX to another ConnectX with a [LinkX interconnect](#), do the same on the other system with an IP address on the same network segment.

For example, to communicate with `192.168.1.1/24` above (`/24 -> 255.255.255.0` submask), setup your other system with an IP between `192.168.1.2` and `192.168.1.254`, and the same `/24` submask.

5.1.2 Enabling GPUDirect RDMA

Note: Only supported on NVIDIA's Quadro/workstation GPUs (not GeForce).

Follow the instructions below to enable [GPUDirect RDMA](#):

dGPU

On dGPU, the GPUDirect RDMA drivers are named `nvidia-peermem`, and are installed with the rest of the NVIDIA dGPU drivers.

Attention: To enable the use of GPUDirect RDMA with a ConnectX SmartNIC (section above), the following steps are required if the MOFED drivers were installed after the `peermem` drivers:

```
nv_driver_version=$(modinfo nvidia | awk '/^version:/ {print $2}' | cut -d. -f1)
sudo dpkg-reconfigure nvidia-dkms-$nv_driver_version # or nvidia-dkms-${nv_driver_
↪version}-server
```

Load the `peermem` kernel module manually:

```
sudo modprobe nvidia-peermem
```

Run the following to load it automatically during boot:

```
echo nvidia-peermem | sudo tee -a /etc/modules
```

iGPU

Warning: At this time, the IGX SW 1.0 DP and JetPack 6.0 DP are missing the `nvidia-p2p` kernel for support for GPU Direct RDMA support. They're planned in the respective GA releases. The instructions below are to load the kernel module once it is packaged in the GA releases.

On iGPU, the GPUDirect RDMA drivers are named `nvidia-p2p`. Run the following to load the kernel module manually:

```
sudo modprobe nvidia-p2p
```

Run the following to load it automatically during boot:

```
echo nvidia-p2p | sudo tee -a /etc/modules
```

5.1.3 Testing with Rivermax

The instructions below describe the steps to test GPUDirect using the [Rivermax](#) SDK. The test applications used by these instructions, `generic_sender` and `generic_receiver`, can then be used as samples in order to develop custom applications that use the Rivermax SDK to optimize data transfers.

Note: The Linux default path where Rivermax expects to find the license file is `/opt/mellanox/rivermax/rivermax.lic`, or you can specify the full path and file name for the environment variable `RIVERMAX_LICENSE_PATH`.

Note: If manually installing the Rivermax SDK from the link above, please note there is no need to follow the steps for installing `MLNX_OFED/MLNX_EN` in the Rivermax documentation.

Running the Rivermax sample applications requires two systems, a sender and a receiver, connected via ConnectX network adapters. If two Developer Kits are used, then the onboard ConnectX can be used on each system. However, if only one Developer Kit is available, then it is expected that another system with an add-in ConnectX network adapter will need to be used. Rivermax supports a wide array of platforms, including both Linux and Windows, but these instructions assume that another Linux based platform will be used as the sender device while the Developer Kit is used as the receiver.

Note: The `$rivermax_sdk` variable referenced below corresponds to the path where the Rivermax SDK package is installed. If the Rivermax SDK was installed via SDK Manager, this path will be:

```
rivermax_sdk=$HOME/Documents/Rivermax/1.31.10
```

If the Rivermax SDK was installed via a manual download, make sure to export your path to the SDK:

```
rivermax_sdk=$DOWNLOAD_PATH/1.31.10
```

The install path might differ in future versions of Rivermax.

1. Determine the logical name for the ConnectX devices that are used by each system. This can be done by using the `lshw -class network` command, finding the `product:` entry for the ConnectX device, and making note of the `logical name:` that corresponds to that device. For example, this output on a Developer Kit shows

the onboard ConnectX device using the `enp9s0f01` logical name (lshw output shortened for demonstration purposes).

```
$ sudo lshw -class network
*-network:0
    description: Ethernet interface
    product: MT28908 Family [ConnectX-6]
    vendor: Mellanox Technologies
    physical id: 0
    bus info: pci@0000:09:00.0
    <b>logical name: enp9s0f0</b>
    version: 00
    serial: 48:b0:2d:13:9b:6b
    capacity: 10Gbit/s
    width: 64 bits
    clock: 33MHz
    capabilities: pciexpress vpd msix pm bus_master cap_list ethernet physical_
    ↳1000bt-fd 10000bt-fd autonegotiation
    configuration: autonegotiation=on broadcast=yes driver=mlx5_core_
    ↳driverversion=5.4-1.0.3 duplex=full firmware=20.27.4006 (NVD0000000001) ip=10.0.0.
    ↳2 latency=0 link=yes multicast=yes
    resources: iomemory:180-17f irq:33 memory:1818000000-1819ffffff
```

The instructions that follow will use the `enp9s0f0` logical name for `ifconfig` commands, but these names should be replaced with the corresponding logical names as determined by this step.

2. Run the `generic_sender` application on the sending system.

a. Bring up the network:

```
$ sudo ifconfig enp9s0f0 up 10.0.0.1
```

b. Build the sample apps:

```
$ cd ${rivermax_sdk}/apps
$ make
```

e. Launch the `generic_sender` application:

```
$ sudo ./generic_sender -l 10.0.0.1 -d 10.0.0.2 -p 5001 -y 1462 -k 8192 -z 500 -v
```

which gives

```
+#####
| Sender index: 0
| Thread ID: 0x7fa1ffb1c0
| CPU core affinity: -1
| Number of streams in this thread: 1
| Memory address: 0x7f986e3010
| Memory length: 59883520[B]
| Memory key: 40308
+#####
| Stream index: 0
| Source IP: 10.0.0.1
| Destination IP: 10.0.0.2
```

(continues on next page)

(continued from previous page)

```

| Destination port: 5001
| Number of flows: 1
| Rate limit bps: 0
| Rate limit max burst in packets: 0
| Memory address: 0x7f986e3010
| Memory length: 59883520[B]
| Memory key: 40308
| Number of user requested chunks: 1
| Number of application chunks: 5
| Number of packets in chunk: 8192
| Packet's payload size: 1462
+*****

```

3. Run the `generic_receiver` application on the receiving system.

a. Bring up the network:

```
$ sudo ifconfig enp9s0f0 up 10.0.0.2
```

b. Build the `generic_receiver` app with GPUDirect support from the [Rivermax GitHub Repo](#). Before following the instructions to [build with CUDA Toolkit support](#), apply the changes to the file `generic_receiver/generic_receiver.cpp` in [this PR](#). This was tested on the NVIDIA IGX Orin Developer Kit with Rivermax 1.31.10.

c. Launch the `generic_receiver` application from the build directory:

```

$ sudo ./generic_receiver -i 10.0.0.2 -m 10.0.0.2 -s 10.0.0.1 -p 5001 -g 0
...
Attached flow 1 to stream.
Running main receive loop...
Got 5877704 GPU packets | 68.75 Gbps during 1.00 sec
Got 5878240 GPU packets | 68.75 Gbps during 1.00 sec
Got 5878240 GPU packets | 68.75 Gbps during 1.00 sec
Got 5877704 GPU packets | 68.75 Gbps during 1.00 sec
Got 5878240 GPU packets | 68.75 Gbps during 1.00 sec
...

```

With both the `generic_sender` and `generic_receiver` processes active, the receiver will continue to print out received packet statistics every second. Both processes can then be terminated with `<ctrl-c>`.

5.2 Enabling G-SYNC

For better performance and to keep up with the high refresh rate of Holoscan applications, we recommend the use of a G-SYNC display.

Tip: Holoscan has been tested with these two G-SYNC displays:

- [Asus ROG Swift PG279QM](#)
- [Asus ROG Swift 360 Hz PG259QNR](#)

Follow these steps to ensure G-SYNC is enabled on your display:

1. Open the “NVIDIA Settings” Graphics application (`nvidia-settings` in Terminal).
2. Click on **X Server Display Configuration** then the **Advanced** button. This will show the **Allow G-SYNC on monitor not validated as G-SYNC compatible** option. Enable the option and click **Apply**:
3. To show the refresh rate and G-SYNC label on the display window, click on **OpenGL Settings** for the selected display. Now click **Allow G-SYNC/G-SYNC Compatible** and **Enable G-SYNC/G-SYNC Compatible Visual Indicator** options, then click **Quit**. This step is shown in the image below. The Gsync indicator is at the top right of the screen once the application is running.

5.3 Disabling Variable Backlight

Various monitors have a Variable Backlight feature. That setting can add up to a frame of latency when enabled. Refer to your monitor’s manufacturer instructions to disable it.

Tip: To disable variable backlight on the ASUS ROG Swift monitors mentioned above, use the joystick button at the back of the display, go to the `image` tag, select `variable backlight`, then switch that setting to `OFF`.

5.4 Enabling Exclusive Display Mode

By default, applications use a borderless full screen window managed by the window manager. Because the window manager also manages other applications, applications may suffer a performance hit. To improve performance, exclusive display mode can be used with Holoscan’s new visualization module (Holoviz), allowing the application to bypass the window manager and render directly to the display. Refer to the [Holoviz documentation](#) for details.

5.5 Use both Integrated and Discrete GPUs on NVIDIA Developer Kits

NVIDIA Developer Kits like the [NVIDIA IGX Orin](#) or the [NVIDIA Clara AGX](#) have both a discrete GPU (dGPU - optional on IGX Orin) and an integrated GPU (iGPU - Tegra SoC).

As of this release, when these developer kits are flashed to leverage the dGPU, there are two limiting factors preventing the use of the iGPU:

1. Conflict between the dGPU kernel mode driver and the iGPU display kernel driver (both named `nvidia.ko`). This conflict is not addressable at this time, meaning that **the iGPU cannot be used for display while the dGPU is enabled**.
2. Conflicts between the user mode driver libraries (ex: `libcuda.so`) and the compute stack (ex: `libcuda_rt.so`) for dGPU and iGPU.

We provide utilities to work around the second conflict:

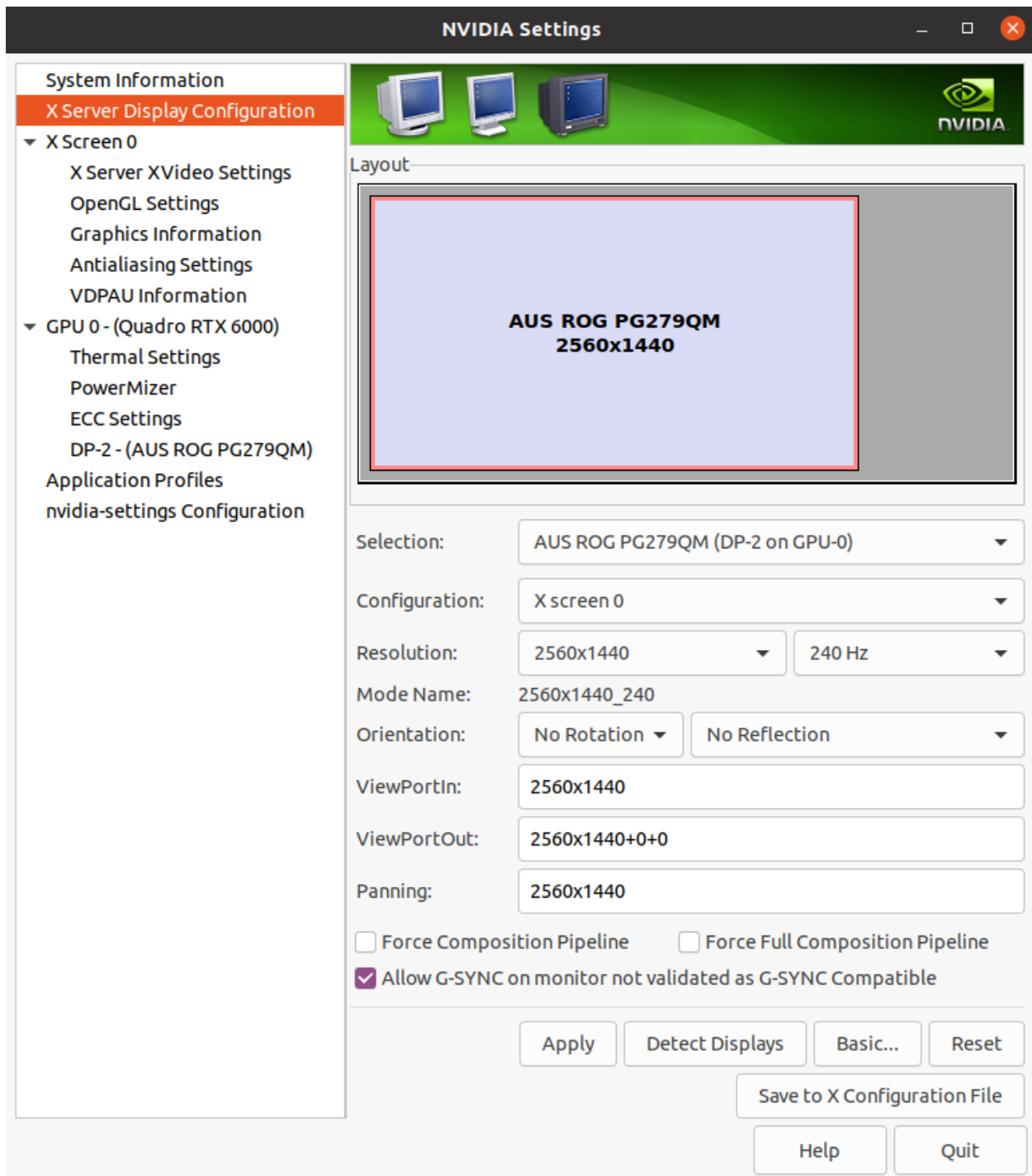


Fig. 5.1: Enable G-SYNC for the current display

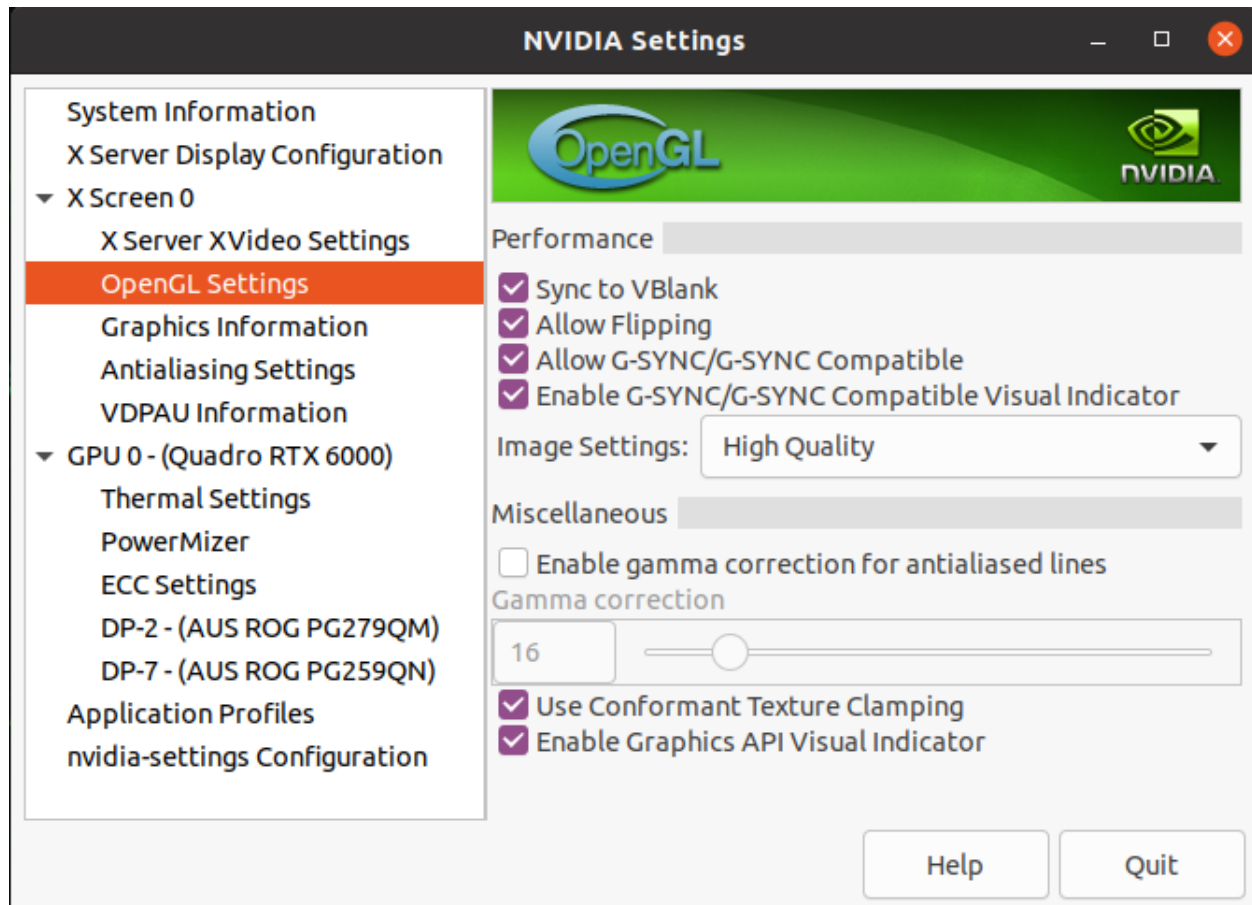


Fig. 5.2: Enable Visual Indicator for the current display

IGX SW 1.0

1. From an IGX developer kit flashed for dGPU, run the following command to enable iGPU container support:

```
sudo /opt/nvidia/l4t-igpu-container-on-dgpu-host-config/l4t-igpu-container-on-dgpu-  
↪host-config.sh configure
```

Refer to the [IGX user guide](#) for details.

2. To leverage both GPUs in Holoscan, you can either:
 1. create separate Holoscan applications running concurrently, where the iGPU application must run in the Holoscan iGPU container, and the dGPU application can run bare metal or in the Holoscan dGPU container. Refer to the [IGX user guide](#) for details on how to launch a Holoscan container using the iGPU.
 2. create a single distributed application that leverages both the iGPU and dGPU by executing separate fragments on the iGPU and on the dGPU.

The example below shows the ping distributed application between the iGPU and dGPU using Holoscan containers:

```
COMMON_DOCKER_FLAGS="--rm -i --init --net=host  
--runtime=nvidia -e NVIDIA_DRIVER_CAPABILITIES=all  
--cap-add CAP_SYS_PTRACE --ipc=host --ulimit memlock=-1 --ulimit stack=67108864  
"  
  
HOLOSCAN_VERSION=2.6.0  
HOLOSCAN_IMG="nvcr.io/nvidia/clara-holoscan/holoscan:v$HOLOSCAN_VERSION"  
HOLOSCAN_DGPU_IMG="$HOLOSCAN_IMG-dgpu"  
HOLOSCAN_IGPU_IMG="$HOLOSCAN_IMG-igpu"  
  
# Pull images  
docker pull $HOLOSCAN_DGPU_IMG  
docker pull $HOLOSCAN_IGPU_IMG  
  
# Run ping distributed (python) in dGPU container  
# - Making this one the `driver`, but could be igpu too  
# - Using & to not block the terminal to run igpu afterwards. Could run igpu in separate_  
↪terminal instead.  
docker run \  
    $COMMON_DOCKER_FLAGS \  
    $HOLOSCAN_DGPU_IMG \  
    bash -c "python3 ./examples/ping_distributed/python/ping_distributed.py --gpu --worker_  
↪--driver" &  
  
# Run ping distributed (c++) in iGPU container  
docker run \  
    $COMMON_DOCKER_FLAGS \  
    -e NVIDIA_VISIBLE_DEVICES=nvidia.com/igpu=0 \  
    $HOLOSCAN_IGPU_IMG \  
    bash -c "./examples/ping_distributed/cpp/ping_distributed --gpu --worker"
```

HoloPack 1.2+

The [L4T Compute Assist](#) is a container on NGC which isolates the iGPU stack by containing the L4T BSP packages in order to enable iGPU compute on the developer kits configured for dGPU. Other applications can run concurrently on the dGPU, natively or in another container.

Attention: These utilities enable using the iGPU for capabilities other than **display** only, since they do not address the first conflict listed above.

5.6 Deployment Software Stack

NVIDIA Holoscan accelerates deployment of production-quality applications by providing a set of **OpenEmbedded** build recipes and reference configurations that can be leveraged to customize and build Holoscan-compatible Linux4Tegra (L4T) embedded board support packages (BSP) on the NVIDIA IGX Developer Kits.

[Holoscan OpenEmbedded/Yocto recipes](#) add OpenEmbedded recipes and sample build configurations to build BSPs for the NVIDIA IGX Developer Kit that feature support for discrete GPUs (dGPU), AJA Video Systems I/O boards, and the Holoscan SDK. These BSPs are built on a developer's host machine and are then flashed onto the NVIDIA IGX Developer Kit using provided scripts.

There are two options available to set up a build environment and start building Holoscan BSP images using OpenEmbedded.

1. The first sets up a local build environment in which all dependencies are fetched and installed manually by the developer directly on their host machine. Please refer to the [Holoscan OpenEmbedded/Yocto recipes README](#) for more information on how to use the local build environment.
2. The second uses a [Holoscan OpenEmbedded/Yocto Build Container](#) that is provided by NVIDIA on NGC, which contains all of the dependencies and configuration scripts so the entire process of building and flashing a BSP can be done with just a few simple commands.

THIRD PARTY HARDWARE SETUP

GPU compute performance is a key component of the Holoscan hardware platforms, and in order to optimize GPU-based video processing applications and provide lowest possible latency, the Holoscan SDK now supports AJA Video Systems capture cards and Emergent Vision Technologies high-speed cameras. The following sections will provide more information on how to setup the system with these technologies.

6.1 AJA Video Systems

AJA provides a wide range of proven, professional video I/O devices, and thanks to a partnership between NVIDIA and AJA, Holoscan provides ongoing support for the AJA NTV2 SDK and device drivers.

The AJA drivers and SDK offer RDMA support for NVIDIA GPUs. This feature allows video data to be captured directly from the AJA card to GPU memory, which significantly reduces latency and system PCI bandwidth for GPU video processing applications as system to GPU copies are eliminated from the processing pipeline.

The following instructions describe the steps required to setup and use an AJA device with RDMA support on NVIDIA Developer Kits with a PCIe slot. Note that the AJA NTV2 SDK support for Holoscan includes all of the [AJA Developer Products](#), though the following instructions have only been verified for the [Corvid 44 12G BNC](#), [KONA XM](#), and [KONA HDMI](#) products, specifically.

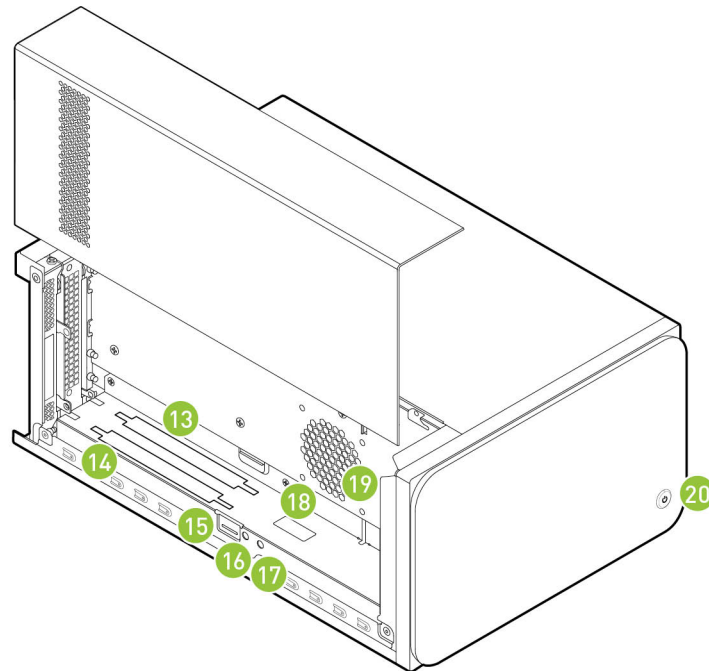
Note: The addition of an AJA device to a NVIDIA Developer Kit is optional. The Holoscan SDK has elements that can be run with an AJA device with the additional features mentioned above, but those elements can also run without AJA. For example, there are Holoscan sample applications that have an AJA live input component, however they can also take in video replay as input. Similarly, the latency measurement tool can measure the latency of the video I/O subsystem with or without an AJA device available.

6.1.1 Installing the AJA Hardware

This section describes how to install the AJA hardware on the Clara AGX Developer Kit. Note that the AJA Hardware is also compatible with the NVIDIA IGX Orin Developer Kit.

To install an AJA Video Systems device into the Clara AGX Developer Kit, remove the side access panel by removing two screws on the back of the Clara AGX. This provides access to the two available PCIe slots, labelled 13 and 14 in the [Clara AGX Developer Kit User Guide](#):

While these slots are physically identical PCIe x16 slots, they are connected to the Clara AGX via different PCIe bridges. Only slot 14 shares the same PCIe bridge as the RTX6000 dGPU, and so the AJA device must be installed into slot 14 for RDMA support to be available. The following image shows a [Corvid 44 12G BNC](#) card installed into slot 14 as needed to enable RDMA support.



6.1.2 Installing the AJA Software

The AJA NTV2 SDK includes both the drivers (kernel module) that are required in order to enable an AJA device, as well as the SDK (headers and libraries) that are used to access an AJA device from an application.

The drivers must be loaded every time the system is rebooted, and they must be loaded natively on the host system (i.e. not inside a container). The drivers must be loaded regardless of whether applications will be run natively or inside a container (see *Using AJA Devices in Containers*).

The SDK only needs to be installed on the native host and/or container that will be used to compile applications with AJA support. The Holoscan SDK containers already have the NTV2 SDK installed, and so no additional steps are required to build AJA-enabled applications (such as the reference Holoscan applications) within these containers. However, installing the NTV2 SDK and utilities natively on the host is useful for the initial setup and testing of the AJA device, so the following instructions cover this native installation.

Note: To summarize, the steps in this section must be performed on the native host, outside of a container, with the following steps **required once**:

- *Downloading the AJA NTV2 SDK Source*
- *Building the AJA NTV2 Drivers*

The following steps **required after every reboot**:

- *Loading the AJA NTV2 Drivers*

And the following steps are **optional** (but recommended during the initial setup):

- *Building and Installing the AJA NTV2 SDK*
 - *Testing the AJA Device*
-

Using the AJA NTV2 Driver and SDK Build Script

Included in the *scripts* directory is the *aja_build.sh* script which can be used to download the AJA NTV2 source, build the drivers and SDK, load the drivers, and run the *ntv2enumerateboards* utility to list the AJA boards that are connected to the system. To download and build the drivers and SDK, simply run the script:

```
$ ./scripts/aja_build.sh
```

To optionally have the script load the drivers and list the connected devices once the build is complete, add the *--load-driver* flag:

```
$ ./scripts/aja_build.sh --load-driver
```

Note: The remainder of the steps in this documentation describe how to manually build and load the AJA NTV2 drivers and SDK, and are not needed when using the build script. However, it will still be required to reload the drivers after rebooting the system by running the *load_ajantv2* command as described in *Loading the AJA NTV2 Drivers*.

Downloading the AJA NTV2 SDK Source

Navigate to a directory where you would like the source code to be downloaded, then perform the following to clone the NTV2 SDK source code.

```
$ git clone https://github.com/nvidia-holoscan/libajantv2.git
$ export NTV2=$(pwd)/libajantv2
```

Note: These instructions use a fork of the official [AJA NTV2 Repository](#) that is maintained by NVIDIA and may contain additional changes that are required for Holoscan SDK support. These changes will be pushed to the official AJA NTV2 repository whenever possible with the goal to minimize or eliminate divergence between the two repositories.

Installing the NVIDIA Open Kernel Modules for RDMA Support

If the AJA NTV2 drivers are going to be built with RDMA support, the open-source NVIDIA kernel modules must be installed instead of the default proprietary drivers. If the drivers were installed from an NVIDIA driver installer package then follow the directions on the [NVIDIA Open GPU Kernel Module Source GitHub](#) page. If the NVIDIA drivers were installed using an Ubuntu package via *apt*, then replace the installed *nvidia-kernel-source* package with the corresponding *nvidia-kernel-open* package. For example, the following shows that the 545 version drivers are installed:

```
S dpkg --get-selections | grep nvidia-kernel-source
ii nvidia-kernel-source-545 545.23.08-0ubuntu1 amd64 NVIDIA kernel_
↳source package
```

And the following will replace those with the corresponding *nvidia-kernel-open* drivers:

```
S sudo apt install -y nvidia-kernel-open-545
$ sudo dpkg-reconfigure nvidia-dkms-545
```

The system must then be rebooted to load the new open kernel modules.

Building the AJA NTV2 Drivers

The following will build the AJA NTV2 drivers with RDMA support enabled. Once built, the kernel module (*ajantv2.ko*) and load/unload scripts (*load_ajantv2* and *unload_ajantv2*) will be output to the `${NTV2}/driver/bin` directory.

```
$ export AJA_RDMA=1 # Or unset AJA_RDMA to disable RDMA support
$ unset AJA_IGPU # Or export AJA_IGPU=1 to run on the integrated GPU of the_
↳IGX Orin Devkit (L4T >= 35.4)
$ make -j --directory ${NTV2}/driver/linux
```

Loading the AJA NTV2 Drivers

Running any application that uses an AJA device requires the AJA kernel drivers to be loaded, even if the application is being run from within a container.

Note: To enable RDMA with AJA, ensure the *NVIDIA GPUDirect RDMA kernel module is loaded* before the AJA NTV2 drivers.

The AJA drivers must be manually loaded every time the machine is rebooted using the **load_ajantv2** script:

```
$ sudo sh ${NTV2}/driver/bin/load_ajantv2
loaded ajantv2 driver module
```

Note: The NTV2 environment variable must point to the NTV2 SDK path where the drivers were previously built as described in *Building the AJA NTV2 Drivers*.

Secure boot must be disabled in order to load unsigned module. If any errors occur while loading the module refer to the *Troubleshooting* section, below.

Building and Installing the AJA NTV2 SDK

Since the AJA NTV2 SDK is already loaded into the Holoscan containers, this step is not strictly required in order to build or run any Holoscan applications. However, this builds and installs various tools that can be useful for testing the operation of the AJA hardware outside of Holoscan containers, and is required for the steps provided in *Testing the AJA Device*.

```
$ sudo apt-get install -y cmake
$ mkdir ${NTV2}/cmake-build
$ cd ${NTV2}/cmake-build
$ export PATH=/usr/local/cuda/bin:${PATH}
$ cmake ..
$ make -j
$ sudo make install
```

Testing the AJA Device

The following steps depend on tools that were built and installed by the previous step, *Building and Installing the AJA NTV2 SDK*. If any errors occur, see the *Troubleshooting* section, below.

1. To ensure that an AJA device has been installed correctly, the `ntv2enumerateboards` utility can be used:

```
$ ntv2enumerateboards
AJA NTV2 SDK version 16.2.0 build 3 built on Wed Feb 02 21:58:01 UTC 2022
1 AJA device(s) found:
AJA device 0 is called 'KonaHDMI - 0'

This device has a deviceID of 0x10767400
This device has 0 SDI Input(s)
This device has 0 SDI Output(s)
This device has 4 HDMI Input(s)
```

(continues on next page)

(continued from previous page)

```
This device has 0 HDMI Output(s)
This device has 0 Analog Input(s)
This device has 0 Analog Output(s)
```

```
47 video format(s):
    1080i50, 1080i59.94, 1080i60, 720p59.94, 720p60, 1080p29.97, 1080p30,
    1080p25, 1080p23.98, 1080p24, 2Kp23.98, 2Kp24, 720p50, 1080p50b,
    1080p59.94b, 1080p60b, 1080p50a, 1080p59.94a, 1080p60a, 2Kp25, 525i59.94,
    625i50, UHDp23.98, UHDp24, UHDp25, 4Kp23.98, 4Kp24, 4Kp25, UHDp29.97,
    UHDp30, 4Kp29.97, 4Kp30, UHDp50, UHDp59.94, UHDp60, 4Kp50, 4Kp59.94,
    4Kp60, 4Kp47.95, 4Kp48, 2Kp60a, 2Kp59.94a, 2Kp29.97, 2Kp30, 2Kp50a,
    2Kp47.95a, 2Kp48a
```

2. To ensure that RDMA support has been compiled into the AJA driver and is functioning correctly, the `rdmawhacker` utility can be used (use `<ctrl-c>` to terminate):

```
$ rdmawhacker

DMA engine 1 WRITE 8388608 bytes  rate: 3975.63 MB/sec  496.95 xfers/sec
Max rate: 4010.03 MB/sec
Min rate: 3301.69 MB/sec
Avg rate: 3923.94 MB/sec
```

6.1.3 Using AJA Devices in Containers

Accessing an AJA device from a container requires the drivers to be loaded natively on the host (see [Loading the AJA NTV2 Drivers](#)), then the device that is created by the `load_ajantv2` script must be shared with the container using the `--device` docker argument, such as `--device /dev/ajantv20:/dev/ajantv20`.

6.1.4 Troubleshooting

1. **Problem:** The `sudo sh ${NTV2}/driver/bin/load_ajantv2` command returns an error.

Solutions:

- a. Make sure the AJA card is properly installed and powered (see 2.a below)
- b. Check if SecureBoot validation is disabled:

```
$ sudo mokutil --sb-state
SecureBoot enabled
SecureBoot validation is disabled in shim
```

If SecureBoot validation is enabled, disable it with the following procedure:

```
$ sudo mokutil --disable-validation
```

- Enter a temporary password and reboot the system.
- Upon reboot press any key when you see the blue screen MOK Management
- Select Change Secure Boot state
- Enter the password your selected

- Select Yes to disable Secure Boot in shim-signed
- After reboot you can verify again that SecureBoot validation is disabled in shim.

2. **Problem:** The `ntv2enumerateboards` command does not find any devices.

Solutions:

- a. Make sure that the AJA device is installed properly and detected by the system (see [Installing the AJA Hardware](#)):

```
$ lspci
0000:00:00.0 PCI bridge: NVIDIA Corporation Device 1ad0 (rev a1)
0000:05:00.0 Multimedia video controller: AJA Video Device eb25 (rev 01)
0000:06:00.0 PCI bridge: Mellanox Technologies Device 1976
0000:07:00.0 PCI bridge: Mellanox Technologies Device 1976
0000:08:00.0 VGA compatible controller: NVIDIA Corporation Device 1e30 (rev a1)
```

- b. Make sure that the AJA drivers are loaded properly (see [Loading the AJA NTV2 Drivers](#)):

```
$ lsmod
Module              Size  Used by
ajantv2             610066  0
nvidia_drm          54950   4
mlx5_ib             170091   0
nvidia_modeset     1250361  8 nvidia_drm
ib_core             211721   1 mlx5_ib
nvidia              34655210 315 nvidia_modeset
```

3. **Problem:** The `rdmawhacker` command outputs the following error:

```
## ERROR: GPU buffer lock failed
```

Solution: The AJA drivers need to be compiled with RDMA support enabled. Follow the instructions in [Building the AJA NTV2 Drivers](#), making sure not to skip the `export AJA_RDMA=1` when building the drivers.

6.2 Emergent Vision Technologies (EVT)

Thanks to a collaboration with [Emergent Vision Technologies](#), the Holoscan SDK now supports EVT high-speed cameras on NVIDIA Developer Kits equipped with a [ConnectX NIC](#) using the [Rivermax SDK](#).

6.2.1 Installing EVT Hardware

The EVT cameras can be connected to NVIDIA Developer Kits through a [Mellanox ConnectX SmartNIC](#), with the most simple connection method being a single cable between a camera and the devkit. For 25 GbE cameras that use the SFP28 interface, this can be achieved by using SFP28 cable with [QSFP28 to SFP28 adaptor](#).

Note: The Holoscan SDK application has been tested using a SFP28 copper cable of 2M or less. Longer copper cables or optical cables and optical modules can be used but these have not been tested as a part of this development.

Refer to the [NVIDIA IGX Orin Developer Kit User Guide](#) for the location of the QSFP28 connector on the device.

For EVT camera setup, refer to Hardware Installation in [EVT Camera User's Manual](#). Users need to log in to find be able to download Camera User's Manual.

Tip: The EVT cameras require the user to buy the lens. Based on the application of camera, the lens can be bought from any [online](#) store.

6.2.2 Installing EVT Software

The Emergent SDK needs to be installed in order to compile and run the Clara Holoscan applications with EVT camera. The latest tested version of the Emergent SDK is eSDK 2.37.05 Linux Ubuntu 20.04.04 Kernel 5.10.65 JP 5.0 HP and can be downloaded from [here](#). The Emergent SDK comes with headers, libraries and examples. To install the SDK refer to the Software Installation section of EVT [Camera User's Manual](#). Users need to log in to find be able to download Camera User's Manual.

Note: The Emergent SDK depends on Rivermax SDK and the Mellanox OFED Network Drivers. If they're already installed on your system, use the following command when installing the Emergent SDK to avoid duplicate installation:

```
sudo ./install_eSdk.sh no_mellanox
```

Ensure the *ConnectX is properly configured* to use it with the Emergent SDK.

6.2.3 Testing the EVT Camera

To test if the EVT camera and SDK was installed correctly, run the eCapture application with sudo privileges. First, ensure that a valid Rivermax license file is under /opt/mellanox/rivermax/rivermax.lic, then follow the instructions under the eCapture section of EVT [Camera User's Manual](#).

6.2.4 Troubleshooting

1. **Problem:** The application fails to find the EVT camera.

Solution:

- Make sure that the MLNX ConnectX SmartNIC is configured with the correct IP address. Follow section *Configure the ConnectX SmartNIC*

2. **Problem:** The application fails to open the EVT camera.

Solutions:

- Make sure that the application was run with sudo privileges.
- Make sure a valid Rivermax license file is located at /opt/mellanox/rivermax/rivermax.lic.

3. **Problem:** Fail to find eCapture application in the home window.

Solution:

- Open the terminal and find it under /opt/EVT/eCapture. The applications needs to be run with sudo privileges.

4. **Problem:** The eCapture application fails to connect to the EVT camera with error message "GVCP ack error".

Solutions: It could be an issue with the HR12 power connection to the camera. Disconnect the HR12 power connector from the camera and try reconnecting it.

5. **Problem:** The IP address of the Emergent camera is reset even after setting up with the above steps.

Solutions: Check whether the NIC settings in Ubuntu is set to “Connect automatically”. Go to **Settings->Network->NIC** for the Camera and then unselect “Connect automatically” and in the IPv6 tab, select **Disable**.

HOLOSCAN CORE CONCEPTS

Note: In its early days, the Holoscan SDK was tightly linked to the *GXF core concepts*. While the Holoscan SDK still relies on GXF as a backend to execute applications, it now offers its own interface, including a C++ API (0.3), a Python API (0.4), and the ability to write native operators (0.4) without requiring to wrap a GXF extension. Read the *Holoscan and GXF* section for additional details.

An **Application** is composed of **Fragments**, each of which runs a graph of **Operators**. The implementation of that graph is sometimes referred to as a pipeline, or workflow, which can be visualized below:

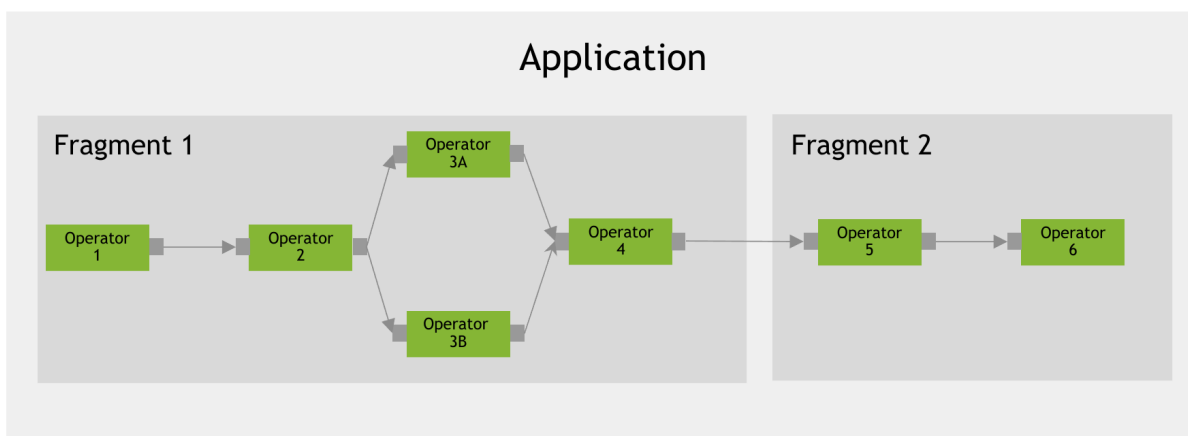


Fig. 7.1: Core concepts: Application

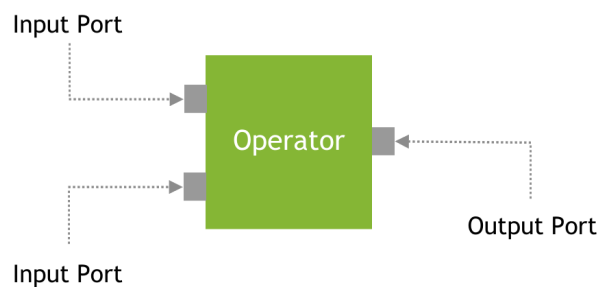


Fig. 7.2: Core concepts: Port

The core concepts of the Holoscan API are:

- **Application:** An application acquires and processes streaming data. An application is a collection of fragments where each fragment can be allocated to execute on a physical node of a Holoscan cluster.
- **Fragment:** A fragment is a building block of the application. It is a directed graph of operators. A fragment can be assigned to a physical node of a Holoscan cluster during execution. The runtime execution manages communication across fragments. In a fragment, Operators (Graph Nodes) are connected to each other by flows (Graph Edges).
- **Operator:** An operator is the most basic unit of work in this framework. An operator receives streaming data at an input port, processes it, and publishes it to one of its output ports. A *Codelet* in GXF would be replaced by an *Operator* in the Holoscan SDK. An *Operator* encapsulates *Receivers* and *Transmitters* of a GXF *Entity* as *Input/Output Ports* of the *Operator*.
- **(Operator) Resource:** Resources such as system memory or a GPU memory pool that an operator needs to perform its job. Resources are allocated during the initialization phase of the application. This matches the semantics of GXF's *Memory Allocator* or any other components derived from the *Component* class in GXF.
- **Condition:** A condition is a predicate that can be evaluated at runtime to determine if an operator should execute. This matches the semantics of GXF's *Scheduling Term*.
- **Port:** A port is an interaction point between two operators. Operators ingest data at Input ports and publish data at Output ports. *Receiver*, *Transmitter*, and *MessageRouter* in GXF would be replaced with the concept of *Input/Output Port* of the *Operator* and the *Flow (Edge)* of the *Application Workflow (DAG)* in the Framework.
- **Message:** A message is a generic data object used by operators to communicate information.
- **Executor:** An executor that manages the execution of a fragment on a physical node. The framework provides a default executor that uses a GXF *Scheduler* to execute an application.

HOLOSCAN BY EXAMPLE

In this section, we demonstrate how to use the Holoscan SDK to build applications through a series of examples. The concepts needed to build your own Holoscan applications will be covered as we go through each example.

Note: Example source code and run instructions can be found in the [examples](#) directory on GitHub, or under `/opt/nvidia/holoscan/examples` in the NGC container and the Debian package, alongside their executables.

8.1 Hello World

For our first example, we look at how to create a Hello World example using the Holoscan SDK.

In this example we'll cover:

- How to define your application class.
- How to define a one-operator workflow.
- How to use a `CountCondition` to limit the number of times an operator is executed.

Note: The example source code and run instructions can be found in the [examples](#) directory on GitHub, or under `/opt/nvidia/holoscan/examples` in the NGC container and the Debian package, alongside their executables.

8.1.1 Defining the HelloWorldApp class

For more details, see the [Defining an Application Class](#) section.

We define the `HelloWorldApp` class that inherits from holoscan's `Application` base class. An instance of the application is created in `main`. The `run()` method will then start the application.

C++

```
26 class HelloWorldApp : public holoscan::Application {
27     public:
28         void compose() override {
29             using namespace holoscan;
30
31             // Define the operators, allowing the hello operator to execute once
32             auto hello = make_operator<ops::HelloWorldOp>("hello", make_condition<CountCondition>
33             ↪(1));
34
35             // Define the workflow by adding operator into the graph
36             add_operator(hello);
37         }
38     };
39
40     int main(int argc, char** argv) {
41         auto app = holoscan::make_application<HelloWorldApp>();
42         app->run();
43
44         return 0;
45     }
46 }
```

Python

```
21 class HelloWorldApp(Application):
22     def compose(self):
23         # Define the operators
24         hello = HelloWorldOp(self, CountCondition(self, 1), name="hello")
25
26         # Define the one-operator workflow
27         self.add_operator(hello)
28
29     def main():
30         app = HelloWorldApp()
31         app.run()
32
33     if __name__ == "__main__":
34         main()
```

8.1.2 Defining the HelloWorldApp workflow

For more details, see the [Application Workflows](#) section.

When defining your application class, the primary task is to define the operators used in your application, and the interconnectivity between them to define the application workflow. The HelloWorldApp uses the simplest form of a workflow which consists of a single operator: HelloWorldOp.

For the sake of this first example, we will ignore the details of defining a custom operator to focus on the highlighted information below: when this operator runs (compute), it will print out Hello World! to the standard output:

C++

```

6 class HelloWorldOp : public Operator {
7 public:
8     HOLOSCAN_OPERATOR_FORWARD_ARGS(HelloWorldOp)
9
10    HelloWorldOp() = default;
11
12    void setup(OperatorSpec& spec) override {
13    }
14
15    void compute(InputContext& op_input, OutputContext& op_output,
16                ExecutionContext& context) override {
17        std::cout << std::endl;
18        std::cout << "Hello World!" << std::endl;
19        std::cout << std::endl;
20    }
21 };

```

Python

```

4 class HelloWorldOp(Operator):
5     """Simple hello world operator.
6
7     This operator has no ports.
8
9     On each tick, this operator prints out hello world.
10    """
11
12    def setup(self, spec: OperatorSpec):
13        pass
14
15    def compute(self, op_input, op_output, context):
16        print("")
17        print("Hello World!")
18        print("")

```

Defining the application workflow occurs within the application's `compose()` method. In there, we first create an instance of the `HelloWorldOp` operator defined above, then add it to our simple workflow using `add_operator()`.

C++

```

26 class HelloWorldApp : public holoscan::Application {
27 public:
28     void compose() override {
29         using namespace holoscan;
30
31         // Define the operators, allowing the hello operator to execute once
32         auto hello = make_operator<ops::HelloWorldOp>("hello", make_condition<CountCondition>
33             ↳(1));

```

(continues on next page)

(continued from previous page)

```

33 // Define the workflow by adding operator into the graph
34 add_operator(hello);
35 }
36 };
37

```

Python

```

21 class HelloWorldApp(Application):
22     def compose(self):
23         # Define the operators
24         hello = HelloWorldOp(self, CountCondition(self, 1), name="hello")
25
26         # Define the one-operator workflow
27         self.add_operator(hello)

```

Holoscan applications deal with streaming data, so an operator's `compute()` method will be called continuously until some situation arises that causes the operator to stop. For our Hello World example, we want to execute the operator only once. We can impose such a condition by passing a `CountCondition` object as an argument to the operator's constructor.

For more details, see the [Configuring operator conditions](#) section.

8.1.3 Running the Application

Running the application should give you the following output in your terminal:

```
Hello World!
```

Congratulations! You have successfully run your first Holoscan SDK application!

8.2 Ping Simple

Most applications will require more than one operator. In this example, we will create two operators where one operator will produce and send data while the other operator will receive and print the data. The code in this example makes use of the built-in **PingTxOp** and **PingRxOp** operators that are defined in the `holoscan::ops` namespace.

In this example we'll cover:

- How to use built-in operators.
- How to use `add_flow()` to connect operators together.

Note: The example source code and run instructions can be found in the [examples](#) directory on GitHub, or under `/opt/nvidia/holoscan/examples` in the NGC container and the Debian package, alongside their executables.

8.2.1 Operators and Workflow

Here is an example workflow involving two operators that are connected linearly.

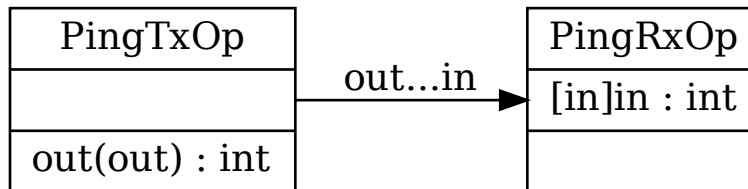


Fig. 8.1: A linear workflow

In this example, the source operator **PingTxOp** produces integers from 1 to 10 and passes it to the sink operator **PingRxOp** which prints the integers to standard output.

8.2.2 Connecting Operators

We can connect two operators by calling `add_flow()` (C++/Python) in the application's `compose()` method.

The `add_flow()` method (C++/Python) takes the source operator, the destination operator, and the optional port name pairs. The port name pair is used to connect the output port of the source operator to the input port of the destination operator. The first element of the pair is the output port name of the upstream operator and the second element is the input port name of the downstream operator. An empty port name ("") can be used for specifying a port name if the operator has only one input/output port. If there is only one output port in the upstream operator and only one input port in the downstream operator, the port pairs can be omitted.

The following code shows how to define a linear workflow in the `compose()` method for our example. Note that when an operator appears in an `add_flow()` statement, it doesn't need to be added into the workflow separately using `add_operator()`.

C++

```

1  #include <holoscan/holoscan.hpp>
2  #include <holoscan/operators/ping_tx/ping_tx.hpp>
3  #include <holoscan/operators/ping_rx/ping_rx.hpp>
4
5  class MyPingApp : public holoscan::Application {
6  public:
7      void compose() override {
8          using namespace holoscan;
9          // Create the tx and rx operators
10         auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
11         auto rx = make_operator<ops::PingRxOp>("rx");
12
13         // Connect the operators into the workflow: tx -> rx
  
```

(continues on next page)

(continued from previous page)

```

14     add_flow(tx, rx);
15 }
16 };
17
18 int main(int argc, char** argv) {
19     auto app = holoscan::make_application<MyPingApp>();
20     app->run();
21
22     return 0;
23 }

```

- The header files that define **PingTxOp** and **PingRxOp** are included on lines 2 and 3 respectively.
- We create an instance of the **PingTxOp** using the `make_operator()` function (line 10) with the name “tx” and constrain its `compute()` method to execute 10 times.
- We create an instance of the **PingRxOp** using the `make_operator()` function (line 11) with the name “rx.”
- The tx and rx operators are connected using `add_flow()` (line 14).

Python

```

1 from holoscan.conditions import CountCondition
2 from holoscan.core import Application
3 from holoscan.operators import PingRxOp, PingTxOp
4
5 class MyPingApp(Application):
6     def compose(self):
7         # Create the tx and rx operators
8         tx = PingTxOp(self, CountCondition(self, 10), name="tx")
9         rx = PingRxOp(self, name="rx")
10
11         # Connect the operators into the workflow: tx -> rx
12         self.add_flow(tx, rx)
13
14
15 def main():
16     app = MyPingApp()
17     app.run()
18
19
20 if __name__ == "__main__":
21     main()

```

- The built-in holoscan operators, **PingRxOp** and **PingTxOp**, are imported on line 3.
- We create an instance of the **PingTxOp** operator with the name “tx” and constrain its `compute()` method to execute 10 times (line 8).
- We create an instance of the **PingRxOp** operator with the name “rx” (line 9).
- The tx and rx operators are connected using `add_flow()` which defines this application’s workflow (line 12).

8.2.3 Running the Application

Running the application should give you the following output in your terminal:

```
Rx message value: 1
Rx message value: 2
Rx message value: 3
Rx message value: 4
Rx message value: 5
Rx message value: 6
Rx message value: 7
Rx message value: 8
Rx message value: 9
Rx message value: 10
```

8.3 Ping Custom Op

In this section, we will modify the previous `ping_simple` example to add a custom operator into the workflow. We've already seen a custom operator defined in the `hello_world` example but skipped over some of the details.

In this example we will cover:

- The details of creating your own custom operator class.
- How to add input and output ports to your operator.
- How to add parameters to your operator.
- The data type of the messages being passed between operators.

Note: The example source code and run instructions can be found in the `examples` directory on GitHub, or under `/opt/nvidia/holoscan/examples` in the NGC container and the Debian package, alongside their executables.

8.3.1 Operators and Workflow

Here is the diagram of the operators and workflow used in this example.

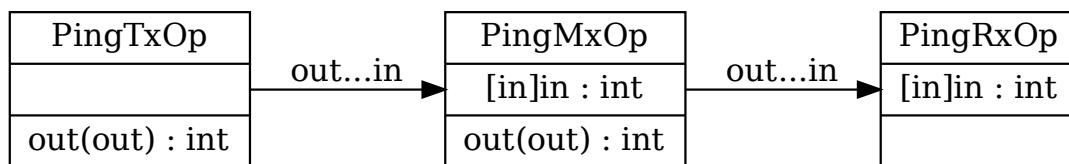


Fig. 8.2: A linear workflow with new custom operator

Compared to the previous example, we are adding a new **PingMxOp** operator between the **PingTxOp** and **PingRxOp** operators. This new operator takes as input an integer, multiplies it by a constant factor, and then sends the new value

to **PingRxOp**. You can think of this custom operator as doing some data processing on an input stream before sending the result to downstream operators.

8.3.2 Configuring Operator Input and Output Ports

Our custom operator needs 1 input and 1 output port and can be added by calling `spec.input()` and `spec.output()` methods within the operator's `setup()` method. This requires providing the data type and name of the port as arguments (for C++ API), or just the port name (for Python API). We will see an example of this in the code snippet below. For more details, see [Specifying operator inputs and outputs \(C++\)](#) or [Specifying operator inputs and outputs \(Python\)](#).

8.3.3 Configuring Operator Parameters

Operators can be made more reusable by customizing their parameters during initialization. The custom parameters can be provided either directly as arguments or accessed from the application's YAML configuration file. We will show how to use the former in this example to customize the “multiplier” factor of our **PingMxOp** custom operator. Configuring operators using a YAML configuration file will be shown in a subsequent [example](#). For more details, see [Configuring operator parameters](#).

The code snippet below shows how to define the **PingMxOp** class.

C++

```

1  #include <holoscan/holoscan.hpp>
2  #include <holoscan/operators/ping_tx/ping_tx.hpp>
3  #include <holoscan/operators/ping_rx/ping_rx.hpp>
4
5  namespace holoscan::ops {
6
7  class PingMxOp : public Operator {
8  public:
9      HOLOSCAN_OPERATOR_FORWARD_ARGS(PingMxOp)
10
11      PingMxOp() = default;
12
13      void setup(OperatorSpec& spec) override {
14          spec.input<int>("in");
15          spec.output<int>("out");
16          spec.param(multiplier_, "multiplier", "Multiplier", "Multiply the input by this value
17      ↪", 2);
18      }
19
20      void compute(InputContext& op_input, OutputContext& op_output, ExecutionContext&)
21      ↪override {
22          auto value = op_input.receive<int>("in");
23
24          std::cout << "Middle message value: " << value << std::endl;
25
26          // Multiply the value by the multiplier parameter
27          value *= multiplier_;
28
29          op_output.emit(value);
30      }
31  };
32  }

```

(continues on next page)

(continued from previous page)

```

28     };
29
30     private:
31         Parameter<int> multiplier_;
32     };
33
34 } // namespace holoscan::ops

```

- The PingMxOp class inherits from the Operator base class (line 7).
- The HOLOSCAN_OPERATOR_FORWARD_ARGS macro (line 9) is syntactic sugar to help forward an operator’s constructor arguments to the Operator base class, and is a convenient shorthand to avoid having to manually define constructors for your operator with the necessary parameters.
- Input/output ports with the names “in”/“out” are added to the operator spec on lines 14 and 15 respectively. The port type of both ports are int as indicated by the template argument <int>.
- We add a “multiplier” parameter to the operator spec (line 16) with a default value of 2. This parameter is tied to the private “multiplier_” data member.
- In the compute() method, we receive the integer data from the operator’s “in” port (line 20), print its value, multiply its value by the multiplicative factor, and send the new value downstream (line 27).
- On line 20, note that the data being passed between the operators has the type int.
- The call to op_output.emit(value) on line 27 is equivalent to op_output.emit(value, "out") since this operator has only 1 output port. If the operator has more than 1 output port, then the port name is required.

Python

```

1  from holoscan.conditions import CountCondition
2  from holoscan.core import Application, Operator, OperatorSpec
3  from holoscan.operators import PingRxOp, PingTxOp
4
5  class PingMxOp(Operator):
6      """Example of an operator modifying data.
7
8      This operator has 1 input and 1 output port:
9          input: "in"
10         output: "out"
11
12         The data from the input is multiplied by the "multiplier" parameter
13
14         """
15
16     def setup(self, spec: OperatorSpec):
17         spec.input("in")
18         spec.output("out")
19         spec.param("multiplier", 2)
20
21     def compute(self, op_input, op_output, context):
22         value = op_input.receive("in")
23         print(f"Middle message value: {value}")
24

```

(continues on next page)

(continued from previous page)

```

25     # Multiply the values by the multiplier parameter
26     value *= self.multiplier
27
28     op_output.emit(value, "out")

```

- The PingMxOp class inherits from the Operator base class (line 5).
- Input/output ports with the names “in”/“out” are added to the operator spec on lines 17 and 18, respectively.
- We add a “multiplier” parameter to the operator spec with a default value of 2 (line 19).
- In the compute() method, we receive the integer data from the operator’s “in” port (line 22), print its value, multiply its value by the multiplicative factor, and send the new value downstream (line 28).

Now that the custom operator has been defined, we create the application, operators, and define the workflow.

C++

```

1  class MyPingApp : public holoscan::Application {
2      public:
3          void compose() override {
4              using namespace holoscan;
5              // Define the tx, mx, rx operators, allowing tx operator to execute 10 times
6              auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
7              auto mx = make_operator<ops::PingMxOp>("mx", Arg("multiplier", 3));
8              auto rx = make_operator<ops::PingRxOp>("rx");
9
10             // Define the workflow: tx -> mx -> rx
11             add_flow(tx, mx);
12             add_flow(mx, rx);
13         }
14     };
15
16     int main(int argc, char** argv) {
17         auto app = holoscan::make_application<MyPingApp>();
18         app->run();
19
20         return 0;
21     }

```

- The tx, mx, and rx operators are created in the compose() method on lines 6–8.
- The custom mx operator is created in exactly the same way with make_operator() (line 7) as the built-in operators, and configured with a “multiplier” parameter initialized to 3 which overrides the parameter’s default value of 2 (in the setup() method).
- The workflow is defined by connecting tx to mx, and mx to rx using add_flow() on lines 11–12.

Python

```

1 class MyPingApp(Application):
2     def compose(self):
3         # Define the tx, mx, rx operators, allowing the tx operator to execute 10 times
4         tx = PingTxOp(self, CountCondition(self, 10), name="tx")
5         mx = PingMxOp(self, name="mx", multiplier=3)
6         rx = PingRxOp(self, name="rx")
7
8         # Define the workflow: tx -> mx -> rx
9         self.add_flow(tx, mx)
10        self.add_flow(mx, rx)
11
12
13 def main():
14     app = MyPingApp()
15     app.run()
16
17
18 if __name__ == "__main__":
19     main()

```

- The tx, mx, and rx operators are created in the `compose()` method on lines 4-6.
- The custom mx operator is created in exactly the same way as the built-in operators (line 5), and configured with a “multiplier” parameter initialized to 3 which overrides the parameter’s default value of 2 (in the `setup()` method).
- The workflow is defined by connecting tx to mx, and mx to rx using `add_flow()` on lines 9-10.

8.3.4 Message Data Types

For the C++ API, the messages that are passed between the operators are the objects of the data type at the inputs and outputs, so the value variable from lines 20 and 25 of the example above has the type `int`. For the Python API, the messages passed between operators can be arbitrary Python objects so no special consideration is needed since it is not restricted to the stricter parameter typing used for C++ API operators.

Let’s look at the code snippet for the built-in **PingTxOp** class and see if this helps to make it clearer.

C++

```

1 #include "holoscan/operators/ping_tx/ping_tx.hpp"
2
3 namespace holoscan::ops {
4
5 void PingTxOp::setup(OperatorSpec& spec) {
6     spec.output<int>("out");
7 }
8
9 void PingTxOp::compute(InputContext&, OutputContext& op_output, ExecutionContext&) {
10     auto value = index_++;
11     op_output.emit(value, "out");

```

(continues on next page)

(continued from previous page)

```

12 }
13
14 } // namespace holoscan::ops

```

- The “out” port of the **PingTxOp** has the type `int` (line 6).
- An integer is published to the “out” port when calling `emit()` (line 11).
- The message received by the downstream **PingMxOp** operator when it calls `op_input.receive<int>()` has the type `int`.

Python

```

1 class PingTxOp(Operator):
2     """Simple transmitter operator.
3
4     This operator has a single output port:
5         output: "out"
6
7     On each tick, it transmits an integer to the "out" port.
8     """
9
10    def setup(self, spec: OperatorSpec):
11        spec.output("out")
12
13    def compute(self, op_input, op_output, context):
14        op_output.emit(self.index, "out")
15        self.index += 1

```

- No special consideration is necessary for the Python version, we simply call `emit()` and pass the integer object (line 14).

Attention: For advanced use cases, e.g., when writing C++ applications where you need interoperability between C++ native and GXF operators, you will need to use the `holoscan::TensorMap` type instead. See [Interoperability between GXF and native C++ operators](#) for more details. If you are writing a Python application which needs a mixture of Python wrapped C++ operators and native Python operators, see [Interoperability between wrapped and native Python operators](#).

8.3.5 Running the Application

Running the application should give you the following output in your terminal:

```

Middle message value: 1
Rx message value: 3
Middle message value: 2
Rx message value: 6
Middle message value: 3
Rx message value: 9
Middle message value: 4
Rx message value: 12

```

(continues on next page)

(continued from previous page)

```

Middle message value: 5
Rx message value: 15
Middle message value: 6
Rx message value: 18
Middle message value: 7
Rx message value: 21
Middle message value: 8
Rx message value: 24
Middle message value: 9
Rx message value: 27
Middle message value: 10
Rx message value: 30

```

8.4 Ping Multi Port

In this section, we look at how to create an application with a more complex workflow where operators may have multiple input/output ports that send/receive a user-defined data type.

In this example we will cover:

- How to send/receive messages with a custom data type.
- How to add a port that can receive any number of inputs.

Note: The example source code and run instructions can be found in the [examples](#) directory on GitHub, or under `/opt/nvidia/holoscan/examples` in the NGC container and the Debian package, alongside their executables.

8.4.1 Operators and Workflow

Here is the diagram of the operators and workflow used in this example.

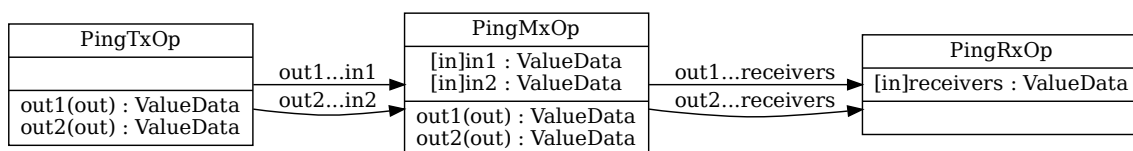


Fig. 8.3: A workflow with multiple inputs and outputs

In this example, PingTxOp sends a stream of odd integers to the `out1` port, and even integers to the `out2` port. PingMxOp receives these values using `in1` and `in2` ports, multiplies them by a constant factor, then forwards them to a single port - `receivers` - on PingRxOp.

8.4.2 User Defined Data Types

In the previous ping examples, the port types for our operators were integers, but the Holoscan SDK can send any arbitrary data type. In this example, we'll see how to configure operators for our user-defined `ValueData` class.

C++

```
1  #include "holoscan/holoscan.hpp"
2
3  class ValueData {
4  public:
5      ValueData() = default;
6      explicit ValueData(int value) : data_(value) {
7          HOLOSCAN_LOG_TRACE("ValueData::ValueData(): {}", data_);
8      }
9      ~ValueData() { HOLOSCAN_LOG_TRACE("ValueData::~~ValueData(): {}", data_); }
10
11     void data(int value) { data_ = value; }
12
13     int data() const { return data_; }
14
15 private:
16     int data_;
17 };
```

The `ValueData` class wraps a simple integer (line 6, 16), but could have been arbitrarily complex.

Note: The `HOLOSCAN_LOG_<LEVEL>()` macros can be used for logging with `fmtlib` syntax (lines 7, 9 above) as demonstrated across this example. See the [Logging](#) section for more details.

Python

```
1  from holoscan.conditions import CountCondition
2  from holoscan.core import Application, IOSpec, Operator, OperatorSpec
3
4  class ValueData:
5      """Example of a custom Python class"""
6
7      def __init__(self, value):
8          self.data = value
9
10     def __repr__(self):
11         return f"ValueData({self.data})"
12
13     def __eq__(self, other):
14         return self.data == other.data
15
16     def __hash__(self):
17         return hash(self.data)
```

The `ValueData` class is a simple wrapper, but could have been arbitrarily complex.

8.4.3 Defining an Explicit Number of Inputs and Outputs

After defining our custom `ValueData` class, we configure our operators' ports to send/receive messages of this type, similarly to the *previous example*.

This is the first operator - `PingTxOp` - sending `ValueData` objects on two ports, `out1` and `out2`:

C++

```

1 namespace holoscan::ops {
2
3 class PingTxOp : public Operator {
4 public:
5     HOLOSCAN_OPERATOR_FORWARD_ARGS(PingTxOp)
6
7     PingTxOp() = default;
8
9     void setup(OperatorSpec& spec) override {
10         spec.output<std::shared_ptr<ValueData>>("out1");
11         spec.output<std::shared_ptr<ValueData>>("out2");
12     }
13
14     void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override {
15         auto value1 = std::make_shared<ValueData>(index_++);
16         op_output.emit(value1, "out1");
17
18         auto value2 = std::make_shared<ValueData>(index_++);
19         op_output.emit(value2, "out2");
20     };
21     int index_ = 1;
22 };

```

- We configure the output ports with the `ValueData` type on lines 10 and 11 using `spec.output<std::shared_ptr<ValueData>>()`. Therefore, the data type for the output ports is an object to a shared pointer to a `ValueData` object.
- The values are then sent out using `op_output.emit()` on lines 16 and 19. The port name is required since there is more than one port on this operator.

Note: Data types of the output ports are shared pointers (`std::shared_ptr`), hence the call to `std::make_shared<ValueData>(...)` on lines 15 and 18.

Python

```

1 class PingTxOp(Operator):
2     """Simple transmitter operator.
3
4     This operator has:
5         outputs: "out1", "out2"
6
7     On each tick, it transmits a `ValueData` object at each port. The

```

(continues on next page)

(continued from previous page)

```

8      transmitted values are even on port1 and odd on port2 and increment with
9      each call to compute.
10     """
11
12     def __init__(self, fragment, *args, **kwargs):
13         self.index = 1
14         # Need to call the base class constructor last
15         super().__init__(fragment, *args, **kwargs)
16
17     def setup(self, spec: OperatorSpec):
18         spec.output("out1")
19         spec.output("out2")
20
21     def compute(self, op_input, op_output, context):
22         value1 = ValueData(self.index)
23         self.index += 1
24         op_output.emit(value1, "out1")
25
26         value2 = ValueData(self.index)
27         self.index += 1
28         op_output.emit(value2, "out2")

```

- We configure the output ports on lines 18 and 19 using `spec.output()`. There is no need to reference the type (`ValueData`) in Python.
- The values are then sent out using `op_output.emit()` on lines 24 and 28.

We then configure the middle operator - `PingMxOp` - to receive that data on ports `in1` and `in2`:

C++

```

1  class PingMxOp : public Operator {
2      public:
3          HOLOSCAN_OPERATOR_FORWARD_ARGS(PingMxOp)
4
5          PingMxOp() = default;
6
7          void setup(OperatorSpec& spec) override {
8              spec.input<std::shared_ptr<ValueData>>("in1");
9              spec.input<std::shared_ptr<ValueData>>("in2");
10             spec.output<std::shared_ptr<ValueData>>("out1");
11             spec.output<std::shared_ptr<ValueData>>("out2");
12             spec.param(multiplier_, "multiplier", "Multiplier", "Multiply the input by this value
13             ↪", 2);
14         }
15
16         void compute(InputContext& op_input, OutputContext& op_output, ExecutionContext&) ↪
17         ↪override {
18             auto value1 = op_input.receive<std::shared_ptr<ValueData>>("in1").value();
19             auto value2 = op_input.receive<std::shared_ptr<ValueData>>("in2").value();
20
21             HOLOSCAN_LOG_INFO("Middle message received (count: {})", count_++);

```

(continues on next page)

(continued from previous page)

```

20 HOLOSCAN_LOG_INFO("Middle message value1: {}", value1->data());
21 HOLOSCAN_LOG_INFO("Middle message value2: {}", value2->data());
22
23
24 // Multiply the values by the multiplier parameter
25 value1->data(value1->data() * multiplier_);
26 value2->data(value2->data() * multiplier_);
27
28 op_output.emit(value1, "out1");
29 op_output.emit(value2, "out2");
30 };
31
32 private:
33 int count_ = 1;
34 Parameter<int> multiplier_;
35 };

```

- We configure the input ports with the `std::shared_ptr<ValueData>` type on lines 8 and 9 using `spec.input<std::shared_ptr<ValueData>>()`.
- The values are received using `op_input.receive()` on lines 16 and 17 using the port names. The received values are of type `std::shared_ptr<ValueData>` as mentioned in the templated `receive()` method.

Python

```

1 class PingMxOp(Operator):
2     """Example of an operator modifying data.
3
4     This operator has:
5         inputs: "in1", "in2"
6         outputs: "out1", "out2"
7
8     The data from each input is multiplied by a user-defined value.
9     """
10
11     def __init__(self, fragment, *args, **kwargs):
12         self.count = 1
13
14         # Need to call the base class constructor last
15         super().__init__(fragment, *args, **kwargs)
16
17     def setup(self, spec: OperatorSpec):
18         spec.input("in1")
19         spec.input("in2")
20         spec.output("out1")
21         spec.output("out2")
22         spec.param("multiplier", 2)
23
24     def compute(self, op_input, op_output, context):
25         value1 = op_input.receive("in1")
26         value2 = op_input.receive("in2")

```

(continues on next page)

(continued from previous page)

```

27     print(f"Middle message received (count: {self.count})")
28     self.count += 1
29
30     print(f"Middle message value1: {value1.data}")
31     print(f"Middle message value2: {value2.data}")
32
33     # Multiply the values by the multiplier parameter
34     value1.data *= self.multiplier
35     value2.data *= self.multiplier
36
37     op_output.emit(value1, "out1")
38     op_output.emit(value2, "out2")

```

Sending messages of arbitrary data types is pretty straightforward in Python. The code to define the operator input ports (lines 18–19), and to receive them (lines 25–26) did not change when we went from passing `int` to `ValueData` objects.

`PingMxOp` processes the data, then sends it out on two ports, similarly to what is done by `PingTxOp` above.

8.4.4 Receiving Any Number of Inputs

In this workflow, `PingRxOp` has a single input port - `receivers` - that is connected to two upstream ports from `PingMxOp`. When an input port needs to connect to multiple upstream ports, we define it with `spec.param()` instead of `spec.input()`. The inputs are then stored in a vector, following the order they were added with `add_flow()`.

C++

```

1  class PingRxOp : public Operator {
2      public:
3          HOLOSCAN_OPERATOR_FORWARD_ARGS(PingRxOp)
4
5          PingRxOp() = default;
6
7          void setup(OperatorSpec& spec) override {
8              // // Since Holoscan SDK v2.3, users can define a multi-receiver input port using
9              // // 'spec.input()'
10             // // with 'IOSpec::kAnySize'.
11             // // The old way is to use 'spec.param()' with 'Parameter<std::vector<IOSpec*>>'
12             // // spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.",
13             // // {});
14             spec.input<std::vector<std::shared_ptr<ValueData>>>("receivers", IOSpec::kAnySize);
15         }
16
17         void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {
18             auto value_vector =
19                 op_input.receive<std::vector<std::shared_ptr<ValueData>>>("receivers").value();
20
21             HOLOSCAN_LOG_INFO("Rx message received (count: {}, size: {})", count_++, value_
22                 .vector.size());
23         }
24     };

```

(continues on next page)

(continued from previous page)

```

21 HOLOSCAN_LOG_INFO("Rx message value1: {}", value_vector[0]->data());
22 HOLOSCAN_LOG_INFO("Rx message value2: {}", value_vector[1]->data());
23 };
24
25 private:
26 // // Since Holoscan SDK v2.3, the following line is no longer needed.
27 // Parameter<std::vector<IOSpec*>> receivers_;
28 int count_ = 1;
29 };

```

- In the operator's `setup()` method, we define an input port `receivers` (line 12) with `holoscan::IOSpec::kAnySize` to allow any number of upstream ports to connect to it.
- The values are retrieved using `op_input.receive<std::vector<std::shared_ptr<ValueData>>>(...)`.
- The type of `value_vector` is `std::vector<std::shared_ptr<ValueData>>` (lines 16-17).

Please see *Receiving any number of inputs (C++)* for more information on how to retrieve any number of inputs in C++.

Python

```

1 class PingRxOp(Operator):
2     """Simple receiver operator.
3
4     This operator has:
5         input: "receivers"
6
7     This is an example of a native operator that can dynamically have any
8     number of inputs connected to its "receivers" port.
9     """
10
11     def __init__(self, fragment, *args, **kwargs):
12         self.count = 1
13         # Need to call the base class constructor last
14         super().__init__(fragment, *args, **kwargs)
15
16     def setup(self, spec: OperatorSpec):
17         # # Since Holoscan SDK v2.3, users can define a multi-receiver input port using
18         # # 'spec.input()' with 'size=IOSpec.ANY_SIZE'.
19         # # The old way is to use 'spec.param()' with 'kind="receivers"'.
20         # spec.param("receivers", kind="receivers")
21         spec.input("receivers", size=IOSpec.ANY_SIZE)
22
23     def compute(self, op_input, op_output, context):
24         values = op_input.receive("receivers")
25         print(f"Rx message received (count: {self.count}, size: {len(values)})")
26         self.count += 1
27         print(f"Rx message value1: {values[0].data}")
28         print(f"Rx message value2: {values[1].data}")

```

- In Python, a port that can be connected to multiple upstream ports is created by defining an input port and setting

the argument `size=IOSpec.ANY_SIZE` (line 21).

- The call to `receive()` returns a tuple of `ValueData` objects (line 24).

Please see *Receiving any number of inputs (Python)* for more information on how to retrieve any number of inputs in Python.

The rest of the code creates the application, operators, and defines the workflow:

C++

```

1 class MyPingApp : public holoscan::Application {
2     public:
3         void compose() override {
4             using namespace holoscan;
5
6             // Define the tx, mx, rx operators, allowing the tx operator to execute 10 times
7             auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
8             auto mx = make_operator<ops::PingMxOp>("mx", Arg("multiplier", 3));
9             auto rx = make_operator<ops::PingRxOp>("rx");
10
11             // Define the workflow
12             add_flow(tx, mx, {"out1", "in1"}, {"out2", "in2"});
13             add_flow(mx, rx, {"out1", "receivers"}, {"out2", "receivers"});
14         }
15     };
16
17     int main(int argc, char** argv) {
18         auto app = holoscan::make_application<MyPingApp>();
19         app->run();
20
21         return 0;
22     }

```

Python

```

105 class MyPingApp(Application):
106     def compose(self):
107         # Define the tx, mx, rx operators, allowing the tx operator to execute 10 times
108         tx = PingTxOp(self, CountCondition(self, 10), name="tx")
109         mx = PingMxOp(self, name="mx", multiplier=3)
110         rx = PingRxOp(self, name="rx")
111
112         # Define the workflow
113         self.add_flow(tx, mx, {"out1", "in1"}, {"out2", "in2"})
114         self.add_flow(mx, rx, {"out1", "receivers"}, {"out2", "receivers"})
115
116
117     def main():
118         app = MyPingApp()
119         app.run()
120

```

(continues on next page)

(continued from previous page)

```

121
122 if __name__ == "__main__":
123     main()

```

- The operators `tx`, `mx`, and `rx` are created in the application's `compose()` similarly to previous examples.
- Since the operators in this example have multiple input/output ports, we need to specify the third, port name pair argument when calling `add_flow()`:
 - `tx/out1` is connected to `mx/in1`, and `tx/out2` is connected to `mx/in2`.
 - `mx/out1` and `mx/out2` are both connected to `rx/receivers`.

8.4.5 Running the Application

Running the application should give you output similar to the following in your terminal.

```

[info] [fragment.cpp:586] Loading extensions from configs...
[info] [gxf_executor.cpp:249] Creating context
[info] [gxf_executor.cpp:1960] Activating Graph...
[info] [gxf_executor.cpp:1992] Running Graph...
[info] [greedy_scheduler.cpp:191] Scheduling 3 entities
[info] [gxf_executor.cpp:1994] Waiting for completion...
[info] [ping_multi_port.cpp:80] Middle message received (count: 1)
[info] [ping_multi_port.cpp:82] Middle message value1: 1
[info] [ping_multi_port.cpp:83] Middle message value2: 2
[info] [ping_multi_port.cpp:116] Rx message received (count: 1, size: 2)
[info] [ping_multi_port.cpp:118] Rx message value1: 3
[info] [ping_multi_port.cpp:119] Rx message value2: 6
[info] [ping_multi_port.cpp:80] Middle message received (count: 2)
[info] [ping_multi_port.cpp:82] Middle message value1: 3
[info] [ping_multi_port.cpp:83] Middle message value2: 4
[info] [ping_multi_port.cpp:116] Rx message received (count: 2, size: 2)
[info] [ping_multi_port.cpp:118] Rx message value1: 9
[info] [ping_multi_port.cpp:119] Rx message value2: 12
...
[info] [ping_multi_port.cpp:80] Middle message received (count: 10)
[info] [ping_multi_port.cpp:82] Middle message value1: 19
[info] [ping_multi_port.cpp:83] Middle message value2: 20
[info] [ping_multi_port.cpp:116] Rx message received (count: 10, size: 2)
[info] [ping_multi_port.cpp:118] Rx message value1: 57
[info] [ping_multi_port.cpp:119] Rx message value2: 60
[info] [greedy_scheduler.cpp:372] Scheduler stopped: Some entities are waiting for
↳ execution, but there are no periodic or async entities to get out of the deadlock.
[info] [greedy_scheduler.cpp:401] Scheduler finished.
[info] [gxf_executor.cpp:1997] Deactivating Graph...
[info] [gxf_executor.cpp:2005] Graph execution finished.
[info] [gxf_executor.cpp:278] Destroying context

```

Note: Depending on your log level you may see more or fewer messages. The output above was generated using the default value of `INFO`. Refer to the [Logging](#) section for more details on how to set the log level.

8.5 Video Replayer

So far we have been working with simple operators to demonstrate Holoscan SDK concepts. In this example, we look at two built-in Holoscan operators that have many practical applications.

In this example we'll cover:

- How to load a video file from disk using **VideoStreamReplayerOp** operator.
- How to display video using **HolovizOp** operator.
- How to configure your operator's parameters using a YAML configuration file.

Note: The example source code and run instructions can be found in the [examples](#) directory on GitHub, or under `/opt/nvidia/holoscan/examples` in the NGC container and the Debian package, alongside their executables.

8.5.1 Operators and Workflow

Here is the diagram of the operators and workflow used in this example.

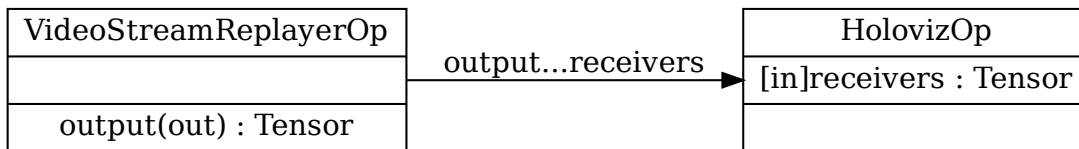


Fig. 8.4: Workflow to load and display video from a file

We connect the “output” port of the replayer operator to the “receivers” port of the Holoviz operator.

8.5.2 Video Stream Replayer Operator

The built-in video stream replayer operator can be used to replay a video stream that has been encoded as gxf entities. You can use the `convert_video_to_gxf_entities.py` script (installed in `/opt/nvidia/holoscan/bin` or available [on GitHub](#)) to encode a video file as gxf entities for use by this operator.

This operator processes the encoded file sequentially and supports realtime, faster than realtime, or slower than realtime playback of prerecorded data. The input data can optionally be repeated to loop forever or only for a specified count. For more details, see `VideoStreamReplayerOp`.

We will use the replayer to read GXF entities from disk and send the frames downstream to the Holoviz operator.

8.5.3 Holoviz Operator

The built-in Holoviz operator provides the functionality to composite realtime streams of frames with multiple different other layers like segmentation mask layers, geometry layers and GUI layers.

We will use Holoviz to display frames that have been sent by the replayer operator to its “receivers” port which can receive any number of inputs. In more intricate workflows, this port can receive multiple streams of input data where, for example, one stream is the original video data, while other streams detect objects in the video to create bounding boxes and/or text overlays.

8.5.4 Application Configuration File (YAML)

The SDK supports reading an optional YAML configuration file and can be used to customize the application’s workflow and operators. For more complex workflows, it may be helpful to use the application configuration file to help separate operator parameter settings from your code. See [Configuring an Application](#) for additional details.

Tip: For C++ applications, the configuration file offers a convenient way to set the behavior of the application at runtime without needing to recompile the code.

This example uses the following configuration file to configure the parameters for the replayer and Holoviz operators. The full list of parameters can be found at `VideoStreamReplayerOp` and `HolovizOp`.

```
%YAML 1.2
replayer:
  directory: "../data/racex"    # Path to gxf entity video data
  basename: "racex"            # Look for <basename>.gxf_{entities|index}
  frame_rate: 0                # Frame rate to replay. (default: 0 follow frame rate in
↪ timestamps)
  repeat: true                 # Loop video? (default: false)
  realtime: true               # Play in realtime, based on frame_rate/timestamps (default:
↪ true)
  count: 0                    # Number of frames to read (default: 0 for no frame count
↪ restriction)

holoviz:
  width: 854                  # width of window size
  height: 480                 # height of window size
  tensors:
    - name: ""                # name of tensor containing input data to display
      type: color              # input type e.g., color, triangles, text, depth_map
      opacity: 1.0            # layer opacity
      priority: 0              # determines render order, higher priority layers are rendered on
↪ top
```

The code below shows our `video_replayer` example. Operator parameters are configured from a configuration file using `from_config()` (C++) and `self.**kwargs()` (Python).

C++

```

1 #include <holoscan/holoscan.hpp>
2 #include <holoscan/operators/video_stream_replayer/video_stream_replayer.hpp>
3 #include <holoscan/operators/holoviz/holoviz.hpp>
4
5 class VideoReplayerApp : public holoscan::Application {
6 public:
7     void compose() override {
8         using namespace holoscan;
9
10        // Define the replayer and holoviz operators and configure using yaml configuration
11        auto replayer = make_operator<ops::VideoStreamReplayerOp>("replayer", from_config(
12        ↪ "replayer"));
13        auto visualizer = make_operator<ops::HolovizOp>("holoviz", from_config("holoviz"));
14
15        // Define the workflow: replayer -> holoviz
16        add_flow(replayer, visualizer, {"output", "receivers"});
17    }
18 };
19
20 int main(int argc, char** argv) {
21     // Get the yaml configuration file
22     auto config_path = std::filesystem::canonical(argv[0]).parent_path();
23     config_path /= std::filesystem::path("video_replayer.yaml");
24     if (argc >= 2) {
25         config_path = argv[1];
26     }
27
28     auto app = holoscan::make_application<VideoReplayerApp>();
29     app->config(config_path);
30     app->run();
31
32     return 0;
33 }

```

- The built-in **VideoStreamReplayerOp** and **HolovizOp** operators are included from lines 1 and 2, respectively.
- We create an instance of **VideoStreamReplayerOp** named “replayer” with parameters initialized from the YAML configuration file using the call to `from_config()` (line 11).
- We create an instance of **HolovizOp** named “holoviz” with parameters initialized from the YAML configuration file using the call to `from_config()` (line 12).
- The “output” port of “replayer” operator is connected to the “receivers” port of the “holoviz” operator and defines the application workflow (line 34).
- The application’s YAML configuration file contains the parameters for our operators, and is loaded on line 28. If no argument is passed to the executable, the application looks for a file with the name “video_replayer.yaml” in the same directory as the executable (lines 21-22), otherwise it treats the argument as the path to the application’s YAML configuration file (lines 23-25).

Python

```

1 import os
2 import sys
3
4 from holoscan.core import Application
5 from holoscan.operators import HolovizOp, VideoStreamReplayerOp
6
7 sample_data_path = os.environ.get("HOLOSCAN_INPUT_PATH", "../data")
8
9
10 class VideoReplayerApp(Application):
11     """Example of an application that uses the operators defined above.
12
13     This application has the following operators:
14
15     - VideoStreamReplayerOp
16     - HolovizOp
17
18     The VideoStreamReplayerOp reads a video file and sends the frames to the HolovizOp.
19     The HolovizOp displays the frames.
20     """
21
22     def compose(self):
23         video_dir = os.path.join(sample_data_path, "racerx")
24         if not os.path.exists(video_dir):
25             raise ValueError(f"Could not find video data: {video_dir}")
26
27         # Define the replayer and holoviz operators
28         replayer = VideoStreamReplayerOp(
29             self, name="replayer", directory=video_dir, **self.kwargs("replayer")
30         )
31         visualizer = HolovizOp(self, name="holoviz", **self.kwargs("holoviz"))
32
33         # Define the workflow
34         self.add_flow(replayer, visualizer, {("output", "receivers")})
35
36
37 def main(config_file):
38     app = VideoReplayerApp()
39     # if the --config command line argument was provided, it will override this config_
40     ↪ file
41     app.config(config_file)
42     app.run()
43
44 if __name__ == "__main__":
45     config_file = os.path.join(os.path.dirname(__file__), "video_replayer.yaml")
46     main(config_file=config_file)

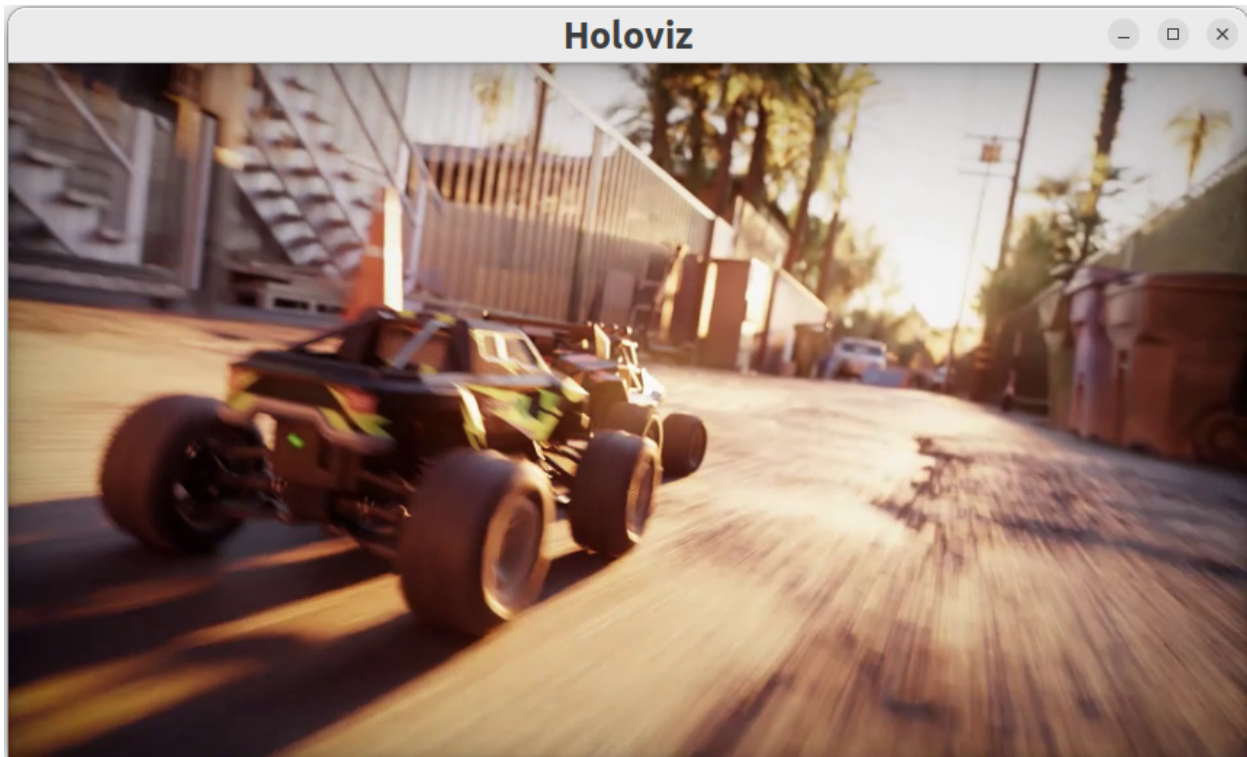
```

- The built-in **VideoStreamReplayerOp** and **HolovizOp** operators are imported on line 5.
- We create an instance of **VideoStreamReplayerOp** named “replayer” with parameters initialized from the YAML configuration file using `**self.kwargs()` (lines 28-30).

- For the Python script, the path to the GXF entity video data is not set in the application configuration file, but determined by the code on lines 7 and 23 and is passed directly as the “directory” argument (line 29). This allows more flexibility for the user to run the script from any directory by setting the HOLOSCAN_INPUT_PATH directory (line 7).
- We create an instance of **HolovizOp** named “holoviz” with parameters initialized from the YAML configuration file using `**self.kwargs()` (line 31).
- The “output” port of “replayer” operator is connected to the “receivers” port of the “holoviz” operator and defines the application workflow (line 34).
- The application’s YAML configuration file contains the parameters for our operators, and is loaded on line 45. If no argument is passed to the Python script, the application looks for a file with the name “video_replayer.yaml” in the same directory as the script (line 39). Otherwise it treats the argument as the path to the app’s YAML configuration file (lines 41-42).

8.5.5 Running the Application

Running the application should bring up video playback of the video referenced in the YAML file.



8.6 Video Replayer (Distributed)

In this example, we extend the previous *video replayer application* into a multi-node *distributed application*. A distributed application is made up of multiple Fragments (C++/Python), each of which may run on its own node.

In the distributed case we will:

- Create one fragment that loads a video file from disk using **VideoStreamReplayerOp** operator.
- Create a second fragment that will display the video using the **HolovizOp** operator.

These two fragments will be combined into a distributed application such that the display of the video frames could occur on a separate node from the node where the data is read.

Note: The example source code and run instructions can be found in the [examples](#) directory on GitHub, or under `/opt/nvidia/holoscan/examples` in the NGC container and the Debian package, alongside their executables.

8.6.1 Operators and Workflow

Here is the diagram of the operators and workflow used in this example.

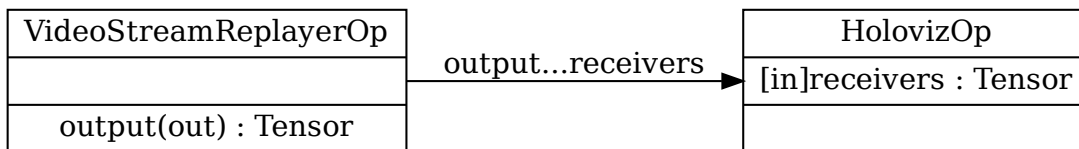


Fig. 8.5: Workflow to load and display video from a file

This is the same workflow as the *single fragment video replayer*. Each operator is assigned to a separate fragment and there is now a network connection between the fragments.

8.6.2 Defining and Connecting Fragments

Distributed applications define fragments explicitly to isolate the different units of work that could be distributed to different nodes. In this example:

- We define two classes that inherit from **Fragment**:
 - **Fragment1** contains an instance of **VideoStreamReplayerOp** named “replayer.”
 - **Fragment2** contains an instance of **HolovizOp** name “holoviz.”
- We create an application, **DistributedVideoReplayerApp**. In its `compose` method:
 - we call `make_fragment` to initialize both fragments.
 - we then connect the “output” port of “replayer” operator in fragment1 to the “receivers” port of the “holoviz” operator in fragment2 to define the application workflow.

- The operators instantiated in the fragments can still be configured with parameters initialized from the YAML configuration ingested by the application using `from_config()` (C++) or `kwargs()` (Python).

C++

```

1  #include <holoscan/holoscan.hpp>
2  #include <holoscan/operators/holoviz/holoviz.hpp>
3  #include <holoscan/operators/video_stream_replayer/video_stream_replayer.hpp>
4
5  class Fragment1 : public holoscan::Fragment {
6  public:
7      void compose() override {
8          using namespace holoscan;
9
10         auto replayer = make_operator<ops::VideoStreamReplayerOp>("replayer", from_config(
11             ↪ "replayer"));
12         add_operator(replayer);
13     }
14 };
15
16 class Fragment2 : public holoscan::Fragment {
17 public:
18     void compose() override {
19         using namespace holoscan;
20
21         auto visualizer = make_operator<ops::HolovizOp>("holoviz", from_config("holoviz"));
22         add_operator(visualizer);
23     }
24 };
25
26 class DistributedVideoReplayerApp : public holoscan::Application {
27 public:
28     void compose() override {
29         using namespace holoscan;
30
31         auto fragment1 = make_fragment<Fragment1>("fragment1");
32         auto fragment2 = make_fragment<Fragment2>("fragment2");
33
34         // Define the workflow: replayer -> holoviz
35         add_flow(fragment1, fragment2, {"replayer.output", "holoviz.receivers"});
36     }
37 };
38
39 int main(int argc, char** argv) {
40     // Get the yaml configuration file
41     auto config_path = std::filesystem::canonical(argv[0]).parent_path();
42     config_path /= std::filesystem::path("video_replayer_distributed.yaml");
43
44     auto app = holoscan::make_application<DistributedVideoReplayerApp>();
45     app->config(config_path);
46     app->run();

```

(continues on next page)

(continued from previous page)

```

47     return 0;
48 }

```

Python

```

1  import os
2
3  from holoscan.core import Application, Fragment
4  from holoscan.operators import HolovizOp, VideoStreamReplayerOp
5
6  sample_data_path = os.environ.get("HOLOSCAN_INPUT_PATH", "../data")
7
8
9  class Fragment1(Fragment):
10     def __init__(self, app, name):
11         super().__init__(app, name)
12
13     def compose(self):
14         # Set the video source
15         video_path = self._get_input_path()
16         logging.info(
17             f"Using video from {video_path}"
18         )
19
20         # Define the replayer and holoviz operators
21         replayer = VideoStreamReplayerOp(
22             self, name="replayer", directory=video_path, **self.kwargs("replayer")
23         )
24
25         self.add_operator(replayer)
26
27     def _get_input_path(self):
28         path = os.environ.get(
29             "HOLOSCAN_INPUT_PATH", os.path.join(os.path.dirname(__file__), "data")
30         )
31         return os.path.join(path, "racerx")
32
33
34  class Fragment2(Fragment):
35     def compose(self):
36         visualizer = HolovizOp(self, name="holoviz", **self.kwargs("holoviz"))
37
38         self.add_operator(visualizer)
39
40
41  class DistributedVideoReplayerApp(Application):
42     """Example of a distributed application that uses the fragments and operators_
43     ↪ defined above.
44
45     This application has the following fragments:

```

(continues on next page)

(continued from previous page)

```

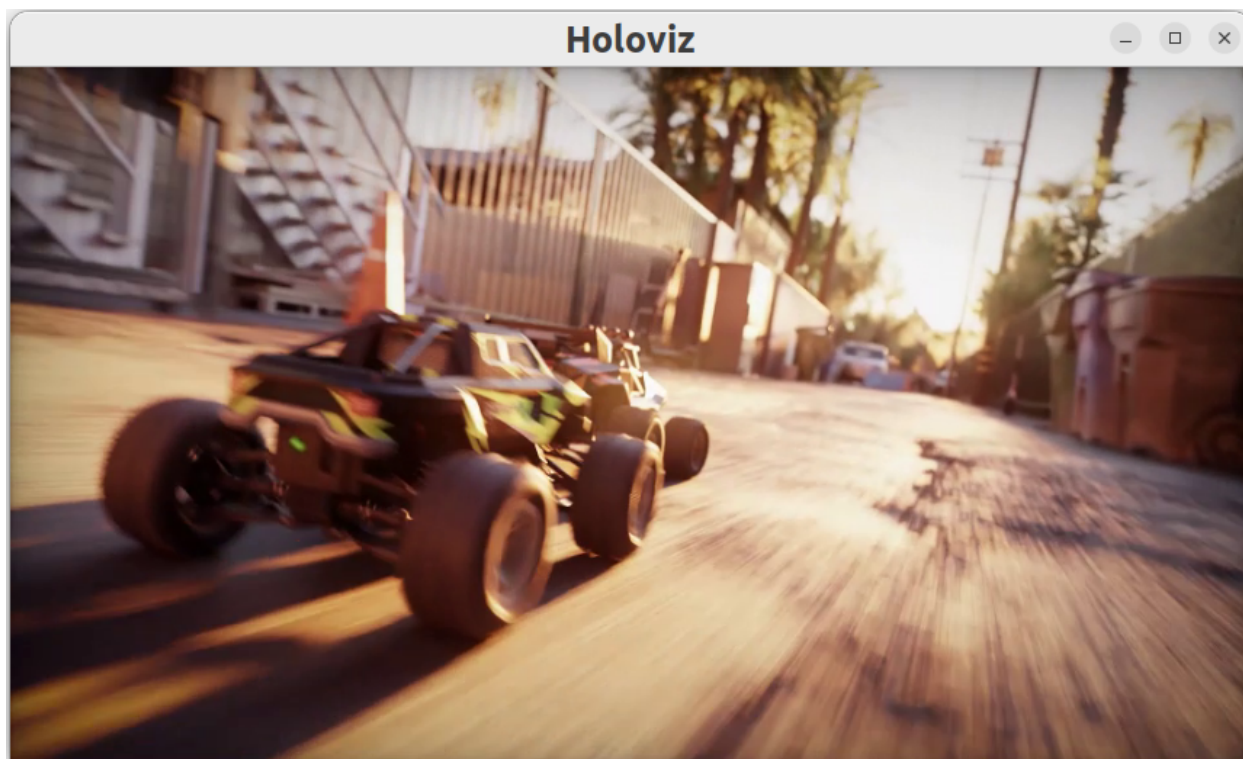
45 - Fragment1
46     - holding VideoStreamReplayerOp
47 - Fragment2
48     - holding HolovizOp
49
50 The VideoStreamReplayerOp reads a video file and sends the frames to the HolovizOp.
51 The HolovizOp displays the frames.
52 """
53
54 def compose(self):
55     # Define the fragments
56     fragment1 = Fragment1(self, name="fragment1")
57     fragment2 = Fragment2(self, name="fragment2")
58
59     # Define the workflow
60     self.add_flow(fragment1, fragment2, {"replayer.output", "holoviz.receivers"})
61
62
63 def main():
64     config_file_path = os.path.join(os.path.dirname(__file__), "video_replayer_
65     ↪distributed.yaml")
66
67     logging.info(f"Reading application configuration from {config_file_path}")
68
69     app = DistributedVideoReplayerApp()
70     app.config(config_file_path)
71     app.run()
72
73 if __name__ == "__main__":
74     main()

```

This particular distributed application only has one operator per fragment, so the operators were added via **add_operator (C++/Python)**. In general, each fragment may have multiple operators and connections between operators within a fragment would be made using `add_flow()` (C++/Python) method within the fragment's `compute()` (C++/Python) method.

8.6.3 Running the Application

Running the application should bring up video playback of the video referenced in the YAML file.



Note: Instructions for running the distributed application involve calling the application from the “driver” node as well as from any worker nodes. For details, see the application run instructions in the [examples](#) directory on GitHub, or under `/opt/nvidia/holoscan/examples/video_replayer_distributed` in the NGC container and the Debian package.

Tip: Refer to [UCX Network Interface Selection](#) when running a distributed application across multiple nodes.

8.7 Bring Your Own Model (BYOM)

The Holoscan platform is optimized for performing AI inferencing workflows. This section shows how the user can easily modify the `bring_your_own_model` example to create their own AI applications.

In this example we’ll cover:

- The usage of `FormatConverterOp`, `InferenceOp`, `SegmentationPostprocessorOp` operators to add AI inference into the workflow.
- How to modify the existing code in this example to create an ultrasound segmentation application to visualize the results from a spinal scoliosis segmentation model.

Note: The example source code and run instructions can be found in the [examples](#) directory on GitHub, or under `/opt/nvidia/holoscan/examples` in the NGC container and the Debian package, alongside their executables.

8.7.1 Operators and Workflow

Here is the diagram of the operators and workflow used in the `byom.py` example.



Fig. 8.6: The BYOM inference workflow

The example code already contains the plumbing required to create the pipeline above where the video is loaded by `VideoStreamReplayer` and passed to two branches. The first branch goes directly to `Holoviz` to display the original video. The second branch in this workflow goes through AI inferencing and can be used to generate overlays such as bounding boxes, segmentation masks, or text to add additional information.

This second branch has three operators we haven't yet encountered.

- **Format Converter:** The input video stream goes through a preprocessing stage to convert the tensors to the appropriate shape/format before being fed into the AI model. It is used here to convert the datatype of the image from `uint8` to `float32` and resized to match the model's expectations.
- **Inference:** This operator performs AI inferencing on the input video stream with the provided model. It supports inferencing of multiple input video streams and models.
- **Segmentation Postprocessor:** This postprocessing stage takes the output of inference, either with the final softmax layer (multiclass) or sigmoid (2-class), and emits a tensor with `uint8` values that contain the highest probability class index. The output of the segmentation postprocessor is then fed into the `Holoviz` visualizer to create the overlay.

8.7.2 Prerequisites

To follow along this example, you can download the ultrasound dataset with the following commands:

```

$ wget --content-disposition \
  https://api.ngc.nvidia.com/v2/resources/nvidia/clara-holoscan/holoscan_ultrasound_
  ↳ sample_data/versions/20220608/zip \
  -O holoscan_ultrasound_sample_data_20220608.zip
$ unzip holoscan_ultrasound_sample_data_20220608.zip -d <SDK_ROOT>/data/ultrasound_
  ↳ segmentation
  
```

You can also follow along using your own dataset by adjusting the operator parameters based on your input video and model, and converting your video and model to a format that is understood by Holoscan.

Input Video

The video stream replayer supports reading video files that are encoded as GXF entities. These files are provided with the ultrasound dataset as the `ultrasound_256x256.gxf_entities` and `ultrasound_256x256.gxf_index` files.

Note: To use your own video data, you can use the `convert_video_to_gxf_entities.py` script (installed in `/opt/nvidia/holoscan/bin` or [on GitHub](#)) to encode your video. Note that - using this script - the metadata in the generated GXF tensor files will indicate that the data should be copied to the GPU on read.

Input model

Currently, the inference operators in Holoscan are able to load [ONNX models](#), or [TensorRT](#) engine files built for the GPU architecture on which you will be running the model. TensorRT engines are automatically generated from ONNX by the operators when the applications run.

If you are converting your model from PyTorch to ONNX, chances are your input is NCHW and will need to be converted to NHWC. We provide an example transformation script named `graph_surgeon.py`, installed in `/opt/nvidia/holoscan/bin` or available [on GitHub](#). You may need to modify the dimensions as needed before modifying your model.

Tip: To get a better understanding of your model, and if this step is necessary, websites such as [netron.app](#) can be used.

8.7.3 Understanding the Application Code

Before modifying the application, let's look at the existing code to get a better understanding of how it works.

Python

```

1  import os
2  from argparse import ArgumentParser
3
4  from holoscan.core import Application
5
6  from holoscan.operators import (
7      FormatConverterOp,
8      HolovizOp,
9      InferenceOp,
10     SegmentationPostprocessorOp,
11     VideoStreamReplayerOp,
12 )
13 from holoscan.resources import UnboundedAllocator
14
15
16 class BYOMApp(Application):
17     def __init__(self, data):
18         """Initialize the application
19 
```

(continues on next page)

(continued from previous page)

```

20  Parameters
21  -----
22  data : Location to the data
23  """
24
25  super().__init__()
26
27  # set name
28  self.name = "BYOM App"
29
30  if data == "none":
31      data = os.environ.get("HOLOSCAN_INPUT_PATH", "../data")
32
33  self.sample_data_path = data
34
35  self.model_path = os.path.join(os.path.dirname(__file__), "../model")
36  self.model_path_map = {
37      "byom_model": os.path.join(self.model_path, "identity_model.onnx"),
38  }
39
40  self.video_dir = os.path.join(self.sample_data_path, "racerx")
41  if not os.path.exists(self.video_dir):
42      raise ValueError(f"Could not find video data: {self.video_dir}")

```

- The built-in `FormatConvertOp`, `InferenceOp`, and `SegmentationPostprocessorOp` operators are imported on lines 7, 9, and 10. These 3 operators make up the preprocessing, inference, and postprocessing stages of our AI pipeline respectively.
- The `UnboundedAllocator` resource is imported on line 13. This is used by our application's operators for memory allocation.
- The paths to the identity model are defined on lines 35–38. This model passes its input tensor to its output, and acts as a placeholder for this example.
- The directory of the video files are defined on line 40.

Next, we look at the operators and their parameters defined in the application YAML file.

Python

```

1  def compose(self):
2      host_allocator = UnboundedAllocator(self, name="host_allocator")
3
4      source = VideoStreamReplayerOp(
5          self, name="replayer", directory=self.video_dir, **self.kwargs("replayer")
6      )
7
8      preprocessor = FormatConverterOp(
9          self, name="preprocessor", pool=host_allocator, **self.kwargs("preprocessor")
10     )
11
12     inference = InferenceOp(
13         self,

```

(continues on next page)

(continued from previous page)

```

14         name="inference",
15         allocator=host_allocator,
16         model_path_map=self.model_path_map,
17         **self.kwargs("inference"),
18     )
19
20     postprocessor = SegmentationPostprocessorOp(
21         self, name="postprocessor", allocator=host_allocator, **self.kwargs(
22             ↪ "postprocessor")
23     )
24
25     viz = HolovizOp(self, name="viz", **self.kwargs("viz"))

```

- An instance of the `UnboundedAllocator` resource class is created (line 2) and used by subsequent operators for memory allocation. This allocator allocates memory dynamically on the host as needed. For applications where latency becomes an issue, an allocator supporting a memory pool such as `BlockMemoryPool` or `RMMAllocator` could be used instead.
- The preprocessor operator (line 8) takes care of converting the input video from the source video to a format that can be used by the AI model.
- The inference operator (line 12) feeds the output from the preprocessor to the AI model to perform inference.
- The postprocessor operator (line 20) postprocesses the output from the inference operator before passing it downstream to the visualizer. Here, the segmentation postprocessor checks the probabilities output from the model to determine which class is most likely and emits this class index. This is then used by the Holoviz operator to create a segmentation mask overlay.

YAML

```

1 %YAML 1.2
2 replayer: # VideoStreamReplayer
3   basename: "racex"
4   frame_rate: 0 # as specified in timestamps
5   repeat: true # default: false
6   realtime: true # default: true
7   count: 0 # default: 0 (no frame count restriction)
8
9 preprocessor: # FormatConverter
10   out_tensor_name: source_video
11   out_dtype: "float32"
12   resize_width: 512
13   resize_height: 512
14
15 inference: # Inference
16   backend: "trt"
17   pre_processor_map:
18     "byom_model": ["source_video"]
19   inference_map:
20     "byom_model": ["output"]
21
22 postprocessor: # SegmentationPostprocessor

```

(continues on next page)

(continued from previous page)

```

23  in_tensor_name: output
24  # network_output_type: None
25  data_format: nchw
26
27  viz: # Holoviz
28      width: 854
29      height: 480
30      color_lut: [
31          [0.65, 0.81, 0.89, 0.1],
32          ]

```

- The preprocessor converts the tensors to float32 values (line 11) and ensures that the image is resized to 512x512 (line 12–13).
- The `pre_processor_map` parameter (lines 17–18) maps the model name(s) to input tensor name(s). Here, “source_video” matches the output tensor name of the preprocessor (line 10). The `inference_map` parameter maps the model name(s) to the output tensor name(s). Here, “output” matches the input tensor name of the postprocessor (line 23). For more details on `InferenceOp` parameters, see [Customizing the Inference Operator](#) or refer to [Inference](#).
- The `network_output_type` parameter is commented out on line 24 to remind ourselves that this second branch is currently not generating anything interesting. If not specified, this parameter defaults to “softmax” for `SegmentationPostprocessorOp`.
- The color lookup table defined on lines 30–32 is used here to create a segmentation mask overlay. The values of each entry in the table are RGBA values between 0.0 and 1.0. For the alpha value, 0.0 is fully transparent and 1.0 is fully opaque.

Finally, we define the application and workflow.

Python

```

1      # Define the workflow
2      self.add_flow(source, viz, {"output", "receivers"})
3      self.add_flow(source, preprocessor, {"output", "source_video"})
4      self.add_flow(preprocessor, inference, {"tensor", "receivers"})
5      self.add_flow(inference, postprocessor, {"transmitter", "in_tensor"})
6      self.add_flow(postprocessor, viz, {"out_tensor", "receivers"})
7
8
9  def main(config_file, data):
10      app = BYOMApp(data=data)
11      # if the --config command line argument was provided, it will override this config_
12      ↪ file
13      app.config(config_file)
14      app.run()
15
16  if __name__ == "__main__":
17      # Parse args
18      parser = ArgumentParser(description="BYOM demo application.")
19      parser.add_argument(
20          "-d",

```

(continues on next page)

(continued from previous page)

```

21     "--data",
22     default="none",
23     help=("Set the data path"),
24 )
25
26 args = parser.parse_args()
27 config_file = os.path.join(os.path.dirname(__file__), "byom.yaml")
28 main(config_file=config_file, data=args.data)

```

- The `add_flow()` on line 2 defines the first branch to display the original video.
- The `add_flow()` commands from line 3-6 defines the second branch to display the segmentation mask overlay.

8.7.4 Modifying the Application for Ultrasound Segmentation

To create the ultrasound segmentation application, we need to swap out the input video and model to use the ultrasound files, and adjust the parameters to ensure the input video is resized correctly to the model's expectations.

We will need to modify the Python and YAML files to change our application to the ultrasound segmentation application.

Python

```

1 class BYOMApp(Application):
2     def __init__(self, data):
3         """Initialize the application
4
5         Parameters
6         -----
7         data : Location to the data
8         """
9
10        super().__init__()
11
12        # set name
13        self.name = "BYOM App"
14
15        if data == "none":
16            data = os.environ.get("HOLOSCAN_INPUT_PATH", "../data")
17
18        self.sample_data_path = data
19
20        self.model_path = os.path.join(self.sample_data_path, "ultrasound_segmentation",
21        ↪ "model")
22        self.model_path_map = {
23            "byom_model": os.path.join(self.model_path, "us_unet_256x256_nhwc.onnx"),
24        }
25
26        self.video_dir = os.path.join(self.sample_data_path, "ultrasound_segmentation",
27        ↪ "video")
28        if not os.path.exists(self.video_dir):

```

(continues on next page)

(continued from previous page)

```
27 raise ValueError(f"Could not find video data: {self.video_dir}")
```

- Update `self.model_path_map` to the ultrasound segmentation model (lines 20–23).
- Update `self.video_dir` to point to the directory of the ultrasound video files (line 25).

YAML

```
1 replayer: # VideoStreamReplayer
2   basename: "ultrasound_256x256"
3   frame_rate: 0 # as specified in timestamps
4   repeat: true # default: false
5   realtime: true # default: true
6   count: 0 # default: 0 (no frame count restriction)
7
8 preprocessor: # FormatConverter
9   out_tensor_name: source_video
10  out_dtype: "float32"
11  resize_width: 256
12  resize_height: 256
13
14 inference: # Inference
15   backend: "trt"
16   pre_processor_map:
17     "byom_model": ["source_video"]
18   inference_map:
19     "byom_model": ["output"]
20
21 postprocessor: # SegmentationPostprocessor
22   in_tensor_name: output
23   network_output_type: softmax
24   data_format: nchw
25
26 viz: # Holoviz
27   width: 854
28   height: 480
29   color_lut: [
30     [0.65, 0.81, 0.89, 0.1],
31     [0.2, 0.63, 0.17, 0.7]
32   ]
```

- Update `basename` to the basename of the ultrasound video files (line 2).
- The AI model expects the width and height of the images to be 256x256, update the preprocessor's parameters to resize the input to 256x256 (line 11–12).
- The AI model's final output layer is a softmax, so we indicate this to the postprocessor (line 23).
- Since this model predicts between two classes, we need another entry in Holoviz's color lookup table (line 31). Note that the alpha value of the first color entry is 0.1 (line 30) so the mask for the background class may not be visible. The second entry we just added is a green color and has an alpha value of 0.7 so it will be easily visible.

The above changes are enough to update the BYOM example to the ultrasound segmentation application.

In general, when deploying your own AI models, you will need to consider the operators in the second branch. This example uses a pretty typical AI workflow:

- **Input:** This could be a video on disk, an input stream from a capture device, or other data stream.
- **Preprocessing:** You may need to preprocess the input stream to convert tensors into the shape and format that is expected by your AI model (e.g., converting datatype and resizing).
- **Inference:** Your model will need to be in ONNX or TensorRT format.
- **Postprocessing:** An operator that postprocesses the output of the model to a format that can be readily used by downstream operators.
- **Output:** The postprocessed stream can be displayed or used by other downstream operators.

The Holoscan SDK comes with a number of [built-in operators](#) that you can use to configure your own workflow. If needed, you can write your own custom operators or visit [Holohub](#) for additional implementations and ideas for operators.

8.7.5 Running the Application

After modifying the application as instructed above, running the application should bring up the ultrasound video with a segmentation mask overlay similar to the image below.

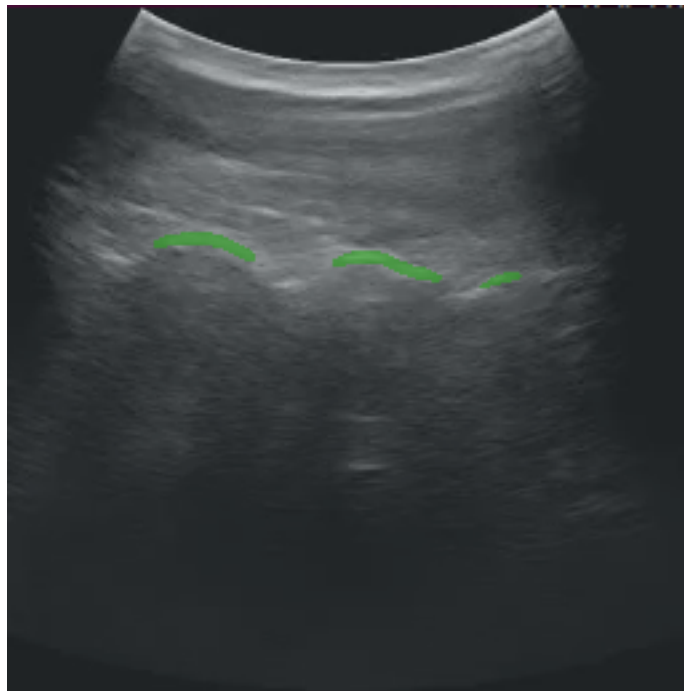


Fig. 8.7: Ultrasound Segmentation

Note: If you run the `byom.py` application without modification and are using the debian installation, you may run into the following error message:

```
[error] Error in Inference Manager ... TRT Inference: failed to build TRT engine file.
```

In this case, modifying the write permissions for the model directory should help (use with caution):

```
sudo chmod a+w /opt/nvidia/holoscan/examples/bring_your_own_model/model
```

8.7.6 Customizing the Inference Operator

The built-in `InferenceOp` operator provides the functionality of the *Inference*. This operator has a `receivers` port that can connect to any number of upstream ports to allow for multi-ai inferencing, and one `transmitter` port to send results downstream. Below is a description of some of the operator's parameters and a general guidance on how to use them.

- `backend`: If the input models are in `tensorrt engine file` format, select `trt` as the backend. If the input models are in `onnx` format select either `trt` or `onnx` as the backend.
- `allocator`: Can be passed to this operator to specify how the output tensors are allocated.
- `model_path_map`: Contains dictionary keys with unique strings that refer to each model. The values are set to the path to the model files on disk. All models must be either in `onnx` or in `tensorrt engine file` format. The Holoscan Inference Module will do the `onnx` to `tensorrt` model conversion if the TensorRT engine files do not exist.
- `pre_processor_map`: This dictionary should contain the same keys as `model_path_map`, mapping to the output tensor name for each model.
- `inference_map`: This dictionary should contain the same keys as `model_path_map`, mapping to the output tensor name for each model.
- `enable_fp16`: Boolean variable indicating if half-precision should be used to speed up inferencing. The default value is `False`, and uses single-precision (32-bit fp) values.
- `input_on_cuda`: Indicates whether input tensors are on device or host.
- `output_on_cuda`: Indicates whether output tensors are on device or host.
- `transmit_on_cuda`: If `True`, it means the data transmission from the inference will be on **Device**, otherwise it means the data transmission from the inference will be on **Host**.

8.7.7 Common Pitfalls Deploying New Models

Color Channel Order

It is important to know what channel order your model expects. This may be indicated by the training data, pre-training transformations performed at training, or the expected inference format used in your application.

For example, if your inference data is RGB, but your model expects BGR, you will need to add the following to your `segmentation_preprocessor` in the yml file: `out_channel_order: [2,1,0]`.

Normalizing Your Data

Similarly, default scaling for streaming data is `[0, 1]`, but dependent on how your model was trained, you may be expecting `[0, 255]`.

For the above case, you would add the following to your `segmentation_preprocessor` in the YAML file:

```
scale_min:  0.0 scale_max:  255.0
```

Network Output Type

Models often have different output types such as `Sigmoid`, `Softmax`, or perhaps something else, and you may need to examine the last few layers of your model to determine which applies to your case.

As in the case of our ultrasound segmentation example above, we added the following in our YAML file:

```
network_output_type:  softmax
```


CREATING AN APPLICATION

In this section, we'll address:

- How to *define an Application class*.
- How to *configure an Application*.
- How to *define different types of workflows*.
- How to *build and run your application*.

Note: This section covers basics of applications running as a single fragment. For multi-fragment applications, refer to the *distributed application documentation*.

9.1 Defining an Application Class

The following code snippet shows an example Application code skeleton:

C++

- We define the `App` class that inherits from the `Application` base class.
- We create an instance of the `App` class in `main()` using the `make_application()` function.
- The `run()` method starts the application which will execute its `compose()` method where the custom workflow will be defined.

```
#include <holoscan/holoscan.hpp>

class App : public holoscan::Application {
public:
    void compose() override {
        // Define Operators and workflow
        // ...
    }
};

int main() {
    auto app = holoscan::make_application<App>();
    app->run();
}
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

Python

- We define the `App` class that inherits from the `Application` base class.
- We create an instance of the `App` class in a `main()` function that is called from `__main__`.
- The `run()` method starts the application which will execute its `compose()` method where the custom workflow will be defined.

```
from holoscan.core import Application

class App(Application):

    def compose(self):
        # Define Operators and workflow
        # ...

def main():
    app = App()
    app.run()

if __name__ == "__main__":
    main()
```

Note: It is recommended to call `run()` from within a separate `main()` function rather than calling it directly from `__main__`. This will ensure that the `Application`'s destructor is called before the Python process exits.

Tip: This is also illustrated in the *hello_world* example.

It is also possible to instead launch the application asynchronously (i.e., non-blocking for the thread launching the application), as shown below:

C++

This can be done simply by replacing the call to `run()` with `run_async()` which returns a `std::future`. Calling `future.get()` will block until the application has finished running and throw an exception if a runtime error occurred during execution.

```
int main() {
    auto app = holoscan::make_application<App>();
    auto future = app->run_async();
    future.get();
    return 0;
}
```

Python

This can be done simply by replacing the call to `run()` with `run_async()` which returns a `Python concurrent.futures.Future`. Calling `future.result()` will block until the application has finished running and raise an exception if a runtime error occurred during execution.

```
def main():
    app = App()
    future = app.run_async()
    future.result()

if __name__ == "__main__":
    main()
```

Tip: This is also illustrated in the `ping_simple_run_async` example.

9.2 Configuring an Application

An application can be configured at different levels:

1. *providing the GXF extensions that need to be loaded* (when using *GXF operators*).
2. configuring parameters for your application, including for: a. *the operators* in the workflow. b. *the scheduler* of your application.
3. *configuring some runtime properties* when deploying for production.

The sections below will describe how to configure each of them, starting with a native support for YAML-based configuration for convenience.

9.2.1 YAML configuration support

Holoscan supports loading arbitrary parameters from a YAML configuration file at runtime, making it convenient to configure each item listed above, or other custom parameters you wish to add on top of the existing API. For C++ applications, it also provides the ability to change the behavior of your application without needing to recompile it.

Note: Usage of the YAML utility is optional. Configurations can be hardcoded in your program, or done using any parser that you choose.

Here is an example YAML configuration:

```
string_param: "test"
float_param: 0.50
bool_param: true
dict_param:
  key_1: value_1
  key_2: value_2
```

Ingesting these parameters can be done using the two methods below:

C++

- The `config()` method takes the path to the YAML configuration file. If the input path is relative, it will be relative to the current working directory.
- The `from_config()` method returns an `ArgList` object for a given key in the YAML file. It holds a list of `Arg` objects, each of which holds a name (key) and a value.
 - If the `ArgList` object has only one `Arg` (when the key is pointing to a scalar item), it can be converted to the desired type using the `as()` method by passing the type as an argument.
 - The key can be a dot-separated string to access nested fields.
- The `config_keys()` method returns an unordered set of the key names accessible via `from_config()`.

```
// Pass configuration file
auto app = holoscan::make_application<App>();
app->config("path/to/app_config.yaml");

// Scalars
auto string_param = app->from_config("string_param").as<std::string>();
auto float_param = app->from_config("float_param").as<float>();
auto bool_param = app->from_config("bool_param").as<bool>();

// Dict
auto dict_param = app->from_config("dict_param");
auto dict_nested_param = app->from_config("dict_param.key_1").as<std::string>();

// Print
std::cout << "string_param: " << string_param << std::endl;
std::cout << "float_param: " << float_param << std::endl;
std::cout << "bool_param: " << bool_param << std::endl;
std::cout << "dict_param:\n" << dict_param.description() << std::endl;
std::cout << "dict_param['key1']: " << dict_nested_param << std::endl;

// // Output
// string_param: test
// float_param: 0.5
// bool_param: 1
// dict_param:
// name: arglist
// args:
//   - name: key_1
//     type: YAML::Node
//     value: value_1
//   - name: key_2
//     type: YAML::Node
//     value: value_2
// dict_param['key1']: value_1
```


Python

- The `config()` method takes the path to the YAML configuration file. If the input path is relative, it will be relative to the current working directory.
- The `kwargs()` method return a regular Python dict for a given key in the YAML file.
 - **Advanced:** this method wraps the `from_config()` method similar to the C++ equivalent, which returns an `ArgList` object if the key is pointing to a map item, or an `Arg` object if the key is pointing to a scalar item. An `Arg` object can be cast to the desired type (e.g., `str(app.from_config("string_param"))`).
- The `config_keys()` method returns a set of the key names accessible via `from_config()`.

```
# Pass configuration file
app = App()
app.config("path/to/app_config.yaml")

# Scalars
string_param = app.kwargs("string_param")["string_param"]
float_param = app.kwargs("float_param")["float_param"]
bool_param = app.kwargs("bool_param")["bool_param"]

# Dict
dict_param = app.kwargs("dict_param")
dict_nested_param = dict_param["key_1"]

# Print
print(f"string_param: {string_param}")
print(f"float_param: {float_param}")
print(f"bool_param: {bool_param}")
print(f"dict_param: {dict_param}")
print(f"dict_param['key_1']: {dict_nested_param}")

# # Output:
# string_param: test
# float_param: 0.5
# bool_param: True
# dict_param: {'key_1': 'value_1', 'key_2': 'value_2'}
# dict_param['key_1']: 'value_1'
```

Warning: `from_config()` cannot be used as inputs to the built-in operators at this time. Therefore, it's recommended to use `kwargs()` in Python.

Tip: This is also illustrated in the [video_replayer](#) example.

Attention: With both `from_config` and `kwargs`, the returned `ArgList`/dictionary will include both the key and its associated item if that item value is a scalar. If the item is a map/dictionary itself, the input key is dropped, and the output will only hold the key/values from that item.

9.2.2 Loading GXF extensions

If you use operators that depend on GXF extensions for their implementations (known as *GXF operators*), the shared libraries (.so) of these extensions need to be dynamically loaded as plugins at runtime.

The SDK already automatically handles loading the required extensions for the *built-in operators* in both C++ and Python, as well as common extensions (listed here). To load additional extensions for your own operators, you can use one of the following approach:

YAML

```
extensions:
- libgxf_myextension1.so
- /path/to/libgxf_myextension2.so
```

C++

```
auto app = holoscan::make_application<App>();
auto exts = {"libgxf_myextension1.so", "/path/to/libgxf_myextension2.so"};
for (auto& ext : exts) {
    app->executor().extension_manager()->load_extension(ext);
}
```

PYTHON

```
from holoscan.gxf import load_extensions
from holoscan.core import Application
app = Application()
context = app.executor.context_uint64
exts = ["libgxf_myextension1.so", "/path/to/libgxf_myextension2.so"]
load_extensions(context, exts)
```

Note: To be discoverable, paths to these shared libraries need to either be absolute, relative to your working directory, installed in the lib/gxf_extensions folder of the holoscan package, or listed under the HOLOSCAN_LIB_PATH or LD_LIBRARY_PATH environment variables.

Please see other examples in the [system tests](#) in the Holoscan SDK repository.

9.2.3 Configuring operators

Operators are defined in the `compose()` method of your application. They are not instantiated (with the `initialize` method) until an application's `run()` method is called.

Operators have three type of fields which can be configured: parameters, conditions, and resources.

Configuring operator parameters

Operators could have parameters defined in their `setup` method to better control their behavior (see details when *creating your own operators*). The snippet below would be the implementation of this method for a minimal operator named `MyOp`, that takes a string and a boolean as parameters; we'll ignore any extra details for the sake of this example:

C++

```
void setup(OperatorSpec& spec) override {
    spec.param(string_param_, "string_param");
    spec.param(bool_param_, "bool_param");
}
```

PYTHON

```
def setup(self, spec: OperatorSpec):
    spec.param("string_param")
    spec.param("bool_param")
    # Optional in python. Could define `self.<param_name>` instead in `def __init__`
```

Tip: Given an instance of an operator class, you can print a human-readable description of its specification to inspect the parameter fields that can be configured on that operator class:

C++

```
std::cout << operator_object->spec()->description() << std::endl;
```

PYTHON

```
print(operator_object.spec)
```

Given this YAML configuration:

```
myop_param:
  string_param: "test"
  bool_param: true

bool_param: false # we'll use this later
```

We can configure an instance of the `MyOp` operator in the application's `compose` method like this:

C++

```
void compose() override {
    // Using YAML
    auto my_op1 = make_operator<MyOp>("my_op1", from_config("myop_param"));

    // Same as above
    auto my_op2 = make_operator<MyOp>("my_op2",
        Arg("string_param", std::string("test")), // can use Arg(key, value)...
        Arg("bool_param") = true                 // ... or Arg(key) = value
    );
}
```

PYTHON

```
def compose(self):
    # Using YAML
    my_op1 = MyOp(self, name="my_op1", **self.kwargs("myop_param"))

    # Same as above
    my_op2 = MyOp(self,
        name="my_op2",
        string_param="test",
        bool_param=True,
    )
```

Tip: This is also illustrated in the [ping_custom_op](#) example.

If multiple ArgList are provided with duplicate keys, the latest one overrides them:

C++

```
void compose() override {
    // Using YAML
    auto my_op1 = make_operator<MyOp>("my_op1",
        from_config("myop_param"),
        from_config("bool_param")
    );

    // Same as above
    auto my_op2 = make_operator<MyOp>("my_op2",
        Arg("string_param", "test"),
        Arg("bool_param") = true,
        Arg("bool_param") = false
    );

    // -> my_op `bool_param` will be set to `false`
}
```

PYTHON

```
def compose(self):
    # Using YAML
    my_op1 = MyOp(self, name="my_op1",
                  from_config("myop_param"),
                  from_config("bool_param"),
                  )

    # Note: We're using from_config above since we can't merge automatically with kwargs
    # as this would create duplicated keys. However, we recommend using kwargs in Python
    # to avoid limitations with wrapped operators, so the code below is preferred.

    # Same as above
    params = self.kwargs("myop_param").update(self.kwargs("bool_param"))
    my_op2 = MyOp(self, name="my_op2", params)

    # -> my_op `bool_param` will be set to `False`
```

Configuring operator conditions

By default, operators with no input ports will continuously run, while operators with input ports will run as long as they receive inputs (as they're configured with the [MessageAvailableCondition](#)).

To change that behavior, one or more other *conditions* classes can be passed to the constructor of an operator to define when it should execute.

For example, we set three conditions on this operator `my_op`:

C++

```
void compose() override {
    // Limit to 10 iterations
    auto c1 = make_condition<CountCondition>("my_count_condition", 10);

    // Wait at least 200 milliseconds between each execution
    auto c2 = make_condition<PeriodicCondition>("my_periodic_condition", "200ms");

    // Stop when the condition calls `disable_tick()`
    auto c3 = make_condition<BooleanCondition>("my_bool_condition");

    // Pass directly to the operator constructor
    auto my_op = make_operator<MyOp>("my_op", c1, c2, c3);
}
```

PYTHON

```
def compose(self):
    # Limit to 10 iterations
    c1 = CountCondition(self, 10, name="my_count_condition")

    # Wait at least 200 milliseconds between each execution
    c2 = PeriodicCondition(self, timedelta(milliseconds=200), name="my_periodic_condition")

    # Stop when the condition calls `disable_tick()`
    c3 = BooleanCondition(self, name="my_bool_condition")

    # Pass directly to the operator constructor
    my_op = MyOp(self, c1, c2, c3, name="my_op")
```

Tip: This is also illustrated in the `conditions`’ examples.

Note: You’ll need to specify a unique name for the conditions if there are multiple conditions applied to an operator.

Configuring operator resources

Some *resources* can be passed to the operator’s constructor, typically an *allocator* passed as a regular parameter.

For example:

C++

```
void compose() override {
    // Allocating memory pool of specific size on the GPU
    // ex: width * height * channels * channel size in bytes
    auto block_size = 640 * 480 * 4 * 2;
    auto p1 = make_resource<BlockMemoryPool>("my_pool1", 1, size, 1);

    // Provide unbounded memory pool
    auto p2 = make_condition<UnboundedAllocator>("my_pool2");

    // Pass to operator as parameters (name defined in operator setup)
    auto my_op = make_operator<MyOp>("my_op",
                                     Arg("pool1", p1),
                                     Arg("pool2", p2));
}
```

PYTHON

```
def compose(self):
    # Allocating memory pool of specific size on the GPU
    # ex: width * height * channels * channel size in bytes
    block_size = 640 * 480 * 4 * 2;
    p1 = BlockMemoryPool(self, name="my_pool1", storage_type=1, block_size=block_size, num_
    ↪ blocks=1)

    # Provide unbounded memory pool
    p2 = UnboundedAllocator(self, name="my_pool2")

    # Pass to operator as parameters (name defined in operator setup)
    auto my_op = MyOp(self, name="my_op", pool1=p1, pool2=p2)
```

Native resource creation

The resources bundled with the SDK are wrapping an underlying GXF component. However, it is also possible to define a “native” resource without any need to create and wrap an underlying GXF component. Such a resource can also be passed conditionally to an operator in the same way as the resources created in the previous section.

For example:

C++

To create a native resource, implement a class that inherits from Resource

```
namespace holoscan {

class MyNativeResource : public holoscan::Resource {
public:
    HOLOSCAN_RESOURCE_FORWARD_ARGS_SUPER(MyNativeResource, Resource)

    MyNativeResource() = default;

    // add any desired parameters in the setup method
    // (a single string parameter is shown here for illustration)
    void setup(ComponentSpec& spec) override {
        spec.param(message_, "message", "Message string", "Message String", std::string(
    ↪ "test message"));
    }

    // add any user-defined methods (these could be called from an Operator's compute_
    ↪ method)
    std::string message() { return message_.get(); }

private:
    Parameter<std::string> message_;
};
} // namespace: holoscan
```

The setup method can be used to define any parameters needed by the resource.

This resource can be used with a C++ operator, just like any other resource. For example, an operator could have a parameter holding a shared pointer to `MyNativeResource` as below.

```
private:

class MyOperator : public holoscan::Operator {
public:
    HOLOSCAN_OPERATOR_FORWARD_ARGS(MyOperator)

    MyOperator() = default;

    void setup(OperatorSpec& spec) override {
        spec.param(message_resource_, "message_resource", "message_resource",
                    "resource printing a message");
    }

    void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override {
        HOLOSCAN_LOG_TRACE("MyOp::compute()");

        // get a resource based on its name (this assumes the app author named the resource
        ↪ "message_resource")
        auto res = resource<MyNativeResource>("message_resource");
        if (!res) {
            throw std::runtime_error("resource named 'message_resource' not found!");
        }

        // call a method on the retrieved resource class
        auto message = res->message();

    };

private:
    Parameter<std::shared_ptr<holoscan::MyNativeResource>> message_resource_;
}
```

The compute method above demonstrates how the templated resource method can be used to retrieve a resource. and the resource could be created and passed via a named argument in the usual way

```
// example code for within Application::compose (or Fragment::compose)

auto message_resource = make_resource<holoscan::MyNativeResource>(
    "message_resource", holoscan::Arg("message", "hello world");

auto my_op = std::make_operator<holoscan::ops::MyOperator>(
    "my_op", holoscan::Arg("message_resource", message_resource));
```

As with GXF-based resources, it is also possible to pass a native resource as a positional argument to the operator constructor.

For a concrete example of native resource use in a real application, see the [volume_rendering_xr application](#) on Holohub. This application uses a native `XrSession resource` type which corresponds to a single OpenXR session. This single “session” resource can then be shared by both the `XrBeginFrameOp` and `XrEndFrameOp` operators.

Python

To create a native resource, implement a class that inherits from `Resource`.

```
class MyNativeResource(Resource):
    def __init__(self, fragment, message="test message", *args, **kwargs):
        self.message = message
        super().__init__(fragment, *args, **kwargs)

    # Could optionally define Parameter as in C++ via spec.param as below.
    # Here, we chose instead to pass message as an argument to __init__ above.
    # def setup(self, spec: ComponentSpec):
    #     spec.param("message", "test message")

    # define a custom method
    def message(self):
        return self.message
```

The below shows how some custom operator could use such a resource in its compute method

```
class MyOperator(Operator):
    def compute(self, op_input, op_output, context):
        resource = self.resource("message_resource")
        if resource is None:
            raise ValueError("expected message resource not found")
        assert isinstance(resource, MyNativeResource)

        print(f"message = {resource.message()}")
```

where this native resource could have been created and passed positionally to `MyOperator` as follows

```
# example code within Application.compose (or Fragment.compose)

message_resource = MyNativeResource(
    fragment=self, message="hello world", name="message_resource")

# pass the native resource as a positional argument to MyOperator
my_op = MyOperator(fragment=self, message_resource)
```

There is a minimal example of native resource use in the `examples/native` folder.

9.2.4 Configuring the scheduler

The *scheduler* controls how the application schedules the execution of the operators that make up its *workflow*.

The default scheduler is a single-threaded *GreedyScheduler*. An application can be configured to use a different scheduler Scheduler (C++/Python) or change the parameters from the default scheduler, using the `scheduler()` function (C++/Python).

For example, if an application needs to run multiple operators in parallel, the *MultiThreadScheduler* or *EventBasedScheduler* can instead be used. The difference between the two is that the *MultiThreadScheduler* is based on actively polling operators to determine if they are ready to execute, while the *EventBasedScheduler* will instead wait for an event indicating that an operator is ready to execute.

The code snippet below shows how to set and configure a non-default scheduler:

C++

- We create an instance of a `holoscan::Scheduler` derived class by using the `make_scheduler()` function. Like operators, parameters can come from explicit `Args` or `ArgList`, or from a YAML configuration.
- The `scheduler()` method assigns the scheduler to be used by the application.

```
auto app = holoscan::make_application<App>();
auto scheduler = app->make_scheduler<holoscan::EventBasedScheduler>(
    "myscheduler",
    Arg("worker_thread_number", 4),
    Arg("stop_on_deadlock", true)
);
app->scheduler(scheduler);
app->run();
```

Python

- We create an instance of a `Scheduler` class in the `schedulers` module. Like operators, parameters can come from an explicit `Arg` or `ArgList`, or from a YAML configuration.
- The `scheduler()` method assigns the scheduler to be used by the application.

```
app = App()
scheduler = holoscan.schedulers.EventBasedScheduler(
    app,
    name="myscheduler",
    worker_thread_number=4,
    stop_on_deadlock=True,
)
app.scheduler(scheduler)
app.run()
```

Tip: This is also illustrated in the [multithread](#) example.

9.2.5 Configuring runtime properties

As described *below*, applications can run simply by executing the C++ or Python application manually on a given node, or by *packaging it* in a *HAP container*. With the latter, runtime properties need to be configured: refer to the *App Runner Configuration* for details.

9.3 Application Workflows

Note: Operators are initialized according to the **topological order** of its fragment-graph. When an application runs, the operators are executed in the same topological order. Topological ordering of the graph ensures that all the data dependencies of an operator are satisfied before its instantiation and execution. Currently, we do not support specifying a different and explicit instantiation and execution order of the operators.

9.3.1 One-operator Workflow

The simplest form of a workflow would be a single operator.

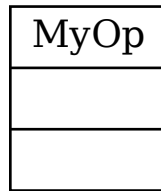


Fig. 9.1: A one-operator workflow

The graph above shows an **Operator** (C++/Python) (named **MyOp**) that has neither inputs nor output ports.

- Such an operator may accept input data from the outside (e.g., from a file) and produce output data (e.g., to a file) so that it acts as both the source and the sink operator.
- Arguments to the operator (e.g., input/output file paths) can be passed as parameters as described in the *section above*.

We can add an operator to the workflow by calling `add_operator` (C++/Python) method in the `compose()` method.

The following code shows how to define a one-operator workflow in `compose()` method of the `App` class (assuming that the operator class `MyOp` is declared/defined in the same file).

CPP

```

1 class App : public holoscan::Application {
2 public:
3     void compose() override {
4         // Define Operators
5         auto my_op = make_operator<MyOp>("my_op");
6
7         // Define the workflow
8         add_operator(my_op);
9     }
10 };

```

PYTHON

```

1 class App(Application):
2
3     def compose(self):
4         # Define Operators
5         my_op = MyOp(self, name="my_op")
6
7         # Define the workflow
8         self.add_operator(my_op)

```

9.3.2 Linear Workflow

Here is an example workflow where the operators are connected linearly:

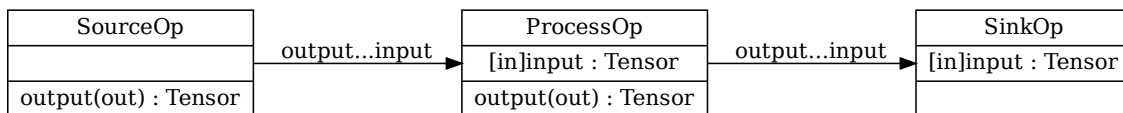


Fig. 9.2: A linear workflow

In this example, **SourceOp** produces a message and passes it to **ProcessOp**. **ProcessOp** produces another message and passes it to **SinkOp**.

We can connect two operators by calling the `add_flow()` method (C++/Python) in the `compose()` method.

The `add_flow()` method (C++/Python) takes the source operator, the destination operator, and the optional port name pairs. The port name pair is used to connect the output port of the source operator to the input port of the destination operator. The first element of the pair is the output port name of the upstream operator and the second element is the input port name of the downstream operator. An empty port name ("") can be used for specifying a port name if the operator has only one input/output port. If there is only one output port in the upstream operator and only one input port in the downstream operator, the port pairs can be omitted.

The following code shows how to define a linear workflow in the `compose()` method of the `App` class (assuming that the operator classes `SourceOp`, `ProcessOp`, and `SinkOp` are declared/defined in the same file).

CPP

```

1 class App : public holoscan::Application {
2 public:
3     void compose() override {
4         // Define Operators
5         auto source = make_operator<SourceOp>("source");
6         auto process = make_operator<ProcessOp>("process");
7         auto sink = make_operator<SinkOp>("sink");
8
9         // Define the workflow
10        add_flow(source, process); // same as `add_flow(source, process, {"output", "input"}
11        ↪);`
12        add_flow(process, sink); // same as `add_flow(process, sink, {"", ""});`
13    }
14 };

```

PYTHON

```

1 class App(Application):
2
3     def compose(self):
4         # Define Operators
5         source = SourceOp(self, name="source")
6         process = ProcessOp(self, name="process")
7         sink = SinkOp(self, name="sink")
8
9         # Define the workflow
10        self.add_flow(source, process) # same as `self.add_flow(source, process, {(
11        ↪"output", "input")})`
12        self.add_flow(process, sink) # same as `self.add_flow(process, sink, {("", "")}
13        ↪)`

```

9.3.3 Complex Workflow (Multiple Inputs and Outputs)

You can design a complex workflow like below where some operators have multi-inputs and/or multi-outputs:

CPP

```

1 class App : public holoscan::Application {
2 public:
3     void compose() override {
4         // Define Operators
5         auto reader1 = make_operator<Reader1>("reader1");
6         auto reader2 = make_operator<Reader2>("reader2");
7         auto processor1 = make_operator<Processor1>("processor1");
8         auto processor2 = make_operator<Processor2>("processor2");
9         auto processor3 = make_operator<Processor3>("processor3");
10        auto writer = make_operator<Writer>("writer");

```

(continues on next page)

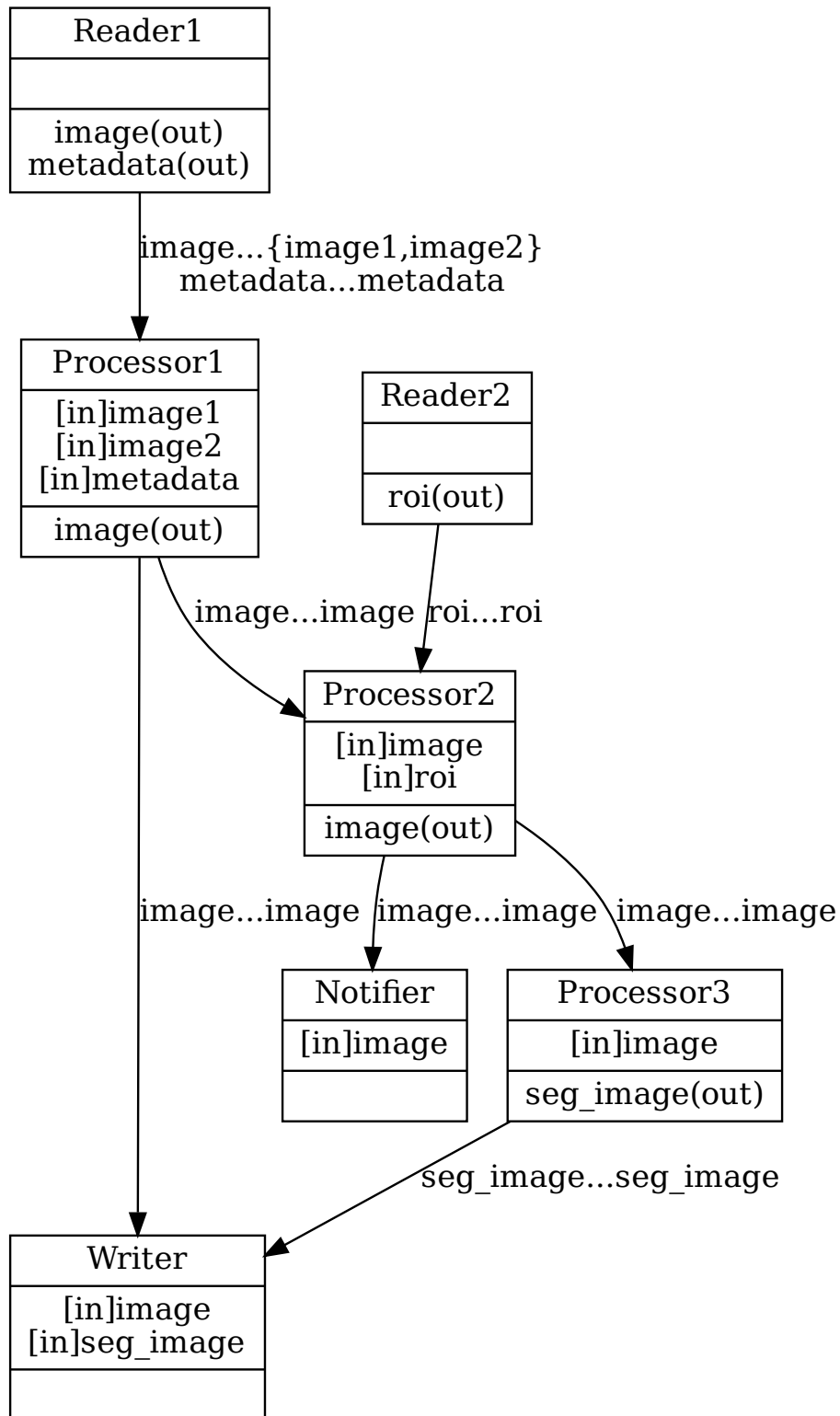


Fig. 9.3: A complex workflow (multiple inputs and outputs)

(continued from previous page)

```

11     auto notifier = make_operator<Notifier>("notifier");
12
13     // Define the workflow
14     add_flow(reader1, processor1, {"image", "image1"}, {"image", "image2"}, {"metadata",
15     ↪ "metadata"}));
16     add_flow(reader1, processor1, {"image", "image2"}));
17     add_flow(reader2, processor2, {"roi", "roi"}));
18     add_flow(processor1, processor2, {"image", "image"}));
19     add_flow(processor1, writer, {"image", "image"}));
20     add_flow(processor2, notifier);
21     add_flow(processor2, processor3);
22     add_flow(processor3, writer, {"seg_image", "seg_image"}));
23 }

```

PYTHON

```

1 class App(Application):
2
3     def compose(self):
4         # Define Operators
5         reader1 = Reader1Op(self, name="reader1")
6         reader2 = Reader2Op(self, name="reader2")
7         processor1 = Processor1Op(self, name="processor1")
8         processor2 = Processor2Op(self, name="processor2")
9         processor3 = Processor3Op(self, name="processor3")
10        notifier = NotifierOp(self, name="notifier")
11        writer = WriterOp(self, name="writer")
12
13        # Define the workflow
14        self.add_flow(reader1, processor1, {"image", "image1"}, {"image", "image2"}, (
15        ↪ "metadata", "metadata"))
16        self.add_flow(reader2, processor2, {"roi", "roi"})
17        self.add_flow(processor1, processor2, {"image", "image"})
18        self.add_flow(processor1, writer, {"image", "image"})
19        self.add_flow(processor2, notifier)
20        self.add_flow(processor2, processor3)
21        self.add_flow(processor3, writer, {"seg_image", "seg_image"})

```

If there is a cycle in the graph with no implicit root operator, the root operator is either the first operator in the first call to `add_flow` method (C++/Python), or the operator in the first call to `add_operator` method (C++/Python).

C++

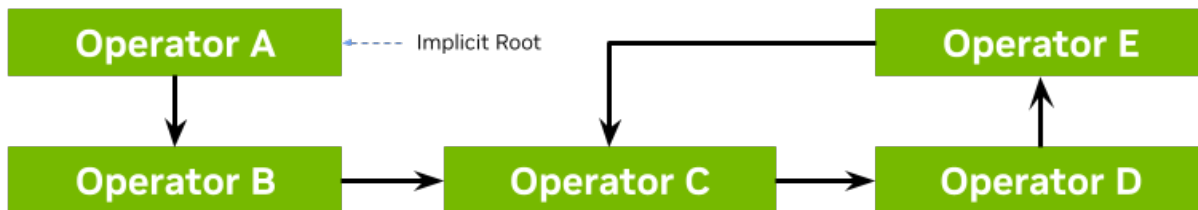
```

auto op1 = make_operator<...>("op1");
auto op2 = make_operator<...>("op2");
auto op3 = make_operator<...>("op3");

add_flow(op1, op2);
add_flow(op2, op3);
add_flow(op3, op1);
// There is no implicit root operator
// op1 is the root operator because op1 is the first operator in the first call to add_
↪ flow

```

If there is a cycle in the graph with an implicit root operator which has no input port, then the initialization and execution orders of the operators are still topologically sorted as far as possible until the cycle needs to be explicitly broken. An example is given below:



Order of operators: Operator A, Operator B, {a combination of Operator C, D and E}

9.4 Building and running your Application

C++

You can build your C++ application using CMake, by calling `find_package(holoscan)` in your `CMakeLists.txt` to load the SDK libraries. Your executable will need to link against:

- `holoscan::core`
- any operator defined outside your `main.cpp` which you wish to use in your app workflow, such as:
 - SDK *built-in operators* under the `holoscan::ops` namespace.
 - operators created separately in your project with `add_library`.
 - operators imported externally using with `find_library` or `find_package`.

Listing 9.1: `<src_dir>/CMakeLists.txt`

```

# Your CMake project
cmake_minimum_required(VERSION 3.20)
project(my_project CXX)

# Finds the holoscan SDK
find_package(holoscan REQUIRED CONFIG PATHS "/opt/nvidia/holoscan")

# Create an executable for your application

```

(continues on next page)

(continued from previous page)

```

add_executable(my_app main.cpp)

# Link your application against holoscan::core and any existing operators you'd like to
↪ use
target_link_libraries(my_app
    PRIVATE
        holoscan::core
        holoscan::ops::<some_built_in_operator_target>
        <some_other_operator_target>
        <...>
)

```

Tip: This is also illustrated in all the examples:

- in `CMakeLists.txt` for the SDK installation directory - `/opt/nvidia/holoscan/examples`.
- in `CMakeLists.min.txt` for the SDK [source directory](#).

Once your `CMakeLists.txt` is ready in `<src_dir>`, you can build in `<build_dir>` with the command line below. You can optionally pass `Holoscan_ROOT` if the SDK installation you'd like to use differs from the `PATHS` given to `find_package(holoscan)` above.

```

# Configure
cmake -S <src_dir> -B <build_dir> -D Holoscan_ROOT="/opt/nvidia/holoscan"
# Build
cmake --build <build_dir> -j

```

You can then run your application by running `<build_dir>/my_app`.

Python

Python applications do not require building. Simply ensure that:

- The `holoscan` python module is installed in your `dist-packages` or is listed under the `PYTHONPATH` env variable so you can import `holoscan.core` and any built-in operator you might need in `holoscan.operators`.
- Any external operators are available in modules in your `dist-packages` or contained in `PYTHONPATH`.

Note: While Python applications do not need to be built, they might depend on operators that wrap C++ operators. All Python operators built-in in the SDK already ship with the Python bindings pre-built. Follow [this section](#) if you are wrapping C++ operators yourself to use in your Python application.

You can then run your application by running `python3 my_app.py`.

Note: Given a CMake project, a pre-built executable, or a Python application, you can also use the [Holoscan CLI](#) to *package and run your Holoscan application* in a OCI-compliant container image.

9.5 Dynamic Application Metadata

As of Holoscan v2.3 it is possible to send metadata alongside the data emitted from an operator's output ports. This metadata can then be used and/or modified by any downstream operators. Currently this feature is only available for C++ applications, but will also be available to Python applications in a future release. The subsections below describe how this feature can be enabled and used.

9.5.1 Enabling application metadata

Currently the metadata feature is disabled by default and must be explicitly enabled as shown in the code block below

C++

```
app = holoscan::make_application<MyApplication>();

// Enable metadata feature before calling app->run() or app->run_async()
app->is_metadata_enabled(true);

app->run();
```

Python

```
app = MyApplication()

# Enable metadata feature before calling app.run() or app.run_async()
app.is_metadata_enabled = True
app.run()
```

9.5.2 Understanding Metadata Flow

Each operator in the workflow has an associated `MetadataDictionary` object. At the start of each operator's `compute()` call this metadata dictionary will be empty (i.e. metadata does not persist from previous compute calls). When any call to `receive()` data is made, any metadata also found in the input message will be merged into the operator's local metadata dictionary. The operator's compute method can then read, append to or remove metadata as explained in the next section. Whenever the operator emits data via a call to `emit()` the current status of the operator's metadata dictionary will be transmitted on that port alongside the data passed via the first argument to the emit call. Any downstream operators will then receive this metadata via their input ports.

9.5.3 Working With Metadata from Operator::compute

Within the operator's `compute()` method, the `metadata()` method can be called to get a shared pointer to the `MetadataDictionary` of the operator. The metadata dictionary provides a similar API to a `std::unordered_map` (C++) or `dict` (Python) where the keys are strings (`std::string` for C++) and the values can store any object type (via a C++ `MetadataObject` holding a `std::any`).

C++

Templated `get()` and `set()` method are provided as demonstrated below to allow directly setting values of a given type without having to explicitly work with the internal `MetadataObject` type.

```
// Receiving from a port updates operator metadata with any metadata found on the port
auto input_tensors = op_input.receive<TensorMap>("in");

// Get a reference to the shared metadata dictionary
auto& meta = metadata();

// Retrieve existing values.
// Use get<Type> to automatically cast the `std::any` contained within the
↳ `holoscan::Message`
auto name = meta->get<std::string>("patient_name");
auto age = meta->get<int>("age");

// Get also provides a two-argument version where a default value to be assigned is
↳ given by
// the second argument. The type of the default value should match the expected type of
↳ the value.
auto flag = meta->get("flag", false);

// Add a new value (if a key already exists, the value will be updated according to the
// operator's metadata_policy).
std::vector<float> spacing{1.0, 1.0, 3.0};
meta->set("pixel_spacing"s, spacing);

// Remove a value
meta->erase("patient_name")

// ... Some processing to produce output `data` could go here ...

// Current state of `meta` will automatically be emitted along with `data` in the call
↳ below
op_output.emit(data, "output1");

// Can clear all items
meta->clear();

// Any emit call after this point would not transmit a metadata object
op_output.emit(data, "output2");
```

See the `MetadataDictionary` API docs for all available methods.

Python

A Pythonic interface is provided for the `MetadataObject` type.

```
# Receiving from a port updates operator metadata with any metadata found on the port
input_tensors = op_input.receive("in")

# self.metadata can be used to access the shared MetadataDictionary
# for example we can check if a key exists
has_key = "my_key" in self.metadata

# get the number of keys
num_keys = len(self.metadata)

# get a list of the keys
print(f"metadata keys = {self.metadata.keys()}")

# iterate over the values in the dictionary using the `items()` method
for key, value in self.metadata.items():
    # process item
    pass

# print a Python dict of the keys/values
print(self.metadata)

# Retrieve existing values. If the underlying value is a C++ class, a conversion to an
↪ equivalent Python object will be made (e.g. `std::vector<std::string>` to `List[str]`).
name = self.metadata["patient_name"]
age = self.metadata["age"]

# It is also supported to use the get method along with an optional default value to use
# if the key is not present.
flag = self.metadata.get("flag", False)

# print the current metadata policy
print(f"metadata policy = {self.metadata_policy}")

# Add a new value (if a key already exists, the value will be updated according to the
# operator's metadata_policy). If the value is set via the indexing operator as below,
# the Python object itself is stored as the value.
spacing = (1.0, 1.0, 3.0)
self.metadata["pixel_spacing"] = spacing

# In some cases, if sending metadata to downstream C++-based operators, it may be desired
# to instead store the metadata value as an equivalent C++ type. In that case, it is
# necessary to instead set the value using the `set` method with `cast_to_cpp=True`.
# Automatic casting is supported for bool, str, and various numeric and iterator or
# sequence types.

# The following would result in the spacing `Tuple[float]` being stored as a
# C++ `std::vector<double>`. Here we show use of the `pop` method to remove a previous
↪ value
```

(continues on next page)

(continued from previous page)

```

# if present.
self.metadata.pop("pixel_spacing", None)
self.metadata.set("pixel_spacing", spacing, cast_to_cpp=True)

# To store floating point elements at a different than the default (double) precision or
# integers at a different precision than int64_t, use the dtype argument and pass a
# numpy.dtype argument corresponding to the desired C++ type. For example, the following
# would instead store `spacing` as a std::vector<float> instead. In this case we show
# use of Python's `del` instead of the pop method to remove an existing item.
del self.metadata["pixel_spacing"]
self.metadata.set("pixel_spacing", spacing, dtype=np.float32, cast_to_cpp=True)

# Remove a value
del self["patient name"]

# ... Some processing to produce output `data` could go here ...

# Current state of `meta` will automatically be emitted along with `data` in the call_
↪ below
op_output.emit(data, "output1")

# Can clear all items
self.metadata.clear()

# Any emit call after this point would not transmit a metadata object
op_output.emit(data, "output2")

```

See the MetadataDictionary API docs for all available methods.

The above code illustrated various ways of working with and updating an operator's metadata.

Note: Pay particular attention to the details of how metadata is set. When working with pure Python applications it is best to just use `self.metadata[key] = value` or `self.metadata.set(key, value)` to pass Python objects as the value. This will just use a shared object and not result in copies to/from corresponding C++ types. However, when interacting with other operators that wrap a C++ implementation, their compute method would expect C++ metadata. In that case, the set method with `cast_to_cpp=True` is needed to cast to the expected C++ type. This was shown in some of the “pixel_spacing” set calls in the example above. For convenience, the value passed to the set method can also be a NumPy array, but note that in this case, a copy into a new C++ `std::vector` is performed. The dtype of the array will be respected when creating the vector. In general, the types that can currently be cast to C++ are scalar numeric values, strings and Python Iterators or Sequences of these (the sequence will be converted to a 1D or 2D C++ `std::vector` so the items in the Python sequence cannot be of mixed type).

Metadata Update Policies

C++

The operator class also has a `metadata_policy()` method that can be used to set a `MetadataPolicy` to use when handling duplicate metadata keys across multiple input ports of the operator. The available options are:

- “update” (`MetadataPolicy::kUpdate`): replace any existing key from a prior receive call with one present in a subsequent receive call.
- “reject” (`MetadataPolicy::kReject`): Reject the new key/value pair when a key already exists due to a prior receive call.
- “raise” (`MetadataPolicy::kRaise`): Throw a `std::runtime_error` if a duplicate key is encountered. This is the default policy.

The metadata policy would typically be set during `compose()` as in the following example:

```
// Example for setting metadata policy from Application::compose()
my_op = make_operator<MyOperator>("my_op");
my_op->metadata_policy(holoscan::MetadataPolicy::kRaise);
```

Python

The operator class also has a `metadata_policy()` property that can be used to set a `MetadataPolicy` to use when handling duplicate metadata keys across multiple input ports of the operator. The available options are:

- “update” (`MetadataPolicy.UPDATE`): replace any existing key from a prior receive call with one present in a subsequent receive call. This is the default policy.
- “reject” (`MetadataPolicy.REJECT`): Reject the new key/value pair when a key already exists due to a prior receive call.
- “raise” (`MetadataPolicy.RAISE`): Throw an exception if a duplicate key is encountered.

The metadata policy would typically be set during `compose()` as in the following example:

```
# Example for setting metadata policy from Application.compose()
my_op = MyOperator(self, name="my_op")
my_op.metadata_policy = holoscan.MetadataPolicy.RAISE
```

The policy only applies to the operator on which it was set.

9.5.4 Use of Metadata in Distributed Applications

Sending metadata between two fragments of a distributed application is supported, but there are a couple of aspects to be aware of.

1. Sending metadata over the network requires serialization and deserialization of the metadata keys and values. The value types supported for this are the same as for data emitted over output ports (see the table in the section on *object serialization*). The only exception is that `Tensor` and `TensorMap` values cannot be sent as metadata values between fragments (this restriction also applies to tensor-like Python objects). Any *custom codecs* registered for the SDK will automatically also be available for serialization of metadata values.

2. There is a practical size limit of several kilobytes in the amount of metadata that can be transmitted between fragments. This is because metadata is currently sent along with other entity header information in the UCX header, which has fixed size limit (the metadata is stored along with other header information within the size limit defined by the `HOLOSCAN_UCX_SERIALIZATION_BUFFER_SIZE` *environment variable*).

The above restrictions only apply to metadata sent **between** fragments. Within a fragment there is no size limit on metadata (aside from system memory limits) and no serialization or deserialization step is needed.

9.5.5 Current limitations

1. The current metadata API is only fully supported for native holoscan Operators and is not currently supported by operators that wrap a GXF codelet (i.e. inheriting from `GXFOperator` or created via `GXFCodeletOp`). Aside from `GXFCodeletOp`, the built-in operators provided under the `holoscan::ops` namespace are all native operators, so the feature will work with these. Currently none of these built-in operators add their own metadata, but any metadata received on input ports will automatically be passed on to their output ports (as long as `app->is_metadata_enabled(true)` was set to enable the metadata feature).

CREATING A DISTRIBUTED APPLICATION

Distributed applications refer to those where the workflow is divided into multiple fragments that may be run on separate nodes. For example, data might be collected via a sensor at the edge, sent to a separate workstation for processing, and then the processed data could be sent back to the edge node for visualization. Each node would run a single fragment consisting of a computation graph built up of operators. Thus one fragment is the equivalent of a non-distributed application. In the distributed context, the application initializes the different fragments and then defines the connections between them to build up the full distributed application workflow.

In this section we'll describe:

- How to *define a distributed application*.
- How to *build and run a distributed application*.

10.1 Defining a Distributed Application Class

Tip: Defining distributed applications is also illustrated in the [video_replayer_distributed](#) and [ping_distributed](#) examples. The [ping_distributed](#) examples also illustrate how to update C++ or Python applications to parse user-defined arguments in a way that works without disrupting support for distributed application command line arguments (e.g., `--driver`, `--worker`).

Defining a single fragment (C++/Python) involves adding operators using `make_operator()` (C++) or the operator constructor (Python), and defining the connections between them using the `add_flow()` method (C++/Python) in the `compose()` method. Thus, defining a fragment is just like defining a non-distributed application except that the class should inherit from `fragment` instead of `application`.

The application will then be defined by initializing fragments within the application's `compose()` method. The `add_flow()` method (C++/Python) can be used to define the connections across fragments.

C++

- We define the `Fragment1` and `Fragment2` classes that inherit from the `Fragment` base class.
- We define the `App` class that inherits from the `Application` base class.
- The `App` class initializes any fragments used and defines the connections between them. Here we have used dummy port and operator names in the example `add_flow` call connecting the fragments since no specific operators are shown in this example.
- We create an instance of the `App` class in `main()` using the `make_application()` function.

- The `run()` method starts the application which will execute its `compose()` method where the custom workflow will be defined.

```
#include <holoscan/holoscan.hpp>

class Fragment1 : public holoscan::Fragment {
public:
    void compose() override {
        // Define Operators and workflow for Fragment1
        // ...
    }
};

class Fragment2 : public holoscan::Fragment {
public:
    void compose() override {
        // Define Operators and workflow for Fragment2
        // ...
    }
};

class App : public holoscan::Application {
public:
    void compose() override {
        using namespace holoscan;

        auto fragment1 = make_fragment<Fragment1>("fragment1");
        auto fragment2 = make_fragment<Fragment2>("fragment2");

        // Define the workflow: replayer -> holoviz
        add_flow(fragment1, fragment2, {{"fragment1_operator_name.output_port_name",
                                         "fragment2_operator_name.input_port_name"}});
    }
};

int main() {
    auto app = holoscan::make_application<App>();
    app->run();
    return 0;
}
```

Python

- We define the `Fragment1` and `Fragment2` classes that inherit from the `Fragment` base class.
- We define the `App` class that inherits from the `Application` base class.
- The `App` class initializes any fragments used and defines the connections between them. Here we have used dummy port and operator names in the example `add_flow` call connecting the fragments since no specific operators are shown in this example.
- We create an instance of the `App` class in `__main__`.

- The `run()` method starts the application which will execute its `compose()` method where the custom workflow will be defined.

```

from holoscan.core import Application, Fragment

class Fragment1(Fragment):

    def compose(self):
        # Define Operators and workflow
        # ...

class Fragment2(Fragment):

    def compose(self):
        # Define Operators and workflow
        # ...

class App(Application):

    def compose(self):
        fragment1 = Fragment1(self, name="fragment1")
        fragment2 = Fragment2(self, name="fragment2")

        self.add_flow(fragment1, fragment2, {("fragment1_operator_name.output_port_name",
                                              "fragment2_operator_name.input_port_name")})
        ↪)

def main():
    app = App()
    app.run()

if __name__ == "__main__":
    main()

```

10.1.1 Serialization of Custom Data Types for Distributed Applications

Transmission of data between fragments of a multi-fragment application is done via the [Unified Communications X \(UCX\)](#) library. In order to transmit data, it must be serialized into a binary form suitable for transmission over a network. For Tensors (C++/Python), strings and various scalar and vector numeric types, serialization is already built in. For more details on concrete examples of how to extend the data serialization support to additional user-defined classes, see the separate page on [serialization](#).

10.2 Building and running a Distributed Application

C++

Building a distributed application works in the same way as for a non-distributed one. See *Building and running your Application*

Python

Python applications do not require building. See *Building and running your Application*.

Running an application in a distributed setting requires launching the application binary on all nodes involved in the distributed application. A single node must be selected to act as the application driver. This is achieved by using the `--driver` command-line option. Worker nodes are initiated by launching the application with the `--worker` command-line option. It's possible for the driver node to also serve as a worker if both options are specified.

The address of the driver node must be specified for each process (both the driver and worker(s)) to identify the appropriate network interface for communication. This can be done via the `--address` command-line option, which takes a value in the form of `[<IPv4/IPv6 address or hostname>][:<port>]` (e.g., `--address 192.168.50.68:10000`):

- The driver's IP (or hostname) **MUST** be set for each process (driver and worker(s)) when running distributed applications on multiple nodes (default: `0.0.0.0`). It can be set without the port (e.g., `--address 192.168.50.68`).
- In a single-node application, the driver's IP (or hostname) can be omitted, allowing any network interface (`0.0.0.0`) to be selected by the UCX library.
- The port is always optional (default: `8765`). It can be set without the IP (e.g., `--address :10000`).

The worker node's address can be defined using the `--worker-address` command-line option (`[<IPv4/IPv6 address or hostname>][:<port>]`). If it's not specified, the application worker will default to the host address (`0.0.0.0`) with a randomly chosen port number between `10000` and `32767` that is not currently in use. This argument automatically sets the `HOLOSCAN_UCX_SOURCE_ADDRESS` environment variable if the worker address is a local IP address. Refer to *this section* for details.

The `--fragments` command-line option is used in combination with `--worker` to specify a comma-separated list of fragment names to be run by a worker. If not specified, the application driver will assign a single fragment to the worker. To indicate that a worker should run all fragments, you can specify `--fragments all`.

The `--config` command-line option can be used to designate a path to a configuration file to be used by the application.

Below is an example launching a three fragment application named `my_app` on two separate nodes:

- The application driver is launched at `192.168.50.68:10000` on the first node (A), with a worker running two fragments, "fragment1" and "fragment3."
- On a separate node (B), the application launches a worker for "fragment2," which will connect to the driver at the address above.

C++

```
# Node A
my_app --driver --worker --address 192.168.50.68:10000 --fragments fragment1,fragment3
# Node B
my_app --worker --address 192.168.50.68:10000 --fragments fragment2
```

Python

```
# Node A
python3 my_app.py --driver --worker --address 192.168.50.68:10000 --fragments fragment1,
↪ fragment3
# Node B
python3 my_app.py --worker --address 192.168.50.68:10000 --fragments fragment2
```

Note:

UCX Network Interface Selection

UCX is used in the Holoscan SDK for communication across fragments in distributed applications. It is designed to select the best network device based on performance characteristics (bandwidth, latency, NUMA locality, etc.). In some scenarios (under investigation), UCX cannot find the correct network interface to use, and the application fails to run. In this case, you can manually specify the network interface to use by setting the UCX_NET_DEVICES environment variable.

For example, if the user wants to use the network interface `eth0`, you can set the environment variable as follows, before running the application:

```
export UCX_NET_DEVICES=eth0
```

Or, if you are running a packaged distributed application with the *Holoscan CLI*, use the `--nic eth0` option to manually specify the network interface to use.

The available network interface names can be found by running the following command:

```
ucx_info -d | grep Device: | awk '{print $3}' | sort | uniq
# or
ip -o -4 addr show | awk '{print $2, $4}' # to show interface name and IP
```

Warning:

Known limitations

The following are known limitations of the distributed application support in the SDK, which will be addressed in future updates:

1. A connection error message is displayed even when the distributed application is running correctly.

The message `Connection dropped with status -25 (Connection reset by remote peer)` appears in the console even when the application is functioning properly. This is a known issue and will be addressed in future updates, ensuring that this message will only be displayed in the event of an actual connection error. It currently is printed once some fragments complete their work and start shutdown. Any connections from those fragments to ones that remain open are disconnected at that point, resulting in the logged message.

2. GPU tensors can only currently be sent/received by UCX from a single device on a given node.

By default, device ID 0 is used by the UCX extensions to send/receive data between fragments. To override this default, the user can set environment variable `HOLOSCAN_UCX_DEVICE_ID`.

3. “Address already in use” errors in distributed applications due to the health check service.

In scenarios where distributed applications have both the driver and workers running on the same host, either within a Docker container or directly on the host, there’s a possibility of encountering “Address already in use” errors. A potential solution is to assign a different port number to the `HOLOSCAN_HEALTH_CHECK_PORT` environment variable (default: 8777), for example, by using `export HOLOSCAN_HEALTH_CHECK_PORT=8780`.

Note:**GXF UCX Extension**

Holoscan’s distributed application feature makes use of the [GXF UCX Extension](#). Its documentation may provide useful additional context into how data is transmitted between fragments.

Tip: Given a CMake project, a pre-built executable, or a Python application, you can also use the *Holoscan CLI* to *package and run your Holoscan application* in a OCI-compliant container image.

10.2.1 Environment Variables for Distributed Applications

Holoscan SDK environment variables

You can set environment variables to modify the default actions of services and the scheduler when executing a distributed application.

- **HOLOSCAN_ENABLE_HEALTH_CHECK** : determines if the health check service should be active, even without specifying `--driver` or `--worker` in the CLI. By default, initiating the AppDriver (`--driver`) or AppWorker (`--worker`) service automatically triggers the [GRPC Health Checking Service](#) so `grpc-health-probe` can monitor liveness/readiness. Interprets values like “true”, “1”, or “on” (case-insensitive) as true (to enable the health check). It defaults to “false” if left unspecified.
- **HOLOSCAN_HEALTH_CHECK_PORT** : designates the port number on which the Health Checking Service is launched. It must be an integer value representing a valid port number. If unspecified, it defaults to 8777.

- **HOLOSCAN_DISTRIBUTED_APP_SCHEDULER** : controls which scheduler is used for distributed applications. It can be set to either `greedy`, `multi_thread` or `event_based`. `multithread` is also allowed as a synonym for `multi_thread` for backwards compatibility. If unspecified, the default scheduler is `multi_thread`.
- **HOLOSCAN_STOP_ON_DEADLOCK** : can be used in combination with **HOLOSCAN_DISTRIBUTED_APP_SCHEDULER** to control whether or not the application will automatically stop on deadlock. Values of “True”, “1” or “ON” will be interpreted as true (enable stop on deadlock). It is “true” if unspecified. This environment variable is only used when **HOLOSCAN_DISTRIBUTED_APP_SCHEDULER** is explicitly set.
- **HOLOSCAN_STOP_ON_DEADLOCK_TIMEOUT** : controls the delay (in ms) without activity required before an application is considered to be in deadlock. It must be an integer value (units are ms).
- **HOLOSCAN_MAX_DURATION_MS** : sets the application to automatically terminate after the requested maximum duration (in ms) has elapsed. It must be an integer value (units are ms). This environment variable is only used when **HOLOSCAN_DISTRIBUTED_APP_SCHEDULER** is explicitly set.
- **HOLOSCAN_CHECK_RECESSION_PERIOD_MS** : controls how long (in ms) the scheduler waits before re-checking the status of operators in an application. It must be a floating point value (units are ms). This environment variable is only used when **HOLOSCAN_DISTRIBUTED_APP_SCHEDULER** is explicitly set.
- **HOLOSCAN_UCX_SERIALIZATION_BUFFER_SIZE** : can be used to override the default 7 kB serialization buffer size. This should typically not be needed as tensor types store only a small header in this buffer to avoid explicitly making a copy of their data. However, other data types do get directly copied to the serialization buffer and in some cases it may be necessary to increase it.
- **HOLOSCAN_UCX_ASYNCHRONOUS** : If set to true, asynchronous transmit of UCX messages between fragments is enabled (this is the default). Setting this to False, forces synchronous transmission. Synchronous mode makes it easier to use an allocator like `BlockMemoryPool` as additional tensors would not be queued before the prior one was sent.
- **HOLOSCAN_UCX_DEVICE_ID** : The GPU ID of the device that will be used by UCX transmitter/receivers in distributed applications. If unspecified, it defaults to 0. A list of discrete GPUs available in a system can be obtained via `nvidia-smi -L`. GPU data sent between fragments of a distributed application must be on this device.
- **HOLOSCAN_UCX_PORTS** : This defines the preferred port numbers for the SDK when specific ports for UCX communication need to be predetermined, such as in a Kubernetes environment. If the distributed application requires three ports (UCX receivers) and the environment variable is unset, the SDK chooses three unused ports sequentially from the range 10000~32767. Specifying a value, for example, **HOLOSCAN_UCX_PORTS=10000**, results in the selection of ports 10000, 10001, and 10002. Multiple starting values can be comma-separated. The system increments from the last provided port if more ports are needed. Any unused specified ports are ignored.
- **HOLOSCAN_UCX_SOURCE_ADDRESS** : This environment variable specifies the local IP address (source) for the UCX connection. This variable is especially beneficial when a node has multiple network interfaces, enabling the user to determine which one should be utilized for establishing a UCX client (`UcxTransmitter`). If it is not explicitly specified, the default address is set to `0.0.0.0`, representing any available interface.

UCX-specific environment variables

Transmission of data between fragments of a multi-fragment application is done via the [Unified Communications X \(UCX\)](#) library, a point-to-point communication framework designed to utilize the best available hardware resources (shared memory, TCP, GPUDirect RDMA, etc). UCX has many parameters that can be controlled via environment variables. A few that are particularly relevant to Holoscan SDK distributed applications are listed below:

- The `UCX_TLS` environment variable can be used to control which transport layers are enabled. By default, `UCX_TLS=all` and UCX will attempt to choose the optimal transport layer automatically.
- The `UCX_NET_DEVICES` environment variable is by default set to `all`, meaning that UCX may choose to use any available network interface controller (NIC). In some cases it may be necessary to restrict UCX to a specific device or set of devices, which can be done by setting `UCX_NET_DEVICES` to a comma separated list of the device names (i.e., as obtained by Linux command `ifconfig -a` or `ip link show`).
- Setting `UCX_TCP_CM_REUSEADDR=y` is recommended to enable ports to be reused without having to wait the full socket `TIME_WAIT` period after a socket is closed.
- The `UCX_LOG_LEVEL` environment variable can be used to control the logging level of UCX. The default is setting is `WARN`, but changing to a lower level such as `INFO` will provide more verbose output on which transports and devices are being used.
- By default, Holoscan SDK will automatically set `UCX_PROTO_ENABLE=y` upon application launch to enable the newer “v2” UCX protocols. If for some reason, the older v1 protocols are needed, one can set `UCX_PROTO_ENABLE=n` in the environment to override this setting. When the v2 protocols are enabled, one can optionally set `UCX_PROTO_INFO=y` to enable detailed logging of what protocols are being used at runtime.
- By default, Holoscan SDK will automatically set `UCX_MEMTYPE_CACHE=n` upon application launch to disable the UCX memory type cache (See [UCX documentation](#) for more information. It can cause about [0.2 microseconds of pointer type checking overhead with the `cudaPointerGetAttributes\(\)` CUDA API](#)). If for some reason the memory type cache is needed, one can set `UCX_MEMTYPE_CACHE=y` in the environment to override this setting.
- By default, the Holoscan SDK will automatically set `UCX_CM_USE_ALL_DEVICES=n` at application startup to disable consideration of all devices for data transfer. If for some reason the opposite behavior is desired, one can set `UCX_CM_USE_ALL_DEVICES=y` in the environment to override this setting. Setting `UCX_CM_USE_ALL_DEVICES=n` can be used to workaround an issue where UCX sometimes defaults to a device that might not be the most suitable for data transfer based on the host’s available devices. On a host with address 10.111.66.60, UCX, for instance, might opt for the `br-80572179a31d` (192.168.49.1) device due to its superior bandwidth as compared to `eno2` (10.111.66.60). With `UCX_CM_USE_ALL_DEVICES=n`, UCX will ensure consistency by using the same device for data transfer that was initially used to establish the connection. This ensures more predictable behavior and can avoid potential issues stemming from device mismatches during the data transfer process.
- Setting `UCX_TCP_PORT_RANGE=<start>-<end>` can be used to define a specific range of ports that UCX should utilize for data transfer. This is particularly useful in environments where ports need to be predetermined, such as in a Kubernetes setup. In such contexts, Pods often have ports that need to be exposed, and these ports must be specified ahead of time. Moreover, in scenarios where firewall configurations are stringent and only allow specified ports, having a predetermined range ensures that the UCX communication does not get blocked. This complements the `HOLOSCAN_UCX_SOURCE_ADDRESS`, which specifies the local IP address for the UCX connection, by giving further control over which ports on that specified address should be used. By setting a port range, users can ensure that UCX operates within the boundaries of the network and security policies of their infrastructure.

Tip: A list of all available UCX environment variables and a brief description of each can be obtained by running `ucx_info -f` from the Holoscan SDK container. Holoscan SDK uses UCX’s active message (AM) protocols, so environment variables related to other protocols such as `tag-mat`.

10.3 Serialization

Distributed applications must serialize any objects that are to be sent between the fragments of a multi-fragment application. Serialization involves binary serialization to a buffer that will be sent from one fragment to another via the Unified Communications X (UCX) library. For tensor types (e.g., `holoscan::Tensor`), no actual copy is made, but instead transmission is done directly from the original tensor's data and only a small amount of header information is copied to the serialization buffer.

A table of the types that have codecs pre-registered so that they can be serialized between fragments using Holoscan SDK is given below.

Type Class	Specific Types
integers	<code>int8_t</code> , <code>int16_t</code> , <code>int32_t</code> , <code>int64_t</code> , <code>uint8_t</code> , <code>uint16_t</code> , <code>uint32_t</code> , <code>uint64_t</code>
floating point	<code>float</code> , <code>double</code> , <code>complex <float></code> , <code>complex<double></code>
boolean	<code>bool</code>
strings	<code>std::string</code>
<code>std::vector<T></code>	T is <code>std::string</code> or any of the boolean, integer or floating point types above
<code>std::vector<std::vector<T>></code>	T is <code>std::string</code> or any of the boolean, integer or floating point types above
<code>std::vector<HolovizOp::InputSpec></code>	a vector of <code>InputSpec</code> objects that are specific to <code>HolovizOp</code>
<code>std::shared_ptr<T></code>	T is any of the scalar, vector or <code>std::string</code> types above
tensor types	<code>holoscan::Tensor</code> , <code>nvidia::gxf::Tensor</code> , <code>nvidia::gxf::VideoBuffer</code> , <code>nvidia::gxf::AudioBuffer</code>
GXF-specific types	<code>nvidia::gxf::TimeStamp</code> , <code>nvidia::gxf::EndOfStream</code>

Warning: If an operator transmitting both CPU and GPU tensors is to be used in distributed applications, the same output port cannot mix both GPU and CPU tensors. CPU and GPU tensor outputs should be placed on separate output ports. This is a limitation of the underlying UCX library being used for zero-copy tensor serialization between operators.

As a concrete example, assume an operator, `MyOperator` with a single output port named “out” defined in its setup method. If the output port is only ever going to connect to other operators within a fragment, but never across fragments then it is okay to have a `TensorMap` with a mixture of host and device arrays on that single port.

C++

```
void MyOperator::setup(OperatorSpec& spec) {
    spec.output<holoscan::TensorMap>("out");
}

void MyOperator::compute(OperatorSpec& spec) {

    // omitted: some computation resulting in multiple holoscan::Tensors
    // (two on CPU ("cpu_coords_tensor" and "cpu_metric_tensor") and one on device (
    ↪ "gpu_tensor").

    TensorMap out_message;

    // insert all tensors in one TensorMap (mixing CPU and GPU tensors is okay when
    ↪ ports only connect within a Fragment)
    out_message.insert({"coordinates", cpu_coords_tensor});
    out_message.insert({"metrics", cpu_metric_tensor});
    out_message.insert({"mask", gpu_tensor});

    op_output.emit(out_message, "out");
}
```

Python

```

class MyOperator:

    def setup(self, spec: OperatorSpec):
        spec.output("out")

    def compute(self, op_input, op_output, context):
        # Omitted: assume some computation resulting in three holoscan::Tensor or
        ↪ tensor-like
        # objects. Two on CPU ("cpu_coords_tensor" and "cpu_metric_tensor") and one
        ↪ on device
        # ("gpu_tensor").

        # mixing CPU and GPU tensors in a single dict is okay only for within-
        ↪ Fragment connections
        op_output.emit(
            dict(
                coordinates=cpu_coords_tensor,
                metrics=cpu_metrics_tensor,
                mask=gpu_tensor,
            ),
            "out"
        )

```

However, this mixing of CPU and GPU arrays on a single port will not work for distributed apps and instead separate ports should be used if it is necessary for an operator to communicate across fragments.

C++

```

void MyOperator::setup(OperatorSpec& spec) {
    spec.output<holoscan::TensorMap>("out_host");
    spec.output<holoscan::TensorMap>("out_device");
}

void MyOperator::compute(OperatorSpec& spec) {
    // some computation resulting in a pair of holoscan::Tensor, one on CPU ("cpu_tensor
    ↪ ") and one on device ("gpu_tensor").
    TensorMap out_message_host;
    TensorMap out_message_device;

    // put all CPU tensors on one port
    out_message_host.insert({"coordinates", cpu_coordinates_tensor});
    out_message_host.insert({"metrics", cpu_metrics_tensor});
    op_output.emit(out_message_host, "out_host");

    // put all GPU tensors on another
    out_message_device.insert({"mask", gpu_tensor});
    op_output.emit(out_message_device, "out_device");
}

```

Python

```

class MyOperator:

    def setup(self, spec: OperatorSpec):
        spec.output("out_host")
        spec.output("out_device")

    def compute(self, op_input, op_output, context):
        # Omitted: assume some computation resulting in three holoscan::Tensor or
        ↪ tensor-like
        # objects. Two on CPU ("cpu_coords_tensor" and "cpu_metric_tensor") and one
        ↪ on device
        # ("gpu_tensor").

        # split CPU and GPU tensors across ports for compatibility with inter-
        ↪ fragment communication
        op_output.emit(
            dict(coordinates=cpu_coords_tensor, metrics=cpu_metrics_tensor),
            "out_host"
        )
        op_output.emit(dict(mask=gpu_tensor), "out_device")

```

10.3.1 Python

For the Python API, any array-like object supporting the `DLPack` interface, `__array_interface__` or `__cuda_array_interface__` will be transmitted using Tensor serialization. This is done to avoid data copies for performance reasons. Objects of type `list[holoscan.HolovizOp.InputSpec]` will be sent using the underlying C++ serializer for `std::vector<HolovizOp::InputSpec>`. All other Python objects will be serialized to/from a `std::string` using the `cloudpickle` library.

Warning: A restriction imposed by the use of `cloudpickle` is that all fragments in a distributed application must be running the same Python version.

Warning: Distributed applications behave differently than single fragment applications when `op_output.emit()` is called to emit a tensor-like Python object. Specifically, for array-like objects such as a PyTorch tensor, the same Python object will **not** be received by any call to `op_input.receive()` in a downstream Python operator (even if the upstream and downstream operators are part of the same fragment). An object of type `holoscan.Tensor` will be received as a `holoscan.Tensor`. Any other array-like objects with data stored on device (GPU) will be received as a CuPy tensor. Similarly, any array-like object with data stored on the host (CPU) will be received as a NumPy array. The user must convert back to the original array-like type if needed (typically possible in a zero-copy fashion via `DLPack` or array interfaces).

10.3.2 C++

For any additional C++ classes that need to be serialized for transmission between fragments in a distributed application, the user must create their own codec and register it with the Holoscan SDK framework. As a concrete example, suppose that we had the following simple `Coordinate` class that we wish to send between fragments.

```
struct Coordinate {
    float x;
    float y;
    float z;
};
```

To create a codec capable of serializing and deserializing this type, one should define a `holoscan::codec` class for it as shown below.

```
#include "holoscan/core/codec_registry.hpp"
#include "holoscan/core/errors.hpp"
#include "holoscan/core/expected.hpp"

namespace holoscan {

template <
struct codec<Coordinate> {
    static expected<size_t, RuntimeError> serialize(const Coordinate& value, Endpoint*
    endpoint) {
        return serialize_trivial_type<Coordinate>(value, endpoint);
    }
    static expected<Coordinate, RuntimeError> deserialize(Endpoint* endpoint) {
        return deserialize_trivial_type<Coordinate>(endpoint);
    }
};

} // namespace holoscan
```

In this example, the first argument to `serialize` is a `const` reference to the type to be serialized and the return value is an `expected` containing the number of bytes that were serialized. The `deserialize` method returns an `expected` containing the deserialized object. The `Endpoint` class is a base class representing the serialization endpoint (For distributed applications, the actual endpoint class used is `UcxSerializationBuffer`).

The helper functions `serialize_trivial_type` (`deserialize_trivial_type`) can be used to serialize (deserialize) any plain-old-data (POD) type. Specifically, POD types can be serialized by just copying `sizeof(Type)` bytes to/from the endpoint. The `read_trivial_type()` and `~holoscan::Endpoint::write_trivial_type` methods could be used directly instead.

```
template <
struct codec<Coordinate> {
    static expected<size_t, RuntimeError> serialize(const Coordinate& value, Endpoint*
    endpoint) {
        return endpoint->write_trivial_type(&value);
    }
    static expected<Coordinate, RuntimeError> deserialize(Endpoint* endpoint) {
        Coordinate encoded;
        auto maybe_value = endpoint->read_trivial_type(&encoded);
```

(continues on next page)

(continued from previous page)

```

    if (!maybe_value) { return forward_error(maybe_value); }
    return encoded;
}
};

```

In practice, one would not actually need to define `codec<Coordinate>` at all since `Coordinate` is a trivially serializable type and the existing codec treats any types for which there is not a template specialization as a trivially serializable type. It is, however, still necessary to register the codec type with the `CodecRegistry` as described below.

For non-trivial types, one will likely also need to use the `read()` and `write()` methods to implement the codec. Example use of these for the built-in codecs can be found in [holoscan/core/codecs.hpp](#).

Once such a codec has been defined, the remaining step is to register it with the static `CodecRegistry` class. This will make the UCX-based classes used by distributed applications aware of the existence of a codec for serialization of this object type. If the type is specific to a particular operator, then one can register it via the `register_codec()` class.

```

#include "holoscan/core/codec_registry.hpp"

namespace holoscan::ops {

void MyCoordinateOperator::initialize() {
    register_codec<Coordinate>("Coordinate");

    // ...

    // parent class initialize() call must be after the argument additions above
    Operator::initialize();
}

} // namespace holoscan::ops

```

Here, the argument provided to `register_codec` is the name the registry will use for the codec. This name will be serialized in the message header so that the deserializer knows which deserialization function to use on the received data. In this example, we chose a name that matches the class name, but that is not a requirement. If the name matches one that is already present in the `CodecRegistry` class, then any existing codec under that name will be replaced by the newly registered one.

It is also possible to directly register the type outside of the context of `initialize()` by directly retrieving the static instance of the codec registry as follows.

```

namespace holoscan {

CodecRegistry::get_instance().add_codec<Coordinate>("Coordinate");

} // namespace holoscan

```

Tip: CLI arguments (such as `--driver`, `--worker`, `--fragments`) are parsed by the `Application` (C++/Python) class and the remaining arguments are available as `app.argv` (C++/Python).

C++

A concrete example of using `app->argv()` in the `ping_distributed.cpp` example is covered in the section on *user-defined command line arguments*.

If you want to get access to the arguments before creating the C++ instance, you can access them through `holoscan::Application().argv()`.

The following example shows how to access the arguments in your application.

```
#include <holoscan/holoscan.hpp>

class MyPingApp : public holoscan::Application {
// ...
};

int main(int argc, char** argv) {
    auto my_argv =
        holoscan::Application({"myapp", "--driver", "my_arg1", "--address=10.0.0.1"}).
        ↪argv();
    HOLOSCAN_LOG_INFO(" my_argv: {}", fmt::join(my_argv, " "));

    HOLOSCAN_LOG_INFO(
        "    argv: {} (argc: {})",
        fmt::join(std::vector<std::string>(argv, argv + argc), " "),
        argc);

    auto app_argv = holoscan::Application().argv(); // do not use reference ('auto&') here.
    ↪(lifetime issue)
    HOLOSCAN_LOG_INFO("app_argv: {} (size: {})", fmt::join(app_argv, " "), app_argv.
    ↪size());

    auto app = holoscan::make_application<MyPingApp>();
    HOLOSCAN_LOG_INFO("app->argv() == app_argv: {}", app->argv() == app_argv);

    app->run();
    return 0;
}

// $ ./myapp --driver --input image.dat --address 10.0.0.20

// my_argv: myapp my_arg1
// argv: ./myapp --driver --input image.dat --address 10.0.0.20 (argc: 6)
// app_argv: ./myapp --input image.dat (size: 3)
// app->argv() == app_argv: true
```

Please see other examples in the [Application unit tests](#) in the Holoscan SDK repository.

Python

A concrete example of usage of `app.argv` in the `ping_distributed.py` example is covered in the section on *user-defined command line arguments*.

If you want to get access to the arguments before creating the Python instance, you can access them through `Application().argv`.

The following example shows how to access the arguments in your application.

```
import argparse
import sys
from holoscan.core import Application

class MyApp(Application):
    def compose(self):
        pass

def main():
    app = MyApp() # or alternatively, MyApp([sys.executable, *sys.argv])
    app.run()

if __name__ == "__main__":

    print("sys.argv:", sys.argv)
    print("Application().argv:", app.argv)

    parser = argparse.ArgumentParser()
    parser.add_argument("--input")
    args = parser.parse_args(app.argv[1:])
    print("args:", args)

    main()

# $ python cli_test.py --address 10.0.0.20 --input image.dat
# sys.argv: ['cli_test.py', '--address', '10.0.0.20', '--input', 'image.dat']
# Application().argv: ['cli_test.py', '--input', 'image.dat']
# args: Namespace(input='a')
```

```
>>> from holoscan.core import Application
>>> import sys
>>> Application().argv == sys.argv
True
>>> Application([]).argv == sys.argv
True
>>> Application([sys.executable, *sys.argv]).argv == sys.argv
True
>>> Application(["python3", "myapp.py", "--driver", "my_arg1", "--address=10.0.0.1"]).
↪ argv
['myapp.py', 'my_arg1']
```

Please see other examples in the [Application unit tests](#) (`TestApplication` class) in the Holoscan SDK repository.

10.3.3 Adding user-defined command line arguments

When adding user-defined command line arguments to an application, one should avoid the use of any of the default command line argument names as `--help`, `--version`, `--config`, `--driver`, `--worker`, `--address`, `--worker-address`, `--fragments` as covered in the section on [running a distributed application](#). It is recommended to parse user-defined arguments from the `argv` ((C++/Python)) method/property of the application as covered in the note above, instead of using C++ `char* argv[]` or Python `sys.argv` directly. This way, only the new, user-defined arguments will need to be parsed.

A concrete example of this for both C++ and Python can be seen in the existing [ping_distributed](#) example where an application-defined boolean argument (`--gpu`) is specified in addition to the default set of application arguments.

C++

```
int main() {
    auto app = holoscan::make_application<App>();

    // Parse args
    bool tensor_on_gpu = false;
    auto& args = app->argv();
    if (std::find(args.begin(), args.end(), "--gpu") != std::end(args)) { tensor_on_gpu =
    true; }

    // configure tensor on host vs. GPU
    app->gpu_tensor(tensor_on_gpu);

    // run the application
    app->run();

    return 0;
}
```

Python

```
def main(on_gpu=False):
    app = MyPingApp()

    tensor_str = "GPU" if on_gpu else "host"
    print(f"Configuring application to use {tensor_str} tensors")
    app.gpu_tensor = on_gpu

    app.run()

if __name__ == "__main__":

    # get the Application's arguments
```

(continues on next page)

(continued from previous page)

```

app_argv = Application().argv

parser = ArgumentParser(description="Distributed ping application.")
parser.add_argument(
    "--gpu",
    action="store_true",
    help="Use a GPU tensor instead of a host tensor",
)
# pass app_argv[1:] to parse_args (app_argv[0] is the path of the application)
args = parser.parse_args(app_argv[1:])
main(on_gpu=args.gpu)

```

For Python, `app_argv[1:]` can be used with an `ArgumentParser` from Python's `argparse` module.

Alternatively, it may be preferable to instead use `parser.parse_known_args()` to allow any arguments not defined by the user's parser to pass through to the application class itself. If one also sets `add_help=False` when constructing the `ArgumentParser`, it is possible to print the parser's help while still preserving the default application help (covering the default set of distributed application arguments). An example of this style is shown in the code block below.

```

parser = ArgumentParser(description="Distributed ping application.", add_help=False)
parser.add_argument(
    "--gpu",
    action="store_true",
    help="Use a GPU tensor instead of a host tensor",
)

# use parse_known_args to ignore other CLI arguments that may be used by Application
args, remaining = parser.parse_known_args()

# can print the parser's help here prior to the Application's help output
if "-h" in remaining or "--help" in remaining:
    print("Additional arguments supported by this application:")
    print(textwrap.indent(parser.format_help(), " "))
main(on_gpu=args.gpu)

```


PACKAGING HOLOSCAN APPLICATIONS

The *Holoscan App Packager*, included as part of the *Holoscan CLI* as the `package` command, allows you to package your Holoscan applications into a *HAP-compliant* container image for distribution and deployment.

11.1 Prerequisites

11.1.1 Dependencies

Ensure the following are installed in the environment where you want to run the *CLI*:

- **PIP dependencies:** This is automatically installed with the Holoscan Python wheel.
- **NVIDIA Container Toolkit with Docker**
 - Developer Kits (aarch64): already included in IGX Software and JetPack
 - x86_64: tested with NVIDIA Container Toolkit 1.13.3 with Docker v24.0.1
- **Docker BuildX plugin**
 1. Check if it is installed:

```
$ docker buildx version
github.com/docker/buildx v0.10.5 86bdced
```

2. If not, run the following commands based on the [official doc](#):

```
# Install Docker dependencies
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg

# Add Docker Official GPG Key
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o_
↪ /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Configure Docker APT Repository
echo \
"deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.
↪ gpg] https://download.docker.com/linux/ubuntu \
"$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

(continues on next page)

(continued from previous page)

```
# Install Docker BuildX Plugin
sudo apt-get update
sudo apt-get install docker-buildx-plugin
```

- **QEMU** (*Optional*)
 - used for packaging container images of different architectures than the host (example: x86_64 -> arm64).

11.1.2 CLI Installation

The Holoscan CLI is installed as part of the Holoscan SDK and can be called with the following instructions depending on your installation:

Python Wheel

- In a virtual environment: the holoscan CLI should already be in the PATH
- System python: ensure that \$HOME/.local/bin is added to your PATH. If using bash, the following command will make it persist across sessions:

```
echo 'export PATH=$HOME/.local/bin:$PATH' >> ~/.bashrc
```

Debian Package

Ensure that /opt/nvidia/holoscan/ is added to your PATH. If using bash, the following command will make it persist across sessions:

```
echo 'alias holoscan=/opt/nvidia/holoscan/bin/holoscan' >> ~/.bashrc
```

From source

If building the SDK from source and starting the build container with `run launch`, the holoscan CLI should already be in the PATH.

If building bare-metal (advanced), ensure that <BUILD_OR_INSTALL_DIR>/bin is added to your PATH. If using bash, the following command will make it persist across sessions:

```
echo 'alias holoscan=<BUILD_OR_INSTALL_DIR>/bin/holoscan' >> ~/.bashrc
```

NGC Container

The NGC container has the CLI installed already, no additional steps are required.

11.2 Package an application

Tip: The packager feature is also illustrated in the [cli_packager](#) and [video_replayer_distributed](#) examples.

Additional arguments are required when launching the container to enable the packaging of Holoscan applications inside the NGC Holoscan container. Please see the [NGC Holoscan container](#) page for additional details.

1. Ensure to use the [HAP environment variables](#) wherever possible when accessing data. For example:

Let's take a look at the distributed video replayer example ([examples/video_replayer_distributed](#)).

- **Using the Application Configuration File**

C++

In the main function, we call the `app->config(config_path)` function with the default configuration file. The `app->config(...)` checks to see if the application was executed with `--config` argument first. If `--config` was set, the method uses the configuration file from the `--config` argument. Otherwise, it checks if the environment variable `HOLOSCAN_CONFIG_PATH` is set and uses that value as the source. If neither were set, the default configuration file (`config_path`) is used.

```
int main(int argc, char** argv) {
    // Get the yaml configuration file
    auto config_path = std::filesystem::canonical(argv[0]).parent_path();
    config_path /= std::filesystem::path("video_replayer_distributed.yaml");

    auto app = holoscan::make_application<DistributedVideoReplayerApp>();
    app->config(config_path);
    app->run();

    return 0;
}
```

Python

In the main function, we call the `app.config(config_file_path)` function with the default configuration file. The `app.config(...)` method checks to see if the application was executed with `--config` argument first. If `--config` was set, the method uses the configuration file from the `--config` argument. Otherwise, it checks if the environment variable `HOLOSCAN_CONFIG_PATH` is set and uses that value as the source. If neither were set, the default configuration file (`config_file_path`) is used.

```
def main():
    input_path = get_input_path()
    config_file_path = os.path.join(os.path.dirname(__file__), "video_replayer_
distributed.yaml")

    logging.info(f"Reading application configuration from {config_file_path}")

    app = DistributedVideoReplayerApp(input_path)
    app.config(config_file_path)
    app.run()
```

- Using Environment Variable HOLOSCAN_INPUT_PATH for Data Input

C++

In `Fragment1`, we try to set the input video directory with the value defined in `HOLOSCAN_INPUT_PATH`. When we instantiate a new Video Stream Replayer operator, we pass in all configuration values from the `from_config("replayer")` call. In addition, we include args that we created with the value from `HOLOSCAN_INPUT_PATH` if available as the last argument to override the directory setting.

```
class Fragment1 : public holoscan::Fragment {
public:
    void compose() override {
        using namespace holoscan;
        ArgList args;
        auto data_directory = std::getenv("HOLOSCAN_INPUT_PATH");
        if (data_directory != nullptr && data_directory[0] != '\0') {
            auto video_directory = std::filesystem::path(data_directory);
            video_directory /= "racerx";
            args.add(Arg("directory", video_directory.string()));
            HOLOSCAN_LOG_INFO("Using video from {}", video_directory.string());
        }
        auto replayer =
            make_operator<ops::VideoStreamReplayerOp>("replayer", from_config(
↪ "replayer"), args);
        add_operator(replayer);
    }
};
```

Python

In `Fragment1`, we try to set the input video directory with the value defined in `HOLOSCAN_INPUT_PATH`. When we instantiate a new Video Stream Replayer operator, we pass in the `video_path` along with all replayer configurations found in the configuration file.

```
class Fragment1(Fragment):
    def __init__(self, app, name):
        super().__init__(app, name)

    def __init__(self, app, name):
        super().__init__(app, name)

    def compose(self):
        # Set the video source
        video_path = self._get_input_path()
        logging.info(
            f"Using video from {video_path}"
        )

        # Define the replayer and holoviz operators
        replayer = VideoStreamReplayerOp(
            self, name="replayer", directory=video_path, **self.kwargs("replayer
↪ ")
```

(continues on next page)

(continued from previous page)

```

    )

    self.add_operator(replayer)

    def _get_input_path(self):
        path = os.environ.get(
            "HOLOSCAN_INPUT_PATH", os.path.join(os.path.dirname(__file__), "data
↪")
        )
        return os.path.join(path, "racerx")

```

2. Include a YAML configuration file as described in the [Application Runner Configuration](#) page.
3. Use the `holoscan package` command to create a HAP container image. For example:

```

holoscan package --platform x64-workstation --tag my-awesome-app --config /path/to/
↪my/awesome/application/config.yaml /path/to/my/awesome/application/

```

11.2.1 Additional Requirements for Non-Distributed Applications

In addition to using HAP-defined environment variables, applications must also handle parsing the `--config` argument when packaging a non-distributed application.

- **Handle `--config` argument**

C++

Define a `parse_arguments` function to handle parsing the data path and the config file path:

```

bool parse_arguments(int argc, char** argv, std::string& data_path, std::string&
↪config_path) {
    static struct option long_options[] = {
        {"data", required_argument, 0, 'd'}, {"config", required_argument, 0, 'c'},
↪{0, 0, 0, 0}};

    while (int c = getopt_long(argc, argv, "d:c:", long_options, NULL)) {
        if (c == -1 || c == '?') break;

        switch (c) {
            case 'c':
                config_path = optarg;
                break;
            case 'd':
                data_path = optarg;
                break;
            default:
                holoscan::log_error("Unhandled option '{}'", static_cast<char>(c));
                return false;
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    return true;
}

```

Use the `parse_arguments()` function defined above in the main function:

```

int main(int argc, char** argv) {
    // Parse the arguments
    std::string config_path = "";
    std::string data_directory = "";
    if (!parse_arguments(argc, argv, data_directory, config_path)) { return 1; }
    if (data_directory.empty()) {
        // Get the input data environment variable
        auto input_path = std::getenv("HOLOSCAN_INPUT_PATH");
        if (input_path != nullptr && input_path[0] != '\0') {
            data_directory = std::string(input_path);
        } else {
            HOLOSCAN_LOG_ERROR(
                "Input data not provided. Use --data or set HOLOSCAN_INPUT_PATH_
↪environment variable.");
            exit(-1);
        }
    }

    if (config_path.empty()) {
        // Get the input data environment variable
        auto config_file_path = std::getenv("HOLOSCAN_CONFIG_PATH");
        if (config_file_path == nullptr || config_file_path[0] == '\0') {
            auto config_file = std::filesystem::canonical(argv[0]).parent_path();
            config_path = config_file / std::filesystem::path("app-config.yaml");
        } else {
            config_path = config_file_path;
        }
    }

    auto app = holoscan::make_application<App>();

    HOLOSCAN_LOG_INFO("Using configuration file from {}", config_path);
    app->config(config_path);

    HOLOSCAN_LOG_INFO("Using input data from {}", data_directory);
    app->set_datapath(data_directory);

    app->run();

    return 0;
}

```


Python

Define a `parse_arguments` function to parse the arguments:

```
def parse_arguments() -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="My Application")
    parser.add_argument(
        "--data",
        type=str,
        required=False,
        default=os.environ.get("HOLOSCAN_INPUT_PATH", None),
        help="Input dataset",
    )
    parser.add_argument(
        "--config",
        type=str,
        required=False,
        default=os.environ.get(
            "HOLOSCAN_CONFIG_PATH",
            os.path.join(os.path.dirname(__file__), "app-config.yaml"),
        ),
        help="Application configurations",
    )

    args, _ = parser.parse_known_args()

    return args
```

Parse the arguments using `parse_arguments()` defined above and start the application:

```
if __name__ == "__main__":
    args = parse_arguments()

    if args.data is None:
        logger.error(
            "Input data not provided. Use --data or set HOLOSCAN_INPUT_PATH_
↪environment variable."
        )
        sys.exit(-1)

    app = MyApp(args.data)
    app.config(args.config)
    app.run()
```

11.2.2 Common Issues When Using Holoscan Packager

DNS Name Resolution Error

The Holoscan Packager may be unable to resolve hostnames in specific networking environments and may show errors similar to the following:

```
curl: (6) Could not resolve host: github.com.
Failed to establish a new connection: [Errno -3] Temporary failure in name resolution...
```

To resolve these errors, edit the `/etc/docker/daemon.json` file to include `dns` and `dns-search` fields as follows:

```
{
  "default-runtime": "nvidia",
  "runtimes": {
    "nvidia": {
      "args": [],
      "path": "nvidia-container-runtime"
    }
  },
  "dns": ["IP-1", "IP-n"],
  "dns-search": ["DNS-SERVER-1", "DNS-SERVER-n"]
}
```

You may need to consult your IT team and replace `IP-x` and `DNS-SERVER-x` with the provided values.

11.3 Run a packaged application

The packaged Holoscan application container image can run with the *Holoscan App Runner*:

```
holoscan run -i /path/to/my/input -o /path/to/application/generated/output my-
↳ application:1.0.1
```

Since the packaged Holoscan application container images are OCI-compliant, they're also compatible with [Docker](#), [Kubernetes](#), and [containerd](#).

Each packaged Holoscan application container image includes tools inside for extracting the embedded application, manifest files, models, etc. To access the tool and to view all available options, run the following:

```
docker run -it my-container-image[:tag] help
```

The command should print following:

```
USAGE: /var/holoscan/tools [command] [arguments]...
Command List
  extract ----- Extract data based on mounted volume paths.
    /var/run/holoscan/export/app      extract the application
    /var/run/holoscan/export/config   extract app.json and pkg.json manifest files.
↳ and application YAML.
    /var/run/holoscan/export/models   extract models
    /var/run/holoscan/export/docs     extract documentation files
    /var/run/holoscan/export         extract all of the above
    IMPORTANT: ensure the directory to be mounted for data extraction is created.
↳ first on the host system.
```

(continues on next page)

(continued from previous page)

```

        and has the correct permissions. If the directory had been created by ↵
↵the container previously
        with the user and group being root, please delete it and manually ↵
↵create it again.
    show ----- Print manifest file(s): [app|pkg] to the ↵
↵terminal.
        app                print app.json
        pkg                print pkg.json
    env ----- Print all environment variables to the terminal.

```

Note: The tools can also be accessed inside the Docker container via `/var/holoscan/tools`.

For example, run the following commands to extract the manifest files and the application configuration file:

```

# create a directory on the host system first
mkdir -p config-files

# mount the directory created to /var/run/holoscan/export/config
docker run -it --rm -v $(pwd)/config-files:/var/run/holoscan/export/config my-container-
↵image[:tag] extract

# include -u 1000 if the above command reports a permission error
docker run -it --rm -u 1000 -v $(pwd)/config-files:/var/run/holoscan/export/config my-
↵container-image[:tag] extract

# If the permission error continues to occur, please check if the mounted directory has ↵
↵the correct permission.
# If it doesn't, please recreate it or change the permissions as needed.

# list files extracted
ls config-files/

# output:
# app.json  app.yaml  pkg.json

```


CREATING OPERATORS

Tip: Creating a custom operator is also illustrated in the *ping_custom_op* example.

12.1 C++ Operators

When assembling a C++ application, two types of operators can be used:

1. **Native C++ operators:** custom operators defined in C++ without using the GXF API, by creating a subclass of `holoscan::Operator`. These C++ operators can pass arbitrary C++ objects around between operators.
2. **GXF Operators:** operators defined in the underlying C++ library by inheriting from the `holoscan::ops::GXFOperator` class. These operators wrap GXF codelets from GXF extensions. Examples are `VideoStreamReplayerOp` for replaying video files, `FormatConverterOp` for format conversions, and `HolovizOp` for visualization.

Note: It is possible to create an application using a mixture of GXF operators and native operators. In this case, some special consideration to cast the input and output tensors appropriately must be taken, as shown in [a section below](#).

12.1.1 Native C++ Operators

Operator Lifecycle (C++)

The lifecycle of a `holoscan::Operator` is made up of three stages:

- `start()` is called once when the operator starts, and is used for initializing heavy tasks such as allocating memory resources and using parameters.
- `compute()` is called when the operator is triggered, which can occur any number of times throughout the operator lifecycle between `start()` and `stop()`.
- `stop()` is called once when the operator is stopped, and is used for deinitializing heavy tasks such as deallocating resources that were previously assigned in `start()`.

All operators on the workflow are scheduled for execution. When an operator is first executed, the `start()` method is called, followed by the `compute()` method. When the operator is stopped, the `stop()` method is called. The `compute()` method is called multiple times between `start()` and `stop()`.

If any of the scheduling conditions specified by *Conditions* are not met (for example, the `CountCondition` would cause the scheduling condition to not be met if the operator has been executed a certain number of times), the operator is stopped and the `stop()` method is called.

We will cover how to use Conditions in the *Specifying operator inputs and outputs (C++)* section of the user guide.

Typically, the `start()` and the `stop()` functions are only called once during the application's lifecycle. However, if the scheduling conditions are met again, the operator can be scheduled for execution, and the `start()` method will be called again.

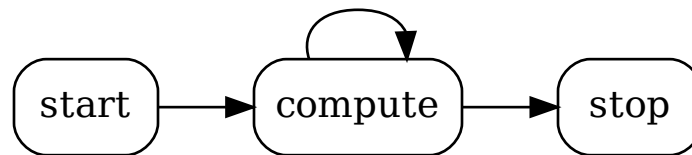


Fig. 12.1: The sequence of method calls in the lifecycle of a Holoscan Operator

Warning: If Python bindings are going to be created for this C++ operator, it is recommended to put any cleanup of resources allocated in the `initialize()` and/or `start()` methods into the `stop()` method of the operator and **not** in its destructor. This is necessary as a workaround to a current issue where it is not guaranteed that the destructor always gets called prior to Python application termination. The `stop()` method will always be explicitly called, so we can be assured that any cleanup happens as expected.

We can override the default behavior of the operator by implementing the above methods. The following example shows how to implement a custom operator that overrides `start`, `stop`, and `compute` methods.

Listing 12.1: The basic structure of a Holoscan Operator (C++)

```

1  #include "holoscan/holoscan.hpp"
2
3  using holoscan::Operator;
4  using holoscan::OperatorSpec;
5  using holoscan::InputContext;
6  using holoscan::OutputContext;
7  using holoscan::ExecutionContext;
8  using holoscan::Arg;
9  using holoscan::ArgList;
10
11 class MyOp : public Operator {
12 public:
13     HOLOSCAN_OPERATOR_FORWARD_ARGS(MyOp)
14
15     MyOp() = default;
16
17     void setup(OperatorSpec& spec) override {
18     }
  
```

(continues on next page)

(continued from previous page)

```

19
20 void start() override {
21     HOLOSCAN_LOG_TRACE("MyOp::start()");
22 }
23
24 void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override {
25     HOLOSCAN_LOG_TRACE("MyOp::compute()");
26 };
27
28 void stop() override {
29     HOLOSCAN_LOG_TRACE("MyOp::stop()");
30 }
31 };

```

Creating a custom operator (C++)

To create a custom operator in C++, it is necessary to create a subclass of `holoscan::Operator`. The following example demonstrates how to use native operators (the operators that do not have an underlying, pre-compiled GXF Codelet).

Code Snippet: `examples/ping_multi_port/cpp/ping_multi_port.cpp`

Listing 12.2: `examples/ping_multi_port/cpp/ping_multi_port.cpp`

```

21 #include "holoscan/holoscan.hpp"
22
23 class ValueData {
24 public:
25     ValueData() = default;
26     explicit ValueData(int value) : data_(value) {
27         HOLOSCAN_LOG_TRACE("ValueData::ValueData(): {}", data_);
28     }
29     ~ValueData() { HOLOSCAN_LOG_TRACE("ValueData::~ValueData(): {}", data_); }
30
31     void data(int value) { data_ = value; }
32
33     int data() const { return data_; }
34
35 private:
36     int data_;
37 };
38
39 namespace holoscan::ops {
40
41 class PingTxOp : public Operator {
42 public:
43     HOLOSCAN_OPERATOR_FORWARD_ARGS(PingTxOp)
44
45     PingTxOp() = default;
46
47     void setup(OperatorSpec& spec) override {
48         spec.output<std::shared_ptr<ValueData>>("out1");

```

(continues on next page)

(continued from previous page)

```

49     spec.output<std::shared_ptr<ValueData>>("out2");
50 }
51
52 void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override {
53     auto value1 = std::make_shared<ValueData>(index_++);
54     op_output.emit(value1, "out1");
55
56     auto value2 = std::make_shared<ValueData>(index_++);
57     op_output.emit(value2, "out2");
58 };
59 int index_ = 1;
60 };
61
62 class PingMxOp : public Operator {
63 public:
64     HOLOSCAN_OPERATOR_FORWARD_ARGS(PingMxOp)
65
66     PingMxOp() = default;
67
68     void setup(OperatorSpec& spec) override {
69         spec.input<std::shared_ptr<ValueData>>("in1");
70         spec.input<std::shared_ptr<ValueData>>("in2");
71         spec.output<std::shared_ptr<ValueData>>("out1");
72         spec.output<std::shared_ptr<ValueData>>("out2");
73         spec.param(multiplier_, "multiplier", "Multiplier", "Multiply the input by this value
↪", 2);
74     }
75
76     void compute(InputContext& op_input, OutputContext& op_output, ExecutionContext&)
↪override {
77         auto value1 = op_input.receive<std::shared_ptr<ValueData>>("in1").value();
78         auto value2 = op_input.receive<std::shared_ptr<ValueData>>("in2").value();
79
80         HOLOSCAN_LOG_INFO("Middle message received (count: {})", count_++);
81
82         HOLOSCAN_LOG_INFO("Middle message value1: {}", value1->data());
83         HOLOSCAN_LOG_INFO("Middle message value2: {}", value2->data());
84
85         // Multiply the values by the multiplier parameter
86         value1->data(value1->data() * multiplier_);
87         value2->data(value2->data() * multiplier_);
88
89         op_output.emit(value1, "out1");
90         op_output.emit(value2, "out2");
91     };
92
93 private:
94     int count_ = 1;
95     Parameter<int> multiplier_;
96 };
97
98 class PingRxOp : public Operator {

```

(continues on next page)

(continued from previous page)

```

99 public:
100 HOLOSCAN_OPERATOR_FORWARD_ARGS(PingRxOp)
101
102 PingRxOp() = default;
103
104 void setup(OperatorSpec& spec) override {
105     // // Since Holoscan SDK v2.3, users can define a multi-receiver input port using
106     // 'spec.input()'
107     // // with 'IOSpec::kAnySize'.
108     // // The old way is to use 'spec.param()' with 'Parameter<std::vector<IOSpec*>>'
109     // 'receivers_';
110     // spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.",
111     // {});
112     spec.input<std::vector<std::shared_ptr<ValueData>>>("receivers", IOSpec::kAnySize);
113 }
114
115 void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {
116     auto value_vector =
117         op_input.receive<std::vector<std::shared_ptr<ValueData>>>("receivers").value();
118
119     HOLOSCAN_LOG_INFO("Rx message received (count: {}, size: {})", count_++, value_
120     vector.size());
121
122     HOLOSCAN_LOG_INFO("Rx message value1: {}", value_vector[0]->data());
123     HOLOSCAN_LOG_INFO("Rx message value2: {}", value_vector[1]->data());
124 };
125
126 private:
127     // // Since Holoscan SDK v2.3, the following line is no longer needed.
128     // Parameter<std::vector<IOSpec*>> receivers_;
129     int count_ = 1;
130 };
131
132 } // namespace holoscan::ops
133
134 class MyPingApp : public holoscan::Application {
135 public:
136     void compose() override {
137         using namespace holoscan;
138
139         // Define the tx, mx, rx operators, allowing the tx operator to execute 10 times
140         auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
141         auto mx = make_operator<ops::PingMxOp>("mx", Arg("multiplier", 3));
142         auto rx = make_operator<ops::PingRxOp>("rx");
143
144         // Define the workflow
145         add_flow(tx, mx, {{"out1", "in1"}, {"out2", "in2"}});
146         add_flow(mx, rx, {{"out1", "receivers"}, {"out2", "receivers"}});
147     }
148 };
149
150 int main(int argc, char** argv) {

```

(continues on next page)

(continued from previous page)

```

147 auto app = holoscan::make_application<MyPingApp>();
148 app->run();
149
150 return 0;
151 }

```

In this application, three operators are created: `PingTxOp`, `PingMxOp`, and `PingRxOp`

1. The `PingTxOp` operator is a source operator that emits two values every time it is invoked. The values are emitted on two different output ports, `out1` (for odd integers) and `out2` (for even integers).
2. The `PingMxOp` operator is a middle operator that receives two values from the `PingTxOp` operator and emits two values on two different output ports. The values are multiplied by the `multiplier` parameter.
3. The `PingRxOp` operator is a sink operator that receives two values from the `PingMxOp` operator. The values are received on a single input, `receivers`, which is a vector of input ports. The `PingRxOp` operator receives the values in the order they are emitted by the `PingMxOp` operator.

As covered in more detail below, the inputs to each operator are specified in the `setup()` method of the operator. Then inputs are received within the `compute()` method via `op_input.receive()` and outputs are emitted via `op_output.emit()`.

Note that for native C++ operators as defined here, any object including a shared pointer can be emitted or received. For large objects such as tensors, it may be preferable from a performance standpoint to transmit a shared pointer to the object rather than making a copy. When shared pointers are used and the same tensor is sent to more than one downstream operator, you should avoid in-place operations on the tensor or race conditions between operators may occur.

If you need to configure arguments or perform other setup tasks before or after the operator is initialized, you can override the `initialize()` method. This method is called once before the `start()` method.

Example:

```

void initialize() override {
    // Register custom type and codec for serialization
    register_converter<std::array<float, 3>>()>();
    register_codec<std::vector<InputSpec>>>("std::vector
→<holoscan::ops::HolovizOp::InputSpec>", true);

    // Set up prerequisite parameters before calling Operator::initialize()
    auto frag = fragment();

    // Check if an argument for 'allocator' exists
    auto has_allocator = std::find_if(
        args().begin(), args().end(), [](const auto& arg) { return (arg.name() ==
→"allocator"); });
    // Create the allocator if no argument is provided
    if (has_allocator == args().end()) {
        allocator_ = frag->make_resource<UnboundedAllocator>("allocator");
        add_arg(allocator_.get());
    }

    // Call the parent class's initialize() method to complete the initialization.
    // Operator::initialize must occur after all arguments have been added.
    Operator::initialize();
}

```

(continues on next page)

(continued from previous page)

```

    // After Operator::initialize(), the operator is ready for use and the parameters
    ↪are set
    int multiplier = multiplier_;
    HOLOSCAN_LOG_INFO("Multiplier: {}", multiplier);
}

```

For details on the `register_converter()` and `register_codec()` methods, refer to `holoscan::Operator::register_converter()` for the custom parameter type and the section on *object serialization* for distributed applications.

Specifying operator parameters (C++)

In the example `holoscan::ops::PingMxOp` operator above, you have a parameter `multiplier` that is declared as part of the class as a private member using the `param()` templated type:

```
Parameter<int> multiplier_;
```

It is then added to the `OperatorSpec` attribute of the operator in its `setup()` method, where an associated string key must be provided. Other properties can also be mentioned such as description and default value:

```

// Provide key, and optionally other information
spec.param(multiplier_, "multiplier", "Multiplier", "Multiply the input by this value",
↪2);

```

Note: If your parameter is of a custom type, you must register that type and provide a YAML encoder/decoder, as documented under `holoscan::Operator::register_converter()`

See the *Configuring operator parameters* section to learn how an application can set these parameters.

Specifying operator inputs and outputs (C++)

To configure the input(s) and output(s) of C++ native operators, call the `spec.input()` and `spec.output()` methods within the `setup()` method of the operator.

The `spec.input()` and `spec.output()` methods should be called once for each input and output to be added. The `OperatorSpec` object and the `setup()` method will be initialized and called automatically by the `Application` class when its `run()` method is called.

These methods (`spec.input()` and `spec.output()`) return an `IOSpec` object that can be used to configure the input/output port.

By default, the `holoscan::MessageAvailableCondition` and `holoscan::DownstreamMessageAffordableCondition` conditions are applied (with a `min_size` of 1) to the input/output ports. This means that the operator's `compute()` method will not be invoked until a message is available on the input port and the downstream operator's input port (queue) has enough capacity to receive the message.

```

void setup(OperatorSpec& spec) override {
    spec.input<std::shared_ptr<ValueData>>("in");
    // Above statement is equivalent to:
    // spec.input<std::shared_ptr<ValueData>>("in")
    // .condition(ConditionType::kMessageAvailable, Arg("min_size") = static_cast
    ↪<uint64_t>(1));
}

```

(continues on next page)

(continued from previous page)

```

    spec.output<std::shared_ptr<ValueData>>("out");
    // Above statement is equivalent to:
    // spec.output<std::shared_ptr<ValueData>>("out")
    // .condition(ConditionType::kDownstreamMessageAffordable, Arg("min_size") =
    ↪ static_cast<uint64_t>(1));
    ...
}

```

In the above example, the `spec.input()` method is used to configure the input port to have the `holoscan::MessageAvailableCondition` with a minimum size of 1. This means that the operator's `compute()` method will not be invoked until a message is available on the input port of the operator. Similarly, the `spec.output()` method is used to configure the output port to have the `holoscan::DownstreamMessageAffordableCondition` with a minimum size of 1. This means that the operator's `compute()` method will not be invoked until the downstream operator's input port has enough capacity to receive the message.

If you want to change this behavior, use the `IOSpec::condition()` method to configure the conditions. For example, to configure the input and output ports to have no conditions, you can use the following code:

```

void setup(OperatorSpec& spec) override {
    spec.input<std::shared_ptr<ValueData>>("in")
        .condition(ConditionType::kNone);

    spec.output<std::shared_ptr<ValueData>>("out")
        .condition(ConditionType::kNone);
    // ...
}

```

The example code in the `setup()` method configures the input port to have no conditions, which means that the `compute()` method will be called as soon as the operator is ready to compute. Since there is no guarantee that the input port will have a message available, the `compute()` method should check if there is a message available on the input port before attempting to read it.

The `receive()` method of the `InputContext` object can be used to access different types of input data within the `compute()` method of your operator class, where its template argument (`DataT`) is the data type of the input. This method takes the name of the input port as an argument (which can be omitted if your operator has a single input port), and returns the input data. If input data is not available, the method returns an object of the `holoscan::expected<std::shared_ptr<ValueData>, holoscan::RuntimeError>` type. The `holoscan::expected<T, E>` class template is used to represent expected objects, which can either hold a value of type `T` or an error of type `E`. The expected object is used to return and propagate errors in a more structured way than using error codes or exceptions. In this case, the expected object can hold either a `std::shared_ptr<ValueData>` object or a `holoscan::RuntimeError` class that contains an error message describing the reason for the failure.

The `holoscan::RuntimeError` class is a derived class of `std::runtime_error` and supports accessing more error information, for example, with the `what()` method.

In the example code fragment below, the `PingRxOp` operator receives input on a port called "in" with data type `std::shared_ptr<ValueData>`. The `receive()` method is used to access the input data. The `maybe_value` is checked to be valid or not with the `if` condition. If there is an error in the input data, the error message is logged and the operator throws the error. If the input data is valid, we can access the reference of the input data using the `value()` method of the expected object. To avoid copying the input data (or creating another shared pointer), the reference of the input data is stored in the `value` variable (using `auto& value = maybe_value.value()`). The `data()` method of the `ValueData` class is then called to get the value of the input data.

```
// ...

class PingRxOp : public holoscan::Operator {
public:
    HOLOSCAN_OPERATOR_FORWARD_ARGS(PingRxOp)
    PingRxOp() = default;

    void setup(holoscan::OperatorSpec& spec) override {
        spec.input<std::shared_ptr<ValueData>>("in");
    }

    void compute(holoscan::InputContext& op_input, holoscan::OutputContext&,
                 holoscan::ExecutionContext&) override {
        auto maybe_value = op_input.receive<std::shared_ptr<ValueData>>("in");

        if (!maybe_value) {
            HOLOSCAN_LOG_ERROR("Failed to receive message - {}", maybe_value.error().what());
            // [error] Failed to receive message - InputContext receive() Error: No message is_
            ←received from the input port with name 'in'
            throw maybe_value.error(); // or `return;`
        }

        auto& value = maybe_value.value();
        HOLOSCAN_LOG_INFO("Message received (value: {})", value->data());
    }
};
```

Internally, message passing in Holoscan is implemented using the `Message` class, which wraps a `std::any` object and provides a type-safe interface to access the input data. The `std::any` class is a type-safe container for single values of any type and is used to store the input and output data of operators. The `std::any` class is part of the C++ standard library and is defined in the `any` header file.

Since the Holoscan SDK uses GXF as an execution engine, the `holoscan::Message` object is also encapsulated in a `nvidia::gxf::Entity` object when passing data among Holoscan native operators and GXF operators. This ensures that the data is compatible with the GXF framework.

If the input data is expected to be from a GXF operator or a tensor (in both cases, the data is an instance of `nvidia::gxf::Entity`), the `holoscan::gxf::Entity` class can be used in the template argument of the `receive` method to access the input data. The `holoscan::gxf::Entity` class is a wrapper around the `nvidia::gxf::Entity` class (which is like a dictionary object) and provides a way to get a tensor and to add a tensor to the entity.

The Holoscan SDK provides built-in data types called **Domain Objects**, defined in the `include/holoscan/core/domain` directory. For example, the `holoscan::Tensor` is a Domain Object class that represents a multi-dimensional array of data, which is interoperable with the underlying GXF class (`nvidia::gxf::Tensor`). The `holoscan::Tensor` class provides methods to access the tensor data, shape, and other properties. Passing `holoscan::Tensor` objects to/from *GXF operators* is supported.

Tip: The `holoscan::Tensor` class is a wrapper around the `DLManagedTensorContext` struct holding a `DLManagedTensor` object. As such, it provides a primary interface to access tensor data and is interoperable with other frameworks that support the `DLPack` interface.

See the [interoperability section](#) for more details.

In the example below, the `TensorRx` operator receives input on a port called “in” with data type

holoscan::gxf::Entity.

```
// ...

class TensorRxOp : public holoscan::Operator {
public:
    HOLOSCAN_OPERATOR_FORWARD_ARGS(TensorRxOp)
    TensorRxOp() = default;

    void setup(holoscan::OperatorSpec& spec) override {
        spec.input<holoscan::gxf::Entity>("in");
    }

    void compute(holoscan::InputContext& op_input, holoscan::OutputContext&,
                 holoscan::ExecutionContext&) override {
        // Type of 'maybe_entity' is holoscan::expected<holoscan::gxf::Entity,>
        ↪holoscan::RuntimeError>
        auto maybe_entity = op_input.receive<holoscan::gxf::Entity>("in");
        if (maybe_entity) {
            auto& entity = maybe_entity.value(); // holoscan::gxf::Entity&
            // Get a tensor from the entity if it exists.
            // Can pass a tensor name as an argument to get a specific tensor.
            auto tensor = entity.get<holoscan::Tensor>(); // std::shared_ptr<holoscan::Tensor>
            if (tensor) {
                HOLOSCAN_LOG_INFO("tensor nbytes: {}", tensor->nbytes());
            }
        }
    }
};
```

If the entity contains a tensor, the `get` method of the `holoscan::gxf::Entity` class can be used to retrieve the tensor. The `get` method returns a `std::shared_ptr<holoscan::Tensor>` object, which can be used to access the tensor data. The `nbytes` method of the `holoscan::Tensor` class is used to get the number of bytes in the tensor.

By using the `holoscan::TensorMap` class, which stores a map of tensor names to tensors (`std::unordered_map<std::string, std::shared_ptr<holoscan::Tensor>>`), the code that receives an entity object containing one or more tensor objects can be updated to receive a `holoscan::TensorMap` object instead of a `holoscan::gxf::Entity` object. The `holoscan::TensorMap` class provides a way to access the tensor data by name, using a `std::unordered_map`-like interface.

```
// ...

class TensorRxOp : public holoscan::Operator {
public:
    HOLOSCAN_OPERATOR_FORWARD_ARGS(TensorRxOp)
    TensorRxOp() = default;

    void setup(holoscan::OperatorSpec& spec) override {
        spec.input<holoscan::TensorMap>("in");
    }

    void compute(holoscan::InputContext& op_input, holoscan::OutputContext&,
                 holoscan::ExecutionContext&) override {
        // Type of 'maybe_entity' is holoscan::expected<holoscan::TensorMap,>
        ↪holoscan::RuntimeError>
```

(continues on next page)

(continued from previous page)

```

auto maybe_tensor_map = op_input.receive<holoscan::TensorMap>("in");
if (maybe_tensor_map) {
    auto& tensor_map = maybe_tensor_map.value(); // holoscan::TensorMap&
    for (const auto& [name, tensor] : tensor_map) {
        HOLOSCAN_LOG_INFO("tensor name: {}", name);
        HOLOSCAN_LOG_INFO("tensor nbytes: {}", tensor->nbytes());
    }
}
};

```

In the above example, the `TensorRxOp` operator receives input on a port called “in” with data type `holoscan::TensorMap`. The `receive` method of the `InputContext` object is used to access the input data. The `receive` method returns an expected object that can hold either a `holoscan::TensorMap` object or a `holoscan::RuntimeError` object. The `holoscan::TensorMap` class is a wrapper around the `std::unordered_map<std::string, std::shared_ptr<holoscan::Tensor>>` class and provides a way to access the tensor data. The `nbytes` method of the `holoscan::Tensor` class is used to get the number of bytes in the tensor.

If the type `std::any` is used for the template argument of the `receive` method, the `receive()` method will return a `std::any` object containing the input of the specified name. In the example below, the `PingRxOp` operator receives input on a port called “in” with data type `std::any`. The `type()` method of the `std::any` object is used to determine the actual type of the input data, and the `std::any_cast()` function is used to retrieve the value of the input data.

```

// ...

class AnyRxOp : public holoscan::Operator {
public:
    HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER(AnyRxOp, holoscan::ops::GXFOperator)
    AnyRxOp() = default;
    void setup(holoscan::OperatorSpec& spec) override {
        spec.input<std::any>("in");
    }
    void compute(holoscan::InputContext& op_input, holoscan::OutputContext&,
        holoscan::ExecutionContext&) override {
        auto maybe_any = op_input.receive<std::any>("in");
        if (!maybe_any) {
            HOLOSCAN_LOG_ERROR("Failed to receive message - {}", maybe_any.error().what());
            return;
        }

        auto& in_any = maybe_any.value();
        const auto& in_any_type = in_any.type();

        try {
            if (in_any_type == typeid(holoscan::gxf::Entity)) {
                auto in_entity = std::any_cast<holoscan::gxf::Entity>(in_any);
                auto tensor = in_entity.get<holoscan::Tensor>(); // std::shared_ptr
                if (tensor) {
                    HOLOSCAN_LOG_INFO("tensor nbytes: {}", tensor->nbytes());
                }
            } else if (in_any_type == typeid(std::shared_ptr<ValueData>)) {

```

(continues on next page)

(continued from previous page)

```

    auto in_value = std::any_cast<std::shared_ptr<ValueData>>(in_any);
    HOLOSCAN_LOG_INFO("Received value: {}", in_value->data());
  } else {
    HOLOSCAN_LOG_ERROR("Invalid message type: {}", in_any_type.name());
  }
} catch (const std::bad_any_cast& e) {
  HOLOSCAN_LOG_ERROR("Failed to cast message - {}", e.what());
}
}
};

```

Receiving any number of inputs (C++)

Instead of assigning a specific number of input ports, it may be preferable to allow the ability to receive any number of objects on a port in certain situations.

Using `IOSpec::kAnySize` for variable input handling

One way to achieve this is to define a multi-receiver input port by calling `spec.input<std::vector<T>>("port_name", IOSpec::kAnySize)` with `IOSpec::kAnySize` as the second argument in the `setup()` method of the operator, where `T` is the type of the input data (as done for `PingRxOp` in the *native operator ping example*).

```

void setup(OperatorSpec& spec) override {
    spec.input<std::vector<std::shared_ptr<ValueData>>>("receivers", IOSpec::kAnySize);
}

```

Listing 12.3: examples/ping_multi_port/cpp/ping_multi_port.cpp

```

98 class PingRxOp : public Operator {
99     public:
100         HOLOSCAN_OPERATOR_FORWARD_ARGS(PingRxOp)
101
102         PingRxOp() = default;
103
104         void setup(OperatorSpec& spec) override {
105             // // Since Holoscan SDK v2.3, users can define a multi-receiver input port using
106             // 'spec.input()'
107             // // with 'IOSpec::kAnySize'.
108             // // The old way is to use 'spec.param()' with 'Parameter<std::vector<IOSpec*>>'
109             // receivers_';
110             // spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.",
111             // {});
112             spec.input<std::vector<std::shared_ptr<ValueData>>>("receivers", IOSpec::kAnySize);
113         }
114
115         void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {
116             auto value_vector =
117                 op_input.receive<std::vector<std::shared_ptr<ValueData>>>("receivers").value();
118         }
119     };

```

(continues on next page)

(continued from previous page)

```

116     HOLOSCAN_LOG_INFO("Rx message received (count: {}, size: {})", count_++, value_
    ↪vector.size());
117
118     HOLOSCAN_LOG_INFO("Rx message value1: {}", value_vector[0]->data());
119     HOLOSCAN_LOG_INFO("Rx message value2: {}", value_vector[1]->data());
120 };
121
122 private:
123     // // Since Holoscan SDK v2.3, the following line is no longer needed.
124     // Parameter<std::vector<IOSpec*>> receivers_;
125     int count_ = 1;
126 };
127
128 } // namespace holoscan::ops
129
130 class MyPingApp : public holoscan::Application {
131 public:
132     void compose() override {
133         using namespace holoscan;
134
135         // Define the tx, mx, rx operators, allowing the tx operator to execute 10 times
136         auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
137         auto mx = make_operator<ops::PingMxOp>("mx", Arg("multiplier", 3));
138         auto rx = make_operator<ops::PingRxOp>("rx");
139
140         // Define the workflow
141         add_flow(tx, mx, {{"out1", "in1"}, {"out2", "in2"}});
142         add_flow(mx, rx, {{"out1", "receivers"}, {"out2", "receivers"}});
143     }
144 };

```

Then, once the following configuration is provided in the `compose()` method,

```
add_flow(mx, rx, {{"out1", "receivers"}, {"out2", "receivers"}});
```

the `PingRxOp` will receive two inputs on the `receivers` port in the `compute()` method:

```
auto value_vector =
    op_input.receive<std::vector<std::shared_ptr<ValueData>>>("receivers").value();
```

Tip: When an input port is defined with `IOSpec::kAnySize`, the framework creates a new input port for each input object received on the port. The input ports are named using the format `<port_name>:<index>`, where `<port_name>` is the name of the input port and `<index>` is the index of the input object received on the port. For example, if the `receivers` port receives two input objects, the input ports will be named `receivers:0` and `receivers:1`.

The framework internally creates a parameter (`receivers`) with the type `std::vector<holoscan::IOSpec*>`, implicitly creates input ports (`receivers:0` and `receivers:1`), and connects them (adding references of the input ports to the `receivers` vector). This way, when the `receive()` method is called, the framework can return the input data from the corresponding input ports as a vector.

```
auto value_vector =
    op_input.receive<std::vector<std::shared_ptr<ValueData>>>("receivers").value();
```

If you add `HOLOSCAN_LOG_INFO(rx->description());` at the end of the `compose()` method, you will see the description of the `PingRxOp` operator as shown below:

```
id: -1
name: rx
fragment: ""
args:
  []
type: kNative
conditions:
  []
resources:
  []
spec:
  fragment: ""
  params:
    - name: receivers
      type: std::vector<holoscan::IOSpec*>
      description: ""
      flag: kNone
  inputs:
    - name: receivers:1
      io_type: kInput
      typeinfo_name: N8holoscan3gxf6EntityE
      connector_type: kDefault
      conditions:
        []
    - name: receivers:0
      io_type: kInput
      typeinfo_name: N8holoscan3gxf6EntityE
      connector_type: kDefault
      conditions:
        []
    - name: receivers
      io_type: kInput
      typeinfo_name: St6vectorISt10shared_ptrI9ValueDataESaIS2_EE
      connector_type: kDefault
      conditions:
        []
  outputs:
    []
```

Configuring input port queue size and message batch condition (C++)

If you want to receive multiple objects on a port and process them in batches, you can increase the queue size of the input port and set the `min_size` parameter of the `MessageAvailableCondition` condition to the desired batch size. This can be done by calling the `connector()` and `condition()` methods with the desired arguments, using the batch size as the `capacity` and `min_size` parameters, respectively.

Setting `min_size` to `N` will ensure that the operator receives `N` objects before the `compute()` method is called.

```
void setup(holoscan::OperatorSpec& spec) override {
    spec.input<std::shared_ptr<ValueData>>("receivers")
        .connector(holoscan::IOSpec::ConnectorType::kDoubleBuffer,
            holoscan::Arg("capacity", static_cast<uint64_t>(2)))
        .condition(holoscan::ConditionType::kMessageAvailable,
            holoscan::Arg("min_size", static_cast<uint64_t>(2)));
}
```

Then, the `receive()` method can be called with the `receivers` port name to receive input data in batches.

```
void compute(holoscan::InputContext& op_input, holoscan::OutputContext&,
    holoscan::ExecutionContext&) override {
    std::vector<std::shared_ptr<ValueData>> value_vector;
    auto maybe_value = op_input.receive<std::shared_ptr<ValueData>>("receivers");
    while (maybe_value) {
        value_vector.push_back(maybe_value.value());
        maybe_value = op_input.receive<std::shared_ptr<ValueData>>("receivers");
    }

    HOLOSCAN_LOG_INFO("Rx message received (size: {})", value_vector.size());
}
```

In the above example, the operator receives input on a port called “`receivers`” with a queue size of 2 and a `min_size` of 2. The `receive()` method is called in a loop to receive the input data in batches of 2. Since the operator does not know the number of objects to be received in advance, the `receive()` method is called in a loop until it returns an error. The input data is stored in a vector, and the size of the vector is logged after all the input data is received.

To simplify the above code, the Holoscan SDK provides a `IOSpec::kPrecedingCount` constant as a second argument to the `OperatorSpec`’s `input()` method to specify the number of preceding connections to the input port (in this case, the number of connections to the `receivers` port is 2) as the batch size. This can be used to receive the input data in batches without the need to call the `receive()` method in a loop.

```
void setup(holoscan::OperatorSpec& spec) override {
    spec.input<std::vector<std::shared_ptr<ValueData>>>("receivers",
    ↪ holoscan::IOSpec::kPrecedingCount);
}
```

Then, the `receive()` method can be called with the `receivers` port name to receive the input data in batches.

```
void compute(holoscan::InputContext& op_input, holoscan::OutputContext&,
    holoscan::ExecutionContext&) override {
    auto value_vector =
        op_input.receive<std::vector<std::shared_ptr<ValueData>>>("receivers").value();

    HOLOSCAN_LOG_INFO("Rx message received (size: {})", value_vector.size());

    HOLOSCAN_LOG_INFO("Rx message value1: {}", value_vector[0]->data());
    HOLOSCAN_LOG_INFO("Rx message value2: {}", value_vector[1]->data());
}
```

In the above example, the operator receives input on a port called “`receivers`” with a batch size of 2. The `receive()` method is called with the `receivers` port name to receive the input data in batches of 2. The input data is stored in a vector, and the size of the vector is logged after all the input data has been received.

If you want to use a specific batch size, you can use `holoscan::IOSpec::IOSize(int64_t)` instead of

`holoscan::IOSpec::kPrecedingCount` to specify the batch size. Using `IOSize` in this way is equivalent to the more verbose `condition()` and `connector()` calls to update the capacity and `min_size` arguments shown near the start of this section.

The main reason to use `condition()` or `connector()` methods instead of the shorter `IOSize` is if additional parameter changes, such as the queue policy, need to be made. See more details on the use of the `condition()` and `connector()` methods in the advanced topics section below (*Further customizing inputs and outputs*).

```
void setup(holoscan::OperatorSpec& spec) override {
    spec.input<std::vector<std::shared_ptr<ValueData>>>("receivers",
↪holoscan::IOSpec::IOSize(2));
}
```

If you want to receive the input data one by one, you can call the `receive()` method without using the `std::vector<T>` template argument.

```
void compute(holoscan::InputContext& op_input, holoscan::OutputContext&,
             holoscan::ExecutionContext&) override {
    while (true) {
        auto maybe_value = op_input.receive<std::shared_ptr<ValueData>>("receivers");
        if (!maybe_value) { break; }
        auto& value = maybe_value.value();
        // Process the input data
        HOLOSCAN_LOG_INFO("Rx message received (value: {})", value->data());
    }
}
```

The above code will receive input data one by one from the `receivers` port. The `receive()` method is called in a loop until it returns an error. The input data is stored in a variable, and the value of the input data is logged.

Note: This approach (receiving the input data one by one) is not applicable for the `holoscan::IOSpec::kAnySize` case. With the `holoscan::IOSpec::kAnySize` argument, the framework creates a new input port for each input object received on the port internally. Each implicit input port (named using the format `<port_name>:<index>`) is associated with a `MessageAvailableCondition` condition that has a `min_size` of 1. Therefore, the `receive()` method needs to be called with the `std::vector<T>` template argument to receive the input data in batches at once.

If you really need to receive the input data one by one for `holoscan::IOSpec::kAnySize` case (though it is not recommended), you can receive the input data from each implicit input port (named `<port_name>:<index>`) one by one using the `receive()` method without the `std::vector<T>` template argument. (e.g., `op_input.receive<std::shared_ptr<ValueData>>("receivers:0")`, `op_input.receive<std::shared_ptr<ValueData>>("receivers:1")`, etc.). To avoid the error message (such as `The operator does not have an input port with label 'receivers:X'`) when calling the `receive()` method for the implicit input port, you need to calculate the number of connections to the `receivers` port in advance and call the `receive()` method for each implicit input port accordingly.

```
void compute(holoscan::InputContext& op_input, holoscan::OutputContext&,
             holoscan::ExecutionContext&) override {
    int input_count = spec()->inputs().size() - 1; // -1 to exclude the 'receivers'
↪input port
    for (int i = 0; i < input_count; i++) {
        auto maybe_value =
            op_input.receive<std::shared_ptr<ValueData>>(fmt::format("receivers:{}", i).c_
↪str());
        if (!maybe_value) { break; }
    }
}
```

(continues on next page)

(continued from previous page)

```

    auto& value = maybe_value.value();
    // Process the input data
    HOLOSCAN_LOG_INFO("Rx message received (value: {})", value->data());
}
}

```

Attention: Using `IOSpec::kPrecedingCount` or `IOSpec::IOSize(int64_t)` appears to show the same behavior as `IOSpec::kAnySize` in the above example. However, the difference is that since `IOSpec::kPrecedingCount` or `IOSpec::IOSize(int64_t)` doesn't use separate `MessageAvailableCondition` conditions for each (internal) input port, it is not guaranteed that the operator will receive the input data in order.

This means the operator may receive the input data in a different order than the order in which the connections are made in the `compose()` method. Additionally, with the multithread scheduler, it is not guaranteed that the operator will receive the input data from each of the connections uniformly. The operator may receive more input data from one connection than from another.

If the order of the input data is important, it is recommended to use `IOSpec::kAnySize` and call the `receive()` method with the `std::vector<T>` template argument to receive the input data in batches at once.

Please see the [C++ system test cases](#) for more examples of receiving multiple inputs in C++ operators.

Building your C++ operator

You can build your C++ operator using CMake, by calling `find_package(holoscan)` in your `CMakeLists.txt` to load the SDK libraries. Your operator will need to link against `holoscan::core`:

Listing 12.4: <src_dir>/CMakeLists.txt

```

# Your CMake project
cmake_minimum_required(VERSION 3.20)
project(my_project CXX)

# Finds the holoscan SDK
find_package(holoscan REQUIRED CONFIG PATHS "/opt/nvidia/holoscan")

# Create a library for your operator
add_library(my_operator SHARED my_operator.cpp)

# Link your operator against holoscan::core
target_link_libraries(my_operator
    PUBLIC holoscan::core
)

```

Once your `CMakeLists.txt` is ready in <src_dir>, you can build in <build_dir> with the command line below. You can optionally pass `Holoscan_ROOT` if the SDK installation you'd like to use differs from the `PATHS` given to `find_package(holoscan)` above.

```

# Configure
cmake -S <src_dir> -B <build_dir> -D Holoscan_ROOT="/opt/nvidia/holoscan"

```

(continues on next page)

(continued from previous page)

```
# Build
cmake --build <build_dir> -j
```

Using your C++ Operator in an Application

- **If the application is configured in the same CMake project as the operator**, you can simply add the operator CMake target library name under the application executable `target_link_libraries` call, as the operator CMake target is already defined.

```
# operator
add_library(my_op my_op.cpp)
target_link_libraries(my_operator PUBLIC holoscan::core)

# application
add_executable(my_app main.cpp)
target_link_libraries(my_operator
    PRIVATE
    holoscan::core
    my_op
)
```

- **If the application is configured in a separate project as the operator**, you need to [export the operator](#) in its own CMake project, and import it in the application CMake project, before being able to list it under `target_link_libraries` also. This is the same as what is done for the SDK [built-in operators](#), available under the `holoscan::ops` namespace.

You can then include the headers to your C++ operator in your application code.

12.1.2 GXF Operators

With the Holoscan C++ API, we can also wrap [GXF Codelets](#) from GXF extensions as Holoscan Operators.

Note: If you do not have an existing GXF extension, we recommend developing native operators using the [C++](#) or [Python](#) APIs to skip the need for wrapping GXF codelets as operators. If you do need to create a GXF Extension, follow the [Creating a GXF Extension](#) section for a detailed explanation of the GXF extension development process.

Tip: The manual codelet wrapping mechanism described below is no longer necessary in order to make use of a GXF Codelet as a Holoscan operator. There is a new `GXFCodeletOp` which allows directly using an existing GXF codelet via `Fragment::make_operator` without having to first create a wrapper class for it. Similarly there is now also a `GXFComponentResource` class which allows a GXF Component to be used as a Holoscan resource via `Fragment::make_resource`. A detailed example of how to use each of these is provided for both C++ and Python applications in the [examples/import_gxf_components](#) folder.

Given an existing GXF extension, we can create a simple “identity” application consisting of a replayer, which reads contents from a file on disk, and our recorder from the last section, which will store the output of the replayer exactly in the same format. This allows us to see whether the output of the recorder matches the original input files.

The `MyRecorderOp` Holoscan Operator implementation below will wrap the `MyRecorder` GXF Codelet shown [here](#).

Operator definition

Listing 12.5: my_recorder_op.hpp

```

1  #ifndef APPS_MY_RECORDER_APP_MY_RECORDER_OP_HPP
2  #define APPS_MY_RECORDER_APP_MY_RECORDER_OP_HPP
3
4  #include "holoscan/core/gxf/gxf_operator.hpp"
5
6  namespace holoscan::ops {
7
8  class MyRecorderOp : public holoscan::ops::GXFOperator {
9  public:
10     HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER(MyRecorderOp, holoscan::ops::GXFOperator)
11
12     MyRecorderOp() = default;
13
14     const char* gxf_type_name() const override { return "MyRecorder"; }
15
16     void setup(OperatorSpec& spec) override;
17
18     void initialize() override;
19
20 private:
21     Parameter<holoscan::IOSpec*> receiver_;
22     Parameter<std::shared_ptr<holoscan::Resource>> my_serializer_;
23     Parameter<std::string> directory_;
24     Parameter<std::string> basename_;
25     Parameter<bool> flush_on_tick_;
26 };
27
28 } // namespace holoscan::ops
29
30 #endif /* APPS_MY_RECORDER_APP_MY_RECORDER_OP_HPP */

```

The `holoscan::ops::MyRecorderOp` class wraps a `MyRecorder` GXF Codelet by inheriting from the `holoscan::ops::GXFOperator` class. The `HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER` macro is used to forward the arguments of the constructor to the base class.

We first need to define the fields of the `MyRecorderOp` class. You can see that fields with the same names are defined in both the `MyRecorderOp` class and the `MyRecorder` GXF codelet.

Listing 12.6: Parameter declarations in
gfx_extensions/my_recorder/my_recorder.hpp

```

22  nvidia::gfx::Parameter<nvidia::gfx::Handle<nvidia::gfx::Receiver>> receiver_;
23  nvidia::gfx::Parameter<nvidia::gfx::Handle<nvidia::gfx::EntitySerializer>> my_
    serializer_;
24  nvidia::gfx::Parameter<std::string> directory_;
25  nvidia::gfx::Parameter<std::string> basename_;
26  nvidia::gfx::Parameter<bool> flush_on_tick_;

```

Comparing the MyRecorderOp holoscan parameter to the MyRecorder gfx codelet:

Holoscan Operator	GXF Codelet
holoscan::Parameter	nvidia::gfx::Parameter
holoscan::IOSpec*	nvidia::gfx::Handle<nvidia::gfx::Receiver>> or nvidia::gfx::Handle<nvidia::gfx::Transmitter>>
std::shared_ptr<holoscan::Resolver>	nvidia::gfx::Handle<T>> example: T is nvidia::gfx::EntitySerializer

We then need to implement the following functions:

- `const char* gfx_type_name()` `const` override: return the GFX type name of the Codelet. The fully-qualified class name (MyRecorder) for the GFX Codelet is specified.
- `void setup(OperatorSpec& spec)` `override`: setup the OperatorSpec with the inputs/outputs and parameters of the Operator.
- `void initialize()` `override`: initialize the Operator.

Setting up parameter specifications

The implementation of the `setup(OperatorSpec& spec)` function is as follows:

Listing 12.7: my_recorder_op.cpp

```

1  #include "../my_recorder_op.hpp"
2
3  #include "holoscan/core/fragment.hpp"
4  #include "holoscan/core/gxf/entity.hpp"
5  #include "holoscan/core/operator_spec.hpp"
6
7  #include "holoscan/core/resources/gxf/video_stream_serializer.hpp"
8
9  namespace holoscan::ops {
10
11  void MyRecorderOp::setup(OperatorSpec& spec) {
12      auto& input = spec.input<holoscan::gfx::Entity>("input");
13      // Above is same with the following two lines (a default condition is assigned to the
    // input port if not specified):
14      //
15      // auto& input = spec.input<holoscan::gfx::Entity>("input")
16      // .condition(ConditionType::kMessageAvailable, Arg("min_size") =
    // static_cast<uint64_t>(1));

```

(continues on next page)

(continued from previous page)

```

17 spec.param(receiver_, "receiver", "Entity receiver", "Receiver channel to log", &
18 ↪input);
19 spec.param(my_serializer_,
20             "serializer",
21             "Entity serializer",
22             "Serializer for serializing input data");
23 spec.param(directory_, "out_directory", "Output directory path", "Directory path to
24 ↪store received output");
25 spec.param(basename_, "basename", "File base name", "User specified file name without
26 ↪extension");
27 spec.param(flush_on_tick_,
28             "flush_on_tick",
29             "Boolean to flush on tick",
30             "Flushes output buffer on every `tick` when true",
31             false);
32 }
33 void MyRecorderOp::initialize() {...}
34 } // namespace holoscan::ops

```

Here, we set up the inputs/outputs and parameters of the Operator. Note how the content of this function is very similar to the MyRecorder GXF codelet's *registerInterface* function.

- In the C++ API, GXF Receiver and Transmitter components (such as DoubleBufferReceiver and DoubleBufferTransmitter) are considered as input and output ports of the Operator so we register the inputs/outputs of the Operator with `input<T>` and `output<T>` functions (where T is the data type of the port).
- Compared to the pure *GXF application* that does the same job, the *SchedulingTerm* of an Entity in the *GXF Application YAML* are specified as Conditions on the input/output ports (e.g., `holoscan::MessageAvailableCondition` and `holoscan::DownstreamMessageAffordableCondition`).

The highlighted lines in `MyRecorderOp::setup` above match the following highlighted statements of *GXF Application YAML*:

Listing 12.8: A part of `apps/my_recorder_app_gxf/my_recorder_gxf.yaml`

```

35 name: recorder
36 components:
37   - name: input
38     type: nvidia::gxf::DoubleBufferReceiver
39   - name: allocator
40     type: nvidia::gxf::UnboundedAllocator
41   - name: component_serializer
42     type: nvidia::gxf::StdComponentSerializer
43     parameters:
44       allocator: allocator
45   - name: entity_serializer
46     type: nvidia::gxf::StdEntitySerializer
47     parameters:
48       component_serializers: [component_serializer]
49   - type: MyRecorder

```

(continues on next page)

(continued from previous page)

```

50     parameters:
51         receiver: input
52         serializer: entity_serializer
53         out_directory: "/tmp"
54         basename: "tensor_out"
55 - type: nvidia::gxf::MessageAvailableSchedulingTerm
56     parameters:
57         receiver: input
58         min_size: 1

```

In the same way, if we had a Transmitter GXF component, we would have the following statements (Please see available constants for `holoscan::ConditionType`):

```

auto& output = spec.output<holoscan::gxf::Entity>("output");
// Above is same with the following two lines (a default condition is assigned to the
// output port if not specified):
//
//     auto& output = spec.output<holoscan::gxf::Entity>("output")
//                               .condition(ConditionType::kDownstreamMessageAffordable, Arg(
// "min_size") = static_cast<uint64_t>(1));

```

Initializing the operator

Next, the implementation of the `initialize()` function is as follows:

Listing 12.9: `my_recorder_op.cpp`

```

1  #include "../my_recorder_op.hpp"
2
3  #include "holoscan/core/fragment.hpp"
4  #include "holoscan/core/gxf/entity.hpp"
5  #include "holoscan/core/operator_spec.hpp"
6
7  #include "holoscan/core/resources/gxf/video_stream_serializer.hpp"
8
9  namespace holoscan::ops {
10
11  void MyRecorderOp::setup(OperatorSpec& spec) {...}
12
13  void MyRecorderOp::initialize() {
14      // Set up prerequisite parameters before calling GXFOperator::initialize()
15      auto frag = fragment();
16      auto serializer =
17          frag->make_resource<holoscan::StdEntitySerializer>("serializer");
18      add_arg(Arg("serializer") = serializer);
19
20      GXFOperator::initialize();
21  }
22
23  } // namespace holoscan::ops

```

Here we set up the pre-defined parameters such as the serializer. The highlighted lines above matches the high-

lighted statements of *GXF Application YAML*:

Listing 12.10: Another part of apps/my_recorder_app_gxf/my_recorder_gxf.yaml

```

35 name: recorder
36 components:
37   - name: input
38     type: nvidia::gxf::DoubleBufferReceiver
39   - name: allocator
40     type: nvidia::gxf::UnboundedAllocator
41   - name: component_serializer
42     type: nvidia::gxf::StdComponentSerializer
43     parameters:
44       allocator: allocator
45   - name: entity_serializer
46     type: nvidia::gxf::StdEntitySerializer
47     parameters:
48       component_serializers: [component_serializer]
49   - type: MyRecorder
50     parameters:
51       receiver: input
52       serializer: entity_serializer
53       out_directory: "/tmp"
54       basename: "tensor_out"
55   - type: nvidia::gxf::MessageAvailableSchedulingTerm
56     parameters:
57       receiver: input
58       min_size: 1

```

Note: The Holoscan C++ API already provides the `holoscan::StdEntitySerializer` class which wraps the `nvidia::gxf::StdEntitySerializer` GXF component, used here as `serializer`.

Building your GXF operator

There are no differences in CMake between building a GXF operator and *building a native C++ operator*, since the GXF codelet is actually loaded through a GXF extension as a plugin, and does not need to be added to `target_link_libraries(my_operator ...)`.

Using your GXF Operator in an Application

There are no differences in CMake between using a GXF operator and *using a native C++ operator in an application*. However, the application will need to load the GXF extension library which holds the wrapped GXF codelet symbols, so the application needs to be configured to find the extension library in its yaml configuration file, as documented [here](#).

12.1.3 Interoperability between GXF and native C++ operators

To support sending or receiving tensors to and from operators (both GXF and native C++ operators), the Holoscan SDK provides the C++ classes below:

- A class template called `holoscan::Map` which inherits from `std::unordered_map<std::string, std::shared_ptr<T>>`. The template parameter `T` can be any type, and it is used to specify the type of the `std::shared_ptr` objects stored in the map.
- A `holoscan::TensorMap` class defined as a specialization of `holoscan::Map` for the `holoscan::Tensor` type.

When a message with a `holoscan::TensorMap` is emitted from a native C++ operator, the message object is always converted to a `holoscan::gxf::Entity` object and sent to the downstream operator.

Then, if the sent GXF Entity object holds only Tensor object(s) as its components, the downstream operator can receive the message data as a `holoscan::TensorMap` object instead of a `holoscan::gxf::Entity` object.

Fig. 12.2 shows the relationship between the `holoscan::gxf::Entity` and `nvidia::gxf::Entity` classes and the relationship between the `holoscan::Tensor` and `nvidia::gxf::Tensor` classes.

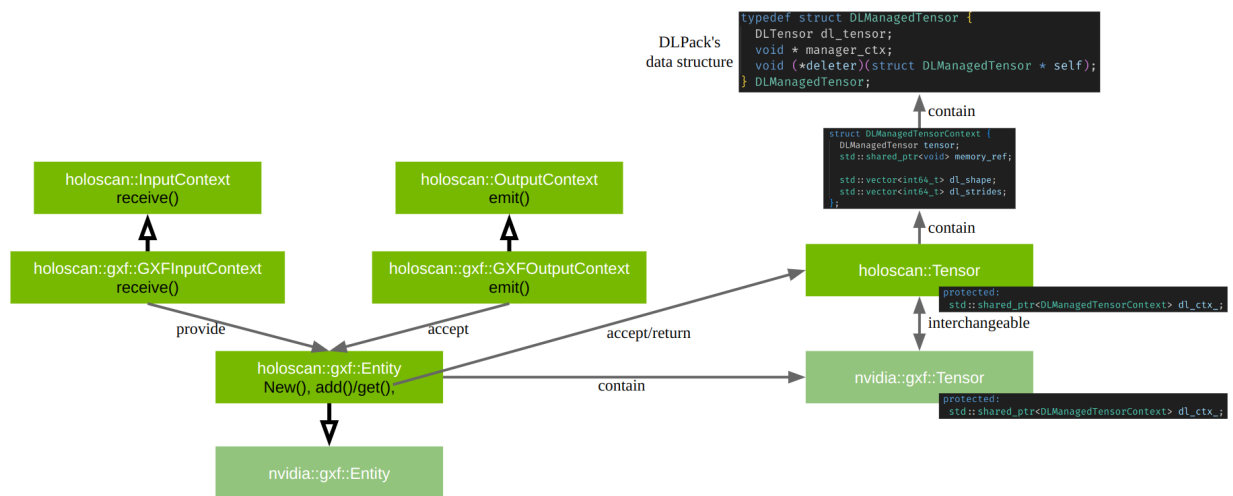


Fig. 12.2: Supporting Tensor Interoperability

Both `holoscan::gxf::Tensor` and `nvidia::gxf::Tensor` are interoperable with each other because they are wrappers around the same underlying `DLManagedTensorContext` struct holding a `DLManagedTensor` object.

The `holoscan::TensorMap` class is used to store multiple tensors in a map, where each tensor is associated with a unique key. The `holoscan::TensorMap` class is used to pass multiple tensors between operators, and it is used in the same way as a `std::unordered_map<std::string, std::shared_ptr<holoscan::Tensor>>` object.

Since both `holoscan::TensorMap` and `holoscan::gxf::Entity` objects hold tensors which are interoperable, the message data between GXF and native C++ operators are also interoperable.

Fig. 12.3 illustrates the use of the `holoscan::TensorMap` class to pass multiple tensors between operators. The `GXFSendTensorOp` operator sends a `nvidia::gfx::Entity` object (containing a `nvidia::gfx::Tensor` object as a GXF component named “tensor”) to the `ProcessTensorOp` operator, which processes the tensors and then forwards the processed tensors to the `GXFReceiveTensorOp` operator.

Consider the following example, where `GXFSendTensorOp` and `GXFReceiveTensorOp` are GXF operators, and where `ProcessTensorOp` is a Holoscan native operator in C++:

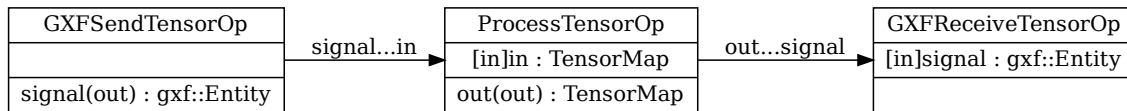


Fig. 12.3: The tensor interoperability between C++ native operator and GXF operator

The following code shows how to implement `ProcessTensorOp`’s `compute()` method as a C++ native operator communicating with GXF operators. Focus on the use of the `holoscan::gfx::Entity`:

Listing 12.11: examples/tensor_interop/cpp/tensor_interop.cpp

```

86 void compute(InputContext& op_input, OutputContext& op_output,
87             ExecutionContext& context) override {
88     // The type of `in_message` is 'holoscan::TensorMap'.
89     auto in_message = op_input.receive<holoscan::TensorMap>("in").value();
90     // The type of out_message is TensorMap
91     TensorMap out_message;
92
93     for (auto& [key, tensor] : in_message) { // Process with 'tensor' here.
94         cudaError_t cuda_status;
95         size_t data_size = tensor->nbytes();
96         std::vector<uint8_t> in_data(data_size);
97         CUDA_TRY(cudaMemcpy(in_data.data(), tensor->data(), data_size,
98         ↪ cudaMemcpyDeviceToHost));
99         HOLOSCAN_LOG_INFO("ProcessTensorOp Before key: '{}', shape: ({}), data: [{}]",
100                          key,
101                          fmt::join(tensor->shape(), ","),
102                          fmt::join(in_data, ","));
103         for (size_t i = 0; i < data_size; i++) { in_data[i] *= 2; }
104         HOLOSCAN_LOG_INFO("ProcessTensorOp After key: '{}', shape: ({}), data: [{}]",
105                          key,
106                          fmt::join(tensor->shape(), ","),
107                          fmt::join(in_data, ","));
108         CUDA_TRY(cudaMemcpy(tensor->data(), in_data.data(), data_size,
109         ↪ cudaMemcpyHostToDevice));
110         out_message.insert({key, tensor});
111     }
112     // Send the processed message.
113     op_output.emit(out_message);
114 }

```

- The input message is of type `holoscan::TensorMap` object.

- Every `holoscan::Tensor` in the `TensorMap` object is copied on the host as `in_data`.
- The data is processed (values multiplied by 2)
- The data is moved back to the `holoscan::Tensor` object on the GPU.
- A new `holoscan::TensorMap` object `out_message` is created to be sent to the next operator with `op_output.emit()`.

Note: A complete example of the C++ native operator that supports interoperability with GXF operators is available in the [examples/tensor_interop/cpp](#) directory.

12.2 Python Operators

When assembling a Python application, two types of operators can be used:

1. **Native Python operators:** custom operators defined in Python, by creating a subclass of `holoscan.core.Operator`. These Python operators can pass arbitrary Python objects around between operators and are not restricted to the stricter parameter typing used for C++ API operators.
2. **Python wrappings of C++ Operators:** operators defined in the underlying C++ library by inheriting from the `holoscan::Operator` class. These operators have Python bindings available within the `holoscan.operators` module. Examples are `VideoStreamReplayerOp` for replaying video files, `FormatConverterOp` for format conversions, and `HolovizOp` for visualization.

Note: It is possible to create an application using a mixture of Python wrapped C++ operators and native Python operators. In this case, some special consideration to cast the input and output tensors appropriately must be taken, as shown in [a section below](#).

12.2.1 Native Python Operator

Operator Lifecycle (Python)

The lifecycle of a `holoscan.core.Operator` is made up of three stages:

- `start()` is called once when the operator starts, and is used for initializing heavy tasks such as allocating memory resources and using parameters.
- `compute()` is called when the operator is triggered, which can occur any number of times throughout the operator lifecycle between `start()` and `stop()`.
- `stop()` is called once when the operator is stopped, and is used for deinitializing heavy tasks such as deallocating resources that were previously assigned in `start()`.

All operators on the workflow are scheduled for execution. When an operator is first executed, the `start()` method is called, followed by the `compute()` method. When the operator is stopped, the `stop()` method is called. The `compute()` method is called multiple times between `start()` and `stop()`.

If any of the scheduling conditions specified by [Conditions](#) are not met (for example, the `CountCondition` would cause the scheduling condition to not be met if the operator has been executed a certain number of times), the operator is stopped and the `stop()` method is called.

We will cover how to use `Conditions` in the [Specifying operator inputs and outputs \(Python\)](#) section of the user guide.

Typically, the `start()` and the `stop()` functions are only called once during the application's lifecycle. However, if the scheduling conditions are met again, the operator can be scheduled for execution, and the `start()` method will be called again.

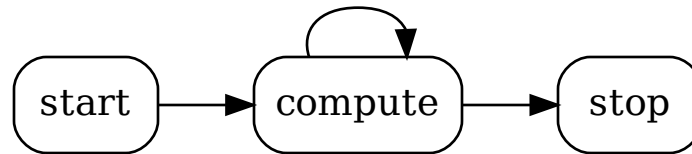


Fig. 12.4: The sequence of method calls in the lifecycle of a Holoscan Operator

We can override the default behavior of the operator by implementing the above methods. The following example shows how to implement a custom operator that overrides start, stop and compute methods.

Listing 12.12: The basic structure of a Holoscan Operator (Python)

```

1  from holoscan.core import (
2      ExecutionContext,
3      InputContext,
4      Operator,
5      OperatorSpec,
6      OutputContext,
7  )
8
9
10 class MyOp(Operator):
11
12     def __init__(self, fragment, *args, **kwargs):
13         super().__init__(fragment, *args, **kwargs)
14
15     def setup(self, spec: OperatorSpec):
16         pass
17
18     def start(self):
19         pass
20
21     def compute(self, op_input: InputContext, op_output: OutputContext, context:
22 ↪ExecutionContext):
23         pass
24
25     def stop(self):
26         pass

```

setup() method vs initialize() vs __init__()

The `setup()` method aims to get the “operator’s spec” by providing a `OperatorSpec` object as a spec param. When `__init__()` is called, it calls C++’s `Operator::spec` method (and also sets the `self.spec` class member) and calls the `setup` method so that the Operator’s spec property holds the operator’s specification. (See the [source code](#) for more details.)

Since the `setup()` method can be called multiple times with other `OperatorSpec` objects (e.g., to enumerate the operator’s description), in the `setup()` method, a user shouldn’t initialize something. Such initialization needs to be done by overriding the `initialize()` method.

```
def initialize(self):
    pass
```

The `__init__()` method is for creating the Operator object and can be used to initialize the operator object itself by passing various arguments. Note that it doesn’t initialize the corresponding GXF entity object. The underlying GXF entity object is initialized when the operator is scheduled for execution.

Please do not forget to call the base class constructor (`super().__init__(fragment, *args, **kwargs)`) at the end of the `__init__` method.

Creating a custom operator (Python)

To create a custom operator in Python it is necessary to create a subclass of `holoscan.core.Operator`. A simple example of an operator that takes a time-varying 1D input array named “signal” and applies convolution with a boxcar (i.e. rect) kernel.

For simplicity, this operator assumes that the “signal” that will be received on the input is already a `numpy.ndarray` or is something that can be cast to one via (`np.asarray`). We will see more details in a later section on how we can interoperate with various tensor classes, including the GXF Tensor objects used by some of the C++-based operators.

Code Snippet: [examples/numpy_native/convolve.py](#)

Listing 12.13: `examples/numpy_native/convolve.py`

```
16 import os
17
18 from holoscan.conditions import CountCondition
19 from holoscan.core import Application, Operator, OperatorSpec
20 from holoscan.logger import LogLevel, set_log_level
21
22 import numpy as np
23
24
25 class SignalGeneratorOp(Operator):
26     """Generate a time-varying impulse.
27
28     Transmits an array of zeros with a single non-zero entry of a
29     specified `height`. The position of the non-zero entry shifts
30     to the right (in a periodic fashion) each time `compute` is
31     called.
32
33     Parameters
34     -----
35     fragment : holoscan.core.Fragment
```

(continues on next page)

(continued from previous page)

```

36     The Fragment (or Application) the operator belongs to.
37     height : number
38         The height of the signal impulse.
39     size : number
40         The total number of samples in the generated 1d signal.
41     dtype : numpy.dtype or str
42         The data type of the generated signal.
43     """
44
45     def __init__(self, fragment, *args, height=1, size=10, dtype=np.int32, **kwargs):
46         self.count = 0
47         self.height = height
48         self.dtype = dtype
49         self.size = size
50         super().__init__(fragment, *args, **kwargs)
51
52     def setup(self, spec: OperatorSpec):
53         spec.output("signal")
54
55     def compute(self, op_input, op_output, context):
56
57         # single sample wide impulse at a time-varying position
58         signal = np.zeros((self.size,), dtype=self.dtype)
59         signal[self.count % signal.size] = self.height
60         self.count += 1
61
62         op_output.emit(signal, "signal")
63
64
65     class ConvolveOp(Operator):
66         """Apply convolution to a tensor.
67
68         Convolves an input signal with a "boxcar" (i.e. "rect") kernel.
69
70         Parameters
71         -----
72         fragment : holoscan.core.Fragment
73             The Fragment (or Application) the operator belongs to.
74         width : number
75             The width of the boxcar kernel used in the convolution.
76         unit_area : bool, optional
77             Whether or not to normalize the convolution kernel to unit area.
78             If False, all samples have implitude one and the dtype of the
79             kernel will match that of the signal. When True the sum over
80             the kernel is one and a 32-bit floating point data type is used
81             for the kernel.
82         """
83
84         def __init__(self, fragment, *args, width=4, unit_area=False, **kwargs):
85             self.count = 0
86             self.width = width
87             self.unit_area = unit_area

```

(continues on next page)

(continued from previous page)

```

88     super().__init__(fragment, *args, **kwargs)
89
90     def setup(self, spec: OperatorSpec):
91         spec.input("signal_in")
92         spec.output("signal_out")
93
94     def compute(self, op_input, op_output, context):
95
96         signal = op_input.receive("signal_in")
97         assert isinstance(signal, np.ndarray)
98
99         if self.unit_area:
100             kernel = np.full((self.width,), 1/self.width, dtype=np.float32)
101         else:
102             kernel = np.ones((self.width,), dtype=signal.dtype)
103
104         convolved = np.convolve(signal, kernel, mode='same')
105
106         op_output.emit(convolved, "signal_out")
107
108
109 class PrintSignalOp(Operator):
110     """Print the received signal to the terminal."""
111
112     def setup(self, spec: OperatorSpec):
113         spec.input("signal")
114
115     def compute(self, op_input, op_output, context):
116         signal = op_input.receive("signal")
117         print(signal)
118
119
120 class ConvolveApp(Application):
121     """Minimal signal processing application.
122
123     Generates a time-varying impulse, convolves it with a boxcar kernel, and
124     prints the result to the terminal.
125
126     A `CountCondition` is applied to the generate to terminate execution
127     after a specific number of steps.
128     """
129
130     def compose(self):
131         signal_generator = SignalGeneratorOp(
132             self,
133             CountCondition(self, count=24),
134             name="generator",
135             **self.kwargs("generator"),
136         )
137         convolver = ConvolveOp(self, name="conv", **self.kwargs("convolve"))
138         printer = PrintSignalOp(self, name="printer")
139         self.add_flow(signal_generator, convolver)

```

(continues on next page)

(continued from previous page)

```

140         self.add_flow(convolver, printer)
141
142
143     def main(config_file):
144         app = ConvolveApp()
145         # if the --config command line argument was provided, it will override this config_
146         ↪ file`
147         app.config(config_file)
148         app.run()
149
150     if __name__ == "__main__":
151         config_file = os.path.join(os.path.dirname(__file__), 'convolve.yaml')
152         main(config_file=config_file)

```

Code Snippet: `examples/numpy_native/convolve.yaml`

Listing 12.14: `examples/numpy_native/convolve.yaml`

```

17 signal_generator:
18     height: 1
19     size: 20
20     dtype: int32
21
22 convolve:
23     width: 4
24     unit_area: false

```

In this application, three native Python operators are created: `SignalGeneratorOp`, `ConvolveOp` and `PrintSignalOp`. The `SignalGeneratorOp` generates a synthetic signal such as `[0, 0, 1, 0, 0, 0]` where the position of the non-zero entry varies each time it is called. `ConvolveOp` performs a 1D convolution with a boxcar (i.e. rect) function of a specified width. `PrintSignalOp` just prints the received signal to the terminal.

As covered in more detail below, the inputs to each operator are specified in the `setup()` method of the operator. Then inputs are received within the `compute` method via `op_input.receive()` and outputs are emitted via `op_output.emit()`.

Note that for native Python operators as defined here, any Python object can be emitted or received. When transmitting between operators, a shared pointer to the object is transmitted rather than a copy. In some cases, such as sending the same tensor to more than one downstream operator, it may be necessary to avoid in-place operations on the tensor in order to avoid any potential race conditions between operators.

Specifying operator parameters (Python)

In the example `SignalGeneratorOp` operator above, we added three keyword arguments in the operator's `__init__` method, used inside the `compose()` method of the operator to adjust its behavior:

```

def __init__(self, fragment, *args, width=4, unit_area=False, **kwargs):
    # Internal counter for the time-dependent signal generation
    self.count = 0

    # Parameters
    self.width = width

```

(continues on next page)

(continued from previous page)

```

self.unit_area = unit_area

# To forward remaining arguments to any underlying C++ Operator class
super().__init__(fragment, *args, **kwargs)

```

Note: As an alternative closer to C++, these parameters can be added through the `OperatorSpec` attribute of the operator in its `setup()` method, where an associated string key must be provided as well as a default value.

```

def setup(self, spec: OperatorSpec):
    spec.param("width", 4)
    spec.param("unit_area", False)

```

The parameters can then be accessed on the `self` object in the operator's methods (including `initialize()`, `start()`, `compute()`, `stop()`) as `self.width` and `self.unit_area`.

Other kwargs properties can also be passed to `spec.param`, such as `headline`, `description` (used by GXF applications), or `kind` (used when *Receiving any number of inputs (Python)*, which is deprecated since v2.3.0).

Note: Native operator parameters added via either of these methods must **not** have a name that overlaps with any of the existing attribute or method names of the base `Operator` class.

See the [Configuring operator parameters](#) section to learn how an application can set these parameters.

Specifying operator inputs and outputs (Python)

To configure the input(s) and output(s) of Python native operators, call the `spec.input()` and `spec.output()` methods within the `setup()` method of the operator.

The `spec.input()` and `spec.output()` methods should be called once for each input and output to be added. The `holoscan.core.OperatorSpec` object and the `setup()` method will be initialized and called automatically by the Application class when its `run()` method is called.

These methods (`spec.input()` and `spec.output()`) return an `IOSpec` object that can be used to configure the input/output port.

By default, the `holoscan.conditions.MessageAvailableCondition` and `holoscan.conditions.DownstreamMessageAffordableCondition` conditions are applied (with a `min_size` of 1) to the input/output ports. This means that the operator's `compute()` method will not be invoked until a message is available on the input port and the downstream operator's input port (queue) has enough capacity to receive the message.

```

def setup(self, spec: OperatorSpec):
    spec.input("in")
    # Above statement is equivalent to:
    # spec.input("in")
    # .condition(ConditionType.MESSAGE_AVAILABLE, min_size = 1)
    spec.output("out")
    # Above statement is equivalent to:
    # spec.output("out")
    # .condition(ConditionType.DOWNSTREAM_MESSAGE_AFFORDABLE, min_size = 1)

```

In the above example, the `spec.input()` method is used to configure the input port to have the `holoscan.conditions.MessageAvailableCondition` with a minimum size of 1. This means that the operator's `compute()` method will not be invoked until a message is available on the input port of the operator. Similarly, the `spec.output()` method is used to configure the output port to have a `holoscan.conditions.DownstreamMessageAffordableCondition` with a minimum size of 1. This means that the operator's `compute()` method will not be invoked until the downstream operator's input port has enough capacity to receive the message.

If you want to change this behavior, use the `IOSpec.condition()` method to configure the conditions. For example, to configure the input and output ports to have no conditions, you can use the following code:

```
from holoscan.core import ConditionType, OperatorSpec
# ...
def setup(self, spec: OperatorSpec):
    spec.input("in").condition(ConditionType.NONE)
    spec.output("out").condition(ConditionType.NONE)
```

The example code in the `setup()` method configures the input port to have no conditions, which means that the `compute()` method will be called as soon as the operator is ready to compute. Since there is no guarantee that the input port will have a message available, the `compute()` method should check if there is a message available on the input port before attempting to read it.

The `receive()` method of the `InputContext` object can be used to access different types of input data within the `compute()` method of your operator class. This method takes the name of the input port as an argument (which can be omitted if your operator has a single input port).

For standard Python objects, `receive()` will directly return the Python object for input of the specified name.

The Holoscan SDK also provides built-in data types called **Domain Objects**, defined in the `include/holoscan/core/domain` directory. For example, the `Tensor` is a Domain Object class that is used to represent a multi-dimensional array of data, which can be used directly by `OperatorSpec`, `InputContext`, and `OutputContext`.

Tip: This `holoscan.core.Tensor` class supports both `DLPack` and NumPy's array interface (`__array_interface__` and `__cuda_array_interface__`) so that it can be used with other Python libraries such as `CuPy`, `PyTorch`, `JAX`, `TensorFlow`, and `Numba`. See the [interoperability section](#) for more details.

In both cases, it will return `None` if there is no message available on the input port:

```
# ...
def compute(self, op_input, op_output, context):
    msg = op_input.receive("in")
    if msg:
        # Do something with msg
```

Receiving any number of inputs (Python)

Instead of assigning a specific number of input ports, it may be preferable to allow the ability to receive any number of objects on a port in certain situations.

Using IOSpec.ANY_SIZE for variable input handling

One way to achieve this is to define a multi-receiver input port by calling `spec.input("port_name", IOSpec.ANY_SIZE)` with `IOSpec.ANY_SIZE` as the second argument in the `setup()` method of the operator (as done for `PingRxOp` in the *native operator ping example*).

```
def setup(self, spec: OperatorSpec):
    spec.input("receivers", size=IOSpec.ANY_SIZE)
```

Code Snippet: `examples/ping_multi_port/python/ping_multi_port.py`

Listing 12.15: `examples/ping_multi_port/python/ping_multi_port.py`

```
113 class PingRxOp(Operator):
114     """Simple receiver operator.
115
116     This operator has:
117         input: "receivers"
118
119     This is an example of a native operator that can dynamically have any
120     number of inputs connected to is "receivers" port.
121     """
122
123     def __init__(self, fragment, *args, **kwargs):
124         self.count = 1
125         # Need to call the base class constructor last
126         super().__init__(fragment, *args, **kwargs)
127
128     def setup(self, spec: OperatorSpec):
129         # # Since Holoscan SDK v2.3, users can define a multi-receiver input port using
130         # # 'spec.input()' with 'size=IOSpec.ANY_SIZE'.
131         # # The old way is to use 'spec.param()' with 'kind="receivers"'.
132         # spec.param("receivers", kind="receivers")
133         spec.input("receivers", size=IOSpec.ANY_SIZE)
134
135     def compute(self, op_input, op_output, context):
136         values = op_input.receive("receivers")
137         print(f"Rx message received (count: {self.count}, size: {len(values)})")
138         self.count += 1
139         print(f"Rx message value1: {values[0].data}")
140         print(f"Rx message value2: {values[1].data}")
141
142
143 # Now define a simple application using the operators defined above
144
145
146 class MyPingApp(Application):
147     def compose(self):
148         # Define the tx, mx, rx operators, allowing the tx operator to execute 10 times
149         tx = PingTxOp(self, CountCondition(self, 10), name="tx")
150         mx = PingMxOp(self, name="mx", multiplier=3)
151         rx = PingRxOp(self, name="rx")
152
```

(continues on next page)

(continued from previous page)

```

153     # Define the workflow
154     self.add_flow(tx, mx, {"out1", "in1"}, {"out2", "in2"})
155     self.add_flow(mx, rx, {"out1", "receivers"}, {"out2", "receivers"})

```

Then, once the following configuration is provided in the `compose()` method,

```
self.add_flow(mx, rx, {"out1", "receivers"}, {"out2", "receivers"})
```

the `PingRxOp` will receive two inputs on the `receivers` port in the `compute()` method:

```
values = op_input.receive("receivers")
```

Tip: When an input port is defined with `IOSpec.ANY_SIZE`, the framework creates a new input port for each input object received on the port. The input ports are named using the format `<port_name>:<index>`, where `<port_name>` is the name of the input port and `<index>` is the index of the input object received on the port. For example, if the `receivers` port receives two input objects, the input ports will be named `receivers:0` and `receivers:1`.

The framework internally creates a parameter (`receivers`) with the type `std::vector<holoscan::IOSpec*>`, implicitly creates input ports (`receivers:0` and `receivers:1`), and connects them (adding references of the input ports to the `receivers` vector). This way, when the `receive()` method is called, the framework can return the input data from the corresponding input ports as a tuple.

```
values = op_input.receive("receivers")
```

If you add `print(rx.description)` at the end of the `compose()` method, you will see the description of the `PingRxOp` operator as shown below:

```

id: -1
name: rx
fragment: ""
args:
  []
type: kNative
conditions:
  []
resources:
  []
spec:
  fragment: ""
  params:
    - name: receivers
      type: std::vector<holoscan::IOSpec*>
      description: ""
      flag: kNone
  inputs:
    - name: receivers:1
      io_type: kInput
      typeinfo_name: N8holoscan3gxf6EntityE
      connector_type: kDefault
      conditions:
        []
    - name: receivers:0

```

(continues on next page)

(continued from previous page)

```

io_type: kInput
typeinfo_name: N8holoscan3gxf6EntityE
connector_type: kDefault
conditions:
  []
- name: receivers
  io_type: kInput
  typeinfo_name: N8holoscan3gxf6EntityE
  connector_type: kDefault
  conditions:
    []
outputs:
  []

```

Configuring input port queue size and message batch condition (Python)

If you want to receive multiple objects on a port and process them in batches, you can increase the queue size of the input port and set the `min_size` parameter of the `MessageAvailableCondition` condition to the desired batch size. This can be done by calling the `connector()` and `condition()` methods with the desired arguments, using the batch size as the `capacity` and `min_size` parameters, respectively.

Setting `min_size` to `N` will ensure that the operator receives `N` objects before the `compute()` method is called.

```

def setup(self, spec: OperatorSpec):
    spec.input("receivers").connector(IOSpec.ConnectorType.DOUBLE_BUFFER,
    capacity=2).condition(
        ConditionType.MESSAGE_AVAILABLE, min_size=2
    )

```

Then, the `receive()` method can be called with the `receivers` port name to receive input data in batches.

```

def compute(self, op_input, op_output, context):
    values = []
    value = op_input.receive("receivers")
    while value:
        values.append(value)
        value = op_input.receive("receivers")

    print(f"Rx message received (size: {len(values)})")

```

In the above example, the operator receives input on a port called “receivers” with a queue size of 2 and a `min_size` of 2. The `receive()` method is called in a loop to receive the input data in batches of 2. Since the operator does not know the number of objects to be received in advance, the `receive()` method is called in a loop until it returns a `None` value. The input data is stored in a list, and the size of the list is logged after all the input data is received.

To simplify the above code, the Holoscan SDK provides a `IOSpec.PRECEDING_COUNT` constant as a second argument to the `spec.input()` method to specify the number of preceding connections to the input port (in this case, the number of connections to the `receivers` port is 2) as the batch size. This can be used to receive the input data in batches without the need to call the `receive()` method in a loop.


```
def setup(self, spec: OperatorSpec):
    spec.input("receivers", size=IOSpec.PRECEDING_COUNT)
```

Then, the `receive()` method can be called with the `receivers` port name to receive the input data in batches.

```
def compute(self, op_input, op_output, context):
    values = op_input.receive("receivers")

    print(f"Rx message received (size: {len(values)})")

    print(f"Rx message value1: {values[0].data}")
    print(f"Rx message value2: {values[1].data}")
```

In the above example, the operator receives input on a port called “receivers” with a batch size of 2. The `receive()` method is called with the `receivers` port name to receive the input data in batches of 2. The input data is stored in a tuple, and the size of the tuple is logged after all the input data has been received.

If you want to use a specific batch size, you can use `holoscan.IOSpec.IOSize(size : int)` instead of `holoscan.IOSpec.PRECEDING_COUNT` to specify the batch size. Using `IOSize` in this way is equivalent to the more verbose `condition()` and `connector()` calls to update the capacity and `min_size` arguments shown near the start of this section.

The main reason to use `condition()` or `connector()` methods instead of the shorter `IOSize` is if additional parameter changes, such as the queue policy, need to be made. See more details on the use of the `condition()` and `connector()` methods in the advanced topics section below (*Further customizing inputs and outputs*).

```
def setup(self, spec: OperatorSpec):
    spec.input("receivers", size=IOSpec.IOSize(2))
```

If you want to receive the input data one by one, you can call the `receive()` method with the `kind="single"` argument.

```
def compute(self, op_input, op_output, context):
    while True:
        value = op_input.receive("receivers", kind="single")
        if value is None:
            break
        # Process the input data
        print(f"Rx message received (value: {value.data})")
```

The above code will receive the input data one by one from the `receivers` port. The `receive()` method is called in a loop until it returns a `None` value. The input data is stored in a variable, and the value of the input data is logged.

Note: This approach (receiving the input data one by one) is not applicable for the `IOSpec.ANY_SIZE` case. With the `IOSpec.ANY_SIZE` argument, the framework creates a new input port for each input object received internally. Each implicit input port (named using the format `<port_name>:<index>`) is associated with a `MessageAvailableCondition` condition that has a `min_size` of 1. Therefore, the `receive()` method **cannot** be called with the `kind="single"` keyword argument to receive the input data one by one. Instead, it can be called without any `kind` argument or with the `kind="multi"` argument for the `IOSpec.ANY_SIZE` case.

If you really need to receive the input data one by one for `IOSpec.ANY_SIZE` case (though it is not recommended), you can receive the input data from each implicit input port (named `<port_name>:<index>`) one by one using the `receive()` method without the `kind` argument. (e.g., `op_input.receive("receivers:0")`, `op_input.receive("receivers:1")`, etc.). To avoid the error message (such as `The operator does not have an input`

port with label 'receivers:X') when calling the `receive()` method for the implicit input port, you need to calculate the number of connections to the `receivers` port in advance and call the `receive()` method for each implicit input port accordingly.

```
def compute(self, op_input, op_output, context):
    input_count = len(self.spec.inputs) - 1 # -1 to exclude the 'receivers' input_
↪port
    for i in range(input_count):
        value = op_input.receive(f"receivers:{i}")
        if value is None:
            break
        # Process the input data
        print(f"Rx message received (value: {value.data})")
```

Attention: Using `IOSpec.PRECEDING_COUNT` or `IOSpec.IOSize(2)` appears to show the same behavior as `IOSpec.ANY_SIZE` in the above example. However, the difference is that since `IOSpec.PRECEDING_COUNT` or `IOSpec.IOSize(2)` doesn't use separate `MessageAvailableCondition` conditions for each (internal) input port, it is not guaranteed that the operator will receive the input data in order.

This means the operator may receive the input data in a different order than the order in which the connections are made in the `compose()` method. Additionally, with the multithread scheduler, it is not guaranteed that the operator will receive the input data from each of the connections uniformly. The operator may receive more input data from one connection than from another.

If the order of the input data is important, it is recommended to use `IOSpec.ANY_SIZE` and call the `receive()` method to receive the input data in batches at once.

Please see the [Python system test cases](#) for more examples of receiving multiple inputs in Python operators.

12.2.2 Python wrapping of a C++ operator

Wrapping an operator developed in C++ for use from Python is covered in a separate section on [creating C++ operator Python bindings](#).

Tip: As of Holoscan 2.1, there is a `GXFCodeletOp` class which can be used to easily wrap an existing GXF codelet from Python without having to first write an underlying C++ wrapper class for it. Similarly there is now also a `GXFComponentResource` class which allows a GXF Component to be used as a Holoscan resource from Python applications. A detailed example of how to use each of these is provided for Python applications in the [examples/import_gxf_components](#) folder.

12.2.3 Interoperability between wrapped and native Python operators

As described in the *Interoperability between GXF and native C++ operators* section, `holoscan::Tensor` objects can be passed to GXF operators using a `holoscan::TensorMap` message that holds the tensor(s). In Python, this is done by sending dict type objects that have tensor names as the keys and holoscan Tensor or array-like objects as the values. Similarly, when a wrapped C++ operator that transmits a single `holoscan::Tensor` is connected to the input port of a Python native operator, calling `op_input.receive()` on that port will return a Python dict containing a single item. That item's key is the tensor name and its value is the corresponding `holoscan.core.Tensor`.

Consider the following example, where `VideoStreamReplayerOp` and `HolovizOp` are Python wrapped C++ operators, and where `ImageProcessingOp` is a Python native operator:

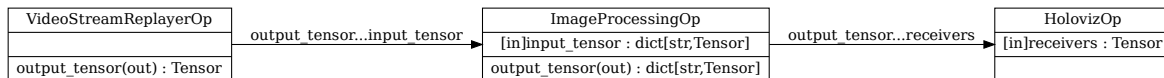


Fig. 12.5: The tensor interoperability between Python native operator and C++-based Python GXF operator

The following code shows how to implement `ImageProcessingOp`'s `compute()` method as a Python native operator communicating with C++ operators:

Listing 12.16: examples/tensor_interop/python/tensor_interop.py

```

62 def compute(self, op_input, op_output, context):
63     # in_message is a dict of tensors
64     in_message = op_input.receive("input_tensor")
65
66     # smooth along first two axes, but not the color channels
67     sigma = (self.sigma, self.sigma, 0)
68
69     # out_message will be a dict of tensors
70     out_message = dict()
71
72     for key, value in in_message.items():
73         print(f"message received (count: {self.count})")
74         self.count += 1
75
76         cp_array = cp.asarray(value)
77
78         # process cp_array
79         cp_array = ndi.gaussian_filter(cp_array, sigma)
80
81         out_message[key] = cp_array
82
83     op_output.emit(out_message, "output_tensor")
  
```

- The `op_input.receive()` method call returns a dict object.
- The `holoscan.core.Tensor` object is converted to a CuPy array by using `cupy.asarray()` method call.
- The CuPy array is used as an input to the `ndi.gaussian_filter()` function call with a parameter `sigma`. The result of the `ndi.gaussian_filter()` function call is a CuPy array.
- Finally, a new dict object is created, `out_message`, to be sent to the next operator with `op_output.emit()`. The CuPy array, `cp_array`, is added to it where the key is the tensor name. CuPy arrays do not

have to explicitly be converted to a `holoscan.core.Tensor` object first since they implement a `DLPack` (and `__cuda__array_interface__`) interface.

Note: A complete example of the Python native operator that supports interoperability with Python wrapped C++ operators is available in the [examples/tensor_interop/python](#) directory.

You can add multiple tensors to a single dict object, as in the example below:

Operator sending a message:

```
out_message = {
    "video": output_array,
    "labels": labels,
    "bbox_coords": bbox_coords,
}

# emit the tensors
op_output.emit(out_message, "outputs")
```

Operator receiving the message, assuming the `outputs` port above is connected to the `inputs` port below with `add_flow()` has the corresponding tensors:

```
in_message = op_input.receive("inputs")
# Tensors and tensor names
video_tensor = in_message["video"]
labels_tensor = in_message["labels"]
bbox_coords_tensor = in_message["bbox_coords"]
```

Note: Some existing operators allow *configuring* the name of the tensors they send/receive. An example is the `tensors` parameter of `HolovizOp`, where the name for each tensor maps to the names of the tensors in the Entity (see the `holoviz` entry in [apps/endoscopy_tool_tracking/python/endoscopy_tool_tracking.yaml](#)).

A complete example of a Python native operator that emits multiple tensors to a downstream C++ operator is available in the [examples/holoviz/python](#) directory.

There is a special serialization code for tensor types for emit/receive of tensor objects over a UCX connection that avoids copying the tensor data to an intermediate buffer. For distributed apps, we cannot just send the Python object as we do between operators in a single fragment app, but instead we need to cast it to `holoscan:Tensor` to use a special zero-copy code path. However, we also transmit a header indicating if the type was originally some other array-like object and attempt to return the same type again on the other side so that the behavior remains more similar to the non-distributed case.

Transmitted object	Received Object
<code>holoscan.Tensor</code>	<code>holoscan.Tensor</code>
dict of array-like	dict of <code>holoscan.Tensor</code>
host array-like object (with <code>__array_interface__</code>)	<code>numpy.ndarray</code>
device array-like object (with <code>__cuda__array_interface__</code>)	<code>cupy.ndarray</code>

This avoids NumPy or CuPy arrays being serialized to a string via cloudpickle so that they can efficiently be transmitted and the same type is returned again on the opposite side. Worth mentioning is that ,if the type emitted was e.g. a PyTorch host/device tensor on emit, the received value will be a `numpy/cupy` array since ANY object implementing the interfaces returns those types.

12.2.4 Automated operator class creation

Holoscan also provides a `holoscan.decorator` module which provides ways to autogenerate Operators by adding decorators to an existing function or class. Please see the separate section on *operator creation via `holoscan.decorator.create_op`*.

12.3 Advanced Topics

12.3.1 Further customizing inputs and outputs

This section complements the information above on basic input and output port configuration given separately in the C++ and Python operator creation guides. The concepts described here are the same for either the C++ or Python APIs.

By default, both the input and output ports of an Operator will use a double-buffered queue that has a capacity of one message and a policy that is set to error if a message arrives while the queue is already full. A single `MessageAvailableCondition` (C++/Python) condition is automatically placed on the operator for each input port so that the `compute` method will not be called until a single message is available at each port. Similarly each output port has a `DownstreamMessageAffordableCondition` (C++/Python) condition that does not let the operator call `compute` until any operators connected downstream have space in their receiver queue for a single message. These default conditions ensure that messages never arrive at a queue when it is already full and that a message has already been received whenever the `compute` method is called. These default conditions make it relatively easy to connect a pipeline where each operator calls `compute` in turn, but may not be suitable for all applications. This section covers how the default behavior can be overridden on request.

It is possible to modify the global default queue policy via the `HOLOSCAN_QUEUE_POLICY` environment variable. Valid options (case insensitive) are:

- “pop”: a new item that arrives when the queue is full replaces the oldest item
- “reject”: a new item that arrives when the queue is discarded
- “fail”: terminate the application if a new item arrives when the queue is full

The default behavior is “fail” when `HOLOSCAN_QUEUE_POLICY` is not specified. If an operator’s `setup` method explicitly sets a receiver or transmitter via the `connector` (C++/Python) method as describe below, that connector’s policy will not be overridden by the default.

Note: Overriding operator port properties is an advanced topic. Developers may want to skip this section until they come across a case where the default behavior is not sufficient for their application.

To override the properties of the queue used for a given port, the `connector` (C++/Python) method can be used as shown in the example below. This example also shows how the `condition` (C++/Python) method can be used to change the condition type placed on the Operator by a port. In general, when an operator has multiple conditions, they are AND combined, so the conditions on **all** ports must be satisfied before an operator can call `compute`.

C++ Example

Consider the following code from within the `holoscan::Operator::setup()` method of an operator.

```
spec.output<TensorMap>("out1")

spec.output<TensorMap>("out2").condition(ConditionType::kNone);

spec.output<TensorMap>("in")
    .connector(IOSpec::ConnectorType::kDoubleBuffer,
               Arg("capacity", static_cast<uint64_t>(2)),
               Arg("policy", static_cast<uint64_t>(1))) // 0=pop, 1=reject, 2=fault_
    ↪ (default)
    .condition(ConditionType::kMessageAvailable,
               Arg("min_size", static_cast<uint64_t>(2)),
               Arg("front_stage_max_size", static_cast<size_t>(2)));
```

This would define

- an output port named “out1” with the default properties
- an output port named “out2” that still has the default connector (a `holoscan::gxf::DoubleBufferTransmitter`), but the default condition of `ConditionType::kDownstreamMessageAffordable` is removed by setting `ConditionType::kNone`. This indicates that the operator will not check if any port downstream of “out2” has space available in its receiver queue before calling compute.
- an input port named “in” where both the connector and condition have different parameters than the default. For example, the queue size is increased to 2 and `policy=1` is “reject”, indicating that if a message arrives when the queue is already full, that message will be rejected in favor of the message already in the queue.

Python Example

Consider the following code from within the `holoscan::Operator::setup()` method of an operator.

```
spec.output("out1")

spec.output("out2").condition(ConditionType.NONE)

spec.input("in").connector(
    IOSpec.ConnectorType.DOUBLE_BUFFER,
    capacity=2,
    policy=1, # 0=pop, 1=reject, 2=fault (default)
).condition(ConditionType.MESSAGE_AVAILABLE, min_size=2, front_stage_max_size=2)
```

This would define

- an output port named “out1” with the default properties
- an output port named “out2” that still has the default connector (a `holoscan.resources.DoubleBufferTransmitter`), but the default condition of `ConditionType.DOWNSTREAM_MESSAGE_AFFORDABLE` is removed by setting `ConditionType.NONE`. This indicates that the operator will not check if any port downstream of “out2” has space available in its receiver queue before calling compute.

- an input port named “in1” where both the connector and condition have different parameters than the default. For example, the queue size is increased to 2 and `policy=1` is “reject”, indicating that if a message arrives when the queue is already full, that message will be rejected in favor of the message already in the queue.

To learn more about overriding connectors and/or conditions there is a [multi_branch_pipeline](#) example which overrides default conditions to allow two branches of a pipeline to run at different frame rates. There is also an example of increasing the queue sizes available in [this Python queue policy test application](#).

12.3.2 Using the Holoscan SDK with Other Libraries

The Holoscan SDK enables seamless integration with various powerful, GPU-accelerated libraries to build efficient, high-performance pipelines.

Please refer to the [Best Practices to Integrate External Libraries into Holoscan Pipelines](#) tutorial in the [HoloHub](#) repository for detailed examples and more information on Holoscan’s tensor interoperability and handling CUDA libraries in the pipeline. This includes [CUDA Python](#), [CuPy](#), [MatX](#) for C++, [cuCIM](#), [CV-CUDA](#), and [OpenCV](#) for integration into Holoscan applications.

LOGGING

13.1 Overview

The Holoscan SDK uses the Logger module to convey messages to the user. These messages are categorized into different severity levels (see below) to inform users of the severity of a message and as a way to control the number and verbosity of messages that are printed to the terminal. There are two settings which can be used for this purpose:

- Logger level
- Logger format

13.1.1 Logger Level

Messages that are logged using the Logger module have a severity level, e.g., messages can be categorized as INFO, WARN, ERROR, etc.

The default logging level for an application is to print out messages with severity INFO or above, i.e., messages that are categorized as INFO, WARN, ERROR, and CRITICAL. You can modify this default by calling `set_log_level()` (C++/Python) in the application code to override the SDK default logging level and give it one of the following log levels.

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- CRITICAL
- OFF

CPP

```
1 #include <holoscan/holoscan.hpp>
2
3 int main() {
4     holoscan::set_log_level(holoscan::LogLevel::WARN);
5     // ...
6     return 0;
7 }
```

PYTHON

```
1 from holoscan.logger import set_log_level
2
3 def main():
4     set_log_level(LogLevel::WARN)
5     # ...
6
7 if __name__ == "__main__":
8     main()
```

Additionally, at runtime, the user can set the HOLOSCAN_LOG_LEVEL environment variable to one of the values listed above. This provides users with the flexibility to enable printing of diagnostic information for debugging purposes when an issue occurs.

```
export HOLOSCAN_LOG_LEVEL=TRACE
```

Note: Under the hood, Holoscan SDK uses GXF to execute the computation graph. By default, this GXF layer uses the same logging level as Holoscan SDK. If it is desired to override the logging level of this executor independently of the Holoscan SDK logging level, environment variable HOLOSCAN_EXECUTOR_LOG_LEVEL can be used. It supports the same levels as HOLOSCAN_LOG_LEVEL.

Note: For distributed applications, it can sometimes be useful to also enable additional logging for the UCX library used to transmit data between fragments. This can be done by setting the UCX environment variable UCX_LOG_LEVEL to one of: fatal, error, warn, info, debug, trace, req, data, async, func, poll. These have the behavior as described here: [UCX log levels](#).

13.2 Logger Format

When a message is printed out, the default message format shows the message severity level, filename:linenumber, and the message to be printed.

For example:

```
[info] [ping_multi_port.cpp:114] Rx message value1: 51
[info] [ping_multi_port.cpp:115] Rx message value2: 54
```

You can modify this default by calling `set_log_pattern()` (C++/Python) in the application code to override the SDK default logging format.

The pattern string can be one of the following pre-defined values

- **SHORT** : prints message severity level, and message
- **DEFAULT** : prints message severity level, filename:linenumber, and message
- **LONG** : prints timestamp, application, message severity level, filename:linenumber, and message
- **FULL** : prints timestamp, thread id, application, message severity level, filename:linenumber, and message

CPP

```
1 #include <holoscan/holoscan.hpp>
2
3 int main() {
4     holoscan::set_log_pattern("SHORT")
5     // ...
6     return 0;
7 }
```

PYTHON

```
1 from holoscan.logger import set_log_pattern
2
3 def main():
4     set_log_pattern("SHORT")
5     # ...
6
7 if __name__ == "__main__":
8     main()
```

With this logger format, the above application would display messages with the following format:

```
[info] Rx message value1: 51
[info] Rx message value2: 54
```

Alternatively, the pattern string can be a custom pattern to customize the logger format. Using this string pattern:

```
"[%Y-%m-%d %H:%M:%S.%e] [%n] [%^%l%$] [%s:%#] %v";
```

The following format will be displayed:

```
[2023-06-27 14:22:36.073] [holoscan] [info] [ping_multi_port.cpp:114] Rx message value1:↵
↵ 51
[2023-06-27 14:22:36.073] [holoscan] [info] [ping_multi_port.cpp:115] Rx message value2:↵
↵ 54
```

For more details on custom formatting and details of each flag, please see the [spdlog wiki page](#).

Additionally, at runtime, the user can also set the `HOLOSCAN_LOG_FORMAT` environment variable to modify the logger format. The accepted string pattern is the same as the string pattern for the `set_log_pattern()` api mentioned above.

13.2.1 Precedence of Logger Level and Logger Format

The `HOLOSCAN_LOG_LEVEL` environment variable takes precedence and overrides the application settings, such as `Logger::set_log_level()` (C++/Python).

When `HOLOSCAN_LOG_LEVEL` is set, it determines the logging level. If this environment variable is unset, the application settings are used if they are available. Otherwise, the SDK's default logging level of `INFO` is applied.

Similarly, the `HOLOSCAN_LOG_FORMAT` environment variable takes precedence and overrides the application settings, such as `Logger::set_log_pattern()` (C++/Python).

When `HOLOSCAN_LOG_FORMAT` is set, it determines the logging format. If this environment variable is unset, the application settings are used if they are available. Otherwise, the SDK's default logging format depending on the current log level (FULL format for `DEBUG` and `TRACE` log levels. DEFAULT format for other log levels) is applied.

13.3 Calling the Logger in Your Application

The **C++ API** uses the `HOLOSCAN_LOG_XXX()` macros to log messages in the application. These macros use the `fmtlib` format string syntax for their format strings.

Note: Holoscan automatically checks `HOLOSCAN_LOG_LEVEL` environment variable and sets the log level when the Application class instance is created. However, those log level settings are for Holoscan core or C++ operator (C++)'s logging message (such as `HOLOSCAN_LOG_INFO` macro), not for Python's logging. Users of the **Python API** should use the built-in `logging` module to log messages. You must configure the logger before use (`logging.basicConfig(level=logging.INFO)`):

```
>>> import logging
>>> logger = logging.getLogger("main")
>>> logger.info('hello')
>>> logging.basicConfig(level=logging.INFO)
>>> logger.info('hello')
INFO:main:hello
```

DEBUGGING

14.1 Overview

The Holoscan SDK is designed to streamline the debugging process for developers working on advanced applications. This comprehensive guide covers the SDK's debugging capabilities, with a focus on Visual Studio Code integration, and provides detailed instructions for various debugging scenarios.

It includes methods for debugging both the C++ and Python components of applications, utilizing tools like GDB, UCX, and Python-specific debuggers.

14.2 Visual Studio Code Integration

14.2.1 VSCode Dev Container

The [Holoscan SDK](#) can be effectively developed using Visual Studio Code, leveraging the capabilities of a development container. This container, defined in the `.devcontainer` folder, is pre-configured with all the necessary tools and libraries, as detailed in [Visual Studio Code's documentation on development containers](#).

14.2.2 Launching VSCode with the Holoscan SDK

- **Local Development:** Use the `./run vscode` command to launch Visual Studio Code in a development container.
- **Remote Development:** For attaching to an existing dev container from a remote machine, use `./run vscode_remote`. Additional instructions can be accessed via `./run vscode_remote -h`.

Upon launching Visual Studio Code, the development container will automatically be built. This process also involves the installation of recommended extensions and the configuration of CMake.

14.2.3 Configuring CMake

For manual adjustments to the CMake configuration:

1. Open the command palette in VSCode (Ctrl + Shift + P).
2. Execute the CMake: Configure command.

14.2.4 Building the Source Code

To build the source code within the development container:

- Either press Ctrl + Shift + B.
- Or use the command palette (Ctrl + Shift + P) and run Tasks: Run Build Task.

14.2.5 Debugging Workflow

For debugging the source code:

1. Open the Run and Debug view in VSCode (Ctrl + Shift + D).
2. Select an appropriate debug configuration from the dropdown.
3. Press F5 to start the debugging session.

The launch configurations are defined in `.vscode/launch.json`([link](#)).

Please refer to [Visual Studio Code's documentation on debugging](#) for more information.

14.2.6 Integrated Debugging for C++ and Python in Holoscan SDK

The Holoscan SDK facilitates seamless debugging of both C++ and Python components within your applications. This is achieved through the integration of the Python C++ Debugger extension in Visual Studio Code, which can be found [here](#).

This powerful extension is specifically designed to enable effective debugging of Python operators that are executed within the C++ runtime environment. Additionally, it provides robust capabilities for debugging C++ operators and various SDK components that are executed via the Python interpreter.

To utilize this feature, debug configurations for Python C++ Debug should be defined within the `.vscode/launch.json` file, available [here](#).

Here's how to get started:

1. Open a Python file within your project, such as `examples/ping_vector/python/ping_vector.py`.
2. In the Run and Debug view of Visual Studio Code, select the Python C++ Debug debug configuration.
3. Set the necessary breakpoints in both your Python and C++ code.
4. Initiate the debugging session by pressing F5.

Upon starting the session, two separate debug terminals will be launched; one for Python and another for C++. In the C++ terminal, you will encounter a prompt regarding superuser access:

```
Superuser access is required to attach to a process. Attaching as superuser can
↪ potentially harm your computer. Do you want to continue? [y/N]
```

Respond with `y` to proceed.

Following this, the Python application initiates, and the C++ debugger attaches to the Python process. This setup allows you to simultaneously debug both Python and C++ code. The `CALL STACK` tab in the `Run` and `Debug` view will display `Python: Debug Current File` and `(gdb) Attach`, indicating active debugging sessions for both languages.

By leveraging this integrated debugging approach, developers can efficiently troubleshoot and enhance applications that utilize both Python and C++ components within the Holoscan SDK.

14.3 Debugging an Application Crash

This section outlines the procedures for debugging an application crash.

14.3.1 Core Dump Analysis

In the event of an application crash, you might encounter messages like `Segmentation fault (core dumped)` or `Aborted (core dumped)`. These indicate the generation of a core dump file, which captures the application's memory state at the time of the crash. This file can be utilized for debugging purposes.

Enabling Core Dump

There are instances where core dumps might be disabled or not generated despite an application crash.

To activate core dumps, it's necessary to configure the `ulimit` setting, which determines the maximum size of core dump files. By default, `ulimit` is set to 0, effectively disabling core dumps. Setting `ulimit` to unlimited enables the generation of core dumps.

```
ulimit -c unlimited
```

Additionally, configuring the `core_pattern` value is required. This value specifies the naming convention for the core dump file. To view the current `core_pattern` setting, execute the following command:

```
cat /proc/sys/kernel/core_pattern
# or
sysctl kernel.core_pattern
```

To modify the `core_pattern` value, execute the following command:

```
echo "coredump_%e_%p" | sudo tee /proc/sys/kernel/core_pattern
# or
sudo sysctl -w kernel.core_pattern=coredump_%e_%p
```

In this case, we have requested that both the executable name (`%e`) and the process id (`%p`) be present in the generated file's name. The various options available are documented in the [core documentation](#).

If you encounter errors like `tee: /proc/sys/kernel/core_pattern: Read-only file system` or `sysctl: setting key "kernel.core_pattern", ignoring: Read-only file system` within a Docker container, it's advisable to set the `kernel.core_pattern` parameter on the host system instead of within the container.

As `kernel.core_pattern` is a system-wide kernel parameter, modifying it on the host should impact all containers. This method, however, necessitates appropriate permissions on the host machine.

Furthermore, when launching a Docker container using `docker run`, it's often essential to include the `--cap-add=SYS_PTRACE` option to enable core dump creation inside the container. Core dump generation typically requires elevated privileges, which are not automatically available to Docker containers.

Using GDB to Debug a Core Dump File

After the core dump file is generated, you can utilize GDB to debug the core dump file.

Consider a scenario where a segmentation fault is intentionally induced at line 29 in `examples/ping_simple/cpp/ping_simple.cpp` by adding the line `*(int*)0 = 0;` to trigger the fault.

```
--- a/examples/ping_simple/cpp/ping_simple.cpp
+++ b/examples/ping_simple/cpp/ping_simple.cpp
@@ -19,7 +19,6 @@
#include <holoscan/operators/ping_tx/ping_tx.hpp>
#include <holoscan/operators/ping_rx/ping_rx.hpp>

-
class MyPingApp : public holoscan::Application {
public:
    void compose() override {
@@ -27,6 +26,7 @@ class MyPingApp : public holoscan::Application {
    // Define the tx and rx operators, allowing the tx operator to execute 10 times
    auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
    auto rx = make_operator<ops::PingRxOp>("rx");
+    *(int*)0 = 0;
```

Upon running `./examples/ping_simple/cpp/ping_simple`, the following output is observed:

```
$ ./examples/ping_simple/cpp/ping_simple
Segmentation fault (core dumped)
```

It's apparent that the application has aborted and a core dump file has been generated.

```
$ ls coredump*
coredump_ping_simple_2160275
```

The core dump file can be debugged using GDB by executing `gdb <application> <coredump_file>`.

```
$ gdb ./examples/ping_simple/cpp/ping_simple coredump_ping_simple_2160275
```

gives

```
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./examples/ping_simple/cpp/ping_simple...
```

(continues on next page)

(continued from previous page)

```
[New LWP 2160275]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1".
Core was generated by `./examples/ping_simple/cpp/ping_simple'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  MyPingApp::compose (this=0x563bd3a3de80) at ../examples/ping_simple/cpp/ping_simple.
    ↪cpp:29
29      *(int*)0 = 0;
(gdb)
```

It is evident that the application crashed at line 29 of examples/ping_simple/cpp/ping_simple.cpp.

To display the backtrace, the bt command can be executed.

```
(gdb) bt
#0  MyPingApp::compose (this=0x563bd3a3de80) at ../examples/ping_simple/cpp/ping_simple.
    ↪cpp:29
#1  0x00007f2a76cdb5ea in holoscan::Application::compose_graph (this=0x563bd3a3de80) at .
    ↪./src/core/application.cpp:325
#2  0x00007f2a76c3d121 in holoscan::AppDriver::check_configuration (this=0x563bd3a42920)
    ↪at ../src/core/app_driver.cpp:803
#3  0x00007f2a76c384ef in holoscan::AppDriver::run (this=0x563bd3a42920) at ../src/core/
    ↪app_driver.cpp:168
#4  0x00007f2a76cda70c in holoscan::Application::run (this=0x563bd3a3de80) at ../src/
    ↪core/application.cpp:207
#5  0x0000563bd2ec4002 in main (argc=1, argv=0x7ffea82c4c28) at ../examples/ping_simple/
    ↪cpp/ping_simple.cpp:38
```

14.3.2 UCX Segmentation Fault Handler

In cases where a distributed application using the UCX library encounters a segmentation fault, you might see stack traces from UCX. This is a default configuration of the UCX library to output stack traces upon a segmentation fault. However, this behavior can be modified by setting the UCX_HANDLE_ERRORS environment variable:

- UCX_HANDLE_ERRORS=bt prints a backtrace during a segmentation fault (default setting).
- UCX_HANDLE_ERRORS=debug attaches a debugger if a segmentation fault occurs.
- UCX_HANDLE_ERRORS=freeze freezes the application on a segmentation fault.
- UCX_HANDLE_ERRORS=freeze , bt both freezes the application and prints a backtrace upon a segmentation fault.
- UCX_HANDLE_ERRORS=none disables backtrace printing during a segmentation fault.

While the default action is to print a backtrace on a segmentation fault, it may not always be helpful.

For instance, if a segmentation fault is intentionally caused at line 139 near the start of PingTensorTxOp::compute in /workspace/holoscan-sdk/src/operators/ping_tensor_tx/ping_tensor_tx.cpp (by adding *(int*)0 = 0;), running ./examples/ping_distributed/cpp/ping_distributed will result in the following output:

```
[holoscan:2097261:0:2097311] Caught signal 11 (Segmentation fault: address not mapped to
    ↪object at address (nil))
==== backtrace (tid:2097311) ====
 0 /opt/ucx/1.15.0/lib/libucs.so.0(ucs_handle_error+0x2e4) [0x7f18db865264]
 1 /opt/ucx/1.15.0/lib/libucs.so.0(+0x3045f) [0x7f18db86545f]
```

(continues on next page)

(continued from previous page)

```

2 /opt/ucx/1.15.0/lib/libucs.so.0(+0x30746) [0x7f18db865746]
3 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x42520) [0x7f18da9ee520]
4 ./examples/ping_distributed/cpp/ping_distributed(+0x103d2b) [0x5651dafc7d2b]
5 /workspace/holoscan-sdk/build-debug-x86_64/lib/libholoscan_core.so.1(_
→ZN8holoscan3gxf10GXFWrapper4tickEv+0x13d) [0x7f18dcbfaafd]
6 /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_core.so(_
→ZN6nvidia3gxf14EntityExecutor10EntityItem11tickCodeletERKNS0_6HandleINS0_
→7CodeletEEE+0x127) [0x7f18db2cb487]
7 /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_core.so(_
→ZN6nvidia3gxf14EntityExecutor10EntityItem4tickElPNS0_6RouterE+0x444) [0x7f18db2cde44]
8 /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_core.so(_
→ZN6nvidia3gxf14EntityExecutor10EntityItem7executeElPNS0_6RouterERl+0x3e9)
→[0x7f18db2ce859]
9 /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_core.so(_
→ZN6nvidia3gxf14EntityExecutor13executeEntityEl+0x41b) [0x7f18db2cf0cb]
10 /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_serialization.so(_
→ZN6nvidia3gxf20MultiThreadScheduler20workerThreadEntranceEPNS0_10ThreadPoolEl+0x3c0)
→[0x7f18daf0cc50]
11 /usr/lib/x86_64-linux-gnu/libstdc++.so.6(+0xdc253) [0x7f18dacb0253]
12 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x94ac3) [0x7f18daa40ac3]
13 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x126660) [0x7f18daad2660]
=====
Segmentation fault (core dumped)

```

Although a backtrace is provided, it may not always be helpful as it often lacks source code information. To obtain detailed source code information, using a debugger is necessary.

By setting the `UCX_HANDLE_ERRORS` environment variable to `freeze, bt` and running `./examples/ping_distributed/cpp/ping_distributed`, we can observe that the thread responsible for the segmentation fault is frozen, allowing us to attach a debugger to it for further investigation.

```

$ UCX_HANDLE_ERRORS=freeze, bt ./examples/ping_distributed/cpp/ping_distributed

[holoscan:37 :1:51] Caught signal 11 (Segmentation fault: address not mapped to object
→at address (nil))
==== backtrace (tid: 51) ====
0 /opt/ucx/1.15.0/lib/libucs.so.0(ucs_handle_error+0x2e4) [0x7f9fc6d75264]
1 /opt/ucx/1.15.0/lib/libucs.so.0(+0x3045f) [0x7f9fc6d7545f]
2 /opt/ucx/1.15.0/lib/libucs.so.0(+0x30746) [0x7f9fc6d75746]
3 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x42520) [0x7f9fc803e520]
4 /workspace/holoscan-sdk/build-x86_64/lib/libholoscan_op_ping_tensor_tx.so.2(_
→ZN8holoscan3ops14PingTensorTxOp7computeERNS_12InputContextERNS_13OutputContextERNS_
→16ExecutionContextE+0x53) [0x7f9fcad9e7f1]
5 /workspace/holoscan-sdk/build-x86_64/lib/libholoscan_core.so.2(_
→ZN8holoscan3gxf10GXFWrapper4tickEv+0x155) [0x7f9fc9e415eb]
6 /workspace/holoscan-sdk/build-x86_64/lib/libgxf_sample.so(_
→ZN6nvidia3gxf14EntityExecutor10EntityItem11tickCodeletERKNS0_6HandleINS0_
→7CodeletEEE+0x1a7) [0x7f9fc88f0347]
7 /workspace/holoscan-sdk/build-x86_64/lib/libgxf_sample.so(_
→ZN6nvidia3gxf14EntityExecutor10EntityItem4tickElPNS0_6RouterE+0x460) [0x7f9fc88f29c0]
8 /workspace/holoscan-sdk/build-x86_64/lib/libgxf_sample.so(_
→ZN6nvidia3gxf14EntityExecutor10EntityItem7executeElPNS0_6RouterERl+0x31e)
→[0x7f9fc88f31ee]

```

(continues on next page)

(continued from previous page)

```

 9 /workspace/holoscan-sdk/build-x86_64/lib/libgxf_sample.so(_
→ZN6nvidia3gxf14EntityExecutor13executeEntityEl+0x2e7) [0x7f9fc88f39d7]
10 /workspace/holoscan-sdk/build-x86_64/lib/libgxf_serialization.so(_
→ZN6nvidia3gxf20MultiThreadScheduler20workerThreadEntranceEPNS0_10ThreadPoolEl+0x419)
→[0x7f9fc8605dd9]
11 /usr/lib/x86_64-linux-gnu/libstdc++.so.6(+0xdc253) [0x7f9fc8321253]
12 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x94ac3) [0x7f9fc8090ac3]
13 /usr/lib/x86_64-linux-gnu/libc.so.6(clone+0x44) [0x7f9fc8121a04]
=====
[holoscan:2127091:0:2127105] Process frozen, press Enter to attach a debugger...

```

It is observed that the thread responsible for the segmentation fault is 51 (tid: 51). To attach a debugger to this thread, simply press Enter.

Upon attaching the debugger, a backtrace will be displayed, but it may not be from the thread that triggered the segmentation fault. To handle this, use the `info threads` command to list all threads, and the `thread <thread_id>` command to switch to the thread that caused the segmentation fault.

```

(gdb) info threads
Id      Target Id                                     Frame
* 1      Thread 0x7f9fc6ce2000 (LWP 37) "ping_distribute" 0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
 2      Thread 0x7f9fc51bb000 (LWP 39) "ping_distribute" 0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
 3      Thread 0x7f9fc11ba000 (LWP 40) "ping_distribute" 0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
 4      Thread 0x7f9fbd1b9000 (LWP 41) "ping_distribute" 0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
 5      Thread 0x7f9fabfff000 (LWP 42) "cuda00001400006" 0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
 6      Thread 0x7f9f99fff000 (LWP 43) "async"          0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
 7      Thread 0x7f9f95ffe000 (LWP 44) "ping_distribute" 0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
 8      Thread 0x7f9f77fff000 (LWP 45) "dispatcher"      0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
 9      Thread 0x7f9f73ffe000 (LWP 46) "async"          0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
10      Thread 0x7f9f6fffd000 (LWP 47) "worker"          0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
11      Thread 0x7f9f5bfff000 (LWP 48) "ping_distribute" 0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
12      Thread 0x7f9f57ffe000 (LWP 49) "dispatcher"      0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
13      Thread 0x7f9f53ffd000 (LWP 50) "async"          0x00007f9fc80e6612 in __libc_
→pause () at ../sysdeps/unix/sysv/linux/pause.c:29
14      Thread 0x7f9f4fffc000 (LWP 51) "worker"          0x00007f9fc80e642f in __GI___
→wait4 (pid=pid@entry=52, stat_loc=stat_loc@entry=0x7f9f4fff6cfc,
→options=options@entry=0, usage=usage@entry=0x0) at ../sysdeps/unix/sysv/linux/wait4.
→c:30

```

It's evident that thread ID 14 is responsible for the segmentation fault (LWP 51). To investigate further, we can switch to this thread using the command `thread 14` in GDB:

```
(gdb) thread 14
```

After switching, we can employ the `bt` command to examine the backtrace of this thread.

```
(gdb) bt
#0 0x00007f9fc80e642f in __GI___wait4 (pid=pid@entry=52, stat_loc=stat_
    ↳loc@entry=0x7f9f4fff6cfc, options=options@entry=0, usage=usage@entry=0x0) at ../
    ↳sysdeps/unix/sysv/linux/wait4.c:30
#1 0x00007f9fc80e63ab in __GI___waitpid (pid=pid@entry=52, stat_loc=stat_
    ↳loc@entry=0x7f9f4fff6cfc, options=options@entry=0) at ./posix/waitpid.c:38
#2 0x00007f9fc6d72587 in ucs_debugger_attach () at /opt/ucx/src/contrib/./src/ucs/
    ↳debug/debug.c:816
#3 0x00007f9fc6d7531d in ucs_error_freeze (message=0x7f9fc6d93c53 "address not mapped_
    ↳to object") at /opt/ucx/src/contrib/./src/ucs/debug/debug.c:919
#4 ucs_handle_error (message=0x7f9fc6d93c53 "address not mapped to object") at /opt/ucx/
    ↳src/contrib/./src/ucs/debug/debug.c:1089
#5 ucs_handle_error (message=0x7f9fc6d93c53 "address not mapped to object") at /opt/ucx/
    ↳src/contrib/./src/ucs/debug/debug.c:1077
#6 0x00007f9fc6d7545f in ucs_debug_handle_error_signal (signo=signo@entry=11,
    ↳cause=0x7f9fc6d93c53 "address not mapped to object", fmt=fmt@entry=0x7f9fc6d93cf5 " at_
    ↳address %p") at /opt/ucx/src/contrib/./src/ucs/debug/debug.c:1038
#7 0x00007f9fc6d75746 in ucs_error_signal_handler (signo=11, info=0x7f9f4fff73b0,
    ↳context=<optimized out>) at /opt/ucx/src/contrib/./src/ucs/debug/debug.c:1060
#8 <signal handler called>
#9 holoscan::ops::PingTensorTxOp::compute (this=0x5643fdcbd540, op_output=..., context=.
    ↳..) at /workspace/holoscan-sdk/src/operators/ping_tensor_tx/ping_tensor_tx.cpp:139
#10 0x00007f9fc9e415eb in holoscan::gxf::GXFWrapper::tick (this=0x5643fdcfef00) at /
    ↳workspace/holoscan-sdk/src/core/gxf/gxf_wrapper.cpp:78
#11 0x00007f9fc88f0347 in
    ↳nvidia::gxf::EntityExecutor::EntityItem::tickCodelet(nvidia::gxf::Handle
    ↳<nvidia::gxf::Codelet> const&) () from /workspace/holoscan-sdk/build-x86_64/lib/libgxf_
    ↳sample.so
#12 0x00007f9fc88f29c0 in nvidia::gxf::EntityExecutor::EntityItem::tick(long,
    ↳nvidia::gxf::Router*) () from /workspace/holoscan-sdk/build-x86_64/lib/libgxf_sample.so
#13 0x00007f9fc88f31ee in nvidia::gxf::EntityExecutor::EntityItem::execute(long,
    ↳nvidia::gxf::Router*, long&) () from /workspace/holoscan-sdk/build-x86_64/lib/libgxf_
    ↳sample.so
#14 0x00007f9fc88f39d7 in nvidia::gxf::EntityExecutor::executeEntity(long, long) () from
    ↳/workspace/holoscan-sdk/build-x86_64/lib/libgxf_sample.so
#15 0x00007f9fc8605dd9 in
    ↳nvidia::gxf::MultiThreadScheduler::workerThreadEntrance(nvidia::gxf::ThreadPool*,
    ↳long) () from /workspace/holoscan-sdk/build-x86_64/lib/libgxf_serialization.so
#16 0x00007f9fc8321253 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#17 0x00007f9fc8090ac3 in start_thread (arg=<optimized out>) at ./nptl/pthread_create.
    ↳c:442
#18 0x00007f9fc8121a04 in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:100
```

Under the backtrace of thread 14, you will find:

```
#8 <signal handler called>
#9 holoscan::ops::PingTensorTxOp::compute (this=0x5643fdcbd540, op_output=..., context=.
    ↳..) at /workspace/holoscan-sdk/src/operators/ping_tensor_tx/ping_tensor_tx.cpp:139
```

This indicates that the segmentation fault occurred at line 139 in `/workspace/holoscan-sdk/src/operators/`

ping_tensor_tx/ping_tensor_tx.cpp.

To view the backtrace of all threads, use the `thread apply all bt` command.

```
(gdb) thread apply all bt

...
Thread 14 (Thread 0x7f9f4fffc000 (LWP 51) "worker"):
#0  0x000007f9fc80e642f in __GI___wait4 (pid=pid@entry=52, stat_loc=stat_
↳ loc@entry=0x7f9f4fff6cfc, options=options@entry=0, usage=usage@entry=0x0) at ../
↳ sysdeps/unix/sysv/linux/wait4.c:30
...

Thread 13 (Thread 0x7f9f53ffd000 (LWP 50) "async"):
#0  0x000007f9fc80e6612 in __libc_pause () at ../sysdeps/unix/sysv/linux/pause.c:29
...
```

14.4 Debugging Holoscan Python Application

The Holoscan SDK provides support for tracing and profiling tools, particularly focusing on the `compute` method of Python operators. Debugging Python operators using Python IDEs can be challenging since this method is invoked from the C++ runtime. This also applies to the `initialize`, `start`, and `stop` methods of Python operators.

Users can leverage IDEs like VSCode/PyCharm (which utilize the [PyDev.Debugger](#)) or other similar tools to debug Python operators:

- For VSCode, refer to [VSCode Python Debugging](#).
- For PyCharm, consult [PyCharm Python Debugging](#).

Subsequent sections will detail methods for debugging, profiling, and tracing Python applications using the Holoscan SDK.

14.4.1 pdb example

The following command initiates a Python application within a `pdb` debugger session:

```
python python/tests/system/test_pytracing.py pdb

# Type the following commands to check if the breakpoints are hit:
#
#  b test_pytracing.py:76
#  c
#  exit
```

This is an interactive session.

Please type the following commands to check if the breakpoints are hit.

```
(Pdb) b test_pytracing.py:76
Breakpoint 1 at /workspace/holoscan-sdk/python/tests/system/test_pytracing.py:76
(Pdb) c
...
> /workspace/holoscan-sdk/python/tests/system/test_pytracing.py(76)start()
```

(continues on next page)

(continued from previous page)

```
-> print("Mx start")
(Pdb) exit
```

For more details, please refer to the `pdb_main()` method in `test_pytracing.py`.

14.4.2 Profiling a Holoscan Python Application

For profiling, users can employ tools like `cProfile` or `line_profiler` for profiling Python applications/operators.

Note that when using a multithreaded scheduler, `cProfile` or the `profile` module might not accurately identify worker threads, or errors could occur.

In such cases with multithreaded schedulers, consider using multithread-aware profilers like `pyinstrument`, `pprofile`, or `yappi`.

For further information, refer to the test case at `test_pytracing.py`.

Using pyinstrument

`pyinstrument` is a call stack profiler for Python, designed to highlight performance bottlenecks in an easily understandable format directly in your terminal as the code executes.

```
python -m pip install pyinstrument
pyinstrument python/tests/system/test_pytracing.py

## Note: With a multithreaded scheduler, the same method may appear multiple times_
↪ across different threads.
# pyinstrument python/tests/system/test_pytracing.py -s multithread
```

```
...
0.107 [root] None
├─ 0.088 MainThread <thread>:140079743820224
│   └─ 0.088 <module> ../../bin/pyinstrument:1
│       └─ 0.088 main pyinstrument/__main__.py:28
│           [7 frames hidden] pyinstrument, <string>, runpy, <built-in>
│           0.087 _run_code runpy.py:63
│               └─ 0.087 <module> test_pytracing.py:1
│                   └─ 0.061 main test_pytracing.py:153
│                       └─ 0.057 MyPingApp.compose test_pytracing.py:141
│                           └─ 0.041 PingMxOp.__init__ test_pytracing.py:59
│                               └─ 0.041 PingMxOp.__init__ ../core/__init__.py:262
│                                   [35 frames hidden] .., numpy, re, sre_compile, sre_
↪ parse...
│                   └─ 0.015 [self] test_pytracing.py
│                       └─ 0.002 [self] test_pytracing.py
│                   └─ 0.024 <module> ../__init__.py:1
│                       [5 frames hidden] .., <built-in>
│                   └─ 0.001 <module> ../conditions/__init__.py:1
│                       [2 frames hidden] .., <built-in>
├─ 0.019 Dummy-1 <thread>:140078275749440
│   └─ 0.019 <module> ../../bin/pyinstrument:1
│       └─ 0.019 main pyinstrument/__main__.py:28
```

(continues on next page)

(continued from previous page)

```
[5 frames hidden] pyinstrument, <string>, runpy
0.019 _run_code runpy.py:63
└─ 0.019 <module> test_pytracing.py:1
    └─ 0.019 main test_pytracing.py:153
        └─ 0.014 [self] test_pytracing.py
            └─ 0.004 PingRxOp.compute test_pytracing.py:118
                └─ 0.004 print <built-in>
```

Using pprofile

pprofile is a line-granularity, thread-aware deterministic and statistic pure-python profiler.

```
python -m pip install pprofile
pprofile --include test_pytracing.py python/tests/system/test_pytracing.py -s multithread
```

```
Total duration: 0.972872s
File: python/tests/system/test_pytracing.py
File duration: 0.542628s (55.78%)
```

Line #	Hits	Time	Time per hit	%	Source code
...					
33	0	0	0	0.00%	
34	2	2.86102e-06	1.43051e-06	0.00%	def setup(self, spec: _
↳ OperatorSpec):					
35	1	1.62125e-05	1.62125e-05	0.00%	spec.output("out")
36	0	0	0	0.00%	
37	2	3.33786e-06	1.66893e-06	0.00%	def initialize(self):
38	1	1.07288e-05	1.07288e-05	0.00%	print("Tx initialize")
39	0	0	0	0.00%	
40	2	1.40667e-05	7.03335e-06	0.00%	def start(self):
41	1	1.23978e-05	1.23978e-05	0.00%	print("Tx start")
42	0	0	0	0.00%	
43	2	3.09944e-05	1.54972e-05	0.00%	def stop(self):
44	1	2.88486e-05	2.88486e-05	0.00%	print("Tx stop")
45	0	0	0	0.00%	
46	4	4.05312e-05	1.01328e-05	0.00%	def compute(self, op_input, op_
↳ output, context):					
47	3	2.57492e-05	8.58307e-06	0.00%	value = self.index
48	3	2.12193e-05	7.07308e-06	0.00%	self.index += 1

Using yappi

yappi is a tracing profiler that is multithreading, asyncio and gevent aware.

```
python -m pip install yappi
# yappi requires setting a context ID callback function to specify the correct context.
↳ ID for
# Holoscan's worker threads.
# For more details, please see `yappi_main()` in `test_pytracing.py`.
python python/tests/system/test_pytracing.py yappi | grep test_pytracing.py
```

(continues on next page)

(continued from previous page)

```
## Note: With a multithreaded scheduler, method hit counts are distributed across
↳ multiple threads.
#python python/tests/system/test_pytracing.py yappi -s multithread | grep test_pytracing.
↳ py
```

```
...
test_pytracing.py main:153 1
test_pytracing.py MyPingApp.compose:141 1
test_pytracing.py PingMxOp.__init__:59 1
test_pytracing.py PingTxOp.__init__:29 1
test_pytracing.py PingMxOp.setup:65 1
test_pytracing.py PingRxOp.__init__:99 1
test_pytracing.py PingRxOp.setup:104 1
test_pytracing.py PingTxOp.setup:34 1
test_pytracing.py PingTxOp.initialize:37 1
test_pytracing.py PingRxOp.stop:115 1
test_pytracing.py PingRxOp.initialize:109 1
test_pytracing.py PingMxOp.initialize:72 1
test_pytracing.py PingMxOp.stop:78 1
test_pytracing.py PingMxOp.compute:81 3
test_pytracing.py PingTxOp.compute:46 3
test_pytracing.py PingRxOp.compute:118 3
test_pytracing.py PingTxOp.start:40 1
test_pytracing.py PingMxOp.start:75 1
test_pytracing.py PingRxOp.start:112 1
test_pytracing.py PingTxOp.stop:43 1
```

Using profile/cProfile

profile/cProfile is a deterministic profiling module for Python programs.

```
python -m cProfile python/tests/system/test_pytracing.py 2>&1 | grep test_pytracing.py
```

```
## Executing a single test case
#python python/tests/system/test_pytracing.py profile
```

1	0.001	0.001	0.107	0.107	test_pytracing.py:1(<module>)
1	0.000	0.000	0.000	0.000	test_pytracing.py:104(setup)
1	0.000	0.000	0.000	0.000	test_pytracing.py:109(initialize)
1	0.000	0.000	0.000	0.000	test_pytracing.py:112(start)
1	0.000	0.000	0.000	0.000	test_pytracing.py:115(stop)
3	0.000	0.000	0.000	0.000	test_pytracing.py:118(compute)
1	0.000	0.000	0.000	0.000	test_pytracing.py:140(MyPingApp)
1	0.014	0.014	0.073	0.073	test_pytracing.py:141(compose)
1	0.009	0.009	0.083	0.083	test_pytracing.py:153(main)
1	0.000	0.000	0.000	0.000	test_pytracing.py:28(PingTxOp)
1	0.000	0.000	0.000	0.000	test_pytracing.py:29(__init__)
1	0.000	0.000	0.000	0.000	test_pytracing.py:34(setup)
1	0.000	0.000	0.000	0.000	test_pytracing.py:37(initialize)

(continues on next page)

(continued from previous page)

1	0.000	0.000	0.000	0.000	test_pytracing.py:40(start)
1	0.000	0.000	0.000	0.000	test_pytracing.py:43(stop)
3	0.000	0.000	0.000	0.000	test_pytracing.py:46(compute)
1	0.000	0.000	0.000	0.000	test_pytracing.py:58(PingMxOp)
1	0.000	0.000	0.058	0.058	test_pytracing.py:59(__init__)
1	0.000	0.000	0.000	0.000	test_pytracing.py:65(setup)
1	0.000	0.000	0.000	0.000	test_pytracing.py:72(initialize)
1	0.000	0.000	0.000	0.000	test_pytracing.py:75(start)
1	0.000	0.000	0.000	0.000	test_pytracing.py:78(stop)
3	0.001	0.000	0.001	0.000	test_pytracing.py:81(compute)
1	0.000	0.000	0.000	0.000	test_pytracing.py:98(PingRxOp)
1	0.000	0.000	0.000	0.000	test_pytracing.py:99(__init__)

Using line_profiler

line_profiler is a module for doing line-by-line profiling of functions.

```
python -m pip install line_profiler
```

```
# Insert `@profile` before the function `def compute(self, op_input, op_output,
↪context):`.
```

```
# The original file will be backed up as `test_pytracing.py.bak`.
```

```
file="python/tests/system/test_pytracing.py"
```

```
pattern="    def compute(self, op_input, op_output, context):"
```

```
insertion="    @profile"
```

```
if ! grep -q "^$insertion" "$file"; then
```

```
    sed -i.bak "/^$pattern/i\
```

```
$insertion" "$file"
```

```
fi
```

```
kernprof -lv python/tests/system/test_pytracing.py
```

```
# Remove the inserted `@profile` decorator.
```

```
mv "$file.bak" "$file"
```

```
...
```

```
Wrote profile results to test_pytracing.py.lprof
```

```
Timer unit: 1e-06 s
```

```
Total time: 0.000304244 s
```

```
File: python/tests/system/test_pytracing.py
```

```
Function: compute at line 46
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
=====					
46					@profile
47					def compute(self, op_input, op_
↪48					output, context):
48	3	2.3	0.8	0.8	value = self.index
49	3	9.3	3.1	3.0	self.index += 1

(continues on next page)

(continued from previous page)

50					
51	3	0.5	0.2	0.2	output = []
52	18	5.0	0.3	1.6	for i in range(0, 5):
53	15	4.2	0.3	1.4	output.append(value)
54	15	2.4	0.2	0.8	value += 1
55					
56	3	280.6	93.5	92.2	op_output.emit(output, "out")
...					

14.4.3 Measuring Code Coverage

The Holoscan SDK provides support for measuring code coverage using [Coverage.py](#).

```
python -m pip install coverage

coverage erase
coverage run examples/ping_vector/python/ping_vector.py
coverage report examples/ping_vector/python/ping_vector.py
coverage html

# Open the generated HTML report in a browser.
xdg-open htmlcov/index.html
```

To record code coverage programmatically, please refer to the `coverage_main()` method in `test_pytracing.py`.

You can execute the example application with code coverage enabled by running the following command:

```
python -m pip install coverage
python python/tests/system/test_pytracing.py coverage
# python python/tests/system/test_pytracing.py coverage -s multithread
```

The following command starts a Python application using the `trace` module:

```
python -m trace --trackcalls python/tests/system/test_pytracing.py | grep test_pytracing
```

```
...
test_pytracing.main -> test_pytracing.MyPingApp.compose
test_pytracing.main -> test_pytracing.PingMxOp.compute
test_pytracing.main -> test_pytracing.PingMxOp.initialize
test_pytracing.main -> test_pytracing.PingMxOp.start
test_pytracing.main -> test_pytracing.PingMxOp.stop
test_pytracing.main -> test_pytracing.PingRxOp.compute
test_pytracing.main -> test_pytracing.PingRxOp.initialize
test_pytracing.main -> test_pytracing.PingRxOp.start
test_pytracing.main -> test_pytracing.PingRxOp.stop
test_pytracing.main -> test_pytracing.PingTxOp.compute
test_pytracing.main -> test_pytracing.PingTxOp.initialize
test_pytracing.main -> test_pytracing.PingTxOp.start
test_pytracing.main -> test_pytracing.PingTxOp.stop
```

A test case utilizing the `trace` module programmatically can be found in the `trace_main()` method in `test_pytracing.py`.

```
python python/tests/system/test_pytracing.py trace
# python python/tests/system/test_pytracing.py trace -s multithread
```


WRITING PYTHON BINDINGS FOR A C++ OPERATOR

For convenience while maintaining high performance, *operators written in C++* can be wrapped in Python. The general approach uses `Pybind11` to concisely create bindings that provide a familiar, Pythonic experience to application authors.

Note: While we provide some utilities to simplify part of the process, this section is designed for advanced developers, since the wrapping of the C++ class using `pybind11` is mostly manual and can vary between each operator.

The existing `Pybind11` documentation is good and it is recommended to read at least the basics on wrapping `functions` and `classes`. The material below will assume some basic familiarity with `Pybind11`, covering the details of creation of the bindings of a C++ `Operator`. As a concrete example, we will cover creation of the bindings for `ToolTrackingPostprocessorOp` from `Holohub` as a simple case and then highlight additional scenarios that might be encountered.

Tip: There are several examples of bindings on `Holohub` in the `operators` folder. The subset of operators that provide a Python wrapper on top of a C++ implementation will have any C++ headers and sources together in a common folder, while any corresponding Python bindings will be in a “python” subfolder (see the `tool_tracking_postprocessor` folder layout, for example).

There are also several `examples of bindings` for the built-in operators of the SDK. Unlike on `Holohub`, for the SDK, the corresponding C++ `headers` and `sources` of an operator are stored under separate directory trees.

Warning: It is recommended to put any cleanup of resources allocated in the C++ operator’s `initialize()` and/or `start()` methods into the `stop()` method of the operator and **not** in its destructor. This is necessary as a workaround to a current issue where it is not guaranteed that the destructor always gets called prior to Python application termination. The `stop()` method will always be explicitly called, so we can be assured that any cleanup happens as expected.

15.1 Tutorial: binding the `ToolTrackingPostprocessorOp` class

15.1.1 Creating a `PyToolTrackingPostprocessorOp` trampoline class

In a C++ file (`tool_tracking_postprocessor.cpp` in this case), create a subclass of the C++ `Operator` class to wrap. The general approach taken is to create a Python-specific class that provides a constructor that takes a `Fragment*`, an explicit list of the operators parameters with default values for any that are optional, and an operator name. This constructor needs to setup the operator as done in `Fragment::make_operator`, so that it is ready for initialization by the GXF executor. We use the convention of prepending “Py” to the C++ class name for this (so, `PyToolTrackingPostprocessorOp` in this case). :

Listing 15.1: tool_tracking_post_processor/python/tool_tracking_post_processor.cpp

```

class PyToolTrackingPostprocessorOp : public ToolTrackingPostprocessorOp {
public:
    /* Inherit the constructors */
    using ToolTrackingPostprocessorOp::ToolTrackingPostprocessorOp;

    // Define a constructor that fully initializes the object.
    PyToolTrackingPostprocessorOp(
        Fragment* fragment, const py::args& args, std::shared_ptr<Allocator> device_
↪ allocator,
        std::shared_ptr<Allocator> host_allocator, float min_prob = 0.5f,
        std::vector<std::vector<float>> overlay_img_colors = VIZ_TOOL_DEFAULT_COLORS,
        std::shared_ptr<holoscan::CudaStreamPool> cuda_stream_pool = nullptr,
        const std::string& name = "tool_tracking_postprocessor")
        : ToolTrackingPostprocessorOp(ArgList{Arg{"device_allocator", device_allocator},
        Arg{"host_allocator", host_allocator},
        Arg{"min_prob", min_prob},
        Arg{"overlay_img_colors", overlay_img_colors}
↪ ,
        }) {
        if (cuda_stream_pool) { this->add_arg(Arg{"cuda_stream_pool", cuda_stream_pool}); }
        add_positional_condition_and_resource_args(this, args);
        name_ = name;
        fragment_ = fragment;
        spec_ = std::make_shared<OperatorSpec>(fragment);
        setup(*spec_.get());
    }
};

```

This constructor will allow providing a Pythonic experience for creating the operator. Specifically, the user can pass Python objects for any of the parameters without having to explicitly create any `holoscan::Arg` objects via `holoscan.core.Arg`. For example, a standard Python float can be passed to `min_prob` and a Python list[`list[float]`] can be passed for `overlay_img_colors` (Pybind11 handles conversion between the C++ and Python types). Pybind11 will also take care of conversion of a Python allocator class like `holoscan.resources.UnboundedAllocator` or `holoscan.resources.BlockMemoryPool` to the underlying C++ `std::shared_ptr<holoscan::Allocator>` type. The arguments `device_allocator` and `host_allocator` correspond to required Parameters of the C++ class and can be provided from Python either positionally or via keyword while the Parameters `min_prob` and `overlay_img_colors` will be optional keyword arguments. `cuda_stream_pool` is also optional, but is only conditionally passed as an argument to the underlying `ToolTrackingPostprocessorOp` constructor when it is not a `nullptr`.

- For all operators, the first argument should be `Fragment* fragment` and is the fragment the operator will be assigned to. In the case of a single fragment application (i.e. not a distributed application), the fragment is just the application itself.
- An (optional) `const std::string& name` argument should be provided to enable the application author to set the operator's name.
- The `const py::args& args` argument corresponds to the `*args` notation in Python. It is a set of 0 or more positional arguments. It is not required to provide this in the function signature, but is recommended in order to enable passing additional conditions such as a `CountCondition` or `PeriodicCondition` as positional arguments to the operator. The call below to

```
add_positional_condition_and_resource_args(this, args);
```

uses a helper function defined in `operator_util.hpp` to add any Condition or Resource arguments found in the list of positional arguments.

- The other arguments all correspond to the various parameters (`holoscan::Parameter`) that are defined for the C++ `ToolTrackingPostProcessorOp` class.
 - All other parameters except `cuda_stream_pool` are passed directly in the argument list to the parent `ToolTrackingPostProcessorOp` class. The parameters present on the C++ operator can be seen in its header [here](#) with default values taken from the `setup` method of the source file [here](#). Note that `CudaStreamHandler` is a utility that will add a parameter of type `Parameter<std::shared_ptr<CudaStreamPool>>`.
 - The `cuda_stream_pool` argument is only conditionally added if it was not `nullptr` (Python's `None`). This is done via

```
if (cuda_stream_pool) { this->add_arg(Arg{"cuda_stream_pool", cuda_stream_pool}
↪); }
```

instead of passing it as part of the `holoscan::ArgList` provided to the `ToolTrackingPostprocessorOp` constructor call above.

The remaining lines of the constructor

```
name_ = name;
fragment_ = fragment;
spec_ = std::make_shared<OperatorSpec>(fragment);
setup(*spec_.get());
```

are required to properly initialize it and should be the same across all operators. These correspond to equivalent code within the `Fragment::make_operator` method.

15.1.2 Defining the Python module

For this operator, there are no other custom classes aside from the operator itself, so we define a module using `PYBIND11_MODULE` as shown below with only a single class definition. This is done in the same `tool_tracking_postprocessor.cpp` file where we defined the `PyToolTrackingPostprocessorOp` trampoline class.

The following header will always be needed.

```
#include <pybind11/pybind11.h>

namespace py = pybind11;
using pybind11::literals::operator""_a;
```

Here, we typically also add defined the `py` namespace as a shorthand for `pybind11` and indicated that we will use the `_a` literal (it provides a shorthand notation when [defining keyword arguments](#)).

Often it will be necessary to include the following header if any parameters to the operator involve C++ standard library containers such as `std::vector` or `std::unordered_map`.

```
#include <pybind11/stl.h>
```

This allows `pybind11` to cast between the C++ container types and corresponding Python types (Python `dict` / C++ `std::unordered_map`, for example).

Listing 15.2: tool_tracking_post_processor/python/tool_tracking_post_processor.cpp

```

PYBIND11_MODULE(_tool_tracking_postprocessor, m) {
    py::class_<ToolTrackingPostprocessorOp,
        PyToolTrackingPostprocessorOp,
        Operator,
        std::shared_ptr<ToolTrackingPostprocessorOp>>(
        m,
        "ToolTrackingPostprocessorOp",
        doc::ToolTrackingPostprocessorOp::doc_ToolTrackingPostprocessorOp_python)
        .def(py::init<Fragment*,
            const py::args& args,
            std::shared_ptr<Allocator>,
            std::shared_ptr<Allocator>,
            float,
            std::vector<std::vector<float>>>,
            std::shared_ptr<holoscan::CudaStreamPool>,
            const std::string&>(),
            "fragment"_a,
            "device_allocator"_a,
            "host_allocator"_a,
            "min_prob"_a = 0.5f,
            "overlay_img_colors"_a = VIZ_TOOL_DEFAULT_COLORS,
            "cuda_stream_pool"_a = py::none(),
            "name"_a = "tool_tracking_postprocessor"s,
            doc::ToolTrackingPostprocessorOp::doc_ToolTrackingPostprocessorOp_python);
} // PYBIND11_MODULE NOLINT

```

Note:

- If you are implementing the python wrapping in Holohub, the `<module_name>` passed to `PYBIND_11_MODULE` **must** match `<CPP_CMAKE_TARGET>` as *covered below*).
- If you are implementing the python wrapping in a standalone CMake project, the `<module_name>` passed to `PYBIND_11_MODULE` **must** match the name of the module passed to the `pybind11::add_module` CMake function.

Using a mismatched name in `PYBIND_11_MODULE` will result in failure to import the module from Python.

The order in which the classes are specified in the `py::class_<>` template call is important and should follow the convention shown here. The first in the list is the C++ class name (`ToolTrackingPostprocessorOp`) and second is the `PyToolTrackingPostprocessorOp` class we defined above with the additional, explicit constructor. We also need to list the parent `Operator` class so that all of the methods such as `start`, `stop`, `compute`, `add_arg`, etc. that were already wrapped for the parent class don't need to be redefined here.

The single `.def(py::init<... call wraps the PyToolTrackingPostprocessorOp constructor we wrote above. As such, the argument types provided to py::init<> must exactly match the order and types of arguments in that constructor's function signature. The subsequent arguments to def are the names and default values (if any) for the named arguments in the same order as the function signature. Note that the const py::args& args (Python *args) argument is not listed as these are positional arguments that don't have a corresponding name. The use of py::none() (Python's None) as the default for cuda_stream_pool corresponds to the nullptr in the C++ function signature. The "_a" literal used in the definition is enabled by the following declaration earlier in the file.`

The final argument to `.def` here is a documentation string that will serve as the Python docstring for the function. It is optional and we chose here to define it in a separate header as described in the next section.

15.1.3 Documentation strings

Prepare documentation strings (const char*) for your python class and its parameters.

Note: Below we use a PYDOC macro defined in the [SDK](#) and available in [HoloHub](#) as a utility to remove leading spaces. In this case, the documentation code is located in header file `tool_tracking_post_processor_pydoc.hpp`, under a custom `holoscan::doc::ToolTrackingPostprocessorOp` namespace. None of this is required, you just need to make any documentation strings available for use as an argument to the `py::class_` constructor or method definition calls.

Listing 15.3: `tool_tracking_post_processor/python/tool_tracking_post_processor_pydoc.hpp`

```
#include "../macros.hpp"

namespace holoscan::doc {

namespace ToolTrackingPostprocessorOp {

// PyToolTrackingPostprocessorOp Constructor
PYDOC(ToolTrackingPostprocessorOp_python, R"doc(
Operator performing post-processing for the endoscopy tool tracking demo.

**==Named Inputs==**

    in : nvidia::gxf::Entity containing multiple nvidia::gxf::Tensor
        Must contain input tensors named "probs", "scaled_coords" and "binary_masks" that
        correspond to the output of the LSTMTensorRTInferenceOp as used in the endoscopy
        tool tracking example applications.

**==Named Outputs==**

    out_coords : nvidia::gxf::Tensor
        Coordinates tensor, stored on the host (CPU).

    out_mask : nvidia::gxf::Tensor
        Binary mask tensor, stored on device (GPU).

Parameters
-----
fragment : Fragment
    The fragment that the operator belongs to.
device_allocator : ``holoscan.resources.Allocator``
    Output allocator used on the device side.
host_allocator : ``holoscan.resources.Allocator``
    Output allocator used on the host side.
min_prob : float, optional
    Minimum probability (in range [0, 1]). Default value is 0.5.
overlay_img_colors : sequence of sequence of float, optional
    Color of the image overlays, a list of RGB values with components between 0 and 1.
    The default value is a qualitative colormap with a sequence of 12 colors.
cuda_stream_pool : ``holoscan.resources.CudaStreamPool``, optional
    ``holoscan.resources.CudaStreamPool`` instance to allocate CUDA streams.
```

(continues on next page)

(continued from previous page)

```

    Default value is ``None``.
name : str, optional
    The name of the operator.
)doc")

} // namespace ToolTrackingPostprocessorOp
} // namespace holoscan::doc

```

We tend to use NumPy-style docstrings for parameters, but also encourage adding a custom section at the top that describes the input and output ports and what type of data is expected on them. This can make it easier for developers to use the operator without having to inspect the source code to determine this information.

15.1.4 Configuring with CMake

We use CMake to configure pybind11 and build the bindings for the C++ operator you wish to wrap. There are two approaches detailed below, one for HoloHub (recommended), one for standalone CMake projects.

Tip: To have your bindings built, ensure the CMake code below is executed as part of a CMake project which already defines the C++ operator as a CMake target, either built in your project (with `add_library`) or imported (with `find_package` or `find_library`).

In HoloHub

We provide a CMake utility function named `pybind11_add_holohub_module` in HoloHub to facilitate configuring and building your python bindings.

In our skeleton code below, a top-level `CMakeLists.txt` which already defined the `tool_tracking_postprocessor` target for the C++ operator would need to do `add_subdirectory(tool_tracking_postprocessor)` to include the following `CMakeLists.txt`. The `pybind11_add_holohub_module` lists that C++ operator target, the C++ class to wrap, and the path to the C++ binding source code we implemented above. Note how the module name provided as the first argument to `PYPBIND11_MODULE` needs to match `_CPP_CMAKE_TARGET` (`_tool_tracking_postprocessor_op` in this case).

Listing 15.4: `tool_tracking_postprocessor/python/CMakeLists.txt`

```

include(pybind11_add_holohub_module)
pybind11_add_holohub_module(
    CPP_CMAKE_TARGET tool_tracking_postprocessor
    CLASS_NAME "ToolTrackingPostprocessorOp"
    SOURCES tool_tracking_postprocessor.cpp
)

```

The key details here are that `CLASS_NAME` should match the name of the C++ class that is being wrapped and is also the name that will be used for the class from Python. `SOURCES` should point to the file where the C++ operator that is being wrapped is defined. The `CPP_CMAKE_TARGET` name will be the name of the holohub package submodule that will contain the operator.

Note that the python subdirectory where this `CMakeLists.txt` resides is reachable thanks to the `add_subdirectory(python)` in the `CMakeLists.txt` one folder above, but that's an arbitrary opinionated location and not a required directory structure.

Standalone CMake

Follow the [pybind11 documentation](#) to configure your CMake project to use pybind11. Then, use the `pybind11_add_module` function with the cpp files containing the code above, and link against `holoscan::core` and the library that exposes your C++ operator to wrap.

Listing 15.5: `my_op_python/CMakeLists.txt`

```
pybind11_add_module(my_python_module my_op_pybind.cpp)
target_link_libraries(my_python_module
    PRIVATE holoscan::core
    PUBLIC my_op
)
```

Example: in the SDK, this is done [here](#).

Warning: The name chosen for `CPP_CMAKE_TARGET` **must** also be used (along with a preceding underscore) as the module name passed as the first argument to the `PYBIND11_MODULE` macro in the bindings.

Note that there is an initial underscore prepended to the name. This is the naming convention used for the shared library and corresponding `__init__.py` file that will be generated by the `pybind11_add_holohub_module` helper function above.

If the name is specified incorrectly, the build will still complete, but at application run time an `ImportError` such as the following would occur

```
[command] python3 /workspace/holohub/applications/endoscopy_tool_tracking/python/
↪endoscopy_tool_tracking.py --data /workspace/holohub/data/endoscopy
Traceback (most recent call last):
  File "/workspace/holohub/applications/endoscopy_tool_tracking/python/endoscopy_tool_
↪tracking.py", line 38, in <module>
    from holohub.tool_tracking_postprocessor import ToolTrackingPostprocessorOp
  File "/workspace/holohub/build/python/lib/holohub/tool_tracking_postprocessor/__
↪init__.py", line 19, in <module>
    from ._tool_tracking_postprocessor import ToolTrackingPostprocessorOp
ImportError: dynamic module does not define module export function (PyInit__tool_
↪tracking_postprocessor)
```

15.1.5 Importing the class in Python

In Holohub

When building your project, two files will be generated inside `<build_or_install_dir>/python/lib/holohub/<CPP_CMAKE_TARGET>` (e.g. `build/python/lib/holohub/tool_tracking_postprocessor/`):

1. the shared library for your bindings (`_tool_tracking_postprocessor_op.cpython-<pyversion>-<arch>-linux-gnu.so`)
2. an `__init__.py` file that makes the necessary imports to expose this in python

Assuming you have `export PYTHONPATH=<build_or_install_dir>/python/lib/`, you should then be able to create an application in Holohub that imports your class via:

```
from holohub.tool_tracking_postprocessor_op import ToolTrackingPostProcessorOp
```

Example: ToolTrackingPostProcessorOp is imported in the Endoscopy Tool Tracking application on HoloHub [here](#).

Standalone CMake

When building your project, a shared library file holding the python bindings and named `my_python_module.cpython-<pyversion>-<arch>-linux-gnu.so` will be generated inside `<build_or_install_dir>/my_op_python` (configurable with `OUTPUT_NAME` and `LIBRARY_OUTPUT_DIRECTORY` respectively in CMake).

From there, you can import it in python via:

```
import holoscan.core
import holoscan.gxf # if your c++ operator uses gxf extensions

from <build_or_install_dir>.my_op_python import MyOp
```

Tip: To imitate HoloHub's behavior, you can also place that file alongside the `.so` file, name it `__init__.py`, and replace `<build_or_install_dir>` by `..`. It can then be imported as a python module, assuming `<build_or_install_dir>` is a module under the `PYTHONPATH` environment variable.

15.2 Additional Examples

In this section we will cover other cases that may occasionally be encountered when writing Python bindings for operators.

15.2.1 Optional arguments

It is also possible to use `std::optional` to handle optional arguments. The `ToolTrackingProcessorOp` example above, for example, has a default argument defined in the spec for `min_prob`.

```
constexpr float DEFAULT_MIN_PROB = 0.5f;
// ...

spec.param(
    min_prob_, "min_prob", "Minimum probability", "Minimum probability.", DEFAULT_MIN_
    ↪PROB);
```

In the tutorial for `ToolTrackingProcessorOp` above we reproduced this default of 0.5 in both the `PyToolTrackingProcessorOp` constructor function signature as well as the Python bindings defined for it. This carries the risk that the default could change at the C++ operator level without a corresponding change being made for Python.

An alternative way to define the constructor would have been to use `std::optional` as follows

```
// Define a constructor that fully initializes the object.
PyToolTrackingPostprocessorOp(
    Fragment* fragment, const py::args& args, std::shared_ptr<Allocator> device_
    ↪allocator,
```

(continues on next page)

(continued from previous page)

```

std::shared_ptr<Allocator> host_allocator, std::optional<float> min_prob = 0.5f,
std::optional<std::vector<std::vector<float>>> overlay_img_colors = VIZ_TOOL_
→DEFAULT_COLORS,
std::shared_ptr<holoscan::CudaStreamPool> cuda_stream_pool = nullptr,
const std::string& name = "tool_tracking_postprocessor")
: ToolTrackingPostprocessorOp(ArgList{Arg{"device_allocator", device_allocator},
                                       Arg{"host_allocator", host_allocator},
                                       }) {
if (cuda_stream_pool) { this->add_arg(Arg{"cuda_stream_pool", cuda_stream_pool}); }
if (min_prob.has_value()) { this->add_arg(Arg{"min_prob", min_prob.value() }); }
if (overlay_img_colors.has_value()) {
    this->add_arg(Arg{"overlay_img_colors", overlay_img_colors.value() });
}
add_positional_condition_and_resource_args(this, args);
name_ = name;
fragment_ = fragment;
spec_ = std::make_shared<OperatorSpec>(fragment);
setup(*spec_.get());
}

```

where now that `min_prob` and `overlay_img_colors` are optional, they are only conditionally added as an argument to `ToolTrackingPostprocessorOp` when they have a value. If this approach is used, the Python bindings for the constructor should be updated to use `py::none()` as the default as follows:

```

.def(py::init<Fragment*,
    const py::args& args,
    std::shared_ptr<Allocator>,
    std::shared_ptr<Allocator>,
    float,
    std::vector<std::vector<float>>,
    std::shared_ptr<holoscan::CudaStreamPool>,
    const std::string&>(),
    "fragment"_a,
    "device_allocator"_a,
    "host_allocator"_a,
    "min_prob"_a = py::none(),
    "overlay_img_colors"_a = py::none(),
    "cuda_stream_pool"_a = py::none(),
    "name"_a = "tool_tracking_postprocessor"s,
    doc::ToolTrackingPostprocessorOp::doc_ToolTrackingPostprocessorOp_python);

```

15.2.2 C++ enum parameters as arguments

Sometimes, operators may use a `Parameter` with an enum type. It is necessary to wrap the C++ enum to be able to use it as a Python type when providing the argument to the operator.

The built-in `holoscan::ops::AJASourceOp` is an example of a C++ operator that takes a `enum Parameter` (an `NTV2Channel` enum).

The enum can easily be wrapped for use from Python via `py::enum_` as shown [here](#). It is recommended in this case to follow Python's convention of using capitalized names in the enum.

15.2.3 (Advanced) Custom C++ classes as arguments

Sometimes it is necessary to accept a custom C++ class type as an argument in the operator's constructor. In this case additional interface code and bindings will likely be necessary to support the type.

A relatively simple example of this is the `DataVecMap` type used by `InferenceProcessorOp`. In that case, the type is a structure that holds an internal `std::map<std::string, std::vector<std::string>>`. The bindings are written to accept a Python dict (`py::dict`) and a [helper function](#) is used within the constructor to convert that dictionary to the corresponding C++ `DataVecMap`.

A more complicated case is the use of a `InputSpec` type in the `HolovizOp` bindings. This case involves creating Python bindings for classes `InputSpec` and `View` as well as a couple of enum types. To avoid the user having to build a `list[holoscan.operators.HolovizOp.InputSpec]` directly to pass as the `tensors` argument, an [additional Python wrapper class](#) was defined in the `__init__.py` to allow passing a simple Python dict for the `tensors` argument and any corresponding `InputSpec` classes are automatically created in its constructor before calling the underlying Python bindings class.

15.2.4 Customizing the C++ types a Python operator can emit or receive

In some instances, users may wish to be able to have a Python operator receive and/or emit a custom C++ type. As a first example, suppose we are wrapping an operator that emits a custom C++ type. We need any downstream native Python operators to be able to receive that type. By default the SDK is able to handle the needed C++ types for built in operators like `std::vector<holoscan::ops::HolovizOp::InputSpec>`. The SDK provides an `EmitterReceiverRegistry` class that 3rd party projects can use to register receiver and emitter methods for any custom C++ type that needs to be handled. To handle a new type, users should implement an `emitter_receiver<T>` struct for the desired type as in the example below. We will first cover the general steps necessary to register such a type and then cover where some steps may be omitted in certain simple cases.

Step 1: define `emitter_receiver::emit` and `emitter_receiver::receive` methods

Here is an example for the built-in `std::vector<holoscan::ops::HolovizOp::InputSpec>` used by `HolovizOp` to define the input specifications for its received tensors.

```
#include <holoscan/python/core/emitter_receiver_registry.hpp>

namespace py = pybind11;

namespace holoscan {

/* Implements emit and receive capability for the HolovizOp::InputSpec type.
 */
template <
struct emitter_receiver<std::vector<holoscan::ops::HolovizOp::InputSpec>> {
    static void emit(py::object& data, const std::string& name, PyOutputContext& op_output,
                    const int64_t acq_timestamp = -1) {
        auto input_spec = data.cast<std::vector<holoscan::ops::HolovizOp::InputSpec>>();
        py::gil_scoped_release release;
        op_output.emit<std::vector<holoscan::ops::HolovizOp::InputSpec>>(input_spec, name.c_
↪str(), acq_timestamp);
        return;
    }

    static py::object receive(std::any result, const std::string& name, PyInputContext& op_
↪input) {
```

(continues on next page)

(continued from previous page)

```

HOLOSCAN_LOG_DEBUG("py_receive: std::vector<HolovizOp::InputSpec> case");
// can directly return vector<InputSpec>
auto specs = std::any_cast<std::vector<holoscan::ops::HolovizOp::InputSpec>>(result);
py::object py_specs = py::cast(specs);
return py_specs;
}
};
}

```

This `emitter_receiver` class defines a `receive` method that takes a `std::any` message and casts it to the corresponding Python `list[HolovizOp.InputSpec]` object. Here the `pybind11::cast` call works because we have wrapped the `HolovizOp::InputSpec` class [here](#).

Similarly, the `emit` method takes a `pybind11::object` (of type `list[HolovizOp.InputSpec]`) and casts it to the corresponding C++ type, `std::vector<holoscan::ops::HolovizOp::InputSpec>`. The conversion between `std::vector` and a Python list is one of `Pbind11`'s built-in conversions (available as long as “`pybind11/stl.h`” has been included).

The signature of the `emit` and `receive` methods must exactly match the case shown here.

Step 2: Create a `register_types` method for adding custom types to the `EmitterReceiverRegistry`.

The bindings in this operators module, should define a method named `register_types` that takes a reference to an `EmitterReceiverRegistry` as its only argument. Within this function there should be a call to `EmitterReceiverRegistry::add_emitter_receiver` for each type that this operator wished to register. The `HolovizOp` defines this method using a lambda function

```

// Import the emitter/receiver registry from holoscan.core and pass it to this_
function to
// register this new C++ type with the SDK.
m.def("register_types", [](EmitterReceiverRegistry& registry) {
    registry.add_emitter_receiver<std::vector<holoscan::ops::HolovizOp::InputSpec>>(
        "std::vector<HolovizOp::InputSpec>"s);
    // array camera pose object
    registry.add_emitter_receiver<std::shared_ptr<std::array<float, 16>>>(
        "std::shared_ptr<std::array<float, 16>>"s);
    // Pose3D camera pose object
    registry.add_emitter_receiver<std::shared_ptr<nvidia::gxf::Pose3D>>(
        "std::shared_ptr<nvidia::gxf::Pose3D>"s);
    // camera_eye_input, camera_look_at_input, camera_up_input
    registry.add_emitter_receiver<std::array<float, 3>>("std::array<float, 3>"s);
});

```

Here the following line registers the `std::vector<holoscan::ops::HolovizOp::InputSpec>` type that we wrote an `emitter_receiver` for above.

```

registry.add_emitter_receiver<std::vector<holoscan::ops::HolovizOp::InputSpec>>(
    "std::vector<HolovizOp::InputSpec>"s);

```

Internally the registry stores a mapping between the C++ `std::type_index` of the type specified in the template argument and the `emitter_receiver` defined for that type. The second argument is a string that the user can choose which is a label for the type. As we will see later, this label can be used from Python to indicate that we want to emit using the `emitter_receiver::emit` method that was registered for a particular label.

Step 3: In the `init.py` file for the Python module defining the operator call `register_types`

To register types with the core SDK, we need to import the `io_type_registry` class (of type `EmitterReceiverRegistry`) from `holoscan.core`. We then pass that class as input to the `register_types` method defined in step 2 to register the 3rd party types with the core SDK.

```
from holoscan.core import io_type_registry

from ._holoviz import register_types as _register_types

# register methods for receiving or emitting list[HolovizOp.InputSpec] and camera pose_
↪ types
_register_types(io_type_registry)
```

where we chose to import `register_types` with an initial underscore as a common Python convention to indicate it is intended to be “private” to this module.

In some cases steps 1 and 3 as shown above are not necessary.

When creating Python bindings for an Operator on Holohub, the `pybind11_add_holohub_module.cmake` utility mentioned above will take care of autogenerating the `__init__.py` as shown in step 3, so it will not be necessary to manually create it in that case.

For types for which Pybind11’s default casting between C++ and Python is adequate, it is not necessary to explicitly define the `emitter_receiver` class as shown in step 1. This is true because there are a couple of [default implementations](#) for `emitter_receiver<T>` and `emitter_receiver<std::shared_ptr<T>>` that already cover common cases. The default `emitter_receiver` works for the `std::vector<HolovizOp::InputSpec>` type shown above, which is why the code shown for illustration there is [not found within the operator’s bindings](#). In that case one could immediately implement `register_types` from step 2 without having to explicitly create an `emitter_receiver` class.

An example where the default `emitter_receiver` would not work is the custom one defined by the SDK for `pybind11::dict`. In this case, to provide convenient emit of multiple tensors via passing a `dict[holoscan::Tensor]` to `op_output.emit` we have special handling of Python dictionaries. The dictionary is inspected and if all keys are strings and all values are tensor-like objects, a single C++ `nvidia::gfx::Entity` containing all of the tensors as an `nvidia::gfx::Tensor` is emitted. If the dictionary is not a tensor map, then it is just emitted as a shared pointer to the Python dict object. The `emitter_receiver` implementations used for the core SDK are defined in [emitter_receivers.hpp](#). These can serve as a reference when creating new ones for additional types.

Runtime behavior of emit and receive

After registering a new type, receive of that type on any input port will automatically be handled. This is because due to the strong typing of C++, any `op_input.receive` call in an operator’s `compute` method can find the registered receive method that matches the `std::type_index` of the type and use that to convert to a corresponding Python object.

Because Python is not strongly typed, on emit, the default behavior remains emitting a shared pointer to the Python object itself. If we instead want to emit a C++ type, we can pass a 3rd argument to `op_output.emit` to specify the name that we used when registering the types via the `add_emitter_receiver` call as above.

Example of emitting a C++ type

As a concrete example, the SDK already registers `std::string` by default. If we wanted, for instance, to emit a Python string as a C++ `std::string` for use by a downstream operator that is wrapping a C++ operator expecting string input, we would add a 3rd argument to the `op_output.emit` call as follows

```
# emit a Python filename string on port "filename_out" using registered type "std::string"
↪
my_string = "filename.raw"
op_output.emit(my_string, "filename_out", "std::string")
```

This specifies that the `emit` method that converts to C++ `std::string` should be used instead of the default behavior of emitting the Python string.

Another example would be to emit a Python `List[float]` as a `std::array<float, 3>` parameter as input to the `camera_eye`, `camera_look_at` or `camera_up` input ports of `HolovizOp`.

```
op_output.emit([0.0, 1.0, 0.0], "camera_eye_out", "std::array<float, 3>")
```

Only types registered with the SDK can be specified by name in this third argument to `emit`.

Table of types registered by the core SDK

The list of types that are registered with the SDK's `EmitterReceiverRegistry` are given in the table below.

C++ Type	name in the EmitterReceiverRegistry
<code>holoscan::Tensor</code>	" <code>holoscan::Tensor</code> "
<code>std::shared_ptr<holoscan::GILGuardedPyObject></code>	" <code>PyObject</code> "
<code>std::string</code>	" <code>std::string</code> "
<code>pybind11::dict</code>	" <code>pybind11::dict</code> "
<code>holoscan::gfx::Entity</code>	" <code>holoscan::gfx::Entity</code> "
<code>holoscan::PyEntity</code>	" <code>holoscan::PyEntity</code> "
<code>nullptr_t</code>	" <code>nullptr_t</code> "
<code>CloudPickleSerializedObject</code>	" <code>CloudPickleSerializedObject</code> "
<code>std::array<float, 3></code>	" <code>std::array<float, 3></code> "
<code>std::shared_ptr<std::array<float, 16>></code>	" <code>std::shared_ptr<std::array<float, 16>></code> "
<code>std::shared_ptr<nvidia::gfx::Pose3D></code>	" <code>std::shared_ptr<nvidia::gfx::Pose3D></code> "
<code>std::vector<holoscan::ops::HolovizOp::InputSpec></code>	" <code>std::vector<HolovizOp::InputSpec></code> "

Note: There is no requirement that the registered name match any particular convention. We generally used the C++ type as the name to avoid ambiguity, but that is not required.

The sections above explain how a `register_types` function can be added to bindings to expand this list. It is also possible to get a list of all currently registered types, including those that have been registered by any additional imported 3rd party modules. This can be done via

```
from holoscan.core import io_type_registry

print(io_type_registry.registered_types())
```


SIMPLIFIED PYTHON OPERATOR CREATION VIA THE `CREATE_OP` DECORATOR

Warning: The `holoscan.decorator.create_op()` decorator and the supporting `holoscan.decorator.Input` and `holoscan.decorator.Output` classes are new in Holoscan v2.2 and are still considered experimental. They are usable now, but it is possible that some backwards incompatible changes to the behavior or API may be made based on initial feedback.

For convenience, a `holoscan.decorator.create_op()` decorator is provided which can be used to automatically convert a simple Python function/generator or a class into a native Python `holoscan.core.Operator`. The wrapped function body (or the `__call__` method if `create_op` is applied to a class) will correspond to the computation to be done in the `holoscan.core.Operator.compute()` method, but without any need to explicitly make any calls to `holoscan.core.InputContext.receive()` to receive inputs or `holoscan.core.OutputContext.emit()` to transmit the output. Any necessary input or output ports will have been automatically generated.

Consider first a simple Python function named `mask_and_offset` that takes `image` and `mask` tensors as input and multiplies them, followed by adding some scalar offset.

```
def mask_and_offset(image, mask, offset=1.5):  
    return image * mask + offset
```

To turn this into an function that returns a corresponding operator we can add the `create_op` decorator like this:

```
from holoscan.decorator import create_op  
  
@create_op(  
    inputs=("image", "mask"),  
    outputs="out",  
)  
def mask_and_offset(image, mask, offset=1.5):  
    return image * mask + offset
```

By supplying the `inputs` argument we are specifying that there are two input ports, named “image” and “mask”. By setting `outputs="out"` we are indicating that the output will be transmitted on a port named “out”. When `inputs` are specified by simple strings in this way, the names used must map to variable names in the wrapped function’s signature. We will see later that it is possible to use the `holoscan.decorator.Input` class to provide more control over how inputs are mapped to function arguments. Similarly, we will see that the `holoscan.decorator.Output` class can be used to provide more control over how the function output is mapped to any output port(s).

There is also an optional, `cast_tensors` argument to `create_op`. For convenience, this defaults to `True`, which results in any tensor-like objects being automatically cast to a NumPy or CuPy array (for host or device tensors, re-

spectively) before they are passed on to the function. If this is not desired (e.g. due to working with a different third party tensor framework than NumPy or CuPy), the user can set `cast_tensors=False`, and manually handle casting of any `holoscan.Tensor` objects to the desired form in the function body. This casting option applies to either single tensors or a tensor map (`dict[Tensor]`).

This decorated function can then be used within the `compose` method of an `Application` to create an operator corresponding to this computation:

```
from holoscan.core import Application, Operator

def MyApp(Application):

    def compose(self)
        mask_op = mask_and_offset(self, name="my_scaling_op", offset=0.0)

        # verify that an Operator was generated
        assert(isinstance(mask_op, Operator))

        # now add any additional operators and create the computation graph using add_
        ↪ flow
```

Note that as for all other `Operator` classes, it is **required** to supply the application (or fragment) as the first argument (`self` here). The name kwarg is always supported and is the name that will be assigned to the operator. Due to the use of this kwarg to specify the operator name, the wrapped function (`mask_and_offset` in this case) should not use name as an argument name. In this case, we specified `offset=0.0` which would override the default value of `offset=1.5` in the function signature.

For completeness, the use of the `create_op` decorator on `mask_and_offset` is equivalent to if the user had defined the following `MaskAndOffsetOp` class and used it in `MyApp.compose`:

```
def MaskAndOffsetOp(Operator):
    def setup(self, spec):
        spec.input("image")
        spec.input("mask")
        spec.output("out")

    def compute(self, op_input, op_output, context):
        # Simplified logic here assumes received values are GPU tensors
        # create_op would add additional logic to handle the inputs
        image = op_input.receive("image")
        image = cp.asarray(image)

        mask = op_input.receive("mask")
        mask = cp.asarray(mask)

        out = image * mask + offset
        op_output.emit(out, "out")
```

16.1 Decorating a function that returns a tuple of arrays

Let's consider another example where function takes in multiple arrays, processes them, and returns a tuple of updated arrays:

```
def scale_and_offset(x1, x2, scale=2.0, offset=1.5):
    y1 = x1 * scale
    y2 = x2 + offset
    return y1, y2
```

To turn this into a corresponding operator we can add the `create_op` decorator like this:

```
@create_op(
    inputs=("x1", "x2"),
    outputs=("out1", "out2"),
)
def scale_and_offset(x1, x2, scale=2.0, offset=1.5):
    y1 = x1 * scale
    y2 = x2 + offset
    return y1, y2
```

As before, the messages received through the ports defined by `inputs`, “x1” and “x2”, will be mapped to respective variables `x1` and `x2`. Likewise, the elements of the output tuple, arrays `y1` and `y2`, will be emitted through ports “out1” and “out2”, respectively. In contrast to input mapping, which is determined by the naming of ports and variables, the output mapping is determined by the ordering of output ports and elements in the tuple returned by the function.

16.2 Using the Input class for more control over input ports

This section will cover additional use cases where using a `str` or `Tuple[str]` for the `inputs` argument is insufficient.

Scenario 1: Assume that the upstream operator sends a `tensormap` to a given input port and we need to specify which tensor(s) in the `tensormap` will map to which input port.

For a concrete example, suppose we want to print a tensor's shape using a function like:

```
def print_shape(tensor):
    print(f"{tensor.shape = }")
```

but the upstream operator outputs a dictionary containing two tensors named “image” and “labels”. We could use this operator by specifying which tensor name on a particular input port would map to the function's “tensor” argument. For example:

```
@create_op(inputs=Input("input_tensor", arg_map={"image": "tensor"}))
def print_shape(tensor):
    print(f"{tensor.shape = }")
```

would create an operator with a single input port named “input_tensor” and no output port. The input port may receive a `tensormap` with any number of tensors, but will only use the tensor named “image”, mapping it to the “tensor” argument of the wrapped function. In general, the `arg_map` is a dictionary mapping tensor names found on the port to their corresponding function argument names.

Scenario 2: we want to override the scheduling condition present on a port. This can be done by specifying `Input` with the `condition` and optionally `condition_kwargs` arguments. For example, to override the `MessageAvailableCondition` that is added to the port by default and allow it to call `compute` even when no input message is available:

```
@create_op(inputs=Input("input_tensor", condition=ConditionType.NONE, condition_kwargs={})
↪))
```

Scenario 3: we want to override the parameters of the receiver present on a port. For example, we could specify a different policy for the double buffer receiver that is used by default (policy=1 corresponds to discarding incoming messages when the queue is already full)

```
@create_op(inputs=Input("input_tensor", connector=ConditionType.DOUBLE_BUFFER, connector_
↪kwargs=dict(capacity=1, policy=1)))
```

16.3 Using the Output class for more control over output ports

To support a case where multiple output ports should be present, the user must have the function return a dict. The `holoscan.decorator.Output` class then has a `tensor_names` keyword argument that can be specified to indicate which items in the dictionary are to be transmitted on a given output port.

For example, assume we have a function that generates three tensors, `x`, `y` and `z` and we want to transmit `x` and `y` on port “out1” while `z` will be transmitted on port “out2”. This can be done by specifying outputs as follows in the `create_op` call:

```
@create_op(
    outputs=(
        Output("out1", tensor_names=("x", "y")),
        Output("out2", tensor_names=("z",)),
    ),
)
def xyz_generator(nx=32, ny=32, nz=16):
    x = cp.arange(nx, dtype=np.uint32)
    y = cp.arange(ny, dtype=np.uint32)
    z = cp.arange(nz, dtype=np.uint32)

    # must return a dict type when Output arg(s) with `tensor_names` is used
    return dict(x=x, y=y, z=z)
```

This operator has no input ports and three optional keyword arguments. It splits the output tensors across two ports as described above. All names used in `tensor_names` must correspond to keys present in the dict emitted by the object. Often the dict values are tensors, but that is not a requirement.

The `holoscan.decorator.Output` class also supports `condition`, `condition_kwargs`, `connector` and `connector_kwargs` that work in the same way as shown for `holoscan.decorator.Input` above. For example, to override the transmitter queue policy for a single output port named “output_tensor”

```
@create_op(inputs=Output("output_tensor", connector=ConditionType.DOUBLE_BUFFER,
↪connector_kwargs=dict(capacity=1, policy=1)))
```

note that `tensor_names` was not specified which means that the returned object does not need to be a dict. The object itself will be emitted on the “output_tensor” port.

Note: When specifying the inputs and outputs arguments to `create_op`, please make sure that all ports have unique names. As a concrete example, if an operator has a single input and output port that are used to send images, one should use unique port names like “image_in” and “image_out” rather than using “image” for both.

16.4 Interoperability with wrapped C++ operators

The SDK includes a `python_decorator` example showing interoperability of wrapped C++ operators (`VideoStreamReplayerOp` and `HolovizOp`) alongside native Python operators created via the `create_op` decorator.

The start of this application imports a couple of the built in C++-based operators with Python bindings (`HolovizOp` and `VideoStreamReplayerOp`). In addition to these, two new operators are created via the `create_op` decorator APIs.

```
import os

from holoscan.core import Application
from holoscan.decorator import Input, Output, create_op
from holoscan.operators import HolovizOp, VideoStreamReplayerOp

sample_data_path = os.environ.get("HOLOSCAN_INPUT_PATH", "../data")

@create_op(
    inputs="tensor",
    outputs="out_tensor",
)
def invert(tensor):
    tensor = 255 - tensor
    return tensor

@create_op(inputs=Input("in", arg_map="tensor"), outputs=Output("out", tensor_names=(
    ↪ "frame",)))
def tensor_info(tensor):
    print(f"tensor from 'in' port: shape = {tensor.shape}, " f"dtype = {tensor.dtype}.
    ↪ name}")
    return tensor
```

The first is created by adding the decorator to a function named `invert` which just inverts the (8-bit RGB) color space values. A second operator, is created by adding the decorator to a function named `tensor_info`, which assumes that the input is a CuPy or NumPy tensor, and prints its shape and data type. Note that `create_op`'s default `cast_tensors=True` option ensures that any host or device tensors are cast to NumPy or CuPy arrays, respectively. This is why it is safe to use NumPy APIs in the function bodies. If the user wants to receive the `holoscan.Tensor` object directly and manually handle the casting to a different type of object in the function body, then `cast_tensors=False` should be specified in the keyword arguments to `create_op`.

Now that we have defined or imported all of the operators, we can build an application in the usual way by inheriting from the `Application` class and implementing the `compose` method. The remainder of the code for this example is shown below.

```
class VideoReplayerApp(Application):
    """Example of an application that uses the operators defined above.

    This application has the following operators:

    - VideoStreamReplayerOp
    - HolovizOp
```

(continues on next page)

(continued from previous page)

```

- invert (created via decorator API)
- tensor_info (created via decorator API)

`VideoStreamReplayerOp` reads a video file and sends the frames to the HolovizOp.
The `invert` operator inverts the color map (the 8-bit `value` in each color channel_
↪ is
set to `255 - value`).
The `tensor_info` operator prints information about the tensors shape and data type.
`HolovizOp` displays the frames.
"""

def compose(self):
    video_dir = os.path.join(sample_data_path, "racerx")
    if not os.path.exists(video_dir):
        raise ValueError(f"Could not find video data: {video_dir}")

    # Define the replayer and holoviz operators
    replayer = VideoStreamReplayerOp(
        self,
        name="replayer",
        directory=video_dir,
        basename="racerx",
        frame_rate=0, # as specified in timestamps
        repeat=False, # default: false
        realtime=True, # default: true
        count=40, # default: 0 (no frame count restriction)
    )
    invert_op = invert(self, name="image_invert")
    info_op = tensor_info(self, name="tensor_info")
    visualizer = HolovizOp(
        self,
        name="holoviz",
        width=854,
        height=480,
        # name="frame" to match Output argument to create_op for tensor_info
        tensors=[dict(name="frame", type="color", opacity=1.0, priority=0)],
    )
    # Define the workflow
    self.add_flow(replayer, invert_op, {("output", "tensor")})
    self.add_flow(invert_op, info_op, {("out_tensor", "in")})
    self.add_flow(info_op, visualizer, {("out", "receivers")})

def main():
    app = VideoReplayerApp()
    app.run()

if __name__ == "__main__":
    main()

```

The highlighted lines show how Operators corresponding to the `invert` and `tensor_info` functions are created by passing the application itself as the first argument. The `invert_op` and `info_op` variables now correspond to a

`holoscan.core.Operator` class and can be connected in the usual way using `add_flow` to define the computation. Note that a name was provided for these operators, via the optional `name` keyword argument. In this case each operator is only used once, but if the same operator is to be used more than once in an application, each should be given a unique name.

16.5 Using `create_op` to turn a generator into an Operator

The `create_op` decorator can be applied to a generator in the same way as for a function. In this case, a `BooleanCondition` will automatically be added to the operator that will stop it from trying to call `compute` again once the generator is exhausted (has no more values to yield). The following is a basic example of decorating a generator for integers from 1 to `count`:

```
@create_op(outputs="out")
def source_generator(count):
    yield from range(1, count + 1)
```

The `compose` method can then create an operator from this decorated generator as follows

```
count_op = source_generator(self, count=100, name="int_source")
```

16.6 Using `create_op` to turn a class into an Operator

The `create_op` decorator can also be applied to a class implementing the `__call__` method, to turn it into an `Operator()`. One reason to choose a class vs. a function is if there is some internal state that needs to be maintained across calls. For example, the operator defined below casts the input data to 32-bit floating point and on even frames also negates the values.

```
@create_op
class NegateEven:
    def __init__(self, start_index=0):
        self.counter = start_index

    def __call__(self, x):
        # cast tensor to 32-bit floating point
        x = x.astype('float32')

        # negate the values if the frame is even
        if self.counter % 2 == 0:
            x = -x
        return x
```

In this case, since there is only a single input and output for the function, we can omit the `inputs` and `outputs` arguments in the call to `create_op`. In this case the input port will have name `"x"`, as determined from the variable name in the function signature. The output port will have an empty name `""`. To use different port names, the `inputs` and/or `outputs` arguments should be specified.

The `compose` method can then create an operator from this decorated generator as follows. Note that any positional or keyword arguments in the `__init__` method would be supplied during the `NegateEven` call. This returns a function (not yet an operator) that can then be called to generate the operator. This is shown below

```
negate_op_creation_func = NegateEven(start_index=0) # `negate_op_
↳ creation_func` is a function that returns an Operator
negate_even_op = negate_op_creation_func(self, name="negate_even") # call the function,
↳ to create an instance of NegateEvenOp
```

or more concisely as just

```
negate_even_op = NegateEven(start_index=0)(self, name="negate_even")
```

Note that the operator class as defined above is approximately equivalent to the Python native operator defined below. We show it here explicitly for reference.

```
import cupy as cp
import numpy as np

class NegateEvenOp(Operator):

    def __init__(self, fragment, *args, start_index=0, **kwargs):
        self.counter = start_index
        super().__init__(fragment, *args, **kwargs)

    def setup(self, spec):
        spec.input("x")
        spec.output("")

    def compute(op_input, op_output, context):
        x = op_input.receive("x")

        # cast to CuPy or NumPy array
        # (validation that `x` is a holoscan.Tensor is omitted for simplicity)
        if hasattr(x, '__cuda_array_interface__'):
            x = cupy.asarray(x)
        else:
            x = numpy.asarray(x)

        # cast tensor to 32-bit floating point
        x = x.astype('float32')

        # negate the values if the frame is even
        if self.counter % 2 == 0:
            x = -x

        op_output.emit(x, "")
```

The primary differences between this `NegateEvenOp` class and the decorated `NegateEven` above are:

- `NegateEven` does not need to define a `setup` method
- `NegateEven` does not inherit from `Operator` and so does not call its `__init__` from the constructor.
- The `NegateEven::__call__` method is simpler than the `NegateEvenOp::compute` method as `receive` and `emit` methods do not need to be explicitly called and casting to a NumPy or CuPy array is automatically handled for `NegateEven`.

BUILT-IN OPERATORS AND EXTENSIONS

The units of work of Holoscan applications are implemented within Operators, as described in the *core concepts* of the SDK. The operators included in the SDK provide domain-agnostic functionalities such as IO, machine learning inference, processing, and visualization, optimized for AI streaming pipelines, relying on a set of *Core Technologies*.

(holoscan-operators=)

17.1 Operators

The operators below are defined under the `holoscan::ops` namespace for C++ and CMake, and under the `holoscan.operators` module in Python.

Class	CMake target/lib	Documentation
AJASourceOp	<code>aja</code>	C++/Python
BayerDemosaicOp	<code>bayer_demosaic</code>	C++/Python
FormatConverterOp	<code>format_converter</code>	C++/Python
HolovizOp	<code>holoviz</code>	C++/Python
InferenceOp	<code>inference</code>	C++/Python
InferenceProcessorOp	<code>inference_processor</code>	C++/Python
PingRxOp	<code>ping_rx</code>	C++/Python
PingTensorRxOp	<code>ping_tensor_rx</code>	C++/Python
PingTensorTxOp	<code>ping_tensor_tx</code>	C++/Python
PingTxOp	<code>ping_tx</code>	C++/Python
SegmentationPostprocessorOp	<code>segmentation_postprocessor</code>	C++/Python
VideoStreamRecorderOp	<code>video_stream_recorder</code>	C++/Python
VideoStreamReplayerOp	<code>video_stream_replayer</code>	C++/Python
V4L2VideoCaptureOp	<code>v4l2</code>	C++/Python

Given an instance of an operator class, you can print a human-readable description of its specification to inspect the inputs, outputs, and parameters that can be configured on that operator class:

C++

```
std::cout << operator_object->spec()->description() << std::endl;
```

Python

```
print(operator_object.spec)
```

Note: The Holoscan SDK uses meta-programming with templating and `std::any` to support arbitrary data types. Because of this, some type information (and therefore values) might not be retrievable by the `description` API. If more details are needed, we recommend inspecting the list of `Parameter` members in the operator `header` to identify their type.

17.2 Extensions

The Holoscan SDK also includes some GXF extensions with GXF codelets, which are typically wrapped as operators, or present for legacy reasons. In addition to the core GXF extensions (`std`, `cuda`, `serialization`, `multimedia`) listed [here](#), the Holoscan SDK includes the following GXF extensions:

- `gxf_holoscan_wrapper`
- `ucx_holoscan`

17.2.1 GXF Holoscan Wrapper

The `gxf_holoscan_wrapper` extension includes the `holoscan::gxf::OperatorWrapper` codelet. It is used as a utility base class to wrap a Holoscan operator to interface with the GXF framework.

Learn more about it in the *Using Holoscan Operators in GXF Applications* section.

17.2.2 UCX (Holoscan)

The `ucx_holoscan` extension includes `nvidia::holoscan::UcxHoloscanComponentSerializer` which is a `nvidia::gxf::ComponentSerializer` that handles serialization of `holoscan::Message` and `holoscan::Tensor` types for transmission using the Unified Communication X (UCX) library. UCX is the library used by Holoscan SDK to enable communication of data between fragments in distributed applications.

Note: The `UcxHoloscanComponentSerializer` is intended for use in combination with other UCX components defined in the GXF UCX extension. Specifically, it can be used by the `UcxEntitySerializer` where it can operate alongside the `UcxComponentSerializer` that serializes GXF-specific types (`nvidia::gxf::Tensor`, `nvidia::gxf::VideoBuffer`, etc.). This way both GXF and Holoscan types can be serialized by distributed applications.

17.2.3 HoloHub

Visit the [HoloHub repository](#) to find a collection of additional Holoscan operators and extensions.

VISUALIZATION

18.1 Overview

Holoviz provides the functionality to composite real-time streams of frames with multiple different other layers like segmentation mask layers, geometry layers, and GUI layers.

For maximum performance, Holoviz makes use of [Vulkan](#), which is already installed as part of the NVIDIA GPU driver.

Holoscan provides the *Holoviz operator* which is sufficient for many, even complex visualization tasks. The *Holoviz operator* is used by multiple Holoscan [example applications](#).

Additionally, for more advanced use cases, the *Holoviz module* can be used to create application-specific visualization operators. The *Holoviz module* provides a C++ API and is also used by the *Holoviz operator*.

The term Holoviz is used for both the *Holoviz operator* and the *Holoviz module* below. Both the operator and the module roughly support the same feature set. Where applicable, information on how to use a feature with the operator and the module is provided. It's explicitly mentioned below when features are not supported by the operator.

18.2 Layers

The core entity of Holoviz are layers. A layer is a two-dimensional image object. Multiple layers are composited to create the final output.

These layer types are supported by Holoviz:

- Image layer
- Geometry layer
- GUI layer

All layers have common attributes which define the look and also the way layers are finally composited.

The priority determines the rendering order of the layers. Before rendering, the layers are sorted by priority. The layers with the lowest priority are rendered first, so that the layer with the highest priority is rendered on top of all other layers. If layers have the same priority, then the render order of these layers is undefined.

The example below draws a transparent geometry layer on top of an image layer (geometry data and image data creation is omitted in the code). Although the geometry layer is specified first, it is drawn last because it has a higher priority (1) than the image layer (0).

Operator

The operator has a `receivers` port which accepts tensors and video buffers produced by other operators. Each tensor or video buffer will result in a layer.

The operator autodetects the layer type for certain input types (e.g., a video buffer will result in an image layer).

For other input types or more complex use cases, input specifications can be provided either at initialization time as a parameter or dynamically at runtime.

```
std::vector<ops::HolovizOp::InputSpec> input_specs;

auto& geometry_spec =
    input_specs.emplace_back(ops::HolovizOp::InputSpec("point_tensor",
↳ops::HolovizOp::InputType::POINTS));
geometry_spec.priority_ = 1;
geometry_spec.opacity_ = 0.5;

auto& image_spec =
    input_specs.emplace_back(ops::HolovizOp::InputSpec("image_tensor",
↳ops::HolovizOp::InputType::IMAGE));
image_spec.priority_ = 0;

auto visualizer = make_operator<ops::HolovizOp>("holoviz", Arg("tensors", input_specs));

// the source provides two tensors named "point_tensor" and "image_tensor" at the
↳"outputs" port.
add_flow(source, visualizer, {"outputs", "receivers"});
```

Module

The definition of a layer is started by calling one of the layer begin functions `viz::BeginImageLayer()`, `viz::BeginGeometryLayer()` or `viz::BeginImGuiLayer()`. The layer definition ends with `viz::EndLayer()`.

The start of a layer definition is resetting the layer attributes like priority and opacity to their defaults. So for the image layer, there is no need to set the opacity to `1.0` since the default is already `1.0`.

```
namespace viz = holoscan::viz;

viz::Begin();

viz::BeginGeometryLayer();
viz::LayerPriority(1);
viz::LayerOpacity(0.5);
/// details omitted
viz::EndLayer();

viz::BeginImageLayer();
viz::LayerPriority(0);
/// details omitted
viz::EndLayer();

viz::End();
```


18.2.1 Image Layers

Operator

Image data can either be on host or device (GPU); both tensors and video buffers are accepted.

```
std::vector<ops::HolovizOp::InputSpec> input_specs;

auto& image_spec =
    input_specs.emplace_back(ops::HolovizOp::InputSpec("image",
ops::HolovizOp::InputType::IMAGE));

auto visualizer = make_operator<ops::HolovizOp>("holoviz", Arg("tensors", input_specs));

// the source provides an image named "image" at the "outputs" port.
add_flow(source, visualizer, {"output", "receivers"});
```

Module

The function `viz::BeginImageLayer()` starts an image layer. An image layer displays a rectangular 2D image.

The image data is defined by calling `viz::ImageCudaDevice()`, `viz::ImageCudaArray()` or `viz::ImageHost()`. Various input formats are supported, see `viz::ImageFormat`.

For single channel image formats image colors can be looked up by defining a lookup table with `viz::LUT()`.

```
viz::BeginImageLayer();
viz::ImageHost(width, height, format, data);
viz::EndLayer();
```

Supported Image Formats

Operator

Supported formats for `nvidia::gxf::VideoBuffer`.

nvidia::gxf::VideoFormat	Supported	Description
GXF_VIDEO_FORMAT_CUSTOM	-	
GXF_VIDEO_FORMAT_YUV420	✓	BT.601 multi planar 4:2:0 YUV
GXF_VIDEO_FORMAT_YUV420_ER	✓	BT.601 multi planar 4:2:0 YUV ER
GXF_VIDEO_FORMAT_YUV420_709	✓	BT.709 multi planar 4:2:0 YUV
GXF_VIDEO_FORMAT_YUV420_709_ER	✓	BT.709 multi planar 4:2:0 YUV ER
GXF_VIDEO_FORMAT_NV12	✓	BT.601 multi planar 4:2:0 YUV with interleaved UV
GXF_VIDEO_FORMAT_NV12_ER	✓	BT.601 multi planar 4:2:0 YUV ER with interleaved UV
GXF_VIDEO_FORMAT_NV12_709	✓	BT.709 multi planar 4:2:0 YUV with interleaved UV
GXF_VIDEO_FORMAT_NV12_709_ER	✓	BT.709 multi planar 4:2:0 YUV ER with interleaved UV
GXF_VIDEO_FORMAT_RGBA	✓	RGBA-8-8-8-8 single plane
GXF_VIDEO_FORMAT_BGRA	✓	BGRA-8-8-8-8 single plane
GXF_VIDEO_FORMAT_ARGB	✓	ARGB-8-8-8-8 single plane
GXF_VIDEO_FORMAT_ABGR	✓	ABGR-8-8-8-8 single plane
GXF_VIDEO_FORMAT_RGBX	✓	RGBX-8-8-8-8 single plane

continues on next page

Table 18.1 – continued from previous page

nvidia::gfx::VideoFormat	Supported	Description
GXF_VIDEO_FORMAT_BGRX	✓	BGRX-8-8-8-8 single plane
GXF_VIDEO_FORMAT_XRGB	✓	XRGB-8-8-8-8 single plane
GXF_VIDEO_FORMAT_XBGR	✓	XBGR-8-8-8-8 single plane
GXF_VIDEO_FORMAT_RGB	✓	RGB-8-8-8 single plane
GXF_VIDEO_FORMAT_BGR	✓	BGR-8-8-8 single plane
GXF_VIDEO_FORMAT_R8_G8_B8	-	RGB - unsigned 8 bit multiplanar
GXF_VIDEO_FORMAT_B8_G8_R8	-	BGR - unsigned 8 bit multiplanar
GXF_VIDEO_FORMAT_GRAY	✓	8 bit GRAY scale single plane
GXF_VIDEO_FORMAT_GRAY16	✓	16 bit GRAY scale single plane
GXF_VIDEO_FORMAT_GRAY32	-	32 bit GRAY scale single plane
GXF_VIDEO_FORMAT_GRAY32F	✓	float 32 bit GRAY scale single plane
GXF_VIDEO_FORMAT_RGB16	-	RGB-16-16-16 single plane
GXF_VIDEO_FORMAT_BGR16	-	BGR-16-16-16 single plane
GXF_VIDEO_FORMAT_RGB32	-	RGB-32-32-32 single plane
GXF_VIDEO_FORMAT_BGR32	-	BGR-32-32-32 single plane
GXF_VIDEO_FORMAT_R16_G16_B16	-	RGB - signed 16 bit multiplanar
GXF_VIDEO_FORMAT_B16_G16_R16	-	BGR - signed 16 bit multiplanar
GXF_VIDEO_FORMAT_R32_G32_B32	-	RGB - signed 32 bit multiplanar
GXF_VIDEO_FORMAT_B32_G32_R32	-	BGR - signed 32 bit multiplanar
GXF_VIDEO_FORMAT_NV24	-	multi planar 4:4:4 YUV with interleaved UV
GXF_VIDEO_FORMAT_NV24_ER	-	multi planar 4:4:4 YUV ER with interleaved UV
GXF_VIDEO_FORMAT_R8_G8_B8_D8	-	RGBD unsigned 8 bit multiplanar
GXF_VIDEO_FORMAT_R16_G16_B16_D16	-	RGBD unsigned 16 bit multiplanar
GXF_VIDEO_FORMAT_R32_G32_B32_D32	-	RGBD unsigned 32 bit multiplanar
GXF_VIDEO_FORMAT_RGBD8	-	RGBD 8 bit unsigned single plane
GXF_VIDEO_FORMAT_RGBD16	-	RGBD 16 bit unsigned single plane
GXF_VIDEO_FORMAT_RGBD32	-	RGBD 32 bit unsigned single plane
GXF_VIDEO_FORMAT_D32F	✓	Depth 32 bit float single plane
GXF_VIDEO_FORMAT_D64F	-	Depth 64 bit float single plane
GXF_VIDEO_FORMAT_RAW16_RGGB	-	RGGB-16-16-16-16 single plane
GXF_VIDEO_FORMAT_RAW16_BGGR	-	BGGR-16-16-16-16 single plane
GXF_VIDEO_FORMAT_RAW16_GRBG	-	GRBG-16-16-16-16 single plane
GXF_VIDEO_FORMAT_RAW16_GBRG	-	GBRG-16-16-16-16 single plane

Image format detection for `nvidia::gfx::Tensor`. Tensors don't have image format information attached. The Holoviz operator detects the image format from the tensor configuration.

nvidia::gfx::PrimitiveType	Channels	Color format	Index for color lookup
kUnsigned8	1	8 bit GRAY scale single plane	✓
kInt8	1	signed 8 bit GRAY scale single plane	✓
kUnsigned16	1	16 bit GRAY scale single plane	✓
kInt16	1	signed 16 bit GRAY scale single plane	✓
kUnsigned32	1	-	✓
kInt32	1	-	✓
kFloat32	1	float 32 bit GRAY scale single plane	✓
kUnsigned8	3	RGB-8-8-8 single plane	-
kInt8	3	signed RGB-8-8-8 single plane	-
kUnsigned8	4	RGBA-8-8-8-8 single plane	-
kInt8	4	signed RGBA-8-8-8-8 single plane	-
kUnsigned16	4	RGBA-16-16-16-16 single plane	-
kInt16	4	signed RGBA-16-16-16-16 single plane	-
kFloat32	4	RGBA-16-16-16-16 single plane	-

Module

See `viz::ImageFormat` for supported image formats. Additionally `viz::ImageComponentMapping()` can be used to map the color components of an image to the color components of the output.

18.2.2 Geometry Layers

A geometry layer is used to draw 2d or 3d geometric primitives. 2d primitives are points, lines, line strips, rectangles, ovals or text and are defined with 2d coordinates (x, y). 3d primitives are points, lines, line strips or triangles and are defined with 3d coordinates (x, y, z).

Coordinates start with (0, 0) in the top left and end with (1, 1) in the bottom right for 2d primitives.

Operator

See `holoviz_geometry.cpp` and `holoviz_geometry.py` for 2d geometric primitives and `holoviz_geometry.py` for 3d geometric primitives.

Module

The function `viz::BeginGeometryLayer()` starts a geometry layer.

See `viz::PrimitiveTopology` for supported geometry primitive topologies.

There are functions to set attributes for geometric primitives like color (`viz::Color()`), line width (`viz::LineWidth()`), and point size (`viz::PointSize()`).

The code below draws a red rectangle and a green text.

```
namespace viz = holoscan::viz;

viz::BeginGeometryLayer();

// draw a red rectangle
viz::Color(1.f, 0.f, 0.f, 0.f);
```

(continues on next page)

(continued from previous page)

```

const float data[]{0.1f, 0.1f, 0.9f, 0.9f};
viz::Primitive(viz::PrimitiveTopology::RECTANGLE_LIST, 1, sizeof(data) / sizeof(data[0]),
    ↪ data);

// draw green text
viz::Color(0.f, 1.f, 0.f, 0.f);
viz::Text(0.5f, 0.5f, 0.2f, "Text");

viz::EndLayer();

```

18.2.3 ImGui Layers

Note: ImGui layers are not supported when using the Holoviz operator.

The Holoviz module supports user interface layers created with [Dear ImGui](#).

Calls to the Dear ImGui API are allowed between `viz::BeginImGuiLayer()` and `viz::EndImGuiLayer()`, and are used to draw to the ImGui layer. The ImGui layer behaves like other layers and is rendered with the layer opacity and priority.

The code below creates a Dear ImGui window with a checkbox used to conditionally show a image layer.

```

namespace viz = holoscan::viz;

bool show_image_layer = false;
while (!viz::WindowShouldClose()) {
    viz::Begin();

    viz::BeginImGuiLayer();

    ImGui::Begin("Options");
    ImGui::Checkbox("Image layer", &show_image_layer);
    ImGui::End();

    viz::EndLayer();

    if (show_image_layer) {
        viz::BeginImageLayer();
        viz::ImageHost(...);
        viz::EndLayer();
    }

    viz::End();
}

```

ImGui is a static library and has no stable API. Therefore, the application and Holoviz have to use the same ImGui version. When the link target `holoscan::viz::imgui` is exported, make sure to link your application against that target.

18.2.4 Depth Map Layers

A depth map is a single channel 2d array where each element represents a depth value. The data is rendered as a 3D object using points, lines, or triangles. The color for the elements can also be specified.

Supported formats for the depth map:

- 8-bit unsigned normalized format that has a single 8-bit depth component
- 32-bit signed float format that has a single 32-bit depth component

Supported format for the depth color map:

- 32-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3

Depth maps are rendered in 3D and support camera movement.

Operator

```
std::vector<ops::HolovizOp::InputSpec> input_specs;

auto& depth_map_spec =
    input_specs.emplace_back(ops::HolovizOp::InputSpec("depth_map",
ops::HolovizOp::InputType::DEPTH_MAP));
depth_map_spec.depth_map_render_mode_ = ops::HolovizOp::DepthMapRenderMode::TRIANGLES;

auto visualizer = make_operator<ops::HolovizOp>("holoviz",
    Arg("tensors", input_specs));

// the source provides an depth map named "depth_map" at the "output" port.
add_flow(source, visualizer, {"output", "receivers"});
```

Module

See [holoviz depth map demo](#).

18.3 Views

By default, a layer will fill the whole window. When using a view, the layer can be placed freely within the window.

Layers can also be placed in 3D space by specifying a 3D transformation matrix.

Note: For geometry layers, there is a default matrix which allows coordinates in the range of [0 ... 1] instead of the Vulkan [-1 ... 1] range. When specifying a matrix for a geometry layer, this default matrix is overwritten.

When multiple views are specified, the layer is drawn multiple times using the specified layer view.

It's possible to specify a negative term for height, which flips the image. When using a negative height, you should also adjust the y value to point to the lower left corner of the viewport instead of the upper left corner.

Operator

See `holoviz_views.py`.

Module

Use `viz::LayerAddView()` to add a view to a layer.

18.4 Camera

When rendering 3d geometry using a geometry layer with 3d primitives or using a depth map layer the camera properties can either be set by the application or interactively changed by the user.

To interactively change the camera, use the mouse:

- Orbit (LMB)
- Pan (LMB + CTRL | MMB)
- Dolly (LMB + SHIFT | RMB | Mouse wheel)
- Look Around (LMB + ALT | LMB + CTRL + SHIFT)
- Zoom (Mouse wheel + SHIFT)

Operator

See `holoviz_camera.cpp`.

Module

Use `viz::SetCamera()` to change the camera.

18.5 Using a Display in Exclusive Mode

Typically, Holoviz opens a normal window on the Linux desktop. In that case, the desktop compositor is combining the Holoviz image with all other elements on the desktop. To avoid this extra compositing step, Holoviz can render to a display directly.

18.5.1 Configure a Display for Exclusive Use

Single display

SSH into the machine and stop the X server:

```
sudo systemctl stop display-manager
```

To resume the display manager, run:

```
sudo systemctl start display-manager
```

Multiple displays

The display to be used in exclusive mode needs to be disabled in the NVIDIA Settings application (nvidia-settings): open the X Server Display Configuration tab, select the display and under Configuration select Disabled. Press Apply.

18.5.2 Enable Exclusive Display in Holoviz

Operator

Arguments to pass to the Holoviz operator:

```
auto visualizer = make_operator<ops::HolovizOp>("holoviz",
  Arg("use_exclusive_display", true), // required
  Arg("display_name", "DP-2"), // optional
  Arg("width", 2560), // optional
  Arg("height", 1440), // optional
  Arg("framerate", 240) // optional
);
```

Module

Provide the name of the display and desired display mode properties to viz::Init().

If the name is nullptr, then the first display is selected.

The name of the display can either be the EDID name as displayed in the NVIDIA Settings, or the output name provided by xrandr or hwinfo --monitor.

Tip:

X11

In this example output of xrandr, DP-2 would be an adequate display name to use:

```
Screen 0: minimum 8 x 8, current 4480 x 1440, maximum 32767 x 32767
DP-0 disconnected (normal left inverted right x axis y axis)
DP-1 disconnected (normal left inverted right x axis y axis)
DP-2 connected primary 2560x1440+1920+0 (normal left inverted right x axis y axis) 600mm
↳x 340mm
  2560x1440    59.98 + 239.97* 199.99  144.00  120.00  99.95
  1024x768     60.00
   800x600     60.32
   640x480     59.94
USB-C-0 disconnected (normal left inverted right x axis y axis)
```

Wayland and X11

In this example output of `hwinfo`, `MSI MPG343CQR` would be an adequate display name to use:

```
$ hwinfo --monitor | grep Model
Model: "MSI MPG343CQR"
```

18.6 CUDA Streams

By default, Holoviz is using CUDA stream 0 for all CUDA operations. Using the default stream can affect concurrency of CUDA operations, see [stream synchronization behavior](#) for more information.

Operator

The operator is using a `holoscan::CudaStreamPool` instance if provided by the `cuda_stream_pool` argument. The stream pool is used to create a CUDA stream used by all Holoviz operations.

```
const std::shared_ptr<holoscan::CudaStreamPool> cuda_stream_pool =
    make_resource<holoscan::CudaStreamPool>("cuda_stream", 0, 0, 0, 1, 5);
auto visualizer =
    make_operator<holoscan::ops::HolovizOp>("visualizer",
        Arg("cuda_stream_pool") = cuda_stream_pool);
```

Module

When providing CUDA resources to Holoviz through (e.g., `viz::ImageCudaDevice()`), Holoviz is using CUDA operations to use that memory. The CUDA stream used by these operations can be set by calling `viz::SetCudaStream()`. The stream can be changed at any time.

18.7 Reading the Frame Buffer

The rendered frame buffer can be read back. This is useful when when doing offscreen rendering or running Holoviz in a headless environment.

Note: Reading the depth buffer is not supported when using the Holoviz operator.

Operator

To read back the color frame buffer, set the `enable_render_buffer_output` parameter to `true` and provide an allocator to the operator.

The frame buffer is emitted on the `render_buffer_output` port.

```
std::shared_ptr<holoscan::ops::HolovizOp> visualizer =
    make_operator<ops::HolovizOp>("visualizer",
        Arg("enable_render_buffer_output", true),
        Arg("allocator") = make_resource<holoscan::UnboundedAllocator>("allocator"),
        Arg("cuda_stream_pool") = cuda_stream_pool);

add_flow(visualizer, destination, {"render_buffer_output", "input"});
```

Module

The rendered color or depth buffer can be read back using `viz::ReadFramebuffer()`.

18.8 sRGB

The sRGB color space is supported for both images and the framebuffer. By default Holoviz is using a linear encoded framebuffer.

Operator

To switch the framebuffer color format set the `framebuffer_srgb` parameter to `true`.

To use sRGB encoded images set the `image_format` field of the `InputSpec` structure to a sRGB image format.

Module

Use the `viz::SetSurfaceFormat()` to set the framebuffer surface format to a sRGB color format.

To use sRGB encoded images set the `fmt` parameter of `viz::ImageCudaDevice()`, `viz::ImageCudaArray()` or `viz::ImageHost()` to a sRGB image format.

18.9 HDR

Holoviz supports selecting the framebuffer surface image format and color space. To enable HDR select a HDR color space.

Operator

Set the `framebuffer_color_space` parameter to a supported HDR color space.

Module

Use the `viz::GetSurfaceFormats()` to query the available surface formats.

Use the `viz::SetSurfaceFormat()` to set the framebuffer surface format to a surface format with a HDR color space.

18.9.1 Distributions supporting HDR

At the time of writing (08/2024) there is currently no official support for HDR on Linux. However there is experimental HDR support for Gnome version 44 and KDE Plasma 6.

KDE Plasma 6

Experimental HDR support is described in this [blog post](#). Three steps are required to use HDR with Holoviz:

1. Enable HDR in the display configuration
2. Install the [Vulkan HDR layer](#)
3. Set the `ENABLE_HDR_WSI` environment variable to 1.

Run `vulkaninfo` to verify that HDR color spaces are reported

```
> vulkaninfo
...
GPU id : 0 (NVIDIA RTX 6000 Ada Generation):
  Surface type = VK_KHR_wayland_surface
  Formats: count = 11
    SurfaceFormat[0]:
      format = FORMAT_A2B10G10R10_UNORM_PACK32
      colorSpace = COLOR_SPACE_SRGB_NONLINEAR_KHR
    SurfaceFormat[1]:
      format = FORMAT_A2R10G10B10_UNORM_PACK32
      colorSpace = COLOR_SPACE_SRGB_NONLINEAR_KHR
    SurfaceFormat[2]:
      format = FORMAT_R8G8B8A8_SRGB
      colorSpace = COLOR_SPACE_SRGB_NONLINEAR_KHR
    SurfaceFormat[3]:
      format = FORMAT_R8G8B8A8_UNORM
      colorSpace = COLOR_SPACE_SRGB_NONLINEAR_KHR
    SurfaceFormat[4]:
      format = FORMAT_B8G8R8A8_SRGB
      colorSpace = COLOR_SPACE_SRGB_NONLINEAR_KHR
    SurfaceFormat[5]:
      format = FORMAT_B8G8R8A8_UNORM
      colorSpace = COLOR_SPACE_SRGB_NONLINEAR_KHR
    SurfaceFormat[6]:
      format = FORMAT_R16G16B16A16_SFLOAT
      colorSpace = COLOR_SPACE_SRGB_NONLINEAR_KHR
    SurfaceFormat[7]:
```

(continues on next page)

(continued from previous page)

```

        format = FORMAT_A2B10G10R10_UNORM_PACK32
        colorSpace = COLOR_SPACE_HDR10_ST2084_EXT
SurfaceFormat[8]:
        format = FORMAT_A2R10G10B10_UNORM_PACK32
        colorSpace = COLOR_SPACE_HDR10_ST2084_EXT
SurfaceFormat[9]:
        format = FORMAT_R16G16B16A16_SFLOAT
        colorSpace = COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT
SurfaceFormat[10]:
        format = FORMAT_R16G16B16A16_SFLOAT
        colorSpace = COLOR_SPACE_BT709_LINEAR_EXT

```

...

Gnome version 44

Gnome version 44 is part of Ubuntu 24.04.

Experimental HDR support had been added with this [MR](#). To enable HDR, make sure Wayland is used (no HDR with X11), press Alt+F2, then type `lg` in the prompt to start the looking glass console. Enter `global.compositor.backend.get_monitor_manager().experimental_hdr = 'on'` to enable HDR, `global.compositor.backend.get_monitor_manager().experimental_hdr = 'off'` to disable HDR.

Gnome is not yet passing the HDR color space to Vulkan. The color space and other information can be queried using `drm_info.check_modes` where `HDR_OUTPUT_METADATA` is not `0`.

For the case below (Microstep MSI MPG343CQR display) the HDR color space is BT2020_RGB with SMPTE ST 2084 (PQ) EOTF. Max luminance is 446 nits.

```
> drm_info | grep -B 1 -A 11 HDR
...
|   |-- "Colorspace": enum {Default, BT2020_RGB, BT2020_YCC} = BT2020_RGB
|   |-- "HDR_OUTPUT_METADATA": blob = 114
|       |-- Type: Static Metadata Type 1
|       |-- EOTF: SMPTE ST 2084 (PQ)
|       |-- Display primaries:
|           |-- Red: (0.6768, 0.3096)
|           |-- Green: (0.2764, 0.6445)
|           |-- Blue: (0.1514, 0.0693)
|       |-- White point: (0.3135, 0.3291)
|       |-- Max display mastering luminance: 446 cd/m²
|       |-- Min display mastering luminance: 0.0001 cd/m²
|       |-- Max content light level: 446 cd/m²
|       |-- Max frame average light level: 446 cd/m²
-
...
```

18.10 Callbacks

Callbacks can be used to receive updates on key presses, mouse position and buttons, and window size.

Operator

C++

```
visualizer = make_operator<ops::HolovizOp>(
    "holoviz",
    Arg("key_callback",
        ops::HolovizOp::KeyCallbackFunction([](ops::HolovizOp::Key key,
↳ops::HolovizOp::KeyAndButtonAction action,
                                                ops::HolovizOp::KeyModifiers↳
↳modifiers) -> void {
    HOLOSCAN_LOG_INFO(
        "key {} action {} modifiers {}", int(key), int(action), *(uint32_t*)&
↳modifiers));
    }));
    )
```

Python

```
def callback(
    key: HolovizOp.Key, action: HolovizOp.KeyAndButtonAction, modifiers: HolovizOp.
↳KeyModifiers
):
    print(key, action, modifiers)

visualizer = HolovizOp(
    self,
    name="holoviz",
    key_callback=callback)
```

Module

```
void key_callback(void *user_pointer, Key key, KeyAndButtonAction action, KeyModifiers↳
↳modifiers) {
    ...
}
...
viz::SetKeyCallback(user_pointer, &key_callback);
...
```

18.11 Holoviz operator

18.11.1 Class Documentation

C++

Python.

18.11.2 Holoviz Operator Examples

There are multiple [examples](#), both in Python and C++, showing how to use various features of the Holoviz operator.

18.12 Holoviz module

18.12.1 Concepts

The Holoviz module uses the concept of the immediate mode design pattern for its API, inspired by the [Dear ImGui](#) library. The difference to the retained mode, for which most APIs are designed, is that there are no objects created and stored by the application. This makes it fast and easy to make visualization changes in a Holoscan application.

18.12.2 Instances

The Holoviz module uses a thread-local instance object to store its internal state. The instance object is created when calling the Holoviz module is first called from a thread. All Holoviz module functions called from that thread use this instance.

When calling into the Holoviz module from other threads other than the thread from which the Holoviz module functions were first called, make sure to call `viz::GetCurrent()` and `viz::SetCurrent()` in the respective threads.

There are usage cases where multiple instances are needed; for example, to open multiple windows. Instances can be created by calling `viz::Create()`. Call `viz::SetCurrent()` to make the instance current before calling the Holoviz module function to be executed for the window the instance belongs to.

18.12.3 Getting Started

The code below creates a window and displays an image.

First, the Holoviz module needs to be initialized. This is done by calling `viz::Init()`.

The elements to display are defined in the render loop; termination of the loop is checked with `viz::WindowShouldClose()`.

The definition of the displayed content starts with `viz::Begin()`, and ends with `viz::End()`. `viz::End()` starts the rendering and displays the rendered result.

Finally, the Holoviz module is shutdown with `viz::Shutdown()`.

```
#include "holoviz/holoviz.hpp"

namespace viz = holoscan::viz;
```

(continues on next page)

(continued from previous page)

```
viz::Init("Holoviz Example");  
  
while (!viz::WindowShouldClose()) {  
    viz::Begin();  
    viz::BeginImageLayer();  
    viz::ImageHost(width, height, viz::ImageFormat::R8G8B8A8_UNORM, image_data);  
    viz::EndLayer();  
    viz::End();  
}  
  
viz::Shutdown();
```

Result:

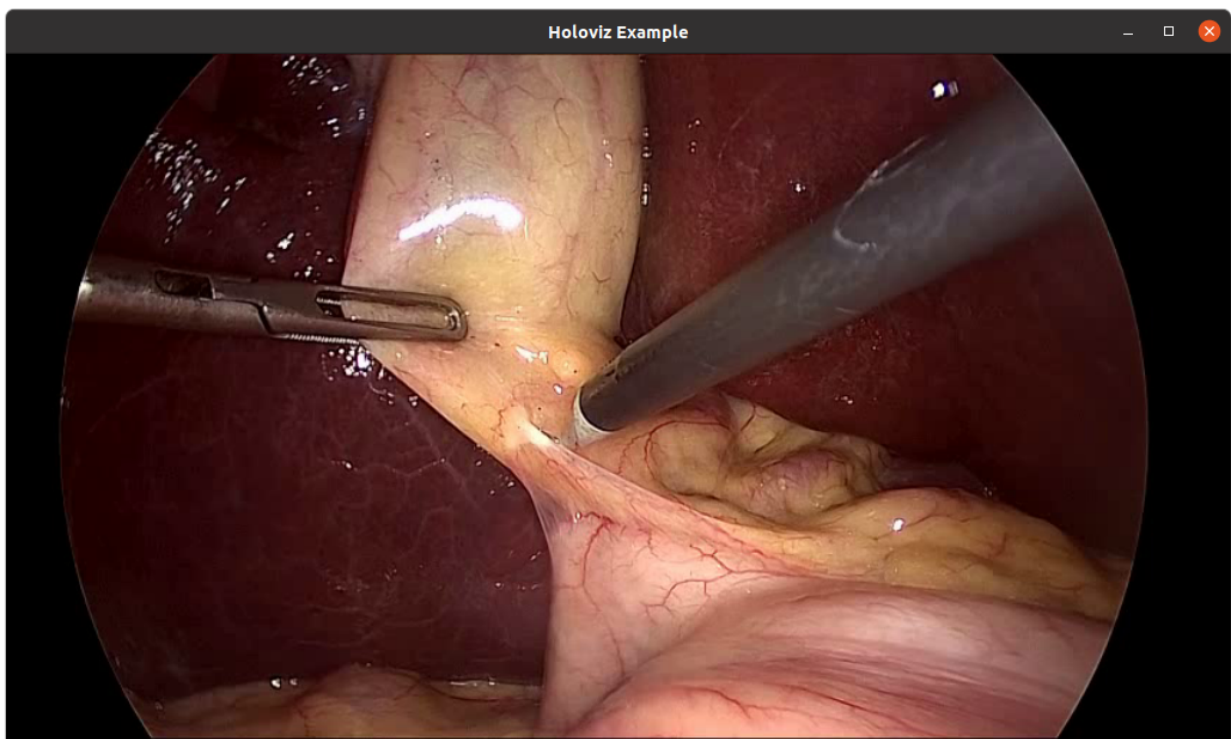


Fig. 18.1: Holoviz example app

18.12.4 API

`namespace_holoscan__viz`

18.12.5 Holoviz Module Examples

There are multiple [examples](#) showing how to use various features of the Holoviz module.

INFERENCE

19.1 Overview

A Holoscan application that needs to run inference will use an inference operator. The built-in *Inference operator* (InferenceOp) can be used, and several related use cases are documented in the Inference operator section below. The use cases are created using the *parameter set* that must be defined in the configuration file of the Holoscan application. If the built-in InferenceOp doesn't cover a specific use case, users can create their own custom inference operator as documented in the *Creating an Inference Operator* section.

The core inference functionality in the Holoscan SDK is provided by the Inference Module, which is a framework that facilitates designing and executing inference and processing applications through its APIs. It is used by the built-in InferenceOp which supports the same parameters as the Inference Module. All parameters required by the Holoscan Inference Module are passed through a parameter set in the configuration file of an application.

19.2 Parameters and Related Features

Required parameters and related features available with the Holoscan Inference Module are listed below.

- Data Buffer Parameters: Parameters are provided in the inference settings to enable data buffer locations at several stages of the inference. As shown in the figure below, three parameters `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` can be set by the user.
 - `input_on_cuda` refers to the location of the data going into the inference.
 - * If value is `true`, it means the input data is on the device.
 - * If value is `false`, it means the input data is on the host.
 - * Default value: `true`
 - `output_on_cuda` refers to the data location of the inferred data.
 - * If value is `true`, it means the inferred data is on the device.
 - * If value is `false`, it means the inferred data is on the host.
 - * Default value: `true`
 - `transmit_on_cuda` refers to the data transmission.
 - * If value is `true`, it means the data transmission from the inference extension will be on **Device**.
 - * If value is `false`, it means the data transmission from the inference extension will be on **Host**.
 - * Default value: `true`
- Inference Parameters

- backend parameter is set to either `trt` for TensorRT, `onnxrt` for ONNX runtime, or `torch` for libtorch. If there are multiple models in the inference application, all models will use the same backend. If it is desired to use different backends for different models, specify the `backend_map` parameter instead.

- * TensorRT:

- CUDA-based inference supported both on `x86_64` and `aarch64`.
- End-to-end CUDA-based data buffer parameters supported. `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` will all be true for end-to-end CUDA-based data movement.
- `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` can be either `true` or `false`.
- TensorRT backend expects input models to be in `tensorrt engine file` format or `onnx` format.
 - if models are in `tensorrt engine file` format, parameter `is_engine_path` must be set to `true`.
 - if models are in `onnx` format, it will be automatically converted into `tensorrt engine file` by the Holoscan inference module.

- * Torch:

- CUDA and CPU based inference supported both on `x86_64` and `aarch64`.
- End-to-end CUDA-based data buffer parameters supported. `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` will all be true for end-to-end CUDA-based data movement.
- `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` can be either `true` or `false`.
- Libtorch and TorchVision are included in the Holoscan NGC container, initially built as part of the [PyTorch NGC container](#). To use the Holoscan SDK torch backend outside of these containers, we recommend you download libtorch and torchvision binaries from [Holoscan's third-party repository](#).
- Torch backend expects input models to be in `torchscript` format.
 - It is recommended to use the same version of torch for `torchscript` model generation, as used in the HOLOSCAN SDK on the respective architectures.
 - Additionally, it is recommended to generate the `torchscript` model on the same architecture on which it will be executed. For example, `torchscript` model must be generated on `x86_64` to be executed in an application running on `x86_64` only.

- * ONNX runtime:

- Data flow via host only. `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` must be `false`.
- CUDA-based inference (supported on `x86_64`).
- CPU-based inference (supported on `x86_64` and `aarch64`).

- `infer_on_cpu` parameter is set to `true` if CPU based inference is desired.

The tables below demonstrate the supported features related to the data buffer and the inference with `trt` and `onnxrt` based backend, on `x86` and `aarch64` system respectively.

x86	input_on_cuda	output_on_cuda	transmit_on_cuda	infer_on_cpu
Supported values for <code>trt</code>	true or false	true or false	true or false	false
Supported values for <code>torch</code>	true or false	true or false	true or false	true or false
Supported values for <code>onnxrt</code>	false	false	true or false	true or false

Aarch64	input_on_cuda	output_on_cuda	transmit_on_cuda	infer_on_cpu
Supported values for trt	true or false	true or false	true or false	false
Supported values for torch	true or false	true or false	true or false	true or false
Supported values for onnxrt	false	false	true or false	true

- **model_path_map**: User can design single or multi AI inference pipeline by populating **model_path_map** in the config file.
 - * With a single entry, it is single inference; with more than one entry, multi AI inference is enabled.
 - * Each entry in **model_path_map** has a unique keyword as key (used as an identifier by the Holoscan Inference Module), and the path to the model as value.
 - * All model entries must have the models either in **onnx** or **tensorrt engine file** or **torchscript** format.
- **pre_processor_map**: input tensor to the respective model is specified in **pre_processor_map** in the config file.
 - * The Holoscan Inference Module supports same input for multiple models or unique input per model.
 - * Each entry in **pre_processor_map** has a unique keyword representing the model (same as used in **model_path_map**), and a vector of tensor names as the value.
 - * The Holoscan Inference Module supports multiple input tensors per model.
- **inference_map**: output tensors per model after inference is specified in **inference_map** in the config file.
 - * Each entry in **inference_map** has a unique keyword representing the model (same as used in **model_path_map** and **pre_processor_map**), and a vector of the output tensor names as the value.
 - * The Holoscan Inference Module supports multiple output tensors per model.
- **parallel_inference**: Parallel or Sequential execution of inferences.
 - * If multiple models are input, you can execute models in parallel.
 - * Parameter **parallel_inference** can be either **true** or **false**. Default value is **true**.
 - * Inferences are launched in parallel without any check of the available GPU resources. You must ensure that there is enough memory and compute available to run all the inferences in parallel.
- **enable_fp16**: Generation of the TensorRT engine files with FP16 option
 - * If backend is set to **trt**, and if the input models are in **onnx** format, then you can generate the engine file with fp16 option to accelerate inferencing.
 - * It takes few minutes to generate the engine files for the first time.
 - * It can be either **true** or **false**. Default value is **false**.
- **is_engine_path**: if the input models are specified in **trt engine format** in **model_path_map**, this flag must be set to **true**. Default value is **false**.
- **in_tensor_names**: Input tensor names to be used by **pre_processor_map**. This parameter is optional. If absent in the parameter map, values are derived from **pre_processor_map**.
- **out_tensor_names**: Output tensor names to be used by **inference_map**. This parameter is optional. If absent in the parameter map, values are derived from **inference_map**.
- **device_map**: Multi-GPU inferencing is enabled if **device_map** is populated in the parameter set.

- * Each entry in `device_map` has a unique keyword representing the model (same as used in `model_path_map` and `pre_processor_map`), and GPU identifier as the value. This GPU ID is used to execute the inference for the specified model.
- * GPUs specified in the `device_map` must have P2P (peer to peer) access and they must be connected to the same PCIE configuration. If P2P access is not possible among GPUs, the host (CPU memory) will be used to transfer the data.
- * Multi-GPU inferencing is supported for all backends.
- `temporal_map`: Temporal inferencing is enabled if `temporal_map` is populated in the parameter set.
 - * Each entry in `temporal_map` has a unique keyword representing the model (same as used in `model_path_map` and `pre_processor_map`), and frame delay as the value. Frame delay represents the frame count that are skipped by the operator in doing the inference for that particular model. A model with the value of 1, is inferred per frame. A model with a value of 10 is inferred for every 10th frame coming into the operator, which is the 1st frame, 11th frame, 21st frame and so on. Additionally, the operator will transmit the last inferred result for all the frames that are not inferred. For example, a model with a value of 10 will be inferred at 11th frame and from 12th to 20th frame, the result from 11th frame is transmitted.
 - * If the `temporal_map` is absent in the parameter set, all models are inferred for all the frames.
 - * All models are not mandatory in the `temporal_map`. The missing models are inferred per frame.
 - * Temporal map based inferencing is supported for all backends.
- `activation_map`: Dynamic inferencing can be enabled with this parameter. It is populated in the parameter set and is updated at runtime.
 - * Each entry in `activation_map` has a unique keyword representing the model (same as used in `model_path_map` and `pre_processor_map`), and activation state as the value. Activation state represents whether the model will be used for inferencing or not on a given frame. Any model(s) with a value of 1 will be active and will be used for inference, and any model(s) with a value of 0 will not run. The activation map must be initialized in the parameter set for all the models that need to be activated or deactivated dynamically.
 - * When the activation state is 0 for a particular model in the `activation_map`, the inference operator will not launch the inference for the model and will emit the last inferred result for the model.
 - * If the `activation_map` is absent in the parameter set, all of the models are inferred for all frames.
 - * All models are not mandatory in the `activation_map`. The missing models are active on every frame.
 - * Activation map based dynamic inferencing is supported for all backends.
- `backend_map`: Multiple backends can be used in the same application with this parameter.
 - * Each entry in `backend_map` has a unique keyword representing the model (same as used in `model_path_map`), and the backend as the value.
 - * A sample `backend_map` is shown below. In the example, `model_1` uses the `tensorRT` backend, and `model_2` and `model_3` uses the `torch` backend for inference.

backend_map:

```
"model_1_unique_identifier": "trt"
"model_2_unique_identifier": "torch"
"model_3_unique_identifier": "torch"
```

- `trt_opt_profile`: This parameter is optional and is activated with TensorRT backend. This parameter is applicable on models with dynamic input shapes.

- * Parameter is specified as a vector of 3 integers. First is the minimum batch size for the input, second is the optimum batch size and third value is the maximum batch size.
 - * Users can specify a batch profile for dynamic input. This profile is then used in engine creation. User must clear the cache to apply the updated optimization profile.
 - * Default value: {1,1,1}
- Other features: The table below illustrates other features and supported values in the current release.

Feature	Supported values
Data type	float32, int32, int8
Inference Back-end	trt, torch, onnxrt
Inputs per model	Multiple
Outputs per model	Multiple
GPU(s) supported	Multi-GPU on same PCIE network
Tensor data dimension	Max 8 supported for onnx and trt backend, 3 (CHW) or 4 (NCHW) for torch.
Model Type	All onnx or all torchscript or all trt engine type or a combination of torch and trt engine

- Multi Receiver and Single Transmitter support
 - The Holoscan Inference Module provides an API to extract the data from multiple receivers.
 - The Holoscan Inference Module provides an API to transmit multiple tensors via a single transmitter.

19.2.1 Parameter Specification

All required inference parameters of the inference application must be specified. Below is a sample parameter set for an application that uses three models for inferencing. You must populate all required fields with appropriate values.

```
inference:
  backend: "trt"
  model_path_map:
    "model_1_unique_identifier": "path_to_model_1"
    "model_2_unique_identifier": "path_to_model_2"
    "model_3_unique_identifier": "path_to_model_3"
  pre_processor_map:
    "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
    "model_2_unique_identifier": ["input_tensor_1_model_2_unique_identifier"]
    "model_3_unique_identifier": ["input_tensor_1_model_3_unique_identifier"]
  inference_map:
    "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
    "model_2_unique_identifier": ["output_tensor_1_model_2_unique_identifier"]
    "model_3_unique_identifier": ["output_tensor_1_model_3_unique_identifier"]
  parallel_inference: true
  infer_on_cpu: false
  enable_fp16: false
  input_on_cuda: true
  output_on_cuda: true
```

(continues on next page)

(continued from previous page)

```
transmit_on_cuda: true
is_engine_path: false
```

19.3 Inference Operator

In Holoscan SDK, the built-in Inference operator (`InferenceOp`) is designed using the Holoscan Inference Module APIs. The Inference operator ingests the inference parameter set (from the configuration file) and the data receivers (from previous connected operators in the application), executes the inference and transmits the inferred results to the next connected operators in the application.

`InferenceOp` is a generic operator that serves multiple use cases via the parameter set. Parameter sets for some key use cases are listed below:

Note: Some parameters have default values set for them in the `InferenceOp`. For any parameters not mentioned in the example parameter sets below, their default is used by the `InferenceOp`. These parameters are used to enable several use cases.

- Single model inference using TensorRT backend.

```
backend: "trt"
model_path_map:
  "model_1_unique_identifier": "path_to_model_1"
pre_processor_map:
  "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
inference_map:
  "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
```

The value of `backend` can be modified for other supported backends, and other parameters related to each backend. You must ensure the correct model type and model path are provided into the parameter set, along with supported values of all parameters for the respective backend.

In this example, `path_to_model_1` must be an `onnx` file, which will be converted to a `tensorRT` engine file at first execution. During subsequent executions, the Holoscan inference module will automatically find the `tensorRT` engine file (if `path_to_model_1` has not changed). Additionally, if you have a pre-built `tensorRT` engine file, `path_to_model_1` must be path to the engine file and the parameter `is_engine_path` must be set to `true` in the parameter set.

- Single model inference using TensorRT backend with multiple outputs.

```
backend: "trt"
model_path_map:
  "model_1_unique_identifier": "path_to_model_1"
pre_processor_map:
  "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
inference_map:
  "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier",
                                "output_tensor_2_model_1_unique_identifier",
                                "output_tensor_3_model_1_unique_identifier"]
```

As shown in example above, the Holoscan Inference module automatically maps the model outputs to the named tensors in the parameter set. You must be sure to use the named tensors in the same sequence in which the model generates the output. Similar logic holds for multiple inputs.

- Single model inference using fp16 precision.

```

backend: "trt"
model_path_map:
    "model_1_unique_identifier": "path_to_model_1"
pre_processor_map:
    "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
inference_map:
    "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier",
                                   "output_tensor_2_model_1_unique_identifier",
                                   "output_tensor_3_model_1_unique_identifier"]

enable_fp16: true

```

If a tensorRT engine file is not available for fp16 precision, it will be automatically generated by the Holoscan Inference module on the first execution. The file is cached for future executions.

- Single model inference on CPU.

```

backend: "onnxrt"
model_path_map:
    "model_1_unique_identifier": "path_to_model_1"
pre_processor_map:
    "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
inference_map:
    "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
infer_on_cpu: true

```

Note that the backend can only be onnxrt or torch for CPU-based inference.

- Single model inference with input/output data on Host.

```

backend: "trt"
model_path_map:
    "model_1_unique_identifier": "path_to_model_1"
pre_processor_map:
    "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
inference_map:
    "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
input_on_cuda: false
output_on_cuda: false

```

Data in the core inference engine is passed through the host and is received on the host. Inference can happen on the GPU. Parameters `input_on_cuda` and `output_on_cuda` define the location of the data before and after inference respectively.

- Single model inference with data transmission via Host.

```

backend: "trt"
model_path_map:
    "model_1_unique_identifier": "path_to_model_1"
pre_processor_map:
    "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
inference_map:
    "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
transmit_on_host: true

```

Data from inference operator to the next connected operator in the application is transmitted via the host.

- Multi model inference with a single backend.

```
backend: "trt"
model_path_map:
  "model_1_unique_identifier": "path_to_model_1"
  "model_2_unique_identifier": "path_to_model_2"
  "model_3_unique_identifier": "path_to_model_3"
pre_processor_map:
  "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
  "model_2_unique_identifier": ["input_tensor_1_model_2_unique_identifier"]
  "model_3_unique_identifier": ["input_tensor_1_model_3_unique_identifier"]
inference_map:
  "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
  "model_2_unique_identifier": ["output_tensor_1_model_2_unique_identifier"]
  "model_3_unique_identifier": ["output_tensor_1_model_3_unique_identifier"]
```

By default, multiple model inferences are launched in parallel. The backend specified via parameter `backend` is used for all models in the application.

- Multi model inference with sequential inference.

```
backend: "trt"
model_path_map:
  "model_1_unique_identifier": "path_to_model_1"
  "model_2_unique_identifier": "path_to_model_2"
  "model_3_unique_identifier": "path_to_model_3"
pre_processor_map:
  "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
  "model_2_unique_identifier": ["input_tensor_1_model_2_unique_identifier"]
  "model_3_unique_identifier": ["input_tensor_1_model_3_unique_identifier"]
inference_map:
  "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
  "model_2_unique_identifier": ["output_tensor_1_model_2_unique_identifier"]
  "model_3_unique_identifier": ["output_tensor_1_model_3_unique_identifier"]
parallel_inference: false
```

`parallel_inference` is set to `true` by default. To launch model inferences in sequence, `parallel_inference` must be set to `false`.

- Multi model inference with multiple backends.

```
backend_map:
  "model_1_unique_identifier": "trt"
  "model_2_unique_identifier": "torch"
  "model_3_unique_identifier": "torch"
model_path_map:
  "model_1_unique_identifier": "path_to_model_1"
  "model_2_unique_identifier": "path_to_model_2"
  "model_3_unique_identifier": "path_to_model_3"
pre_processor_map:
  "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
  "model_2_unique_identifier": ["input_tensor_1_model_2_unique_identifier"]
  "model_3_unique_identifier": ["input_tensor_1_model_3_unique_identifier"]
```

(continues on next page)

(continued from previous page)

```
inference_map:
  "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
  "model_2_unique_identifier": ["output_tensor_1_model_2_unique_identifier"]
  "model_3_unique_identifier": ["output_tensor_1_model_3_unique_identifier"]
```

In the above sample parameter set, the first model will do inference using the `tensorRT` backend, and model 2 and 3 will do inference using the `torch` backend.

Note: The combination of backends in `backend_map` must support all other parameters that will be used during the inference. For example, `onnxrt` and `tensorRT` combination with CPU-based inference is not supported.

- Multi model inference with a single backend on multi-GPU.

```
backend: "trt"
device_map:
  "model_1_unique_identifier": "1"
  "model_2_unique_identifier": "0"
  "model_3_unique_identifier": "1"
model_path_map:
  "model_1_unique_identifier": "path_to_model_1"
  "model_2_unique_identifier": "path_to_model_2"
  "model_3_unique_identifier": "path_to_model_3"
pre_processor_map:
  "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
  "model_2_unique_identifier": ["input_tensor_1_model_2_unique_identifier"]
  "model_3_unique_identifier": ["input_tensor_1_model_3_unique_identifier"]
inference_map:
  "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
  "model_2_unique_identifier": ["output_tensor_1_model_2_unique_identifier"]
  "model_3_unique_identifier": ["output_tensor_1_model_3_unique_identifier"]
```

In the sample above, model 1 and model 3 will do inference on the GPU with ID 1 and model 2 will do inference on the GPU with ID 0. GPUs must have P2P (peer to peer) access among them. If it is not enabled, the Holoscan inference module enables it by default. If P2P access is not possible between GPUs, then the data transfer will happen via the Host.

- Multi model inference with multiple backends on multiple GPUs.

```
backend_map:
  "model_1_unique_identifier": "trt"
  "model_2_unique_identifier": "torch"
  "model_3_unique_identifier": "torch"
device_map:
  "model_1_unique_identifier": "1"
  "model_2_unique_identifier": "0"
  "model_3_unique_identifier": "1"
model_path_map:
  "model_1_unique_identifier": "path_to_model_1"
  "model_2_unique_identifier": "path_to_model_2"
  "model_3_unique_identifier": "path_to_model_3"
pre_processor_map:
  "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
```

(continues on next page)

(continued from previous page)

```

    "model_2_unique_identifier": ["input_tensor_1_model_2_unique_identifier"]
    "model_3_unique_identifier": ["input_tensor_1_model_3_unique_identifier"]
  inference_map:
    "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
    "model_2_unique_identifier": ["output_tensor_1_model_2_unique_identifier"]
    "model_3_unique_identifier": ["output_tensor_1_model_3_unique_identifier"]

```

In the sample above, three models are used during the inference. Model 1 uses the trt backend and runs on the GPU with ID 1, model 2 uses the torch backend and runs on the GPU with ID 0, and model 3 uses the torch backend and runs on the GPU with ID 1.

19.4 Creating an Inference Operator

The Inference operator is the core inference unit in an inference application. The built-in Inference operator (InferenceOp) can be used for inference, or you can create your own custom inference operator as explained in this section. In Holoscan SDK, the inference operator can be designed using the Holoscan Inference Module APIs.

Arguments in the code sections below are referred to as

- Parameter Validity Check: Input inference parameters via the configuration (from step 1) are verified for correctness.

```
auto status = HoloInfer::inference_validity_check(...);
```

- Inference specification creation: For a single AI, only one entry is passed into the required entries in the parameter set. There is no change in the API calls below. Single AI or multi AI is enabled based on the number of entries in the parameter specifications from the configuration (in step 1).

```

// Declaration of inference specifications
std::shared_ptr<HoloInfer::InferenceSpecs> inference_specs_;

// Creation of inference specification structure
inference_specs_ = std::make_shared<HoloInfer::InferenceSpecs>(...);

```

- Inference context creation.

```

// Pointer to inference context.
std::unique_ptr<HoloInfer::InferContext> holoscan_infer_context_;
// Create holoscan inference context
holoscan_infer_context_ = std::make_unique<HoloInfer::InferContext>();

```

- Parameter setup with inference context: All required parameters of the Holoscan Inference Module are transferred in this step, and relevant memory allocations are initiated in the inference specification.

```

// Set and transfer inference specification to inference context
auto status = holoscan_infer_context_->set_inference_params(inference_specs_);

```

- Data extraction and allocation: The following API is used from the Holoinfer utility to extract and allocate data for the specified tensor.

```

// Extract relevant data from input, and update inference specifications
gxf_result_t stat = HoloInfer::get_data_per_model(...);

```

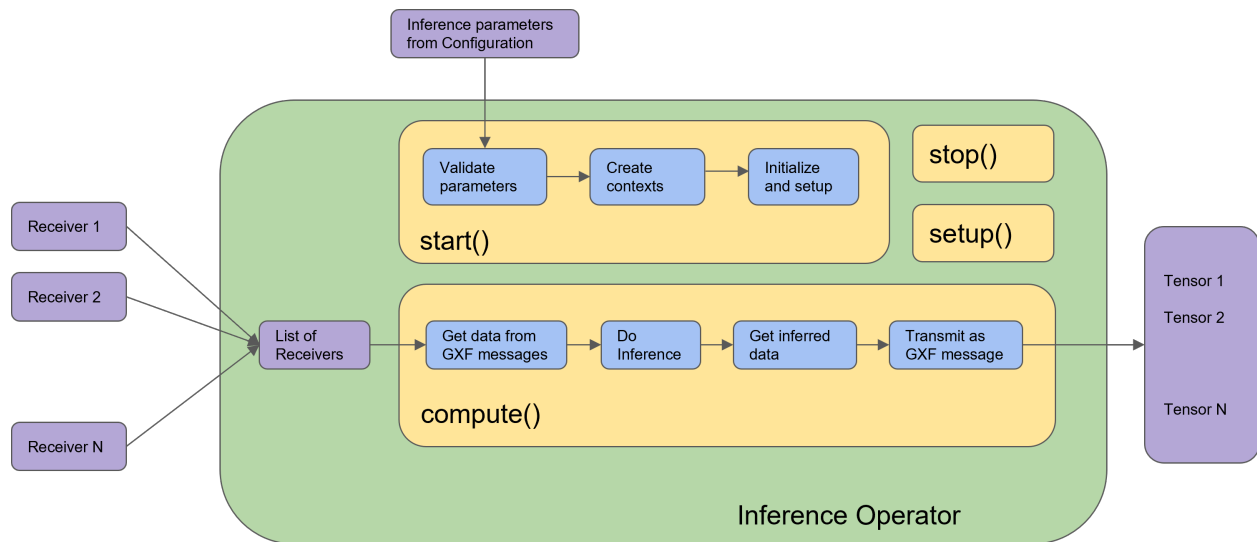
- Inference execution

```
// Execute inference and populate output buffer in inference specifications
auto status = holoscan_infer_context_->execute_inference(inference_specs_->data_per_
    ↪model_,
                                                         inference_specs_->output_
    ↪per_model_);
```

- Transmit inferred data:

```
// Transmit output buffers
auto status = HoloInfer::transmit_data_per_model(...);
```

The figure below demonstrates the Inference operator in the Holoscan SDK. All blocks with blue color are the API calls from the Holoscan Inference Module.



SCHEDULERS

The Scheduler component is a critical part of the system responsible for governing the execution of operators in a graph by enforcing conditions associated with each operator. Its primary responsibility includes orchestrating the execution of all operators defined in the graph while keeping track of their execution states.

The Holoscan SDK offers multiple schedulers that can cater to various use cases. These schedulers are:

1. *Greedy Scheduler*: This basic single-threaded scheduler tests conditions in a greedy manner. It is suitable for simple use cases and provides predictable execution. However, it may not be ideal for large-scale applications as it may incur significant overhead in condition execution.
2. *MultiThread Scheduler*: The multithread scheduler is designed to handle complex execution patterns in large-scale applications. This scheduler consists of a dispatcher thread that monitors the status of each operator and dispatches it to a thread pool of worker threads responsible for executing them. Once execution is complete, worker threads enqueue the operator back on the dispatch queue. The multithread scheduler offers superior performance and scalability over the greedy scheduler.
3. *Event-Based Scheduler*: The event-based scheduler is also a multi-thread scheduler, but as the name indicates it is event-based rather than polling based. Instead of having a thread that constantly polls for the execution readiness of each operator, it instead waits for an event to be received which indicates that an operator is ready to execute. The event-based scheduler will have a lower latency than using the multi-thread scheduler with a long polling interval (`check_recession_period_ms`), but without the high CPU usage seen for a multi-thread scheduler with a very short polling interval.

It is essential to select the appropriate scheduler for the use case at hand to ensure optimal performance and efficient resource utilization. Since most parameters of the schedulers overlap, it is easy to switch between them to test which may be most performant for a given application.

Note: Detailed APIs can be found here: [C++/Python](#)).

20.1 Greedy Scheduler

The greedy scheduler has a few parameters that the user can configure.

- The *clock* used by the scheduler can be set to either a *realtime* or *manual* clock.
 - The realtime clock is what should be used for applications as it pauses execution as needed to respect user-specified conditions (e.g., operators with periodic conditions will wait the requested period before executing again).
 - The manual clock is of benefit mainly for testing purposes as it causes operators to run in a time-compressed fashion (e.g., periodic conditions are not respected and operators run in immediate succession).

- The user can specify a `max_duration_ms` that will cause execution of the application to terminate after a specified maximum duration. The default value of `-1` (or any other negative value) will result in no maximum duration being applied.
- This scheduler also has a Boolean parameter, `stop_on_deadlock` that controls whether the application will terminate if a deadlock occurs. A deadlock occurs when all operators are in a `WAIT` state, but there is no periodic condition pending to break out of this state. This parameter is `true` by default.
- When setting the `stop_on_deadlock_timeout` parameter, the scheduler will wait this amount of time (in ms) before determining that it is in deadlock and should stop. It will reset if a job comes in during the wait. A negative value means no stop on deadlock. This parameter only applies when `stop_on_deadlock=true`.

20.2 Multithread Scheduler

The multithread scheduler has several parameters that the user can configure. These are a superset of the parameters available for the `GreedyScheduler` (described in the section above). Only the parameters unique to the multithread scheduler are described here. The multi-thread scheduler uses a dedicated thread to poll the status of operators and schedule any that are ready to execute. This will lead to high CPU usage by this polling thread when `check_recession_period_ms` is close to 0.

- The number of worker threads used by the scheduler can be set via `worker_thread_number`, which defaults to 1. This should be set based on a consideration of both the workflow and the available hardware. For example, the topology of the computation graph will determine how many operators it may be possible to run in parallel. Some operators may potentially launch multiple threads internally, so some amount of performance profiling may be required to determine optimal parameters for a given workflow.
- The value of `check_recession_period_ms` controls how long the scheduler will sleep before checking a given condition again. In other words, this is the polling interval for operators that are in a `WAIT` state. The default value for this parameter is 5 ms.

20.3 Event-Based Scheduler

The event-based scheduler is also a multi-thread scheduler, but it is event-based rather than polling based. As such, there is no `check_recession_period_ms` parameter, and this scheduler will not have the high CPU usage that can occur when polling at a short interval. Instead, the scheduler only wakes up when an event is received indicating that an operator is ready to execute. The parameters of this scheduler are a superset of the parameters available for the `GreedyScheduler` (described above). Only the parameters unique to the event-based scheduler are described here.

- The number of worker threads used by the scheduler can be set via `worker_thread_number`, which defaults to 1. This should be set based on a consideration of both the workflow and the available hardware. For example, the topology of the computation graph will determine how many operators it may be possible to run in parallel. Some operators may potentially launch multiple threads internally, so some amount of performance profiling may be required to determine optimal parameters for a given workflow.

CONDITIONS

The following table shows various states of the scheduling status of an operator:

Scheduling Status	Description
NEVER	Operator will never execute again
READY	Operator is ready for execution
WAIT	Operator may execute in the future
WAIT_TIME	Operator will be ready for execution after specified duration
WAIT_EVENT	Operator is waiting on an asynchronous event with unknown time interval

Note:

- A failure in execution of any single operator stops the execution of all the operators.
 - Operators are naturally unscheduled from execution when their scheduling status reaches NEVER state.
-

By default, operators are always READY, meaning they are scheduled to continuously execute their `compute()` method. To change that behavior, some condition classes can be assigned to an operator. There are various conditions currently supported in the Holoscan SDK:

- `MessageAvailableCondition`
- `ExpiringMessageAvailableCondition`
- `DownstreamMessageAffordableCondition`
- `CountCondition`
- `BooleanCondition`
- `PeriodicCondition`
- `AsynchronousCondition`

These conditions fall under various types as detailed below. Often, conditions are explicitly added to an operator by the application author, but it should also be noted that unless the default is overridden a `MessageAvailableCondition` is automatically added for each of an operator's input ports and a `DownstreamMessageAffordableCondition` is automatically added for each of its output ports.

Note: Detailed APIs can be found here: [C++/Python](#).

Conditions are AND-combined

An Operator can be associated with multiple conditions which define its execution behavior. Conditions are AND combined to describe the current state of an operator. For an operator to be executed by the scheduler, all the conditions must be in `READY` state and conversely, the operator is unscheduled from execution whenever any one of the scheduling terms reaches `NEVER` state. The priority of various states during AND combine follows the order `NEVER`, `WAIT_EVENT`, `WAIT`, `WAIT_TIME`, and `READY`.

21.1 Condition Types

The following table gives a rough categorization of the available condition types to help better understand their purpose and how they are assigned. More detailed descriptions of the individual conditions are given in the following sections.

Condition Name	Classification	Associated With
<code>MessageAvailableCondition</code>	message-driven	single input port
<code>ExpiringMessageAvailableCondition</code>	message-driven	single input port
<code>DownstreamMessageAffordableCondition</code>	message-driven	single output port
<code>PeriodicCondition</code>	clock-driven	operator as a whole
<code>CountCondition</code>	other	operator as a whole
<code>BooleanCondition</code>	execution-driven	operator as a whole
<code>AsynchronousCondition</code>	execution-driven	operator as a whole

Here, the various message-driven conditions are associated with an input port (receiver) or output port (transmitter). Message-driven conditions are typically assigned via the `IOSpec::condition` method (C++/Python) method as called from an operator's `setup` (C++/Python) method. All other condition types are typically passed as either a positional or keyword argument during operator construction in the application's `compose` method (i.e. passed to `make_operator()` in C++ or the operator class's constructor in Python). Once these conditions are assigned, they automatically enforce the associated criteria for that transmitter/receiver as part of the conditions controlling whether the operator will call `compute`. Due to the AND combination of conditions discussed above, all ports must meet their associated conditions in order for an operator to call `compute`.

The `PeriodicCondition` is clock-driven. It automatically takes effect based on timing from its associated clock. The `CountCondition` is another condition type that automatically takes effect, stopping execution of an operator after a specified count is reached.

The conditions that are marked as execution-driven, by contrast, require an application or operator thread to explicitly trigger a change in the condition. For example, the built-in `HolovizOp` operator's `compute` method implements logic to update an associated `BooleanCondition` to disable the operator when a user closes the display window. Similarly, the `AsynchronousCondition` requires some thread to emit events to trigger an update of its state.

21.2 MessageAvailableCondition

An operator associated with `MessageAvailableCondition` (C++/Python) is executed when the associated queue of the input port has at least a certain number of elements. This condition is associated with a specific input port of an operator through the `condition()` method on the return value (`IOSpec`) of the `OperatorSpec`'s `input()` method.

The minimum number of messages that permits the execution of the operator is specified by `min_size` parameter (default: 1). An optional parameter for this condition is `front_stage_max_size`, the maximum front stage message count. If this parameter is set, the condition will only allow execution if the number of messages in the queue does not exceed this count. It can be used for operators which do not consume all messages from the queue.

21.3 ExpiringMessageAvailableCondition

An operator associated with `ExpiringMessageAvailableCondition` (C++/Python) is executed when the first message received in the associated queue is expiring or when there are enough messages in the queue. This condition is associated with a specific input or output port of an operator through the `condition()` method on the return value (`IOSpec`) of the `OperatorSpec`'s `input()` or `output()` method.

The parameters `max_batch_size` and `max_delay_ns` dictate the maximum number of messages to be batched together and the maximum delay from first message to wait before executing the entity respectively. Please note that `ExpiringMessageAvailableCondition` requires that the input messages sent to any port using this condition must contain a timestamp. This means that the upstream operator has to emit using a timestamp .

21.4 DownstreamMessageAffordableCondition

The `DownstreamMessageAffordableCondition` (C++/Python) condition specifies that an operator shall be executed if the input port of the downstream operator for a given output port can accept new messages. This condition is associated with a specific output port of an operator through the `condition()` method on the return value (`IOSpec`) of the `OperatorSpec`'s `output()` method. The minimum number of messages that permits the execution of the operator is specified by `min_size` parameter (default: 1).

21.5 CountCondition

An operator associated with `CountCondition` (C++/Python) is executed for a specific number of times specified using its `count` parameter. The scheduling status of the operator associated with this condition can either be in `READY` or `NEVER` state. The scheduling status reaches the `NEVER` state when the operator has been executed `count` number of times. The `count` parameter can be set to a negative value to indicate that the operator should be executed an infinite number of times (default: 1).

21.6 BooleanCondition

An operator associated with `BooleanCondition` (C++/Python) is executed when the associated boolean variable is set to `true`. The boolean variable is set to `true/false` by calling the `enable_tick()/disable_tick()` methods on the `BooleanCondition` object. The `check_tick_enabled()` method can be used to check if the boolean variable is set to `true/false`. The scheduling status of the operator associated with this condition can either be in `READY` or `NEVER` state. If the boolean variable is set to `true`, the scheduling status of the operator associated with this condition is set to `READY`. If the boolean variable is set to `false`, the scheduling status of the operator associated with this condition is set to `NEVER`. The `enable_tick()/disable_tick()` methods can be called from any operator in the workflow.

C++

```
void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override {
    // ...
    if (<condition expression>) {           // e.g. if (index_ >= 10)
        auto my_bool_condition = condition<BooleanCondition>("my_bool_condition");
        if (my_bool_condition) {           // if condition exists (not true or false)
            my_bool_condition->disable_tick(); // this will stop the operator
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
    // ...
}

```

PYTHON

```

def compute(self, op_input, op_output, context):
    # ...
    if <condition expression>:           # e.g, self.index >= 10
        my_bool_condition = self.conditions.get("my_bool_condition")
        if my_bool_condition:           # if condition exists (not true or false)
            my_bool_condition.disable_tick() # this will stop the operator
    # ...

```

21.7 PeriodicCondition

An operator associated with `PeriodicCondition` (C++/Python) is executed after periodic time intervals specified using its `recess_period` parameter. The scheduling status of the operator associated with this condition can either be in `READY` or `WAIT_TIME` state. For the first time or after periodic time intervals, the scheduling status of the operator associated with this condition is set to `READY` and the operator is executed. After the operator is executed, the scheduling status is set to `WAIT_TIME`, and the operator is not executed until the `recess_period` time interval.

21.8 AsynchronousCondition

`AsynchronousCondition` (C++/Python) is primarily associated with operators which are working with asynchronous events happening outside of their regular execution performed by the scheduler. Since these events are non-periodic in nature, `AsynchronousCondition` prevents the scheduler from polling the operator for its status regularly and reduces CPU utilization. The scheduling status of the operator associated with this condition can either be in `READY`, `WAIT`, `WAIT_EVENT`, or `NEVER` states based on the asynchronous event it's waiting on.

The state of an asynchronous event is described using `AsynchronousEventState` and is updated using the `event_state()` API.

AsynchronousEventState	Description
READY	Init state, first execution of <code>compute()</code> method is pending
WAIT	Request to async service yet to be sent, nothing to do but wait
EVENT_WAITING	Request sent to an async service, pending event done notification
EVENT_DONE	Event done notification received, operator ready to be ticked
EVENT_NEVER	Operator does not want to be executed again, end of execution

Operators associated with this scheduling term most likely have an asynchronous thread which can update the state of the condition outside of its regular execution cycle performed by the scheduler. When the asynchronous event state is in `WAIT` state, the scheduler regularly polls for the scheduling state of the operator. When the asynchronous event state is in `EVENT_WAITING` state, schedulers will not check the scheduling status of the operator again until they receive an event notification. Setting the state of the asynchronous event to `EVENT_DONE` automatically sends the event notification to the scheduler. Operators can use the `EVENT_NEVER` state to indicate the end of its execution cycle. As for all of the condition types, the condition type can be used with any of the schedulers.

RESOURCES

Resource classes represent resources such as allocators, clocks, transmitters, or receivers that may be used as a parameter for operators or schedulers. The resource classes that are likely to be directly used by application authors are documented here.

Note: There are a number of other resources classes used internally which are not documented here, but appear in the API Documentation (C++/Python).

22.1 Allocator

22.1.1 UnboundedAllocator

An allocator that uses dynamic host or device memory allocation without an upper bound. This allocator does not take any user-specified parameters. This memory pool is easy to use and is recommended for initial prototyping. Once an application is working, switching to a `BlockMemoryPool` instead may help provide additional performance.

22.1.2 BlockMemoryPool

This is a memory pool which provides a user-specified number of equally sized blocks of memory. Using this memory pool provides a way to allocate memory blocks once and reuse the blocks on each subsequent call to an Operator's `compute` method. This saves overhead relative to allocating memory again each time `compute` is called. For the built-in operators which accept a memory pool parameter, there is a section in its API docstrings titled "Device Memory Requirements" which provides guidance on the `num_blocks` and `block_size` needed for use with this memory pool.

- The `storage_type` parameter can be set to determine the memory storage type used by the operator. This can be 0 for page-locked host memory (allocated with `cudaMallocHost`), 1 for device memory (allocated with `cudaMalloc`) or 2 for system memory (allocated with C++ `new`).
- The `block_size` parameter determines the size of a single block in the memory pool in bytes. Any allocation requests made of this allocator must fit into this block size.
- The `num_blocks` parameter controls the total number of blocks that are allocated in the memory pool.
- The `dev_id` parameter is an optional parameter that can be used to specify the CUDA ID of the device on which the memory pool will be created.

22.1.3 RMMAllocator

This allocator provides a pair of memory pools (one is a CUDA device memory pool and the other corresponds to pinned host memory). The underlying implementation is based on the [RAPIDS memory manager](#) (RMM) and uses a pair of `rmm::mr::pool_memory_resource` resource types (The device memory pool is a `rmm::mr::cuda_memory_resource` and the host pool is a `rmm::mr::pinned_memory_resource`). Unlike `BlockMemoryPool`, this allocator can be used with operators like `VideoStreamReplayerOp` that require an allocator capable of allocating both host and device memory. Rather than fixed block sizes, it uses just an initial memory size to allocate and a maximum size that the pool can expand to.

- The `device_memory_initial_size` parameter specifies the initial size of the device (GPU) memory pool. This is an optional parameter that defaults to 8 MB on aarch64 and 16 MB on x86_64. See note below on the format used to specify the value.
- The `device_memory_max_size` parameter specifies the maximum size of the device (GPU) memory pool in MiB. This is an optional parameter that defaults to twice the value of `device_memory_initial_size`. See note below on the format used to specify the value.
- The `host_memory_initial_size` parameter specifies the initial size of the device (GPU) memory pool in MiB. This is an optional parameter that defaults to 8 MB on aarch64 and 16 MB on x86_64. See note below on the format used to specify the value.
- The `host_memory_max_size` parameter specifies the maximum size of the device (GPU) memory pool in MiB. This is an optional parameter that defaults to twice the value of `host_memory_initial_size`. See note below on the format used to specify the value.
- The `dev_id` parameter is an optional parameter that can be used to specify the GPU device ID (as an integer) on which the memory pool will be created.

Note: The values for the memory parameters, such as `device_memory_initial_size` must be specified in the form of a string containing a non-negative integer value followed by a suffix representing the units. Supported units are B, KB, MB, GB and TB where the values are powers of 1024 bytes (e.g. MB = 1024 * 1024 bytes). Examples of valid units are “512MB”, “256 KB”, “1 GB”. If a floating point number is specified that decimal portion will be truncated (i.e. the value is rounded down to the nearest integer).

22.1.4 CudaStreamPool

This allocator creates a pool of CUDA streams.

- The `stream_flags` parameter specifies the flags sent to `cudaStreamCreateWithPriority` when creating the streams in the pool.
- The `stream_priority` parameter specifies the priority sent to `cudaStreamCreateWithPriority` when creating the streams in the pool. Lower values have a higher priority.
- The `reserved_size` parameter specifies the initial number of CUDA streams created in the pool upon initialization.
- The `max_size` parameter is an optional parameter that can be used to specify a maximum number of CUDA streams that can be present in the pool. The default value of 0 means that the size of the pool is unlimited.
- The `dev_id` parameter is an optional parameter that can be used to specify the CUDA ID of the device on which the stream pool will be created.

22.2 Clock

Clock classes can be provided via a `clock` parameter to the `Scheduler` classes to manage the flow of time.

All clock classes provide a common set of methods that can be used at runtime in user applications.

- The `time()` method returns the current time in seconds (floating point).
- The `timestamp()` method returns the current time as an integer number of nanoseconds.
- The `sleep_for()` method sleeps for a specified duration in ns. An overloaded version of this method allows specifying the duration using a `std::chrono::duration<Rep, Period>` from the C++ API or a `datetime.timedelta` from the Python API.
- The `sleep_until()` method sleeps until a specified target time in ns.

22.2.1 Realtime Clock

The `RealtimeClock` respects the true duration of conditions such as `PeriodicCondition`. It is the default clock type and the one that would likely be used in user applications.

In addition to the general clock methods documented above:

- This class has a `set_time_scale()` method which can be used to dynamically change the time scale used by the clock.
- The parameter `initial_time_offset` can be used to set an initial offset in the time at initialization.
- The parameter `initial_time_scale` can be used to modify the scale of time. For instance, a scale of 2.0 would cause time to run twice as fast.
- The parameter `use_time_since_epoch` makes times relative to the [POSIX epoch](#) (`initial_time_offset` becomes an offset from epoch).

22.2.2 Manual Clock

The `ManualClock` compresses time intervals (e.g., `PeriodicCondition` proceeds immediately rather than waiting for the specified period). It is provided mainly for use during testing/development.

The parameter `initial_timestamp` controls the initial timestamp on the clock in ns.

22.3 Transmitter (advanced)

Typically users don't need to explicitly assign transmitter or receiver classes to the `IOSpec` ports of Holoscan SDK operators. For connections between operators a `DoubleBufferTransmitter` will automatically be used, while for connections between fragments in a distributed application, a `UcxTransmitter` will be used. When data frame flow tracking is enabled any `DoubleBufferTransmitter` will be replaced by an `AnnotatedDoubleBufferTransmitter` which also records the timestamps needed for that feature.

22.3.1 DoubleBufferTransmitter

This is the transmitter class used by output ports of operators within a fragment.

22.3.2 UcxTransmitter

This is the transmitter class used by output ports of operators that connect fragments in a distributed applications. It takes care of sending UCX active messages and serializing their contents.

22.4 Receiver (advanced)

Typically users don't need to explicitly assign transmitter or receiver classes to the IOSpec ports of Holoscan SDK operators. For connections between operators, a `DoubleBufferReceiver` will be used, while for connections between fragments in a distributed application, the `UcxReceiver` will be used. When data frame flow tracking is enabled, any `DoubleBufferReceiver` will be replaced by an `AnnotatedDoubleBufferReceiver` which also records the timestamps needed for that feature.

22.4.1 DoubleBufferReceiver

This is the receiver class used by input ports of operators within a fragment.

22.4.2 UcxReceiver

This is the receiver class used by input ports of operators that connect fragments in a distributed applications. It takes care of receiving UCX active messages and deserializing their contents.

23.1 Data Exporter API

The new Data Exporter C++ API (`DataExporter` and `CsvDataExporter`) is now available. This API can be used to export output from Holoscan applications to comma separated value (CSV) files for Holoscan Federated Analytics applications. `DataExporter` is a base class to support exporting Holoscan application output in different formats. `CsvDataExporter` is a class derived from `DataExporter` to support exporting Holoscan application output to CSV files.

The data root directory can be specified using the environment variable `HOLOSCAN_ANALYTICS_DATA_DIRECTORY`. If not specified, it defaults to the current directory. The data file name can be specified using the environment variable `HOLOSCAN_ANALYTICS_DATA_FILE_NAME`. If not specified, it defaults to the name `data.csv`. All the generated data will be stored inside a directory with the same name as the application name that is passed to the `DataExporter` constructor. On each run, a new directory inside the `<root_dir>\<app_dir>\` will be created and a new data file will be created inside it. Each new data directory will be named with the current timestamp. This timestamp convention prevents a given run of the application from overwriting any data stored previously by an earlier run of that same application.

23.1.1 Sample usage of the API

```
// Include Header
#include <holoscan/core/analytics/csv_data_exporter.hpp>

// Define CsvDataExporter member variable
CsvDataExporter exporter

// Initialize CsvDataExporter
exporter("app_name", std::vector<std::string>({"column1", "column2", "column3"}))

// Export data (typically called within an Operator::compute method)
exporter.export_data(std::vector<std::string>({"value1", "value2", "value3"}))
```

23.2 Using Data Exporter API with DataProcessor

The Holoscan applications like Endoscopy Out of Body Detection uses Inference Processor operator (`InferenceProcessorOp`) to output the binary classification results. The `DataProcessor` class used by the inference processor operator (`InferenceProcessorOp`) is now updated to support writing output to CSV files which can then be used as input to analytics applications. Also any other application using `InferenceProcessorOp` can now export the binary classification output to the CSV files.

Below is an example application config using the new export operation:

```
inference_processor_op:
  process_operations:
    "out_of_body_inferred": ["export_results_to_csv,
                             out_of_body_detection,
                             In-body,
                             Out-of-body,
                             Confidence Score"]
  in_tensor_names: ["out_of_body_inferred"]
```

This will create a folder named `out_of_body_detection` in the specified root directory, creates another folder inside it with current timestamp on each run, and creates a `.csv` file with specified name and three columns - In-body, Out-of-body, and Confidence Score. The lines in the `data.csv` file will look like:

```
In-body,Out-of-body,Confidence Score
1,0,0.972435
1,0,0.90207
1,0,0.897973
0,1,0.939281
0,1,0.948691
0,1,0.94994
```


HOLOSCAN APPLICATION PACKAGE SPECIFICATION (HAP)

24.1 Introduction

The Holoscan Application Package specification extends the MONAI Deploy Application Package specification to provide the streaming capabilities, multi-fragment, and other features of the Holoscan SDK.

24.2 Overview

This document includes the specification of the Holoscan Application Package (HAP). A HAP is a containerized application or service which is self-descriptive, as defined by this document.

24.2.1 Goal

This document aims to define the structure and purpose of a HAP, including which parts are optional and which are required so that developers can easily create conformant HAPs.

24.2.2 Assumptions, Constraints, Dependencies

The following assumptions relate to HAP execution, inspection and general usage:

- Containerized applications will be based on Linux x64 (AMD64) and/or ARM64 (aarch64).
- Containerized applications' host environment will be based on Linux x64 (AMD64) and/or ARM64 (aarch64) with container support.
- Developers expect the local execution of their applications to behave identically to the execution of the containerized version.
- Developers expect the local execution of their containerized applications to behave identically to the execution in deployment.
- Developers and operations engineers want the application packages to be self-describing.
- Applications may be created using tool other than that provided in the Holoscan SDK or the MONAI Deploy App SDK.
- Holoscan Application Package may be created using a tool other than that provided in the Holoscan SDK or the MONAI Deploy App SDK.
- Pre-existing, containerized applications must be “converted” into Holoscan Application Packages.

- A Holoscan Application Package may contain a classical application (non-fragment based), a single-fragment application, or a multi-fragment application. (Please see the definition of fragment in [Definitions, Acronyms, Abbreviations](#))
- The scalability of a multi-fragment application based on Holoscan SDK is outside the scope of this document.
- Application packages are expected to be deployed in one of the supported environments. For additional information, see [Holoscan Operating Environments](#).

24.3 Definitions, Acronyms, Abbreviations

Term	Definition
ARM64	Or, AARCH64. See Wikipedia for details.
Container	See What's a container?
Fragment	A fragment is a building block of the Application. It is a directed graph of operators. For details, please refer to the MONAI Deploy App SDK or Holoscan App SDK.
Gigabytes (GiB)	A gibibyte (GiB) is a unit of measurement used in computer data storage that equals to 1,073,741,824 bytes.
HAP	Holoscan Application Package. A containerized application or service which is self-descriptive.
Hosting Service	A service that hosts and orchestrates HAP containers.
MAP	MONAI Application Package. A containerized application or service which is self-descriptive.
Mebibytes (MiB)	A mebibyte (MiB) is a unit of measurement used in computer data storage that equals to 1,048,576 bytes.
MONAI	Medical Open Network for Artificial Intelligence.
SDK	Software Development Kit.
Semantic Version	See Semantic Versioning 2.0 .
x64	Or, x86-64 or AMD64. See Wikipedia for details.

24.4 Requirements

The following requirements **MUST** be met by the HAP specification to be considered complete and approved. All requirements marked as **MUST** or **SHALL** **MUST** be implemented in order to be supported by a HAP-ready hosting service.

24.4.1 Single Artifact

- A HAP **SHALL** comprise a single container, meeting the minimum requirements set forth by this document.
- A HAP **SHALL** be a containerized application to maximize the portability of its application.

24.4.2 Self-Describing

- A HAP MUST be self-describing and provide a mechanism for extracting its description.
 - A HAP SHALL provide a method to print the metadata files to the console.
 - A HAP SHALL provide a method to copy the metadata files to a user-specified directory.
- The method of description SHALL be in a machine-readable and writable format.
- The method of description SHALL be in a human-readable format.
- The method of description SHOULD be a human writable format.
- The method of description SHALL be declarative and immutable.
- The method of description SHALL provide the following information about the HAP:
 - Execution requirements such as dependencies and restrictions.
 - Resource requirements include CPU cores, system memory, shared memory, GPU, and GPU memory.

24.4.3 Runtime Characteristics of the HAP

- A HAP SHALL start the packaged Application when it is executed by the users when arguments are specified.
- A HAP SHALL describe the packaged Application as a long-running service or an application so an external agent can manage its lifecycle.

24.4.4 IO Specification

- A HAP SHALL provide information about its expected inputs such that an external agent can determine if the HAP can receive a workload.
- A HAP SHALL provide sufficient information about its outputs so that an external agent knows how to handle the results.

24.4.5 Local Execution

A HAP MUST be in a format that supports local execution in a development environment.

[Note] See *Holoscan Operating Environments* for additional information about supported environments.

24.4.6 Compatible with Kubernetes

- A HAP SHALL support deployment using Kubernetes.

24.4.7 OCI Compliance

The containerized portion of a HAP SHALL comply with [Open Container Initiative](#) format standards.

Image Annotations

All annotations for the containerized portion of a HAP MUST adhere to the specifications laid out by [The OpenContainers Annotations Spec](#)

- `org.opencontainers.image.title`: A HAP container image SHALL provide a human-readable title (string).
- `org.opencontainers.image.version`: A HAP container image SHALL provide a version of the packaged application using the semantic versioning format. This value is the same as the value defined in `/etc/holoscan/app.json#version` in the *Table of Application Manifest Fields*.
- All other OpenContainers predefined keys SHOULD be provided when available.

24.4.8 Hosting Environment

The HAP Hosting Environment executes the HAP and provides the application with a customized set of environment variables and command line options as part of the invocation.

- The Hosting Service MUST, by default, execute the application as defined by `/etc/holoscan/app.json#command` and then exit when the application or the service completes.
- The Hosting Service MUST provide any environment variables specified by `/etc/holoscan/app.json#environment`.
- The Hosting Service SHOULD monitor the Application process and record its CPU, system memory, and GPU utilization metrics.
- The Hosting Service SHOULD monitor the Application process and enforce any timeout value specified in `/etc/holoscan/app.json#timeout`.

Table of Environment Variables

A HAP SHALL contain the following environment variables and their default values, if not specified by the user, in the Application Manifest `/etc/holoscan/app.json#environment`.

Variable	Default	Format	Description
HOLOSCAN_INPUT_PATH	<code>/var/holoscan/input/</code>	Folder Path	Path to the input folder for the Application.
HOLOSCAN_OUTPUT_PATH	<code>/var/holoscan/output/</code>	Folder Path	Path to the output folder for the Application.
HOLOSCAN_WORKDIR	<code>/var/holoscan/</code>	Folder Path	Path to the Application's working directory.
HOLOSCAN_MODEL_PATH	<code>/opt/holoscan/models/</code>	Folder Path	Path to the Application's models directory.
HOLOSCAN_CONFIG_PATH	<code>/var/holoscan/app.yaml</code>	File Path	Path to the Application's configuration file.
HOLOSCAN_APP_MANIFEST_PATH	<code>/var/holoscan/app.config</code>	File Path	Path to the Application's configuration file.
HOLOSCAN_PKG_MANIFEST_PATH	<code>/var/holoscan/pkg.config</code>	File Path	Path to the Application's configuration file.
HOLOSCAN_DOCS	<code>/opt/holoscan/docs</code>	Folder Path	Path to the folder containing application documentation and licenses.
HOLOSCAN_LOGS	<code>/var/holoscan/logs</code>	Folder Path	Path to the Application's logs.

24.5 Architecture & Design

24.5.1 Description

The Holoscan Application Package (HAP) is a functional package designed to act on datasets of a prescribed format. A HAP is a container image that adheres to the specification provided in this document.

24.5.2 Application

The primary component of a HAP is the application. The application is provided by an application developer and incorporated into the HAP using the Holoscan Application Packager.

All application code and binaries SHALL be in the `/opt/holoscan/app/` folder, except for any dependencies installed by the Holoscan Application Packager during the creation of the HAP.

All AI models (PyTorch, TensorFlow, TensorRT, etc.) SHOULD be in separate sub-folders of the `/opt/holoscan/models/` folder. In specific use cases where the app package developer is prevented from enclosing the model files in the package/container due to intellectual property concerns, the models can be supplied from the host system when the app package is run, e.g., via the volume mount mappings and the use of container env variables.

24.5.3 Manifests

A HAP SHALL contain two manifests: the Application Manifest and the Package Manifest. The Package Manifest shall be stored in `/etc/holoscan/pkg.json`, and the Application Manifest shall be stored in `/etc/holoscan/app.json`. Once a HAP is created, its manifests are expected to be immutable.

Application Manifest

Table of Application Manifest Fields

Name	Required	Default	Type	Format
<code>apiVersion</code>	No	0.0.0	string	semantic version
<code>command</code>	Yes	N/A	string	shell command
<code>environment</code>	No	N/A	object	object w/ name-value pairs
<code>input</code>	Yes	N/A	object	object
<code>input.formats</code>	Yes	N/A	array	array of objects
<code>input.path</code>	No	<code>input/</code>	string	relative file-system path
<code>readiness</code>	No	N/A	object	object
<code>readiness.type</code>	Yes	N/A	string	string
<code>readiness.command</code>	Yes (when type is <code>command</code>)	N/A	array	shell command
<code>readiness.port</code>	Yes (when type is <code>tcp</code> , <code>grpc</code> , or <code>http-get</code>)	N/A	integer	number
<code>readiness.path</code>	Yes (when type is <code>http-get</code>)	N/A	string	string
<code>readiness.initialDelaySeconds</code>	No	1	integer	number
<code>readiness.periodSeconds</code>	No	10	integer	number
<code>readiness.timeoutSeconds</code>	No	1	integer	number
<code>readiness.failureThreshold</code>	No	3	integer	number
<code>liveness</code>	No	N/A	object	object
<code>liveness.type</code>	Yes	N/A	string	string
<code>liveness.command</code>	Yes (when type is <code>command</code>)	N/A	array	shell command
<code>liveness.port</code>	Yes (when type is <code>tcp</code> , <code>grpc</code> , or <code>http-get</code>)	N/A	integer	number
<code>liveness.path</code>	Yes (when type is <code>http-get</code>)	N/A	string	string
<code>liveness.initialDelaySeconds</code>	No	1	integer	number
<code>liveness.periodSeconds</code>	No	10	integer	number
<code>liveness.timeoutSeconds</code>	No	1	integer	number
<code>liveness.failureThreshold</code>	No	3	integer	number
<code>output</code>	Yes	N/A	object	object
<code>output.format</code>	Yes	N/A	object	object
<code>output.path</code>	No	<code>output/</code>	string	relative file-system path
<code>sdk</code>	No	N/A	string	string
<code>sdkVersion</code>	No	0.0.0	string	semantic version
<code>timeout</code>	No	0	integer	number
<code>version</code>	No	0.0.0	string	semantic version
<code>workingDirectory</code>	No	<code>/var/holoscan/</code>	string	absolute file-system path

The Application Manifest file provides information about the HAP's Application.

- The Application Manifest MUST define the type of the containerized application (`/etc/holoscan/app.json#type`).
 - Type SHALL have the value of either `service` or `application`.

- The Application Manifest MUST define the command used to run the Application (`/etc/holoscan/app.json#command`).
- The Application Manifest SHOULD define the version of the manifest file schema (`/etc/holoscan/app.json#apiVersion`).
 - The Manifest schema version SHALL be provided as a [semantic version](#) string.
 - When not provided, the default value `0.0.0` SHALL be assumed.
- The Application Manifest SHOULD define the SDK used to create the Application (`/etc/holoscan/app.json#sdk`).
- The Application Manifest SHOULD define the version of the SDK used to create the Application (`/etc/holoscan/app.json#sdkVersion`).
 - SDK version SHALL be provided as a [semantic version](#) string.
 - When not provided, the default value `0.0.0` SHALL be assumed.
- The Application Manifest SHOULD define the version of the application itself (`/etc/holoscan/app.json#version`).
 - The Application version SHALL be provided as a [semantic version](#) string.
 - When not provided, the default value `0.0.0` SHALL be assumed.
- The Application Manifest SHOULD define the application's working directory (`/etc/holoscan/app.json#workingDirectory`).
 - The Application will execute with its current directory set to this value.
 - The value provided must be an absolute path (the first character is `/`).
 - The default path `/var/holoscan/` SHALL be assumed when not provided.
- The Application Manifest SHOULD define the data input path, relative to the working directory, used by the Application (`/etc/holoscan/app.json#input.path`).
 - The input path SHOULD be a relative to the working directory or an absolute file-system path to a directory.
 - * When the value is a relative file-system path (the first character is not `/`), it is relative to the application's working directory.
 - * When the value is an absolute file-system path (the first character is `/`), the file-system path is used as-is.
 - When not provided, the default value `input/` SHALL be assumed.
- The Application Manifest SHOULD define input data formats supported by the Application (`/etc/holoscan/app.json#input.formats`).
 - Possible values include, but are not limited to, `none`, `network`, `file`.
- The Application Manifest SHOULD define the output path relative to the working directory used by the Application (`/etc/holoscan/app.json#output.path`).
 - The output path SHOULD be relative to the working directory or an absolute file-system path to a directory.
 - * When the value is a relative file-system path (the first character is not `/`), it is relative to the application's working directory.
 - * When the value is an absolute file-system path (the first character is `/`), the file-system path is used as-is.
 - When not provided, the default value `output/` SHALL be assumed.

- The Application Manifest SHOULD define the output data format produced by the Application (`/etc/holoscan/app.json#output.format`).
 - Possible values include, but are not limited to, `none`, `screen`, `file`, `network`.
- The Application Manifest SHOULD configure a check to determine whether or not the application is “ready.”
 - The Application Manifest SHALL define the probe type to be performed (`/etc/holoscan/app.json#readiness.type`).
 - * Possible values include `tcp`, `grpc`, `http-get`, and `command`.
 - The Application Manifest SHALL define the probe commands to execute when the type is `command` (`/etc/holoscan/app.json#readiness.command`).
 - * The data structure is expected to be an array of strings.
 - The Application Manifest SHALL define the port to perform the readiness probe when the type is `grpc`, `tcp`, or `http-get`. (`/etc/holoscan/app.json#readiness.port`)
 - * The value provided must be a valid port number ranging from 1 through 65535. (Please note that port numbers below 1024 are root-only privileged ports.)
 - The Application Manifest SHALL define the path to perform the readiness probe when the type is `http-get` (`/etc/holoscan/app.json#readiness.path`).
 - * The value provided must be an absolute path (the first character is `/`).
 - The Application Manifest SHALL define the number of seconds after the container has started before the readiness probe is initiated. (`/etc/holoscan/app.json#readiness.initialDelaySeconds`).
 - * The default value `0` SHALL be assumed when not provided.
 - The Application Manifest SHALL define how often to perform the readiness probe (`/etc/holoscan/app.json#readiness.periodSeconds`).
 - * When not provided, the default value `10` SHALL be assumed.
 - The Application Manifest SHALL define the number of seconds after which the probe times out (`/etc/holoscan/app.json#readiness.timeoutSeconds`).
 - * When not provided, the default value `1` SHALL be assumed.
 - The Application Manifest SHALL define the number of times to perform the probe before considering the service is not ready (`/etc/holoscan/app.json#readiness.failureThreshold`).
 - * The default value `3` SHALL be assumed when not provided.
- The Application Manifest SHOULD configure a check to determine whether or not the application is “live” or not.
 - The Application Manifest SHALL define the type of probe to be performed (`/etc/holoscan/app.json#liveness.type`).
 - * Possible values include `tcp`, `grpc`, `http-get`, and `command`.
 - The Application Manifest SHALL define the probe commands to execute when the type is `command` (`/etc/holoscan/app.json#liveness.command`).
 - * The data structure is expected to be an array of strings.
 - The Application Manifest SHALL define the port to perform the liveness probe when the type is `grpc`, `tcp`, or `http-get`. (`/etc/holoscan/app.json#liveness.port`)
 - * The value provided must be a valid port number ranging from 1 through 65535. (Please note that port numbers below 1024 are root-only privileged ports.)

- The Application Manifest SHALL define the path to perform the liveness probe when the type is `http-get` (`/etc/holoscan/app.json#liveness.path`).
 - * The value provided must be an absolute path (the first character is `/`).
- The Application Manifest SHALL define the number of seconds after the container has started before the liveness probe is initiated. (`/etc/holoscan/app.json#liveness.initialDelaySeconds`).
 - * The default value `0` SHALL be assumed when not provided.
- The Application Manifest SHALL define how often to perform the liveness probe (`/etc/holoscan/app.json#liveness.periodSeconds`).
 - * When not provided, the default value `10` SHALL be assumed.
- The Application Manifest SHALL define the number of seconds after which the probe times out (`/etc/holoscan/app.json#liveness.timeoutSeconds`).
 - * The default value `1` SHALL be assumed when not provided.
- The Application Manifest SHALL define the number of times to perform the probe before considering the service is not alive (`/etc/holoscan/app.json#liveness.failureThreshold`).
 - * When not provided, the default value `3` SHALL be assumed.
- The Application Manifest SHOULD define any timeout applied to the Application (`/etc/holoscan/app.json#timeout`).
 - When the value is `0`, timeout SHALL be disabled.
 - When not provided, the default value `0` SHALL be assumed.
- The Application Manifest MUST enable the specification of environment variables for the Application (`/etc/holoscan/app.json#environment`).
 - The data structure is expected to be in `"name": "value"` members of the object.
 - The field's name will be the name of the environment variable verbatim and must conform to all requirements for environment variables and JSON field names.
 - The field's value will be the value of the environment variable and must conform to all requirements for environment variables.

Package Manifest

Table of Package Manifest Fields

Name	Required	Default	Type	Format	Description
apiVersion	No	0.0.0	string	semantic version	Version of the manifest file schema.
applicationRoot	Yes	/opt/holoscan/app/	string	absolute file-system path	Absolute file-system path to the folder which contains the Application
modelRoot	No	/opt/holoscan/models/	string	absolute file-system path	Absolute file-system path to the folder which contains the model(s).
models	No	N/A	array	array of objects	Array of objects which describe models in the package.
models[*].name	Yes	N/A	string	string	Name of the model.
models[*].path	No	N/A	string	Relative file-system path	File-system path to the folder which contains the model that is relative to the value defined in modelRoot.
resources	No	N/A	object	object	Object describing resource requirements for the Application.
resources.cpu	No	1	decimal (2)	number	Number of CPU cores required by the Application or the Fragment.
resources.cpuLimit	No	N/A	decimal (2)	number	The CPU core limit for the Application or the Fragment. (1)
resources.gpu	No	0	decimal (2)	number	Number of GPU devices required by the Application or the Fragment.
resources.gpuLimit	No	N/A	decimal (2)	number	The GPU device limit for the Application or the Fragment. (1)
resources.memory	No	1Gi	string	memory size	The memory required by the Application or the Fragment.
resources.memoryLimit	No	N/A	string	memory size	The memory limit for the Application or the Fragment. (1)
resources.gpuMemory	No	N/A	string	memory size	The GPU memory required by the Application or the Fragment.
resources.gpuMemoryLimit	No	N/A	string	memory size	The GPU memory limit for the Application or the Fragment. (1)
resources.sharedMemory	No	64Mi	string	memory size	The shared memory required by the Application or the Fragment.
resources.fragments	No	N/A	object	objects	Nested objects which describe resources for a Multi-Fragment Application.
resources.fragments.<fragment-name>	Yes	N/A	string	string	Name of the fragment.
resources.fragments.<fragment-name>.cpu	No	1	decimal (2)	number	Number of CPU cores required by the Fragment.
resources.fragments.<fragment-name>.	No	N/A	decimal (2)	number	The CPU core limit for the Fragment. (1)
resources.fragments.<fragment-name>.gpu	No	0	decimal (2)	number	Number of GPU devices required by the Fragment.
resources.fragments.<fragment-name>.gpuLimit	No	N/A	decimal (2)	number	The GPU device limit for the Fragment. (1)

[Notes] (1) Use of resource limits depend on the orchestration service or the hosting environment's configuration and implementation. (2) Consider rounding up to a whole number as decimal values may not be supported by all orchestration/hosting services.

The Package Manifest file provides information about the HAP's package layout. It is not intended as a mechanism for controlling how the HAP is used or how the HAP's Application is executed.

- The Package Manifest **MUST** be UTF-8 encoded and use the JavaScript Object Notation (JSON) format.
- The Package Manifest **SHOULD** support either CRLF or LF style line endings.
- The Package Manifest **SHOULD** specify the folder which contains the application (`/etc/holoscan/pkg.json#applicationRoot`).
 - When not provided, the default path `/opt/holoscan/app/` will be assumed.
- The Package Manifest **SHOULD** provide the version of the package file manifest schema (`/etc/holoscan/pkg.json#apiVersion`).
 - The Manifest schema version **SHALL** be provided as a [semantic version](#) string.
- The Package Manifest **SHOULD** provide the package version of itself (`/etc/holoscan/pkg.json#version`).
 - The Package version **SHALL** be provided as a [semantic version](#) string.
- The Package Manifest **SHOULD** provide the directory path to the user-provided models. (`/etc/holoscan/pkg.json#modelRoot`).
 - The value provided must be an absolute path (the first character is `/`).
 - When not provided, the default path `/opt/holoscan/models/` **SHALL** be assumed.
- The Package Manifest **SHOULD** list the models used by the application (`/etc/holoscan/pkg.json#models`).
 - Models **SHALL** be defined by name (`/etc/holoscan/pkg.json#models[*].name`).
 - * Model names **SHALL NOT** contain any Unicode whitespace or control characters.
 - * Model names **SHALL NOT** exceed 128 bytes in length.
 - Models **SHOULD** provide a file-system path if they're included in the HAP itself (`/etc/holoscan/pkg.json#models[*].path`).
 - * When the value is a relative file-system path (the first character is not `/`), it is relative to the model root directory defined in `/etc/holoscan/pkg.json#modelRoot`.
 - * When the value is an absolute file-system path (the first character is `/`), the file-system path is used as-is.
 - * When no value is provided, the name is assumed as the name of the directory relative to the model root directory defined in `/etc/holoscan/pkg.json#modelRoot`.
- The Package Manifest **SHOULD** specify the resources required to execute the Application and the fragments for a Multi-Fragment Application.

This information is used to provision resources when running the containerized application using a compatible application deployment service.
- A classic Application or a single Fragment Application **SHALL** define its resources in the `/etc/holoscan/pkg.json#resource` object.
 - The `/etc/holoscan/pkg.json#resource` object is for the whole application. It **CAN** also be used as a catchall for all fragments in a multi-fragment application where applicable.
 - CPU requirements **SHALL** be denoted using the decimal count of CPU cores (`/etc/holoscan/pkg.json#resources.cpu`).

- Optional CPU limits SHALL be denoted using the decimal count of CPU cores (`/etc/holoscan/pkg.json#resources.cpuLimit`)
- GPU requirements SHALL be denoted using the decimal count of GPUs (`/etc/holoscan/pkg.json#resources.gpu`).
- Optional GPU limits SHALL be denoted using the decimal count of GPUs (`/etc/holoscan/pkg.json#resources.gpuLimit`)
- Memory requirements SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.memory`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
- Optional memory limits SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.memoryLimit`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
- GPU memory requirements SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.gpuMemory`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
- Optional GPU memory limits SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.gpuMemoryLimit`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
- Shared memory requirements SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.sharedMemory`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
- Optional shared memory limits SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.sharedMemoryLimit`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
- Integer values MUST be positive and not contain any position separators.
 - * Example legal values: 1, 42, 2048
 - * Example illegal values: -1, 1.5, 2,048
- Decimal values MUST be positive, rounded to the nearest tenth, use the dot (.) character to separate whole and fractional values, and not contain any positional separators.
 - * Example legal values: 1, 1.0, 0.5, 2.5, 1024
 - * Example illegal values: 1,024, -1.0, 3.14
- When not provided, the default values of `cpu=1`, `gpu=0`, `memory="1Gi"`, and `sharedMemory="64Mi"` will be assumed.

- A Multi-Fragment Application SHOULD define its resources in the `/etc/holoscan/pkg.json#resource.fragments.<fragment-name>` object.
 - When a matching `fragment-name` cannot be found, the `/etc/holoscan/pkg.json#resource` definition is used.
 - Fragment names (`fragment-name`) SHALL NOT contain any Unicode whitespace or control characters.
 - Fragment names (`fragment-name`) SHALL NOT exceed 128 bytes in length.
 - CPU requirements for fragments SHALL be denoted using the decimal count of CPU cores (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.cpu`).
 - Optional CPU limits for fragments SHALL be denoted using the decimal count of CPU cores (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.cpuLimit`).
 - GPU requirements for fragments SHALL be denoted using the decimal count of GPUs (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.gpu`).
 - Optional GPU limits for fragments SHALL be denoted using the decimal count of GPUs (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.gpuLimit`).
 - Memory requirements for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.memory`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
 - Optional memory limits for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.memoryLimit`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
 - GPU memory requirements for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.gpuMemory`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
 - Optional GPU memory limits for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.gpuMemoryLimit`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
 - Shared memory requirements for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.sharedMemory`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
 - Optional shared memory limits for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.sharedMemoryLimit`).
 - * Supported units SHALL be mebibytes (MiB) and gibibytes (GiB).
 - Example: 1.5Gi, 2048Mi
 - Integer values MUST be positive and not contain any position separators.
 - * Example legal values: 1, 42, 2048

- * Example illegal values: -1, 1.5, 2,048
- Decimal values MUST be positive, rounded to the nearest tenth, use the dot (.) character to separate whole and fractional values, and not contain any positional separators.
 - * Example legal values: 1, 1.0, 0.5, 2.5, 1024
 - * Example illegal values: 1,024, -1.0, 3.14
- When not provided, the default values of `cpu=1`, `gpu=0`, `memory="1Gi"`, and `sharedMemory="64Mi"` will be assumed.

24.6 Supplemental Application Files

- A HAP SHOULD package supplemental application files provided by the user.
 - Supplemental files SHOULD be in sub-folders of the `/opt/holoscan/docs/` folder.
 - Supplemental files include, but are not limited to, the following:
 - * `README.md`
 - * `License.txt`
 - * `Changelog.txt`
 - * `EULA`
 - * `Documentation`
 - * `Third-party licenses`

24.6.1 Container Behavior and Interaction

A HAP is a single container supporting the following defined behaviors when started.

Default Behavior

When a HAP is started from the CLI or other means without any parameters, the HAP shall execute the contained application. The HAP internally may use `Entrypoint`, `CMD`, or a combination of both.

Manifest Export

A HAP SHOULD provide at least one method to access the *embedded application*, *models*, *licenses*, *README*, or *manifest files*, namely, `app.json` and `package.json`.

- The Method SHOULD provide a container command, `show`, to print one or more manifest files to the console.
- The Method SHOULD provide a container command, `export`, to copy one or more manifest files to a mounted volume path, as described below
 - `/var/run/holoscan/export/app/`: when detected, the Method copies the contents of `/opt/holoscan/app/` to the folder.
 - `/var/run/holoscan/export/config/`: when detected, the Method copies `/var/holoscan/app.yaml`, `/etc/holoscan/app.json` and `/etc/holoscan/pkg.json` to the folder.
 - `/var/run/holoscan/export/models/`: when detected, the Method copies the contents of `/opt/holoscan/models/` to the folder.

- `/var/run/holoscan/export/docs/`: when detected, the Method copies the contents of `/opt/holoscan/docs/` to the folder.
- `/var/run/holoscan/export/`: when detected without any of the above being detected, the Method SHALL copy all of the above.

Since a HAP is an OCI compliant container, a user can also run a HAP and log in to an interactive shell, using a method supported by the container engine and its command line interface; e.g., Docker supports this by setting the `entrypoint` option. The files in the HAP can then be opened or copied to the mapped volumes with shell commands or scripts. A specific implementation of a HAP may choose to streamline such a process with scripts and applicable user documentation.

24.6.2 Table of Important Paths

Path	Purpose
<code>/etc/holoscan/</code>	HAP manifests and immutable configuration files.
<code>/etc/holoscan/app.json</code>	Application Manifest file.
<code>/etc/holoscan/pkg.json</code>	Package Manifest file.
<code>/opt/holoscan/app/</code>	Application code, scripts, and other files.
<code>/opt/holoscan/models/</code>	AI models. Each model should be in a separate sub-folder.
<code>/opt/holoscan/docs/</code>	Documentation, licenses, EULA, changelog, etc...
<code>/var/holoscan/</code>	Default working directory.
<code>/var/holoscan/input/</code>	Default input directory.
<code>/var/holoscan/output/</code>	Default output directory.
<code>/var/run/holoscan/export/</code>	Special case folder, causes the Script to export contents related to the app. (see: Manifest Export)
<code>/var/run/holoscan/export/app/</code>	Special case folder, causes the Script to export the contents of <code>/opt/holoscan/app/</code> to the folder.
<code>/var/run/holoscan/export/config/</code>	Special case folder, causes the Script to export <code>/etc/holoscan/app.json</code> and <code>/etc/holoscan/pkg.json</code> to the folder.
<code>/var/run/holoscan/export/models/</code>	Special case folder, causes the Script to export the contents of <code>/opt/holoscan/models/</code> to the folder.

24.7 Operating Environments

Holoscan SDK supports the following operating environments.

Operating Environment Name	Characteristics
AGX Devkit	Clara AGX devkit with RTX 6000 dGPU only
IGX Orin Devkit	Clara Holoscan devkit with A6000 dGPU only
IGX Orin Devkit - integrated GPU only	IGX Orin Devkit, iGPU only
IGX Orin Devkit with discrete GPU	IGX Orin Devkit, with RTX A6000 dGPU
Jetson AGX Orin Devkit	Jetson Orin Devkit, iGPU only
Jetson Orin Nano Devkit	Jetson Orin Nano Devkit, iGPU only
X86_64	dGPU only on Ubuntu 18.04 and 20.04

HOLOSCAN CLI

holoscan - a command-line interface for packaging and running your Holoscan applications into *HAP-compliant* containers.

25.1 Synopsis

holoscan [*--help*|-h] [*--log-level*|-l {DEBUG,INFO,WARN,ERROR,CRITICAL}] {*package,run,version,nics*}

25.2 Positional Arguments

25.2.1 Holoscan CLI - Package Command

holoscan package - generate *HAP-compliant* container for your application.

Synopsis

holoscan package [*--help*|-h] [*--log-level*|-l {DEBUG,INFO,WARN,ERROR,CRITICAL}] *--config*|-c CONFIG [*-docs*|-d DOCS] [*--models*|-m MODELS] *--platform* PLATFORM [*--platform-config* PLATFORM_CONFIG] [*--timeout* TIMEOUT] [*--version* VERSION] [*--base-image* BASE_IMAGE] [*--build-image* BUILD_IMAGE] [*--includes* [{debug,holoviz,torch,onnx}]] [*--build-cache* BUILD_CACHE] [*--cmake-args* CMAKE_ARGS] [*--no-cache*|-n] [*--sdk* SDK] [*--source* URL|FILE] [*--sdk-version* SDK_VERSION] [*--holoscan-sdk-file* HOLOSCAN_SDK_FILE] [*--monai-deploy-sdk-file* MONAI_DEPLOY_SDK_FILE] [*--output*|-o OUTPUT] *--tag*|-t TAG [*--username* USERNAME] [*--uid* UID] [*--gid* GID] application [*--source* URL|FILE]

Examples

The code below package a python application for x86_64 systems:

```
# Using a Python directory as input
# Required: a `__main__.py` file in the application directory to execute
# Optional: a `requirements.txt` file in the application directory to install dependencies
holoscan package --platform x64-workstation --tag my-awesome-app --config /path/to/my/
➔ awesome/application/config.yaml /path/to/my/awesome/application/

# Using a Python file as input
holoscan package --platform x64-workstation --tag my-awesome-app --config /path/to/my/
➔ awesome/application/config.yaml /path/to/my/awesome/application/my-app.py
```

The code below package a C++ application for the IGX Orin DevKit (aarch64) with a discrete GPU:

```
# Using a C++ source directory as input
# Required: a `CMakeLists.txt` file in the application directory
holoscan package --platform igx-orin-devkit --platform-config dgpu --tag my-awesome-app -
↪-config /path/to/my/awesome/application/config.yaml /path/to/my/awesome/application/

# Using a C++ pre-compiled executable as input
holoscan package --platform igx-orin-devkit --platform-config dgpu --tag my-awesome-app -
↪-config /path/to/my/awesome/application/config.yaml /path/to/my/awesome/bin/
↪application-executable
```

Note: The commands above load the generated image onto Docker to make the image accessible with docker images.

If you need to package for a different platform or want to transfer the generated image to another system, use the `--output /path/to/output` flag so the generated package can be saved to the specified location.

Positional Arguments

application

Path to the application to be packaged. The following inputs are supported:

- **C++ source code:** you may pass a directory path with your C++ source code with a `CMakeLists.txt` file in it, and the **Packager** will attempt to build your application using CMake and include the compiled application in the final package.
- **C++ pre-compiled executable:** A pre-built executable binary file may be directly provided to the **Packager**.
- **Python application:** you may pass either:
 - a directory which includes a `__main__.py` file to execute (required) and an optional `requirements.txt` file that defined dependencies for your Python application, or
 - the path to a single python file to execute

Warning: Python (PyPI) modules are installed into the user's (via `[--username USERNAME]` argument) directory with the user ID specified via `[--uid UID]`. Therefore, when running a packaged Holoscan application on Kubernetes or other service providers, running Docker with non root user, and running Holoscan CLI `run` command where the logged-on user's ID is different, ensure to specify the `USER` ID that is used when building the application package.

For example, include the `securityContext` when running a Holoscan packaged application with `UID=1000` using Argo:

```
spec:
  securityContext:
    runAsUser: 1000
    runAsNonRoot: true
```

Flags

--config|-c CONFIG

Path to the application's *configuration file*. The configuration file must be in YAML format with a `.yaml` file extension.

[--docs|-d DOCS]

An optional directory path of documentation, README, licenses that shall be included in the package.

[--models|-m MODELS]

An optional directory path to a model file, a directory with a single model, or a directory with multiple models.

Single model example:

```
my-model/
├── surgical_video.gxf_entities
└── surgical_video.gxf_index

my-model/
└── model
    ├── surgical_video.gxf_entities
    └── surgical_video.gxf_index
```

Multi-model example:

```
my-models/
├── model-1
│   ├── my-first-model.gxf_entities
│   └── my-first-model.gxf_index
└── model-2
    └── my-other-model.ts
```

--platform PLATFORM

A comma-separated list of platform types to generate. Each platform value specified generates a standalone container image. If you are running the **Packager** on the same architecture, the generated image is automatically loaded onto Docker and is available with `docker images`. Otherwise, use `--output` flag to save the generated image onto the disk.

PLATFORM must be one of: `clara-agx-devkit`, `igx-orin-devkit`, `jetson-agx-orin-devkit`, `x64-workstation`.

- `igx-orin-devkit`: IGX Orin DevKit
- `jetson-agx-orin-devkit`: Orin AGX DevKit
- `x64-workstation`: systems with a `x86-64` processor(s)

`[--platform-config PLATFORM_CONFIG]`

Specifies the platform configuration to generate. PLATFORM_CONFIG must be one of: `igpu`, `igpu-assist`, `dgpu`.

- `igpu`: Supports integrated GPU
- `igpu-assist`: Supports compute-only tasks on iGPU in presence of a dGPU
- `dgpu`: Supports dedicated GPU

Note: `--platform-config` is required when `--platform` is not `x64-workstation` (which uses `dgpu`).

`[--timeout TIMEOUT]`

An optional timeout value of the application for the supported orchestrators to manage the application's lifecycle. Defaults to 0.

`[--version VERSION]`

An optional version number of the application. When specified, it overrides the value specified in the *configuration file*.

`[--base-image BASE_IMAGE]`

Optionally specifies the base container image for building packaged application. It must be a valid Docker image tag either accessible online or via ``docker`` images. By default, the **Packager** picks a base image to use from [NGC](#).

`[--build-image BUILD_IMAGE]`

Optionally specifies the build container image for building C++ applications. It must be a valid Docker image tag either accessible online or via ``docker`` images. By default, the **Packager** picks a build image to use from [NGC](#).

`[--includes [{debug,holoviz,torch,onnx}]]`

To reduce the size of the packaged application container, the CLI Packager, by default, includes minimum runtime dependencies to run applications designed for Holoscan. You can specify additional runtime dependencies to be included in the packaged application using this option. The following options are available:

- `debug`: includes debugging tools, such as `gdb`
- `holoviz`: includes dependencies for Holoviz rendering on x11 and Wayland
- `torch`: includes `libtorch` and `torchvision` runtime dependencies
- `onnx`: includes `onnxruntime` runtime, `libnvinfer-plugin8`, `libnconnxparser8` dependencies.

Note: Refer to [Developer Resources](#) for dependency versions.

Usage:

```
holoscan package --includes holoviz torch onnx
```

`[--build-cache BUILD_CACHE]`

Specifies a directory path for storing Docker cache. Defaults to `~/holoscan_build_cache`. If the `$HOME` directory is inaccessible, the CLI uses the `/tmp` directory.

`[--cmake-args CMAKE_ARGS]`

A comma-separated list of *cmake* arguments to be used when building C++ applications.

For example:

```
holoscan package --cmake-args "-DCMAKE_BUILD_TYPE=DEBUG -DCMAKE_ARG=VALUE"
```

`[--no-cache|-n]`

Do not use cache when building image.

`[--sdk SDK]`

SDK for building the application: Holoscan or MONAI-Deploy. SDK must be one of: holoscan, monai-deploy.

`[--source URL|FILE]`

Override the artifact manifest source with a securely hosted file or from the local file system.

E.g. `https://my.domain.com/my-file.json`

`[--sdk-version SDK_VERSION]`

Set the version of the SDK that is used to build and package the Application. If not specified, the packager attempts to detect the installed version.

`[--holoscan-sdk-file HOLOSCAN_SDK_FILE]`

Path to the Holoscan SDK Debian or PyPI package. If not specified, the packager downloads the SDK file from the internet depending on the SDK version detected/specified. The `HOLOSCAN_SDK_FILE` filename must have `.deb` or `.whl` file extension for Debian package or PyPI wheel package, respectively.

[--monai-deploy-sdk-file MONAI_DEPLOY_SDK_FILE]

Path to the MONAI Deploy App SDK Debian or PyPI package. If not specified, the packager downloads the SDK file from the internet based on the SDK version. The MONAI_DEPLOY_SDK_FILE package filename must have .whl or .gz file extension.

[--output|-o OUTPUT]

Output directory where result images will be written.

Note: If this flag isn't present, the packager will load the generated image onto Docker to make the image accessible with `docker images`. The `--output` flag is therefore required when building a packaging for a different target architecture than the host system that runs the packager.

--tag|-t TAG

Name and optionally a tag (format: `name:tag`).

For example:

```
my-company/my-application:latest
my-company/my-application:1.0.0
my-application:1.0.1
my-application
```

[--username USERNAME]

Optional *username* to be created in the container execution context. Defaults to `holoscan`.

[--uid UID]

Optional *user ID* to be associated with the user created with `--username` with default of `1000`.

Warning: A very large UID value may result in a very large image due to an open [issue](#) with Docker. It is recommended to use the default value of `1000` when packaging an application and use your current UID/GID when running the application.

`[--gid GID]`

Optional *group ID* to be associated with the user created with `--username` with default of `1000`

25.2.2 Holoscan CLI - Run Command

`holoscan run` - simplifies running a packaged Holoscan application by reducing the number of arguments required compared to `docker run`. In addition, it follows the guidelines of *HAP specification* when launching your packaged Holoscan application.

Warning: When running a packaged Holoscan application on Kubernetes or other service providers, running Docker with non root user, and running Holoscan CLI `run` command where the logged-on user's ID is different, ensure to specify the `USER ID` that is used when building the application package.

For example, include the `securityContext` when running a Holoscan packaged application with `UID=1000` using Argo:

```
spec:
  securityContext:
    runAsUser: 1000
    runAsNonRoot: true
```

Synopsis

```
holoscan run [--help|-h] [--log-level|-l {DEBUG,INFO,WARN,ERROR,CRITICAL}] [--address ADDRESS] [--driver]
[--input|-i INPUT] [--output|-o OUTPUT] [--fragments|-f FRAGMENTS] [--worker] [--worker-address WORKER_ADDRESS]
[--config CONFIG] [--network|-n NETWORK] [--nic NETWORK_INTERFACE] [--use-all-nics] [--render|-r] [--quiet|-q]
[--shm-size] [--terminal] [--device] [--gpu] [--uid UID] [--gid GID] image:[tag]
```

Examples

To run a packaged Holoscan application:

```
holoscan run -i /path/to/my/input -o /path/to/application/generated/output my-
↪ application:1.0.1
```

Positional Arguments

`image: [tag]`

Name and tag of the Docker container image to execute.

Flags

`[--address ADDRESS]`

Address (`[<IP or hostname>][:<port>]`) of the *App Driver*. If not specified, the *App Driver* uses the default host address (`0.0.0.0`) with the default port number (8765).

For example:

```
--address my_app_network
--address my_app_network:8765
```

Note: Ensure that the IP address is not blocked and the port is configured with the firewall accordingly.

`[--driver]`

Run the **App Driver** on the current machine. Can be used together with the `[--worker]` option to run both the **App Driver** and the **App Worker** on the same machine.

`[--input|-i INPUT]`

Specifies a directory path with input data for the application to process. When specified, a directory mount is set up to the value defined in the environment variable `HOLOSCAN_INPUT_PATH`.

Note: Ensure that the directory on the host is accessible by the current user or the user specified with `-uid`.

Note: Use the host system path when running applications inside Docker (DooD).

`[--output|-o OUTPUT]`

Specifies a directory path to store application-generated artifacts. When specified, a directory mount is set up to the value defined in the environment variable `HOLOSCAN_OUTPUT_PATH`.

Note: Ensure that the directory on the host is accessible by the current user or the user specified with `-uid`.

[--fragments|-f FRAGMENTS]

Comma-separated names of the fragments to be executed by the **App Worker**. If not specified, only one fragment (selected by the **App Driver**) will be executed. `all` can be used to run all the fragments.

[--worker]

Run the **App Worker**.

[--worker-address WORKER_ADDRESS]

The address ([<IP or hostname>][:<port>]) of the **App Worker**. If not specified, the **App Worker** uses the default host address (0.0.0.0) with a randomly chosen port number between 10000 and 32767 that is not currently in use. This argument automatically sets the `HOLOSCAN_UCX_SOURCE_ADDRESS` environment variable if the worker address is a local IP address. Refer to *Environment Variables for Distributed Applications* for details.

For example:

```
--worker-address my_app_network
--worker-address my_app_network:10000
```

Note: Ensure that the IP address is not blocked and the port is configured with the firewall accordingly.

[--config CONFIG]

Path to the application configuration file. If specified, it overrides the embedded configuration file found in the environment variable `HOLOSCAN_CONFIG_PATH`.

[--network|-n NETWORK]

The Docker network that the application connects to for communicating with other containers. The **Runner** use the host network by default if not specified. Otherwise, the specified value is used to create a network with the bridge driver.

For advanced uses, first create a network using `docker network create` and pass the name of the network to the `--network` option. Refer to *Docker Networking* documentation for additional details.

[--nic NETWORK_INTERFACE]

Name of the network interface to use with a distributed multi-fragment application. This option sets `UCX_NET_DEVICES` environment variable with the value specified and is required when running a distributed multi-fragment application across multiple nodes. See *UCX Network Interface Selection* for details.

`[--use-all-nics]`

When set, this option allows UCX to control the selection of network interface cards for data transfer. Otherwise, the network interface card specified with ‘`–nic`’ is used. This option sets the environment variable `UCX_CM_USE_ALL_DEVICES` to `y` (default: `False`).

When this option is not set, the CLI runner always sets `UCX_CM_USE_ALL_DEVICES` to `n`.

`[--render|-r]`

Enable graphic rendering from your application. Defaults to `False`.

`[--quiet|-q]`

Suppress the `STDOUT` and print only `STDERR` from the application. Defaults to `False`.

`[--shm-size]`

Sets the size of `/dev/shm`. The format is `<number(int,float)>[MB|m|GB|g|Mi|MiB|Gi|GiB]`. Use `config` to read the shared memory value defined in the `app.json` manifest. By default, the container is launched using `--ipc=host` with host system’s `/dev/shm` mounted.

`[--terminal]`

Enters terminal with all configured volume mappings and environment variables.

`[--device]`

Map host devices into the application container.

By default, the CLI searches the `/dev/` path for devices unless the specified string starts with `/`.

For example:

```
# mount all AJA capture cards
--device ajantv*
# mount AJA capture card 0 and 1
--device ajantv0 ajantv1
# mount V4L2 video device 1 and AJAX capture card 2
--device video1 --device /dev/ajantv2
```

Warning: When using the `--device` option, append `--` after the last item to avoid misinterpretation by the CLI. For example:

```
holoscan run --render --device ajantv0 video1 -- my-application-image:1.0
```

[--gpu]

Override the value of the `NVIDIA_VISIBLE_DEVICES` environment variable with the default value set to the value defined in the *package manifest file* or `all` if undefined.

Refer to the [GPU Enumeration](#) page for all available options.

Note: The default value is `nvidia.com/igpu=0` when running a HAP built for iGPU on a system with both iGPU and dGPU,

Note: A single integer value translates to the device index, not the number of GPUs.

[--uid UID]

Run the application with the specified user ID (UID). Defaults to the current user's UID.

[--gid GID]

Run the application with the specified group ID (GID). Defaults to the current user's GID.

Note: The Holoscan Application supports various environment variables for configuration. Refer to *Environment Variables for Distributed Applications* for details.

25.2.3 Holoscan CLI - Version Command

`holoscan version` - print version information for the Holoscan SDK

Synopsis

`holoscan version [--help|-h] [--log-level|-l {DEBUG,INFO,WARN,ERROR,CRITICAL}]`

package

Package a Holoscan application

run

Run a packaged Holoscan application

version

Print version information for the Holoscan SDK

nics

Print all available network interface cards and its assigned IP address

25.3 CLI-Wide Flags

25.3.1 [--help|-h]

Display detailed help.

25.3.2 [--log-level|-l {DEBUG,INFO,WARN,ERROR,CRITICAL}]

Override the default logging verbosity. Defaults to INFO.

APPLICATION RUNNER CONFIGURATION

The Holoscan runner requires a YAML configuration file to define some properties necessary to deploy an application.

Note: That file is the same configuration file commonly used to configure other aspects of an application, documented [here](#).

26.1 Configuration

The configuration file can be defined in two ways:

- At package time, with the `--config` flag of the `holoscan package` command (Required/Default).
- At runtime, with the `--config` flag of the `holoscan run` command (Optional/Override).

26.2 Properties

The `holoscan run` command parses two specific YAML nodes from the configuration file:

- A required `application` parameter group to generate a *HAP-compliant* container image for the application, including:
 - The `title` (name) and `version` of the application.
 - Optionally, `inputFormats` and `outputFormats` if the application expects any inputs or outputs respectively.
- An optional `resources` parameter group that defines the system resources required to run the application, such as the number of CPUs, GPUs and amount of memory required. If the application contains multiple fragments for distributed workloads, resource definitions can be assigned to each fragment.

26.3 Example

Below is an example configuration file with the application and optional resources parameter groups, for an application with two-fragments (first-fragment and second-fragment):

```
application:
  title: My Application Title
  version: 1.0.1
  inputFormats: ["files"] # optional
  outputFormats: ["screen"] # optional

resources: # optional
  # non-distributed app
  cpu: 1 # optional
  cpuLimit: 5 # optional
  gpu: 1 # optional
  gpuLimit: 5 # optional
  memory: 1Mi # optional
  memoryLimit: 2Gi # optional
  gpuMemory: 1Gi # optional
  gpuMemoryLimit: 1.5Gi # optional
  sharedMemory: 1Gi # optional

  # distributed app
  fragments: # optional
    first-fragment: # optional
      cpu: 1 # optional
      cpuLimit: 5 # optional
      gpu: 1 # optional
      gpuLimit: 5 # optional
      memory: 100Mi # optional
      memoryLimit: 1Gi # optional
      gpuMemory: 1Gi # optional
      gpuMemoryLimit: 10Gi # optional
      sharedMemory: 1Gi # optional
    second-fragment: # optional
      cpu: 1 # optional
      cpuLimit: 2 # optional
      gpu: 1 # optional
      gpuLimit: 2 # optional
      memory: 1Gi # optional
      memoryLimit: 2Gi # optional
      gpuMemory: 1Gi # optional
      gpuMemoryLimit: 5Gi # optional
      sharedMemory: 10Mi # optional
```

For details, please refer to the *HAP specification*.

GXF CORE CONCEPTS

Here is a list of the key GXF terms used in this section:

- **Applications** are built as compute graphs.
- **Entities** are nodes of the graph. They are nothing more than a unique identifier.
- **Components** are parts of an entity and provide their functionality.
- **Codelets** are special components which allow the execution of custom code. They can be derived by overriding the C++ functions `initialize`, `start`, `tick`, `stop`, `deinitialize`, and `registerInterface` (for defining configuration parameters).
- **Connections** are edges of the graph, which connect components.
- **Scheduler and Scheduling Terms**: components that determine how and when the `tick()` of a Codelet executes. This can be single or multithreaded, support conditional execution, asynchronous scheduling, and other custom behavior.
- **Memory Allocator**: provides a system for allocating a large contiguous memory pool up-front and then reusing regions as needed. Memory can be pinned to the device (enabling zero-copy between Codelets when messages are not modified) or host, or customized for other potential behavior.
- **Receivers, Transmitters, and Message Router**: a message passing system between Codelets that supports zero-copy.
- **Tensor**: the common message type is a tensor. It provides a simple abstraction for numeric data that can be allocated, serialized, sent between Codelets, etc. Tensors can be rank 1 to 7 supporting a variety of common data types like arrays, vectors, matrices, multi-channel images, video, regularly sampled time-series data, and higher dimensional constructs popular with deep learning flows.
- **Parameters**: configuration variables used by the Codelet. In GXF applications, they are loaded from the application YAML file and are modifiable without recompiling.

In comparison, the core concepts of the Holoscan SDK can be found [here](#).

HOLOSCAN AND GXF

28.1 Design differences

There are 2 main elements at the core of Holoscan and GXF designs:

1. How to define and execute application graphs.
2. How to define nodes' functionality.

How Holoscan SDK interfaces with GXF on those topics varies as Holoscan SDK evolves, as described below:

28.1.1 Holoscan SDK v0.2

Holoscan SDK was tightly coupled with GXF's existing interface:

1. GXF application graphs are defined in **YAML** configuration files. **GXE** (Graph Execution Engine) is used to execute AI application graphs. Its inputs are the YAML configuration file, and a list of GXF Extensions to load as plugins (manifest YAML file). This design allows entities to be swapped or updated without needing to recompile an application.
2. Components are made available by registering them within a **GXF extension**, each of which maps to a shared library and header(s).

Those concepts are illustrated in the *GXF by example* section.

The only additions that Holoscan provided on top of GXF were:

- Domain-specific reference applications
- New extensions
- CMake configurations for building extensions and applications

28.1.2 Holoscan SDK v0.3

The Holoscan SDK shifted to provide a more developer-friendly interface with C++:

1. GXF application graphs, memory allocation, scheduling, and message routing can be defined using a C++ API, with the ability to read parameters and required GXF extension names from a YAML configuration file. The backend used is still GXF as Holoscan uses the GXF C API, but this bypasses GXE and the full YAML definition.
2. The C++ **Operator** class was added to wrap and expose GXF extensions to that new application interface (See *dev guide*).

28.1.3 Holoscan SDK v0.4

The Holoscan SDK added Python wrapping and native operators to further increase ease of use:

1. The C++ API is also wrapped in Python. GXF is still used as the backend.
2. The Operator class supports **native operators**, i.e., operators that do not require that you implement and register a GXF Extension. An important feature is the ability to support messaging between native and GXF operators without any performance loss (i.e., zero-copy communication of tensors).

28.1.4 Holoscan SDK v0.5

1. The built-in Holoscan GXF extensions are loaded automatically and don't need to be listed in the YAML configuration file of Holoscan applications. This allows Holoscan applications to be defined without requiring a YAML configuration file.
2. No significant changes to build operators. However, most built-in operators were switched to native implementations, with the ability to *convert native operators to GXF codelets* for GXF application developers.

28.1.5 Holoscan SDK v1.0

1. The remaining GXF-based DemosaicOp operator was switched to a native implementation. Now all operators provided by the SDK are native operators.

28.2 Current Limitations

Here is a list of GXF capabilities not yet available in the Holoscan SDK which are planned to be supported in future releases:

- *Job Statistics*

The GXF capabilities below are not available in the Holoscan SDK either. There is no plan to support them at this time:

- *Graph Composer*
- *Behavior Trees*
- *Epoch Scheduler*
- *Target Time Scheduling Term*
- *Multi-Message Available Scheduling Term*
- *Expiring Message Available Scheduling Term*

GXF BY EXAMPLE

Warning: This section is legacy (0.2), as we recommend developing extensions and applications using the C++ or Python APIs. Refer to the developer guide for up-to-date recommendations.

29.1 Inner Workings of a GXF Entity

Let us look at an example of a GXF entity to try to understand its general anatomy. As an example let's start with the entity definition for an image format converter entity named `format_converter_entity` as shown below.

Listing 29.1: An example GXF Application YAML snippet

```
1 %YAML 1.2
2 ---
3 # other entities declared
4 ---
5 name: format_converter_entity
6 components:
7   - name: in_tensor
8     type: nvidia::gxf::DoubleBufferReceiver
9   - type: nvidia::gxf::MessageAvailableSchedulingTerm
10    parameters:
11      receiver: in_tensor
12      min_size: 1
13   - name: out_tensor
14     type: nvidia::gxf::DoubleBufferTransmitter
15   - type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
16    parameters:
17      transmitter: out_tensor
18      min_size: 1
19   - name: pool
20     type: nvidia::gxf::BlockMemoryPool
21    parameters:
22      storage_type: 1
23      block_size: 4919040 # 854 * 480 * 3 (channel) * 4 (bytes per pixel)
24      num_blocks: 2
25   - name: format_converter_component
26     type: nvidia::holoscan::formatconverter::FormatConverter
27    parameters:
28      in: in_tensor
```

(continues on next page)

(continued from previous page)

```

29     out: out_tensor
30     out_tensor_name: source_video
31     out_dtype: "float32"
32     scale_min: 0.0
33     scale_max: 255.0
34     pool: pool
35 ---
36 # other entities declared
37 ---
38 components:
39   - name: input_connection
40     type: nvidia::gxf::Connection
41     parameters:
42       source: upstream_entity/output
43       target: format_converter/in_tensor
44 ---
45 components:
46   - name: output_connection
47     type: nvidia::gxf::Connection
48     parameters:
49       source: format_converter/out_tensor
50       target: downstream_entity/input
51 ---
52 name: scheduler
53 components:
54   - type: nvidia::gxf::GreedyScheduler

```

Above:

1. The entity `format_converter_entity` receives a message in its `in_tensor` message from an upstream entity `upstream_entity` as declared in the `input_connection`.
2. The received message is passed to the `format_converter_component` component to convert the tensor element precision from `uint8` to `float32` and scale any input in the `[0, 255]` intensity range.
3. The `format_converter_component` component finally places the result in the `out_tensor` message so that its result is made available to a downstream entity (`downstream_entity` as declared in `output_connection`).
4. The `Connection` components tie the inputs and outputs of various components together, in the above case `upstream_entity/output -> format_converter_entity/in_tensor` and `format_converter_entity/out_tensor -> downstream_entity/input`.
5. The `scheduler` entity declares a `GreedyScheduler` “system component” which orchestrates the execution of the entities declared in the graph. In the specific case of `GreedyScheduler` entities are scheduled to run exclusively, where no more than one entity can run at any given time.

The YAML snippet above can be visually represented as follows.

In the image, as in the YAML, you will notice the use of `MessageAvailableSchedulingTerm`, `DownstreamReceptiveSchedulingTerm`, and `BlockMemoryPool`. These are components that play a “supporting” role to `in_tensor`, `out_tensor`, and `format_converter_component` components respectively. Specifically:

- `MessageAvailableSchedulingTerm` is a component that takes a `Receiver` (in this case `DoubleBufferReceiver` named `in_tensor`) and alerts the graph `Executor` that a message is available. This alert triggers `format_converter_component`.
- `DownstreamReceptiveSchedulingTerm` is a component that takes a `Transmitter` (in this case

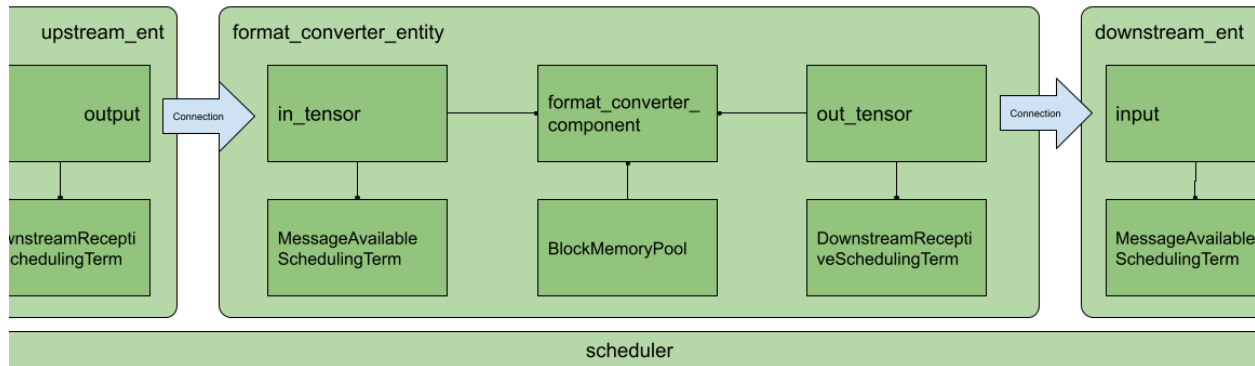


Fig. 29.1: Arrangement of components and entities in a Holoscan application

DoubleBufferTransmitter named `out_tensor`) and alerts the graph Executor that a message has been placed on the output.

- `BlockMemoryPool` provides two blocks of almost 5MB allocated on the GPU device and is used by `format_converter_entity` to allocate the output tensor where the converted data will be placed within the format converted component.

Together these components allow the entity to perform a specific function and coordinate communication with other entities in the graph, via the declared scheduler.

More generally, an entity can be thought of as a collection of components where components can be passed to one another to perform specific subtasks (e.g., event triggering or message notification, format conversion, memory allocation), and an application as a graph of entities.

The scheduler is a component of type `nvidia::gxf::System` which orchestrates the execution components in each entity at application runtime based on triggering rules.

29.2 Data Flow and Triggering Rules

Entities communicate with one another via messages which may contain one or more payloads. Messages are passed and received via a component of type `nvidia::gxf::Queue` from which both `nvidia::gxf::Receiver` and `nvidia::gxf::Transmitter` are derived. Every entity that receives and transmits messages has at least one receiver and one transmitter queue.

Holoscan uses the `nvidia::gxf::SchedulingTerm` component to coordinate data access and component orchestration for a Scheduler which invokes execution through the `tick()` function in each `Codelet`.

Tip: A `SchedulingTerm` defines a specific condition that is used by an entity to let the scheduler know when it's ready for execution.

In the above example, we used a `MessageAvailableSchedulingTerm` to trigger the execution of the components waiting for data from `in_tensor` receiver queue, namely `format_converter_component`.

Listing 29.2: MessageAvailableSchedulingTerm

```
1 - type: nvidia::gxf::MessageAvailableSchedulingTerm
2 parameters:
```

(continues on next page)

(continued from previous page)

```

3 receiver: in_tensor
4 min_size: 1

```

Similarly, `DownStreamReceptiveSchedulingTerm` checks whether the `out_tensor` transmitter queue has at least one outgoing message in it. If there are one or more outgoing messages, `DownStreamReceptiveSchedulingTerm` will notify the scheduler, which in turn attempts to place the message in the receiver queue of a downstream entity. If, however, the downstream entity has a full receiver queue, the message is held in the `out_tensor` queue as a means to handle back-pressure.

Listing 29.3: `DownstreamReceptiveSchedulingTerm`

```

1 - type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
2 parameters:
3   transmitter: out_tensor
4   min_size: 1

```

If we were to draw the entity in [Fig. 29.1](#) in greater detail it would look something like the following.

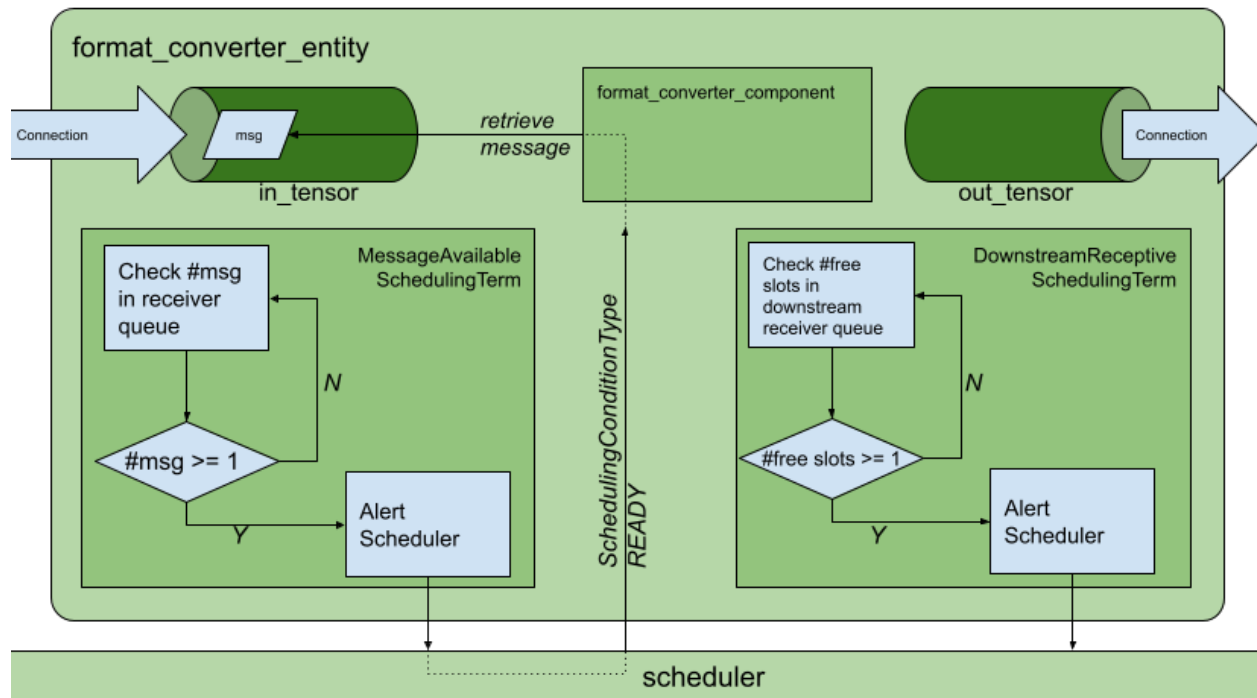


Fig. 29.2: Receive and transmit Queues and SchedulingTerms in entities.

Up to this point, we have covered the “entity component system” at a high level and showed the functional parts of an entity; namely, the messaging queues and the scheduling terms that support the execution of components in the entity. To complete the picture, the next section covers the anatomy and lifecycle of a component, and how to handle events within it.

29.3 Creating a GXF Extension

GXF components in Holoscan can perform a multitude of sub-tasks ranging from data transformations, to memory management, to entity scheduling. In this section, we will explore an `nvidia::gxf::Codelet` component which in Holoscan is known as a “GXF extension.” *Holoscan (GXF) extensions* are typically concerned with application-specific sub-tasks such as data transformations, AI model inference, and the like.

29.3.1 Extension Lifecycle

The lifecycle of a `Codelet` is composed of the following five stages.

1. **initialize** - called only once when the codelet is created for the first time, and use of lightweight initialization.
2. **deinitialize** - called only once before the codelet is destroyed, and used for lightweight deinitialization.
3. **start** - called multiple times over the lifecycle of the codelet according to the order defined in the lifecycle, and used for heavy initialization tasks such as allocating memory resources.
4. **stop** - called multiple times over the lifecycle of the codelet according to the order defined in the lifecycle, and used for heavy deinitialization tasks such as deallocation of all resources previously assigned in **start**.
5. **tick** - called when the codelet is triggered, and is called multiple times over the codelet lifecycle; even multiple times between **start** and **stop**.

The flow between these stages is detailed in [Fig. 29.3](#).

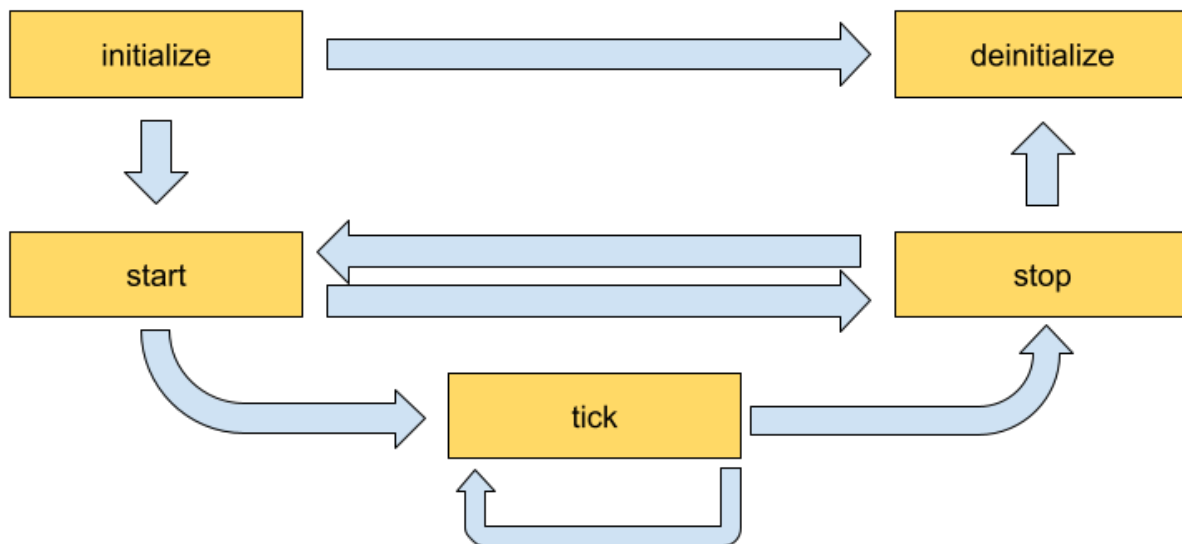


Fig. 29.3: Sequence of method calls in the lifecycle of a Holoscan extension

29.3.2 Implementing an Extension

In this section, we will implement a simple recorder that will highlight the actions we would perform in the lifecycle methods. The recorder receives data in the input queue and records the data to a configured location on the disk. The output format of the recorder files is the GXF-formatted index/binary replayer files (the format is also used for the data in the sample applications), where the `gxf_index` file contains timing and sequence metadata that refer to the binary/tensor data held in the `gxf_entities` file.

Declare the Class That Will Implement the Extension Functionality

The developer can create their Holoscan extension by extending the `Codelet` class, implementing the extension functionality by overriding the lifecycle methods, and defining the parameters the extension exposes at the application level via the `registerInterface` method. To define our recorder component, we would need to implement some of the methods in the `Codelet`.

First, clone the Holoscan project from [here](#) and create a folder to develop our extension such as under `gxf_extensions/my_recorder`.

Tip: Using Bash we create a Holoscan extension folder as follows.

```
git clone https://github.com/nvidia-holoscan/holoscan-sdk.git
cd clara-holoscan-embedded-sdk
mkdir -p gxf_extensions/my_recorder
```

In our extension folder, we create a header file `my_recorder.hpp` with a declaration of our Holoscan component.

Listing 29.4: `gxf_extensions/my_recorder/my_recorder.hpp`

```
1  #include <string>
2
3  #include "gxf/core/handle.hpp"
4  #include "gxf/std/codelet.hpp"
5  #include "gxf/std/receiver.hpp"
6  #include "gxf/std/transmitter.hpp"
7  #include "gxf/serialization/file_stream.hpp"
8  #include "gxf/serialization/entity_serializer.hpp"
9
10
11 class MyRecorder : public nvidia::gxf::Codelet {
12 public:
13     gxf_result_t registerInterface(nvidia::gxf::Registrar* registrar) override;
14     gxf_result_t initialize() override;
15     gxf_result_t deinitialize() override;
16
17     gxf_result_t start() override;
18     gxf_result_t tick() override;
19     gxf_result_t stop() override;
20
21 private:
22     nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::Receiver>> receiver_;
23     nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::EntitySerializer>> my_
    ↪ serializer_;
```

(continues on next page)

(continued from previous page)

```

24  nvidia::gxf::Parameter<std::string> directory_;
25  nvidia::gxf::Parameter<std::string> basename_;
26  nvidia::gxf::Parameter<bool> flush_on_tick_;
27
28  // File stream for data index
29  nvidia::gxf::FileStream index_file_stream_;
30  // File stream for binary data
31  nvidia::gxf::FileStream binary_file_stream_;
32  // Offset into binary file
33  size_t binary_file_offset_;
34  };

```

Declare the Parameters to Expose at the Application Level

Next, we can start implementing our lifecycle methods in the `my_recorder.cpp` file, which we also create in `gxf_extensions/my_recorder` path.

Our recorder will need to expose the `nvidia::gxf::Parameter` variables to the application so the parameters can be modified by configuration.

Listing 29.5: `registerInterface` in `gxf_extensions/my_recorder/my_recorder.cpp`

```

1  #include "my_recorder.hpp"
2
3  gxf_result_t MyRecorder::registerInterface(nvidia::gxf::Registrar* registrar) {
4      nvidia::gxf::Expected<void> result;
5      result &= registrar->parameter(
6          receiver_, "receiver", "Entity receiver",
7          "Receiver channel to log");
8      result &= registrar->parameter(
9          my_serializer_, "serializer", "Entity serializer",
10         "Serializer for serializing input data");
11     result &= registrar->parameter(
12         directory_, "out_directory", "Output directory path",
13         "Directory path to store received output");
14     result &= registrar->parameter(
15         basename_, "basename", "File base name",
16         "User specified file name without extension",
17         nvidia::gxf::Registrar::NoDefaultParameter(), GXF_PARAMETER_FLAGS_OPTIONAL);
18     result &= registrar->parameter(
19         flush_on_tick_, "flush_on_tick", "Boolean to flush on tick",
20         "Flushes output buffer on every `tick` when true", false); // default value `false`
21     return nvidia::gxf::ToResultCode(result);
22 }

```

For pure GXF applications, our component's parameters can be specified in the following format in the YAML file:

Listing 29.6: Example parameters for `MyRecorder` component

```

1  name: my_recorder_entity
2  components:
3  - name: my_recorder_component

```

(continues on next page)

(continued from previous page)

```

4   type: MyRecorder
5   parameters:
6     receiver: receiver
7     serializer: my_serializer
8     out_directory: /home/user/out_path
9     basename: my_output_file # optional
10    # flush_on_tick: false    # optional

```

Note that all the parameters exposed at the application level are mandatory except for `flush_on_tick`, which defaults to `false`, and `basename`, whose default is handled at `initialize()` below.

Implement the Lifecycle Methods

This extension does not need to perform any heavy-weight initialization tasks, so we will concentrate on `initialize()`, `tick()`, and `deinitialize()` methods, which define the core functionality of our component. At initialization, we will create a file stream and keep track of the bytes we write on `tick()` via `binary_file_offset`.

Listing 29.7: `initialize` in `gxf_extensions/my_recorder/my_recorder.cpp`

```

24 gxf_result_t MyRecorder::initialize() {
25     // Create path by appending receiver name to directory path if basename is not provided
26     std::string path = directory_.get() + '/';
27     if (const auto& basename = basename_.try_get()) {
28         path += basename.value();
29     } else {
30         path += receiver_>name();
31     }
32
33     // Initialize index file stream as write-only
34     index_file_stream_ = nvidia::gxf::FileStream("", path +
↳ nvidia::gxf::FileStream::kIndexFileExtension);
35
36     // Initialize binary file stream as write-only
37     binary_file_stream_ = nvidia::gxf::FileStream("", path +
↳ nvidia::gxf::FileStream::kBinaryFileExtension);
38
39     // Open index file stream
40     nvidia::gxf::Expected<void> result = index_file_stream_.open();
41     if (!result) {
42         return nvidia::gxf::ToResultCode(result);
43     }
44
45     // Open binary file stream
46     result = binary_file_stream_.open();
47     if (!result) {
48         return nvidia::gxf::ToResultCode(result);
49     }
50     binary_file_offset_ = 0;
51
52     return GXF_SUCCESS;
53 }

```

When de-initializing, our component will take care of closing the file streams that were created at initialization.

Listing 29.8: deinitialize in `gxf_extensions/my_recorder/my_recorder.cpp`

```

55 gxf_result_t MyRecorder::deinitialize() {
56     // Close binary file stream
57     nvidia::gxf::Expected<void> result = binary_file_stream_.close();
58     if (!result) {
59         return nvidia::gxf::ToResultCode(result);
60     }
61
62     // Close index file stream
63     result = index_file_stream_.close();
64     if (!result) {
65         return nvidia::gxf::ToResultCode(result);
66     }
67
68     return GFX_SUCCESS;
69 }

```

In our recorder, no heavyweight initialization tasks are required, so we implement the following; however, we would use `start()` and `stop()` methods for heavyweight tasks such as memory allocation and deallocation.

Listing 29.9: start/stop in `gxf_extensions/my_recorder/my_recorder.cpp`

```

71 gxf_result_t MyRecorder::start() {
72     return GFX_SUCCESS;
73 }
74
75 gxf_result_t MyRecorder::stop() {
76     return GFX_SUCCESS;
77 }

```

Tip: For a detailed implementation of `start()` and `stop()`, and how memory management can be handled therein, please refer to the implementation of the [AJA Video source extension](#).

Finally, we write the component-specific functionality of our extension by implementing `tick()`.

Listing 29.10: tick in `gxf_extensions/my_recorder/my_recorder.cpp`

```

79 gxf_result_t MyRecorder::tick() {
80     // Receive entity
81     nvidia::gxf::Expected<nvidia::gxf::Entity> entity = receiver_->receive();
82     if (!entity) {
83         return nvidia::gxf::ToResultCode(entity);
84     }
85
86     // Write entity to binary file
87     nvidia::gxf::Expected<size_t> size = my_serializer_->serializeEntity(entity.value(), &
88     ↪ binary_file_stream_);
89     if (!size) {
90         return nvidia::gxf::ToResultCode(size);
91     }

```

(continues on next page)

(continued from previous page)

```

91
92 // Create entity index
93 nvidia::gxf::EntityIndex index;
94 index.log_time = std::chrono::system_clock::now().time_since_epoch().count();
95 index.data_size = size.value();
96 index.data_offset = binary_file_offset_;
97
98 // Write entity index to index file
99 nvidia::gxf::Expected<size_t> result = index_file_stream_.writeTrivialType(&index);
100 if (!result) {
101     return nvidia::gxf::ToResultCode(result);
102 }
103 binary_file_offset_ += size.value();
104
105 if (flush_on_tick_) {
106     // Flush binary file output stream
107     nvidia::gxf::Expected<void> result = binary_file_stream_.flush();
108     if (!result) {
109         return nvidia::gxf::ToResultCode(result);
110     }
111
112     // Flush index file output stream
113     result = index_file_stream_.flush();
114     if (!result) {
115         return nvidia::gxf::ToResultCode(result);
116     }
117 }
118
119 return GXF_SUCCESS;
120 }

```

Register the Extension as a Holoscan Component

As a final step, we must register our extension so it is recognized as a component and loaded by the application executor. For this we create a simple declaration in `my_recorder_ext.cpp` as follows.

Listing 29.11: `gxf_extensions/my_recorder/my_recorder_ext.cpp`

```

1 #include "gxf/std/extension_factory_helper.hpp"
2
3 #include "my_recorder.hpp"
4
5 GXF_EXT_FACTORY_BEGIN()
6 GXF_EXT_FACTORY_SET_INFO(0xb891cef3ce754825, 0x9dd3dcac9bbd8483, "MyRecorderExtension",
7     "My example recorder extension", "NVIDIA", "0.1.0", "LICENSE");
8 GXF_EXT_FACTORY_ADD(0x2464fabf91b34ccf, 0xb554977fa22096bd, MyRecorder,
9     nvidia::gxf::Codelet, "My example recorder codelet.");
10 GXF_EXT_FACTORY_END()

```

`GXF_EXT_FACTORY_SET_INFO` configures the extension with the following information in order:

- UUID which can be generated using `scripts/generate_extension_uuids.py` which defines the **extension**

id

- extension name
- extension description
- author
- extension version
- license text

GXF_EXT_FACTORY_ADD registers the newly built extension as a valid Codelet component with the following information in order:

- UUID which can be generated using `scripts/generate_extension_uuids.py` which defines the **component id** (this must be different from the extension id),
- fully qualified extension class,
- fully qualifies base class,
- component description

To build a shared library for our new extension which can be loaded by a Holoscan application at runtime we use a CMake file under `gxf_extensions/my_recorder/CMakeLists.txt` with the following content.

Listing 29.12: `gxf_extensions/my_recorder/CMakeLists.txt`

```

1  # Create library
2  add_library(my_recorder_lib SHARED
3      my_recorder.cpp
4      my_recorder.hpp
5  )
6  target_link_libraries(my_recorder_lib
7      PUBLIC
8          GXF::std
9          GXF::serialization
10         yaml-cpp
11 )
12
13 # Create extension
14 add_library(my_recorder SHARED
15     my_recorder_ext.cpp
16 )
17 target_link_libraries(my_recorder
18     PUBLIC my_recorder_lib
19 )
20
21 # Install GXF extension as a component 'holoscan-gxf_extensions'
22 install_gxf_extension(my_recorder) # this will also install my_recorder_lib
23 # install_gxf_extension(my_recorder_lib) # this statement is not necessary because this_
    ↳ library follows `<extension library name>_lib` convention.

```

Here, we create a library `my_recorder_lib` with the implementation of the lifecycle methods, and the extension `my_recorder` which exposes the C API necessary for the application runtime to interact with our component.

To make our extension discoverable from the project root we add the line:

```
add_subdirectory(my_recorder)
```

to the CMake file `gxf_extensions/CMakeLists.txt`.

Tip: To build our extension, we can follow the steps in the [README](#).

At this point, we have a complete extension that records data coming into its receiver queue to the specified location on the disk using the GXF-formatted binary/index files.

29.4 Creating a GXF Application

For our application, we create the directory `apps/my_recorder_app_gxf` with the application definition file `my_recorder_gxf.yaml`. The `my_recorder_gxf.yaml` application is as follows:

Listing 29.13: `apps/my_recorder_app_gxf/my_recorder_gxf.yaml`

```

1 %YAML 1.2
2 ---
3 name: replayer
4 components:
5   - name: output
6     type: nvidia::gxf::DoubleBufferTransmitter
7   - name: allocator
8     type: nvidia::gxf::UnboundedAllocator
9   - name: component_serializer
10    type: nvidia::gxf::StdComponentSerializer
11    parameters:
12      allocator: allocator
13   - name: entity_serializer
14     type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
15     ↪ nvidia::gxf::EntitySerializer
16     parameters:
17       component_serializers: [component_serializer]
18   - type: nvidia::holoscan::stream_playback::VideoStreamReplayer
19     parameters:
20       transmitter: output
21       entity_serializer: entity_serializer
22       boolean_scheduling_term: boolean_scheduling
23       directory: "/workspace/data/racerox"
24       basename: "racerox"
25       frame_rate: 0 # as specified in timestamps
26       repeat: false # default: false
27       realtime: true # default: true
28       count: 0 # default: 0 (no frame count restriction)
29   - name: boolean_scheduling
30     type: nvidia::gxf::BooleanSchedulingTerm
31   - type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
32     parameters:
33       transmitter: output
34       min_size: 1

```

(continues on next page)

(continued from previous page)

```

34 ---
35 name: recorder
36 components:
37   - name: input
38     type: nvidia::gxf::DoubleBufferReceiver
39   - name: allocator
40     type: nvidia::gxf::UnboundedAllocator
41   - name: component_serializer
42     type: nvidia::gxf::StdComponentSerializer
43     parameters:
44       allocator: allocator
45   - name: entity_serializer
46     type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
47     ↪ nvidia::gxf::EntitySerializer
48     parameters:
49       component_serializers: [component_serializer]
50   - type: MyRecorder
51     parameters:
52       receiver: input
53       serializer: entity_serializer
54       out_directory: "/tmp"
55       basename: "tensor_out"
56   - type: nvidia::gxf::MessageAvailableSchedulingTerm
57     parameters:
58       receiver: input
59       min_size: 1
60 ---
61 components:
62   - name: input_connection
63     type: nvidia::gxf::Connection
64     parameters:
65       source: replayer/output
66       target: recorder/input
67 ---
68 name: scheduler
69 components:
70   - name: clock
71     type: nvidia::gxf::RealtimeClock
72   - name: greedy_scheduler
73     type: nvidia::gxf::GreedyScheduler
74     parameters:
75       clock: clock

```

Above:

- The replayer reads data from `/workspace/data/racerx/racerx.gxf_[index|entities]` files, deserializes the binary data to a `nvidia::gxf::Tensor` using `VideoStreamSerializer`, and puts the data on an output message in the `replayer/output` transmitter queue.
- The `input_connection` component connects the `replayer/output` transmitter queue to the `recorder/input` receiver queue.
- The recorder reads the data in the input receiver queue, uses `StdEntitySerializer` to convert the received `nvidia::gxf::Tensor` to a binary stream, and outputs to the `/tmp/tensor_out.gxf_[index|entities]`

location specified in the parameters.

- The scheduler component, while not explicitly connected to the application-specific entities, performs the orchestration of the components discussed in the *Data Flow and Triggering Rules*.

Note the use of the `component_serializer` in our newly built recorder. This component is declared separately in the following entity:

```
- name: entity_serializer
  type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
  ↪nvidia::gxf::EntitySerializer
  parameters:
    component_serializers: [component_serializer]
```

This is then passed into `MyRecorder` via the `serializer` parameter, which we exposed in the *extension development* section (*Declare the Parameters to Expose at the Application Level*).

```
- type: MyRecorder
  parameters:
    receiver: input
    serializer: entity_serializer
    directory: "/tmp"
    basename: "tensor_out"
```

For our app to be able to load (and also compile where necessary) the extensions required at runtime, we need to declare a CMake file `apps/my_recorder_app_gxf/CMakeLists.txt` as follows.

Listing 29.14: `apps/my_recorder_app_gxf/CMakeLists.txt`

```
1 create_gxe_application(
2   NAME my_recorder_gxf
3   YAML my_recorder_gxf.yaml
4   EXTENSIONS
5     GXF::std
6     GXF::cuda
7     GXF::multimedia
8     GXF::serialization
9     my_recorder
10    stream_playback
11 )
12
13 # Download the associated dataset if needed
14 if(HOLOSCAN_DOWNLOAD_DATASETS)
15   add_dependencies(my_recorder_gxf racerx_data)
16 endif()
```

In the declaration of `create_gxe_application` we list:

- `my_recorder` component declared in the CMake file of the *extension development* section under the `EXTENSIONS` argument
- the existing `stream_playback` Holoscan extension which reads data from disk

To make our newly built application discoverable by the build, in the root of the repository, we add the following line to `apps/CMakeLists.txt`:


```
add_subdirectory(my_recorder_app_gxf)
```

We now have a minimal working application to test the integration of our newly built MyRecorder extension.

29.5 Running the GXF Recorder Application

To run our application in a local development container:

1. Follow the instructions under the [Using a Development Container](#) section steps 1-5 (try clearing the CMake cache by removing the build folder before compiling).

You can execute the following commands to build:

```
./run build
# ./run clear_cache # if you want to clear build/install/cache folders
```

2. Our test application can now be run in the development container using the command:

```
./apps/my_recorder_app_gxf/my_recorder_gxf
```

from inside the development container.

(You can execute `./run launch` to run the development container.)

```
@LINUX:/workspace/holoscan-sdk/build$ ./apps/my_recorder_app_gxf/my_recorder_gxf
2022-08-24 04:46:47.333 INFO gxf/gxe/gxe.cpp@230: Creating context
2022-08-24 04:46:47.339 INFO gxf/gxe/gxe.cpp@107: Loading app: 'apps/my_recorder_
↳ app_gxf/my_recorder_gxf.yaml'
2022-08-24 04:46:47.339 INFO gxf/std/yaml_file_loader.cpp@117: Loading GXF_
↳ entities from YAML file 'apps/my_recorder_app_gxf/my_recorder_gxf.yaml'...
2022-08-24 04:46:47.340 INFO gxf/gxe/gxe.cpp@291: Initializing...
2022-08-24 04:46:47.437 INFO gxf/gxe/gxe.cpp@298: Running...
2022-08-24 04:46:47.437 INFO gxf/std/greedy_scheduler.cpp@170: Scheduling 2_
↳ entities
2022-08-24 04:47:14.829 INFO /workspace/holoscan-sdk/gxf_extensions/stream_
↳ playback/video_stream_replayer.cpp@144: Reach end of file or playback count_
↳ reaches to the limit. Stop ticking.
2022-08-24 04:47:14.829 INFO gxf/std/greedy_scheduler.cpp@329: Scheduler stopped:_
↳ Some entities are waiting for execution, but there are no periodic or async_
↳ entities to get out of the deadlock.
2022-08-24 04:47:14.829 INFO gxf/std/greedy_scheduler.cpp@353: Scheduler finished.
2022-08-24 04:47:14.829 INFO gxf/gxe/gxe.cpp@320: Deinitializing...
2022-08-24 04:47:14.863 INFO gxf/gxe/gxe.cpp@327: Destroying context
2022-08-24 04:47:14.863 INFO gxf/gxe/gxe.cpp@333: Context destroyed.
```

A successful run (it takes about 30 secs) will result in output files (`tensor_out.gxf_index` and `tensor_out.gxf_entities` in `/tmp`) that match the original input files (`racerox.gxf_index` and `racerox.gxf_entities` under `data/racerox`) exactly.

```
@LINUX:/workspace/holoscan-sdk/build$ ls -al /tmp/
total 821384
drwxrwxrwt 1 root root      4096 Aug 24 04:37 .
drwxr-xr-x 1 root root      4096 Aug 24 04:36 ..
```

(continues on next page)

(continued from previous page)

```
drwxrwxrwt 2 root root      4096 Aug 11 21:42 .X11-unix
-rw-r--r-- 1 1000 1000    729309 Aug 24 04:47 gxf_log
-rw-r--r-- 1 1000 1000 840054484 Aug 24 04:47 tensor_out.gxf_entities
-rw-r--r-- 1 1000 1000    16392 Aug 24 04:47 tensor_out.gxf_index
```

```
@LINUX:/workspace/holoscan-sdk/build$ ls -al ../data/racerx
```

```
total 839116
drwxr-xr-x 2 1000 1000      4096 Aug 24 02:08 .
drwxr-xr-x 4 1000 1000      4096 Aug 24 02:07 ..
-rw-r--r-- 1 1000 1000 19164125 Jun 17 16:31 racerx-medium.mp4
-rw-r--r-- 1 1000 1000 840054484 Jun 17 16:31 racerx.gxf_entities
-rw-r--r-- 1 1000 1000    16392 Jun 17 16:31 racerx.gxf_index
```

USING HOLOSCAN OPERATORS IN GXF APPLICATIONS

For users who are familiar with the GXF development ecosystem (used in Holoscan SDK 0.2), we provide an export feature to leverage native Holoscan operators as GXF codelets to execute in GXF applications and GraphComposer.

We demonstrate how to wrap a native C++ Holoscan operator as a GXF codelet in the [wrap_operator_as_gxf_extension](#) example on [GitHub](#), as described below.

30.1 1. Creating compatible Holoscan Operators

Note: This section assumes you are already familiar with *how to create a native C++ operator*.

To be compatible with GXF codelets, inputs and outputs specified in `Operator::setup(OperatorSpec& spec)` must be of type `holoscan::gxf::Entity`, as shown in the [PingTxNativeOp](#) and the [PingRxNativeOp](#) implementations of this example, in contrast to the [PingTxOp](#) and [PingRxOp](#) built-in operators of the SDK.

For more details regarding the use of `holoscan::gxf::Entity`, follow the documentation on *Interoperability between GXF and native C++ operators*.

30.2 2. Creating the GXF extension that wraps the operator

To wrap the native operator as a GXF codelet in a GXF extension, we provide the `CMake wrap_operator_as_gxf_extension` function in the SDK. An example of how it wraps `PingTxNativeOp` and `PingRxNativeOp` can be found [here](#).

- It leverages the CMake target names of the operators defined in their respective `CMakeLists.txt` (`ping_tx_native_op`, `ping_rx_native_op`)
- The function parameters are documented at the top of the [WrapOperatorAsGXFExtension.cmake](#) file (ignore implementation below).

Warning:

- A unique GXF extension is currently needed for each native operator to export (operators cannot be bundled in a single extension at this time).
- Wrapping other GXF entities than operators (as codelets) is not currently supported.

30.3 3. Using your wrapped operator in a GXF application

Note: This section assumes you are familiar with [how to create a GXF application](#).

As shown in the `gxf_app/CMakeLists.txt` [here](#), you need to list the following extensions in `create_gxe_application()` to use your wrapped codelets:

- `GXF::std`
- `gxf_holoscan_wrapper`
- The name of the CMake target for the created extension, defined by the `EXTENSION_TARGET_NAME` argument passed to `wrap_operator_as_gxf_extension` in the previous section.

The codelet class name (defined by the `CODELET_NAMESPACE::CODELET_NAME` arguments passed to `wrap_operator_as_gxf_extension` in the previous section) can then be used as a component `type` in a GXF application node, as shown in the [YAML app definition](#) of the example, connecting the two ping operators.

GXF USER GUIDE

31.1 Graph Specification

Graph Specification is a format to describe high-performance AI applications in a modular and extensible way. It allows writing applications in a standard format and sharing components across multiple applications without code modification. Graph Specification is based on entity-composition pattern. Every object in graph is represented with entity (aka Node) and components. You can implement custom components, which can be added to entity to achieve the required functionality.

31.1.1 Concepts

The graph contains nodes which follow an entity-component design pattern implementing the “composition over inheritance” paradigm. A node itself is just a lightweight object which owns components. Components define how a node interacts with the rest of the applications. For example, nodes be connected to pass data between each other. A special component, called compute component, is used to execute the code based on certain rules. Typically a compute component would receive data, execute some computation, and publish data.

Graph

A graph is a data-driven representation of an AI application. Implementing an application by using programming code to create and link objects results in a monolithic and hard to maintain program. Instead, a graph object is used to structure an application. The graph can be created using specialized tools and it can be analyzed to identify potential problems or performance bottlenecks. The graph is loaded by the graph runtime to be executed.

The functional blocks of a graph are defined by the set of nodes which the graph owns. Nodes can be queried via the graph using certain query functions. For example, it is possible to search for a node by its name.

SubGraph

A subgraph is a graph with additional node for interfaces. It points to the components which are accessible outside this graph. In order to use a subgraph in an existing graph or subgraph, you need to create an entity where a component of the type `nvidia::gxf::Subgraph` is contained. Inside the Subgraph component, a corresponding subgraph can be loaded from the `yaml` file indicated by `location` property and instantiated in the parent graph.

The system makes the components from interface available to the parent graph when a sub-graph is loaded in the parent graph. It allows users to link subgraphs in parent with defined interface.

A subgraph interface can be defined as follows:

```
---
interfaces:
  - name: iname # the name of the interface for the access from the parent graph
    target: n_entity/n_component # the true component in the subgraph that is represented
    ↪by the interface
```

Node

Graph Specification uses an entity-component design principle for nodes. This means that a node is a lightweight object whose main purpose is to own components. A node is a composition of components. Every component is in exactly one node. In order to customize a node, you do not derive from node as a base class, but instead compose objects out of components. Components can be used to provide a rich set of functionality to a node and thus to an application.

Components

Components are the main functional blocks of an application. Graph runtime provides a couple of components which implement features like properties, code execution, rules and message passing. It also allows you to extend the runtime by injecting your own custom components with custom features to fit a specific use case.

The most common component is a codelet or compute component which is used for data processing and code execution. To implement a custom codelet, you'll need to implement a certain set of functions like *start* and *stop*. A special system – the *scheduler* – will call these functions at the specified time. Typical examples of triggering code execution are: receiving a new message from another node, or performing work on a regular schedule based on a time trigger.

Edges

Nodes can receive data from other nodes by connecting them with an edge. This essential feature allows a graph to represent a compute pipeline or a complicated AI application. An input to a node is called sink while an output is called source. There can be zero, one, or multiple inputs and outputs. A source can be connected to multiple sinks, and a sink can be connected to multiple sources.

Extension

An extension is a compiled shared library of a logical group of component type definitions and their implementations along with any other asset files that are required for execution of the components. Some examples of asset files are model files, shared libraries that the extension library links to and hence required to run, header and development files that enable development of additional components, and extensions that use components from the extension.

An extension library is a runtime loadable module compiled with component information in a standard format that allows the graph runtime to load the extension and retrieve further information from it to:

- Allow the runtime to create components using the component types in the extension.
- Query information regarding the component types in the extension:
 - The component type name
 - The base type of the component
 - A string description of the component
 - Information of parameters of the component – parameter name, type, description, etc.
- Query information regarding the extension itself – Name of the extension, version, license, author, and a string description of the extension.

The section :doc: *GraphComposer_Dev_Workflow* talks more about this with a focus on developing extensions and components.

31.1.2 Graph File Format

The graph file stores list of entities. Each entity has a unique name and list of components. Each component has a name, a type, and properties. Properties are stored as key-value pairs.

```
%YAML 1.2
---
name: source
components:
- name: signal
  type: sample::test::ping
- type: nvidia::gxf::CountSchedulingTerm
  parameters:
    count: 10
---
components:
- type: nvidia::gxf::GreedyScheduler
  parameters:
    realtime: false
    max_duration_ms: 1000000
```

31.2 Graph Execution Engine

Graph Execution Engine is used to execute AI application graphs. It accepts multiple graph files as input, and all graphs are executed in same process context. It also needs manifest files as input which includes list of extensions to load. It must list all extensions required for the graph.

```
gxe --help
Flags from gxf/gxe/gxe.cpp:
  -app (GXF app file to execute. Multiple files can be comma-separated)
    type: string default: ""
  -graph_directory (Path to a directory for searching graph files.)
    type: string default: ""
  -log_file_path (Path to a file for logging.) type: string default: ""
  -manifest (GXF manifest file with extensions. Multiple files can be
    comma-separated) type: string default: ""
  -severity (Set log severity levels: 0=None, 1=Error, 2=Warning, 3=Info,
    4=Debug. Default: Info) type: int32 default: 3
```

31.3 Graph Specification TimeStamping

31.3.1 Message Passing

Once the graph is built, the communication between various entities occur by passing around messages (messages are entities themselves). Specifically, one component/codelet can publish a message entity and another can receive it. When publishing, a message should always have an associated `Timestamp` component with the name `"timestamp"`. A `Timestamp` component contains two different time values (See the `gfx/std/timestamp.hpp` header file for more information.):

1. **acqtime** - This is the time when the message entity is acquired; for instance, this would generally be the driver time of the camera when it captures an image. You must provide this timestamp if you are publishing a message in a codelet.
2. **pubtime** - This is the time when the message entity is published by a node in the graph. This will automatically get updated using the clock of the scheduler.

In a codelet, when publishing message entities using a Transmitter (tx), there are two ways to add the required Timestamp:

1. `tx.publish(Entity message)`: You can manually add a component of type `Timestamp` with the name “timestamp” and set the `acqtime`. The `pubtime` in this case should be set to `0`. The message is published using the `publish(Entity message)`. **This will be deprecated in the next release.**
2. `tx.publish(Entity message, int64_t acqtime)`: You can simply call `publish(Entity message, int64_t acqtime)` with the `acqtime`. `Timestamp` will be added automatically.

31.4 The GXF Scheduler

The execution of entities in a graph is governed by the scheduler and the scheduling terms associated with every entity. A scheduler is a component responsible for orchestrating the execution of all the entities defined in a graph. A scheduler typically keeps track of the graph entities and their current execution states and passes them on to a `nvidia::gxf::EntityExecutor` component when ready for execution. The following diagram depicts the flow for an entity execution.

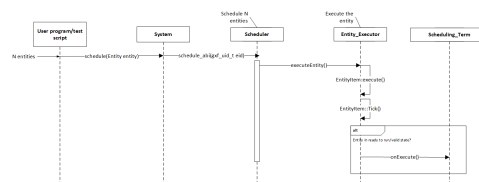


Figure: Entity execution sequence

As shown in the sequence diagram, the schedulers begin executing the graph entities via the `nvidia::gxf::System::runAsync(_abi)` interface, and continue this process until it meets the certain ending criteria. A single entity can have multiple codelets. These codelets are executed in the same order in which they were defined in the entity. A failure in execution of any single codelet stops the execution of all the entities. Entities are naturally unscheduled from execution when any one of their scheduling term reaches the `NEVER` state.

Scheduling terms are components used to define the execution readiness of an entity. An entity can have multiple scheduling terms associated with it and each scheduling term represents the state of an entity using `SchedulingCondition`.

The table below shows various states of `nvidia::gfx::SchedulingConditionType` described using `nvidia::gfx::SchedulingCondition`.

SchedulingConditionType	Description
NEVER	Entity will never execute again
READY	Entity is ready for execution
WAIT	Entity may execute in the future
WAIT_TIME	Entity will be ready for execution after specified duration
WAIT_EVENT	Entity is waiting on an asynchronous event with unknown time interval

Schedulers define deadlock as a condition when there are no entities which are in `READY`, `WAIT_TIME` or `WAIT_EVENT` states, which guarantee execution at a future point in time. This implies all the entities are in the `WAIT` state, for which the scheduler does not know if they ever will reach the `READY` state in the future. The scheduler can be configured to stop when it reaches such a state using the `stop_on_deadlock` parameter, else the entities are polled to check if any of them have reached the `READY` state. The `max_duration` configuration parameter can be used to stop execution of all entities, regardless of their state, after a specified amount of time has elapsed.

There are two types of schedulers currently supported by GXF

1. Greedy Scheduler
2. Multithread Scheduler

31.4.1 Greedy Scheduler

This is a basic single threaded scheduler which tests scheduling term greedily. It is great for simple use cases and predictable execution, but may incur a large overhead of scheduling term execution, making it unsuitable for large applications. The scheduler requires a clock to keep track of time. Based on the choice of clock the scheduler will execute differently. If a `Realtime` clock is used, the scheduler will execute in real-time. This means pausing execution – sleeping the thread – until periodic scheduling terms are due again. If a `ManualClock` is used, scheduling will happen “time-compressed.” This means the flow of time is altered to execute codelets in immediate succession.

The `GreedyScheduler` maintains a running count of entities which are in `READY`, `WAIT_TIME` and `WAIT_EVENT` states. The following activity diagram depicts the gist of the decision making for scheduling an entity by the Greedy Scheduler:

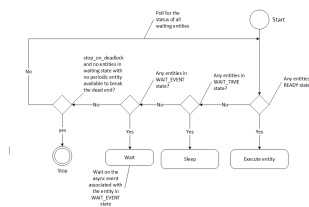


Figure: Greedy Scheduler Activity Diagram

Greedy Scheduler Configuration

The Greedy Scheduler takes in the following parameters from the configuration file:

Parameter name	Description
clock	The clock used by the scheduler to define the flow of time. Typical choices are <code>RealtimeClock</code> or <code>ManualClock</code>
max_duration_ms	The maximum duration for which the scheduler will execute (in ms). If not specified, the scheduler will run until all work is done. If periodic terms are present this means the application will run indefinitely
stop_on_deadlock	If <code>stop_on_deadlock</code> is disabled, the <code>GreedyScheduler</code> constantly polls for the status of all the waiting entities to check if any of them are ready for execution.

Example usage – The following code snippet configures a Greedy Scheduler with a ManualClock option specified.

```

name: scheduler
components:
- type: nvidia::gxf::GreedyScheduler
  parameters:
    max_duration_ms: 3000
    clock: misc/clock
    stop_on_deadlock: true
---
name: misc
components:
- name: clock
  type: nvidia::gxf::ManualClock

```

31.4.2 Multithread Scheduler

The MultiThread Scheduler is more suitable for large applications with complex execution patterns. The scheduler consists of a dispatcher thread which checks the status of an entity, and dispatches it to a thread pool of worker threads responsible for executing them. Worker threads enqueue the entity back on to the dispatch queue upon completion of execution. The number of worker threads can be configured using the `worker_thread_number` parameter. The MultiThread Scheduler also manages a dedicated queue and thread to handle asynchronous events. The following activity diagram demonstrates the gist of the multithread scheduler implementation.

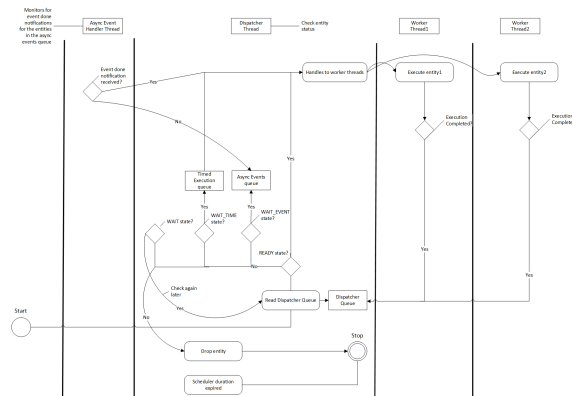


Figure: MultiThread Scheduler Activity Diagram

As depicted in the diagram, when an entity reaches the `WAIT_EVENT` state, it's moved to a queue where they wait to receive event done notification. The asynchronous event handler thread is responsible for moving entities to the dispatcher upon receiving event done notification. The dispatcher thread also maintains a running count of the number of entities in `READY`, `WAIT_EVENT` and `WAIT_TIME` states, and uses these statistics to check if the scheduler has reached a deadlock. The scheduler also needs a clock component to keep track of time and it is configured using the `clock` parameter.

The MultiThread Scheduler is more resource efficient compared to the Greedy Scheduler, and does not incur any additional overhead for constantly polling the states of scheduling terms. The `check_recession_period_ms` parameter can be used to configure the time interval the scheduler must wait to poll the state of entities which are in the `WAIT` state.

Multithread Scheduler Configuration

The Multithread Scheduler takes in the following parameters from the configuration file:

Parameter name	Description
clock	The clock used by the scheduler to define the flow of time. Typical choices are RealtimeClock or ManualClock.
max_duration_ms	The maximum duration for which the scheduler will execute (in ms). If not specified, the scheduler will run until all work is done. If periodic terms are present this means the application will run indefinitely.
check_recess_period_ms	Duration to sleep before checking the condition of an entity again [ms]. This is the maximum duration for which the scheduler would wait when an entity is not yet ready to run.
stop_on_deadlock	If enabled the scheduler will stop when all entities are in a waiting state, but no periodic entity exists to break the dead end. Should be disabled when scheduling conditions can be changed by external actors, for example by clearing queues manually.
worker_thread_number	Number of threads.

Example usage – The following code snippet configures a Multithread Scheduler with the number of worked threads and max duration specified:

```
name: scheduler
components:
- type: nvidia::gfx::MultiThreadScheduler
  parameters:
    max_duration_ms: 5000
    clock: misc/clock
    worker_thread_number: 5
    check_recession_period_ms: 3
    stop_on_deadlock: false
---
name: misc
components:
- name: clock
  type: nvidia::gfx::RealtimeClock
```

31.4.3 Epoch Scheduler

The Epoch Scheduler is used for running loads in externally managed threads. Each run is called an Epoch. The scheduler goes over all entities that are known to be active and executes them one by one. If the epoch budget is provided (in ms), it would keep running all codelets until the budget is consumed or no codelet is ready. It might run over budget since it guarantees to cover all codelets in an epoch. In case the budget is not provided, it would go over all the codelets once and execute them only once.

The Epoch Scheduler takes in the following parameters from the configuration file:

Parameter name	Description
clock	The clock used by the scheduler to define the flow of time. Typical choice is a RealtimeClock.

Example usage – The following code snippet configures an Epoch Scheduler:

```
name: scheduler
components:
- name: clock
  type: nvidia::gxf::RealtimeClock
- name: epoch
  type: nvidia::gxf::EpochScheduler
parameters:
  clock: clock
```

Note that the Epoch Scheduler is intended to run from an external thread. The `runEpoch(float budget_ms);` can be used to set the `budget_ms` and run the scheduler from the external thread. If the specified budget is not positive, all the nodes are executed once.

31.4.4 SchedulingTerms

A `SchedulingTerm` defines a specific condition that is used by an entity to let the scheduler know when it's ready for execution. There are various scheduling terms currently supported by GXF.

PeriodicSchedulingTerm

An entity associated with `nvidia::gxf::PeriodicSchedulingTerm` is ready for execution after periodic time intervals specified using its `recess_period` parameter. The `PeriodicSchedulingTerm` can either be in `READY` or `WAIT_TIME` state.

Example usage:

```
- name: scheduling_term
  type: nvidia::gxf::PeriodicSchedulingTerm
  parameters:
    recess_period: 500000000
```

CountSchedulingTerm

An entity associated with `nvidia::gxf::CountSchedulingTerm` is executed for a specific number of times specified using its `count` parameter. The `CountSchedulingTerm` can either be in `READY` or `NEVER` state. The scheduling term reaches the `NEVER` state when the entity has been executed `count` number of times.

Example usage:

```
- name: scheduling_term
  type: nvidia::gxf::CountSchedulingTerm
  parameters:
    count: 42
```

MessageAvailableSchedulingTerm

An entity associated with `nvidia::gxf::MessageAvailableSchedulingTerm` is executed when the associated receiver queue has at least a certain number of elements. The receiver is specified using the `receiver` parameter of the scheduling term. The minimum number of messages that permits the execution of the entity is specified by `min_size`. An optional parameter for this scheduling term is `front_stage_max_size`, the maximum front stage message count. If this parameter is set, the scheduling term will only allow execution if the number of messages in the queue does not exceed this count. It can be used for codelets which do not consume all messages from the queue.

In the example shown below, the minimum size of the queue is configured to be 4. This means the entity will not be executed until there are at least 4 messages in the queue.

```
- type: nvidia::gxf::MessageAvailableSchedulingTerm
  parameters:
    receiver: tensors
    min_size: 4
```

MultiMessageAvailableSchedulingTerm

An entity associated with `nvidia::gxf::MultiMessageAvailableSchedulingTerm` is executed when a list of provided input receivers combined have at least a given number of messages. The `receivers` parameter is used to specify a list of the input channels/receivers. The minimum number of messages needed to permit the entity execution is set by `min_size` parameter.

Consider the example shown below. The associated entity will be executed when the number of messages combined for all the three receivers is at least the `min_size`; i.e., 5.

```
- name: input_1
  type: nvidia::gxf::test::MockReceiver
  parameters:
    max_capacity: 10
- name: input_2
  type: nvidia::gxf::test::MockReceiver
  parameters:
    max_capacity: 10
- name: input_3
  type: nvidia::gxf::test::MockReceiver
  parameters:
    max_capacity: 10
- type: nvidia::gxf::MultiMessageAvailableSchedulingTerm
  parameters:
    receivers: [input_1, input_2, input_3]
    min_size: 5
```

BooleanSchedulingTerm

An entity associated with `nvidia::gxf::BooleanSchedulingTerm` is executed when its internal state is set to tick. The parameter `enable_tick` is used to control the entity execution. The scheduling term also has two APIs `enable_tick()` and `disable_tick()` to toggle its internal state. The entity execution can be controlled by calling these APIs. If `enable_tick` is set to false, the entity is not executed (Scheduling condition is set to `NEVER`). If `enable_tick` is set to true, the entity will be executed (Scheduling condition is set to `READY`). Entities can toggle the state of the scheduling term by maintaining a handle to it.

Example usage:

```
- type: nvidia::gxf::BooleanSchedulingTerm
  parameters:
    enable_tick: true
```

AsynchronousSchedulingTerm

`AsynchronousSchedulingTerm` is primarily associated with entities which are working with asynchronous events happening outside of their regular execution performed by the scheduler. Since these events are non-periodic in nature, `AsynchronousSchedulingTerm` prevents the scheduler from polling the entity for its status regularly and reduces CPU utilization. `AsynchronousSchedulingTerm` can either be in `READY`, `WAIT`, `WAIT_EVENT` or `NEVER` states based on asynchronous event it's waiting on.

The state of an asynchronous event is described using `nvidia::gxf::AsynchronousEventState` and is updated using the `setEventState` API.

AsynchronousEventState	Description
READY	Init state, first tick is pending
WAIT	Request to async service yet to be sent, nothing to do but wait
EVENT_WAITING	Request sent to an async service, pending event done notification
EVENT_DONE	Event done notification received, entity ready to be ticked
EVENT_NEVER	Entity does not want to be ticked again, end of execution

Entities associated with this scheduling term most likely have an asynchronous thread which can update the state of the scheduling term outside of its regular execution cycle performed by the GXF scheduler. When the scheduling term is in `WAIT` state, the scheduler regularly polls for the state of the entity. When the scheduling term is in `EVENT_WAITING` state, schedulers will not check the status of the entity again until they receive an event notification which can be triggered using the `GxfEntityEventNotify` API. Setting the state of the scheduling term to `EVENT_DONE` automatically sends this notification to the scheduler. Entities can use the `EVENT_NEVER` state to indicate the end of its execution cycle.

Example usage:

```
- name: async_scheduling_term
  type: nvidia::gxf::AsynchronousSchedulingTerm
```

DownstreamReceptiveSchedulingTerm

This scheduling term specifies that an entity shall be executed if the receiver for a given transmitter can accept new messages.

Example usage:

```
- name: downstream_st
  type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
  parameters:
    transmitter: output
    min_size: 1
```

TargetTimeSchedulingTerm

This scheduling term permits execution at a user-specified timestamp. The timestamp is specified on the clock provided.

Example usage:

```
- name: target_st
  type: nvidia::gxf::TargetTimeSchedulingTerm
  parameters:
    clock: clock/manual_clock
```

ExpiringMessageAvailableSchedulingTerm

This scheduling waits for a specified number of messages in the receiver. The entity is executed when the first message received in the queue is expiring or when there are enough messages in the queue. The `receiver` parameter is used to set the receiver to watch on. The parameters `max_batch_size` and `max_delay_ns` dictate the maximum number of messages to be batched together and the maximum delay from first message to wait before executing the entity respectively.

In the example shown below, the associated entity will be executed when the number of messages in the queue is greater than `max_batch_size` (i.e., 5), or when the delay from the first message to current time is greater than `max_delay_ns` (i.e., 100000000).

```
- name: target_st
  type: nvidia::gxf::ExpiringMessageAvailableSchedulingTerm
  parameters:
    receiver: signal
    max_batch_size: 5
    max_delay_ns: 100000000
    clock: misc/clock
```

AND Combined

An entity can be associated with multiple scheduling terms which define its execution behavior. Scheduling terms are AND combined to describe the current state of an entity. For an entity to be executed by the scheduler, all the scheduling terms must be in READY state and conversely, the entity is unscheduled from execution whenever any one of the scheduling term reaches NEVER state. The priority of various states during AND combine follows the order NEVER, WAIT_EVENT, WAIT, WAIT_TIME, and READY.

Example usage:

```
components:
- name: integers
  type: nvidia::gxf::DoubleBufferTransmitter
- name: fibonacci
  type: nvidia::gxf::DoubleBufferTransmitter
- type: nvidia::gxf::CountSchedulingTerm
  parameters:
    count: 100
- type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
  parameters:
    transmitter: integers
    min_size: 1
```

BTSchedulingTerm

A BT (Behavior Tree) scheduling term is used to schedule a behavior tree entity itself and its child entities (if any) in a Behavior tree.

Example usage:

```
name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_codelet
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: [ child1/child1_st ]
    s_term: root_st
    controller: root_controller
```


31.5 Behavior Trees

Behavior tree codelets are one of the mechanisms to control the flow of tasks in GXF. They follow the same general behavior as classical behavior trees, with some useful additions for robotics applications. This document gives an overview of the general concept, the available behavior tree node types, and some examples of how to use them individually or in conjunction with each other.

31.5.1 General Concept

Behavior trees consist of n-ary trees of entities that can have zero or more children. The conditional execution of parent entity is based on the status of execution of the children. A behavior tree is graphically represented as a directed tree in which the nodes are classified as root, control flow nodes, or execution nodes (tasks). For each pair of connected nodes, the outgoing node is called parent and the incoming node is called child.

The execution of a behavior tree starts from the root which sends ticks with a certain frequency to its child. When the execution of a node in the behavior tree is allowed, it returns to the parent a status **running** if its execution has not finished yet, **success** if it has achieved its goal, or **failure** otherwise. The behavior tree also uses a controller component for controlling the entity's termination policy and the execution status. One of the controller behaviors currently implemented for Behavior Tree is `EntityCountFailureRepeatController`, which repeats the entity on failure up to `repeat_count` times before deactivating it.

GXF supports several behavior tree codelets which are explained in the following section.

31.5.2 Behavior Tree Codelets

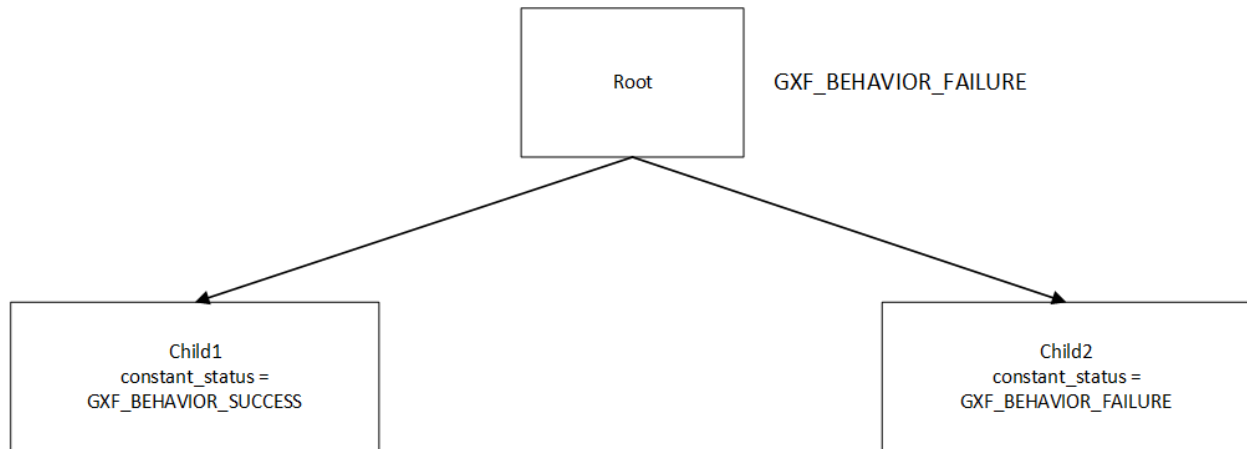
Each behavior tree codelet can have a set of parameters defining how it should behave. Note that in all the examples given below, the naming convention for configuring the children parameter for root codelets is `[child_codelet_name\child_codelet_scheduling_term]`.

Constant Behavior

After each tick period, switches its own status to the configured desired constant status.

Parameter	Description
<code>s_term</code>	scheduling term used for scheduling the entity itself
<code>constant_status</code>	The desired status to switch to during each tick time.

An example diagram depicting Constant behavior used in conjunction with a Sequence behavior defined for root entity is shown below:



Here, the child1 is configured to return a constant status of success (`GXF_BEHAVIOR_SUCCESS`) and child2 returns failure (`GXF_BEHAVIOR_FAILURE`), resulting into the root node (configured to exhibit sequence behavior) returning `GXF_BEHAVIOR_FAILURE`.

The controller for each child can be configured to repeat the execution on failure. A code snippet of configuring the example described is shown below.

```

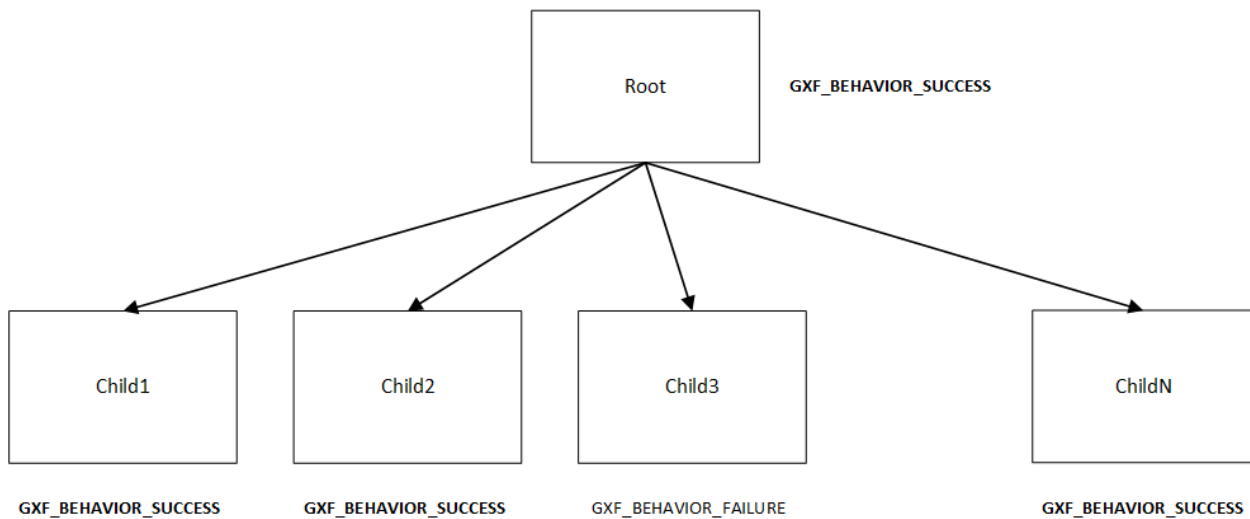
name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_codelet
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: [ child1/child1_st, child2/child2_st ]
    s_term: root_st
---
name: child2
components:
- name: child2_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 3
    return_behavior_running_if_failure_repeat: true
- name: child2_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: child2_codelet
  type: nvidia::gxf::ConstantBehavior
  parameters:
    s_term: child2_st
    constant_status: 1
  
```

Parallel Behavior

Runs its child nodes in parallel. By default, succeeds when all child nodes succeed, and fails when all child nodes fail. This behavior can be customized using the parameters below.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
children	Child entities
success_threshold	Number of successful children required for success. A value of -1 means all children must succeed for this node to succeed.
failure_threshold	Number of failed children required for failure. A value of -1 means all children must fail for this node to fail.

The diagram below shows a graphical representation of a parallel behavior configured with `failure_threshold` configured as -1. Hence, the root node returns `GXF_BEHAVIOR_SUCCESS` even if one child returns a failure status.



A code snippet to configure the example described is shown below.

```

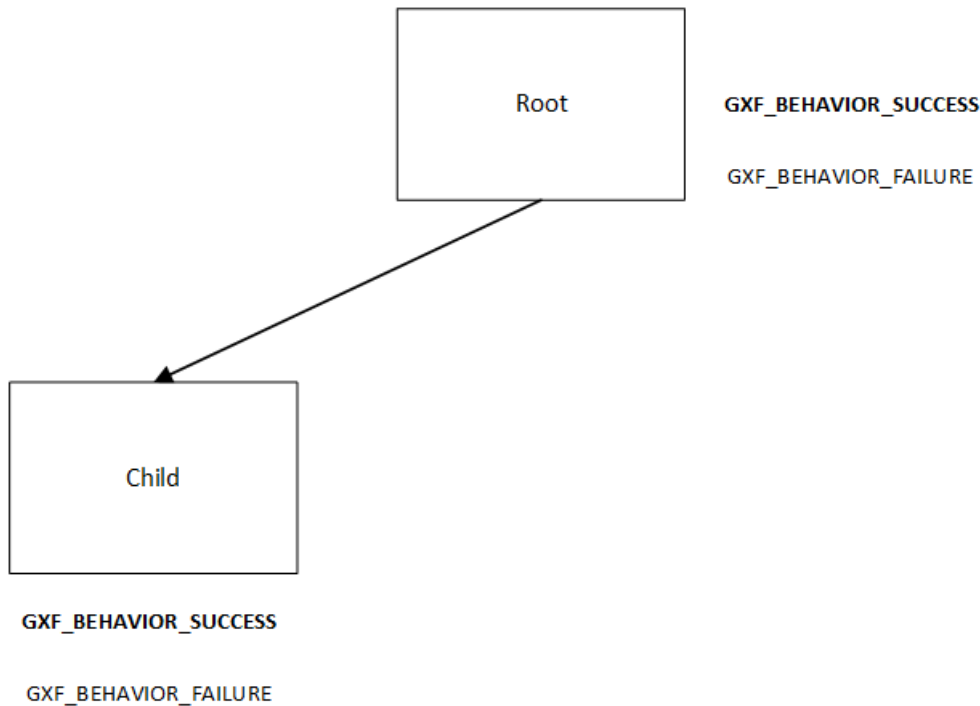
name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_codelet
  type: nvidia::gxf::ParallelBehavior
  parameters:
    children: [ child1/child1_st, child2/child2_st ]
    s_term: root_st
    success_threshold: 1
    failure_threshold: -1
  
```

Repeat Behavior

Repeats its only child entity. By default, won't repeat when the child entity fails. This can be customized using the parameters below.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
repeat_after_failure	Denotes whether to repeat the child after it has failed.

The diagram below shows a graphical representation of a repeat behavior. The root entity can be configured to repeat the only child to repeat after failure. It succeeds when the child entity succeeds.



A code snippet to configure a repeat behavior is as shown below -

```

name: repeat_knock
components:
- name: repeat_knock_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: repeat_knock_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: repeat_codelet
  type: nvidia::gxf::RepeatBehavior
  parameters:
    s_term: repeat_knock_st
    children: [ knock_on_door/knock_on_door_st ]
    repeat_after_failure: true
  
```

(continues on next page)

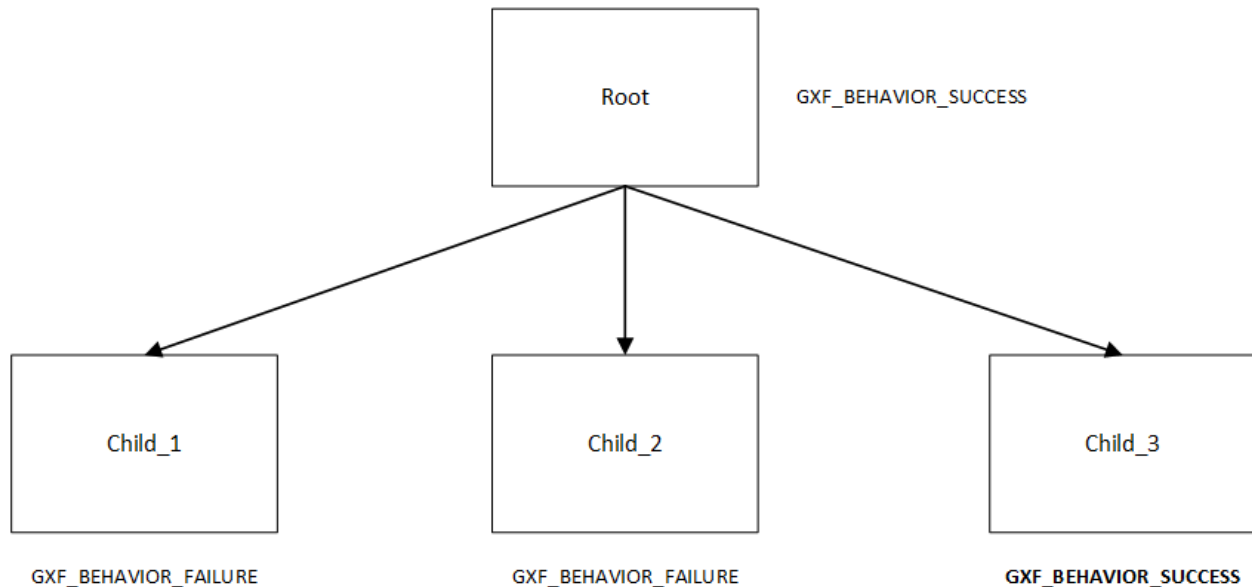
(continued from previous page)

Selector Behavior

Runs all child entities in sequence until one succeeds, then reports success. If all child entities fail (or no child entities are present), this codelet fails.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
children	Child entities

The diagram below shows a graphical representation of a Selector behavior. The root entity starts `child_1`, `child_2`, and `child_3` in a sequence. Although `child_1` and `child_2` fail, the root entity will return success since `child_3` returns successfully.



A code snippet to configure a selector behavior is as shown below.

```

name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_sel_codelet
  type: nvidia::gxf::SelectorBehavior
  parameters:
    children: [ door_distance/door_distance_st, door_detected/door_detected_st, knock/
↪knock_st ]
  
```

(continues on next page)

(continued from previous page)

```

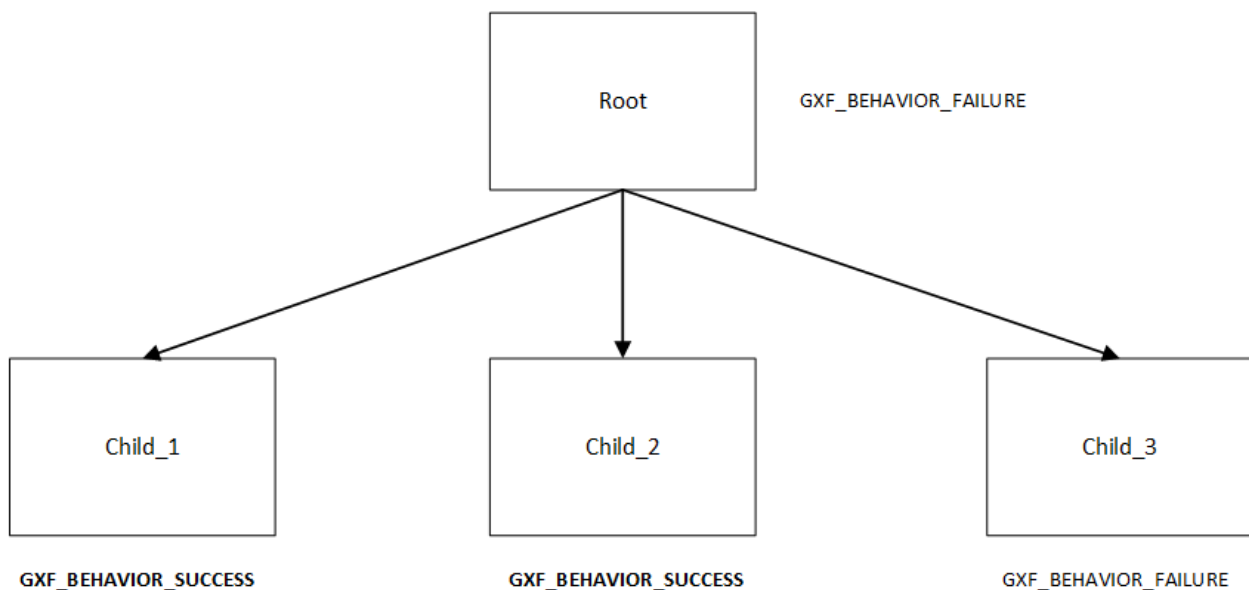
    s_term: root_st
---
name: door_distance
components:
- name: door_distance_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: door_distance_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: door_dist
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: []
    s_term: door_distance_st
---
```

Sequence Behavior

Runs its child entities in sequence, in the order in which they are defined. Succeeds when all child entities succeed or fails as soon as one child entity fails.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
children	Child entities

The diagram below shows a graphical representation of a Sequence behavior. The root entity starts `child_1`, `child_2` and `child_3` in a sequence. Although `child_1` and `child_2` pass, the root entity will return failure since `child_3` returns failure.



A code snippet to configure a sequence behavior is as shown below.

```
name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_codelet
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: [ child1/child1_st, child2/child2_st ]
    s_term: root_st
```

Switch Behavior

Runs the child entity with the index defined as `desired_behavior`.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
children	Child entities
desired_behavior	The index of child entity to switch to when this entity runs

In the code snippet shown below, the desired behavior of the root entity is designated to be the the child at index 1 (scene). Hence, that is the entity that is run.

```
name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_switch_codelet
  type: nvidia::gxf::SwitchBehavior
  parameters:
    children: [ scene/scene_st, ref/ref_st ]
    s_term: root_st
    desired_behavior: 0
---
name: scene
components:
- name: scene_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
```

(continues on next page)

(continued from previous page)

```

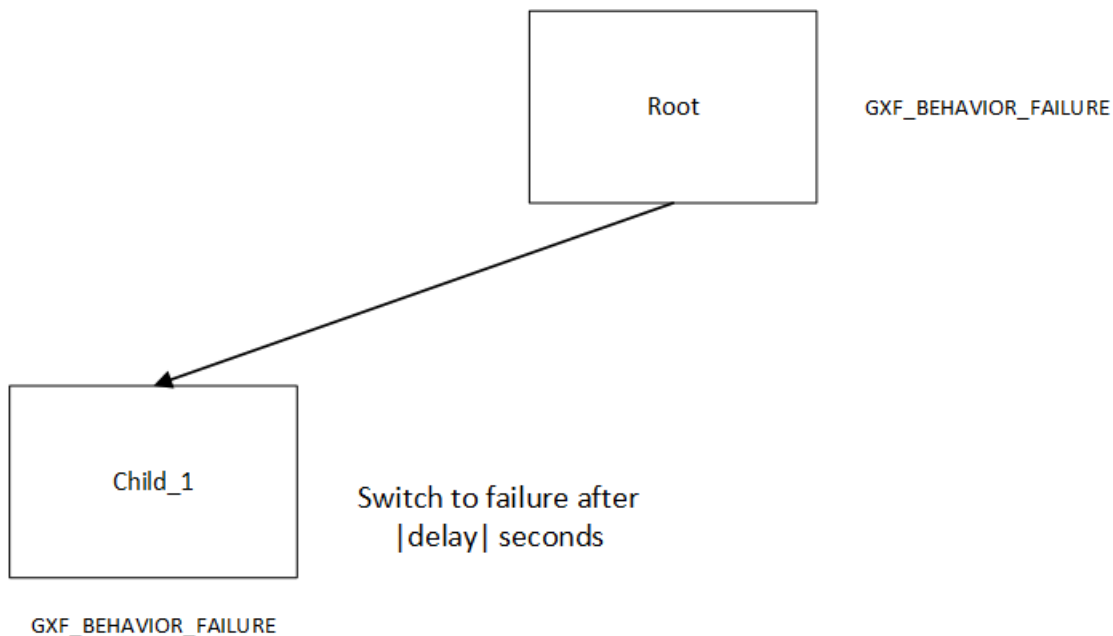
    max_repeat_count: 0
- name: scene_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: scene_seq
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: [ pose/pose_st, det/det_st, seg/seg_st ]
    s_term: scene_st
---
```

Timer Behavior

Waits for a specified amount of time delay and switches to the configured result switch_status afterwards.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
clock	Clock
switch_status	Configured result to switch to after the specified delay
delay	Configured delay

In the diagram shown below, the child entity switches to failure after a configured delay period. The root entity hence returns failure.



A code snippet for the same shown below.

```

name: knock_on_door
components:
- name: knock_on_door_controller

```

(continues on next page)

(continued from previous page)

```

type: nvidia::gxf::EntityCountFailureRepeatController
parameters:
  max_repeat_count: 10
- name: knock_on_door_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: knock
  type: nvidia::gxf::TimerBehavior
  parameters:
    switch_status: 1
    clock: sched/clock
    delay: 1
    s_term: knock_on_door_st
---
```

31.6 GXF Core C APIs

31.6.1 Context

Create context

```
gxf_result_t GxfContextCreate(gxf_context_t* context);
```

Creates a new GXF context

A GXF context is required for all almost all GXF operations. The context must be destroyed with `GxfContextDestroy`. Multiple contexts can be created in the same process, however they can not communicate with each other.

parameter: context The new GXF context is written to the given pointer.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Create a context from a shared context

```
gxf_result_t GxfContextCreate1(gxf_context_t shared, gxf_context_t* context);
```

Creates a new runtime context from shared context.

A shared runtime context is used for sharing entities between graphs running within the same process.

parameter: shared A valid GXF shared context.

parameter: context The new GXF context is written to the given pointer.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Destroy context

```
gxf_result_t GxfContextDestroy(gxf_context_t context);
```

Destroys a GXF context

Every GXF context must be destroyed by calling this function. The context must have been previously created with `GxfContextCreate`. This will also destroy all entities and components which were created as part of the context.

parameter: `context` A valid GXF context.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

31.6.2 Extensions

Maximum number of extensions in a context can be 1024.

Load Extensions from a file

```
gxf_result_t GxfLoadExtension(gxf_context_t context, const char* filename);
```

Loads extension in the given context from file.

parameter: `context` A valid GXF context.

parameter: `filename` A valid filename.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

This function will be deprecated.

Load Extension libraries

```
gxf_result_t GxfLoadExtensions(gxf_context_t context, const GxfLoadExtensionsInfo* info);
```

Loads GXF extension libraries

Loads one or more extensions either directly by their filename or indirectly by loading manifest files. Before a component can be added to a GXF entity the GXF extension shared library providing the component must be loaded. An extension must only be loaded once.

To simplify loading multiple extensions at once the developer can create a manifest file which lists all extensions he needs. This function will then load all extensions listed in the manifest file. Multiple manifest may be loaded, however each extension may still be loaded only a single time.

A manifest file is a YAML file with a single top-level entry `extensions` followed by a list of filenames of GXF extension shared libraries.

Example: — START OF FILE — extensions: - gxf/std/libgxf_std.so - gxf/npp/libgxf_npp.so — END OF FILE —

parameter: `context` A valid GXF context.

parameter: `filename` A valid filename.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

```
gxf_result_t GxfLoadExtensionManifest(gxf_context_t context, const char*
manifest_filename);
```

Loads extensions from manifest file.

parameter: `context` A valid GXF context.

parameter: `filename` A valid filename.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

This function will be deprecated.

Load Metadata files

```
gxf_result_t GxfLoadExtensionMetadataFiles(gxf_context_t context, const char* const*
filenames, uint32_t count);
```

Loads an extension registration metadata file.

Reads a metadata file of the contents of an extension used for registration. These metadata files can be used to resolve typename and TID's of components for other extensions which depend on them. Metadata files do not contain the actual implementation of the extension and must be loaded only to run the extension query API's on extension libraries which have the actual implementation and only depend on the metadata for type resolution.

If some components of extension B depend on some components in extension A: - Load metadata file for extension A. - Load extension library for extension B using `GxfLoadExtensions`. - Run extension query APIs on extension B and its components.

parameter: `context` A valid GXF context.

parameter: `filenames` absolute paths of metadata files.

parameter: `count` The number of metadata files to be loaded.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Register component

```
gxf_result_t GxfRegisterComponent(gxf_context_t context, gxf_tid_t tid, const char* name,
const char* base_name);
```

Registers a component with a GXF extension.

A GXF extension need to register all of its components in the extension factory function. For convenience, the helper macros in `gxf/std/extension_factory_helper.hpp` can be used.

The developer must choose a unique GXF tid with two random 64-bit integers. The developer must ensure that every GXF component has a unique tid. The name of the component must be the fully qualified C++ type name of the component. A component may only have a single base class and that base class must be specified with its fully qualified C++ type name as the parameter `base_name`.

ref: `gxf/std/extension_factory_helper.hpp` ref: `core/type_name.hpp`

parameter: `context` A valid GXF context.

parameter: `tid` The chosen GXF tid.

parameter: `name` The type name of the component.

parameter: `base_name` The type name of the base class of the component.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

31.6.3 Graph Execution

Loads a list of entities from YAML file

```
gxf_result_t GxfGraphLoadFile(gxf_context_t context, const char* filename, const char* parameters_override[], const uint32_t num_overrides);
```

parameter: context A valid GXF context.

parameter: filename A valid YAML filename.

parameter: params_override An optional array of strings used for override parameters in YAML file.

parameter: num_overrides Number of optional override parameter strings.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Set the root folder for searching YAML files during loading

```
gxf_result_t GxfGraphSetRootPath(gxf_context_t context, const char* path);
```

parameter: context A valid GXF context.

parameter: path Path to root folder for searching YAML files during loading.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Loads a list of entities from YAML text

```
gxf_result_t GxfGraphParseString(gxf_context_t context, const char* tex, const char* parameters_override[], const uint32_t num_overrides);
```

parameter: context A valid GXF context.

parameter: text A valid YAML text.

parameter: params_override An optional array of strings used for override parameters in yaml file.

parameter: num_overrides Number of optional override parameter strings.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Activate all system components

```
gxf_result_t GxfGraphActivate(gxf_context_t context);
```

parameter: context A valid GXF context.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Deactivate all System components

```
gxf_result_t GxfGraphDeactivate(gxf_context_t context);
```

parameter: context A valid GXF context.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Starts the execution of the graph asynchronously

```
gxf_result_t GxfGraphRunAsync(gxf_context_t context);
```

parameter: context A valid GXF context.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Interrupt the execution of the graph

```
gxf_result_t GxfGraphInterrupt(gxf_context_t context);
```

parameter: context A valid GXF context.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Waits for the graph to complete execution

```
gxf_result_t GxfGraphWait(gxf_context_t context);
```

parameter: context A valid GXF context.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.`

Runs all System components and waits for their completion

```
gxf_result_t GxfGraphRun(gxf_context_t context);
```

parameter: context A valid GXF context.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

31.6.4 Entities

Create an entity

```
gxf_result_t GxfEntityCreate(gxf_context_t context, gxf_uid_t* eid);
```

Creates a new entity and updates the eid to the unique identifier of the newly created entity.

This method will be deprecated.

```
gxf_result_t GxfCreateEntity((gxf_context_t context, const GxfEntityCreateInfo* info,
gxf_uid_t* eid);
```

Create a new GXF entity.

Entities are lightweight containers to hold components and form the basic building blocks of a GXF application. Entities are created when a GXF file is loaded, or they can be created manually using this function. Entities created with this function must be destroyed using `GxfEntityDestroy`. After the entity was created components can be added to it

with `GxfComponentAdd`. To start execution of codelets on an entity, the entity needs to be activated first. This can happen automatically using `GXF_ENTITY_CREATE_PROGRAM_BIT` or manually using `GxfEntityActivate`.

parameter `context`: GXF context that creates the entity. parameter `info`: pointer to a `GxfEntityCreateInfo` structure containing parameters affecting the creation of the entity. parameter `eid`: pointer to a `gxf_uid_t` handle in which the resulting entity is returned. returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Activate an entity

```
gxf_result_t GxfEntityActivate(gxf_context_t context, gxf_uid_t eid);
```

Activates a previously created and inactive entity.

Activating an entity generally marks the official start of its lifetime and has multiple implications: - If mandatory parameters (i.e., parameters which do not have the flag `optional`) are not set, the operation will fail.

- All components on the entity are initialized.
- All codelets on the entity are scheduled for execution. The scheduler will start calling `start`, `tick`, and `stop` functions as specified by scheduling terms.
- After activation, trying to change a dynamic parameters will result in a failure.
- Adding or removing components of an entity after activation will result in a failure.

parameter: `context` A valid GXF context.

parameter: `eid` UID of a valid entity.

returns: GXF error code.

Deactivate an entity

```
gxf_result_t GxfEntityDeactivate(gxf_context_t context, gxf_uid_t eid);
```

Deactivates a previously activated entity.

Deactivating an entity generally marks the official end of its lifetime and has multiple implications:

- All codelets are removed from the schedule. Already running entities are run to completion.
- All components on the entity are deinitialized.
- Components can be added or removed again once the entity was deactivated.
- Mandatory and non-dynamic parameters can be changed again.

Note: In case that the entity is currently executing, this function will wait and block until

the current execution is finished.

parameter: `context` A valid GXF context.

parameter: `eid` UID of a valid entity.

returns: GXF error code.

Destroy an entity

```
gxf_result_t GxfEntityDestroy(gxf_context_t context, gxf_uid_t eid);
```

Destroys a previously created entity.

Destroys an entity immediately. The entity is destroyed even if the reference count has not yet reached 0. If the entity is active, it is deactivated first.

Note: This function can block for the same reasons as ‘GxfEntityDeactivate’.

parameter: `context` A valid GXF context.

parameter: `eid` The returned UID of the created entity.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Find an entity

```
gxf_result_t GxfEntityFind(gxf_context_t context, const char* name, gxf_uid_t* eid);
```

Finds an entity by its name.

parameter: `context` A valid GXF context.

parameter: `name` A C string with the name of the entity. Ownership is not transferred.

parameter: `eid` The returned UID of the entity.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Find all entities

```
gxf_result_t GxfEntityFindAll(gxf_context_t context, uint64_t* num_entities, gxf_uid_t* entities);
```

Finds all entities in the current application.

Finds and returns all entity ids for the current application. If more than *max_entities* exist only *max_entities* will be returned. The order and selection of entities returned is arbitrary.

parameter: `context` A valid GXF context.

parameter: `num_entities` In/Out: the max number of entities that can fit in the buffer/the number of entities that exist in the application.

parameter: `entities` A buffer allocated by the caller for returned UIDs of all entities, with capacity for *num_entities*.

returns: `GXF_SUCCESS` if the operation was successful, `GXF_QUERY_NOT_ENOUGH_CAPACITY` if more entities exist in the application than *max_entities*, or otherwise one of the GXF error codes.

Increase reference count of an entity

```
gxf_result_t GxfEntityRefCountInc(gxf_context_t context, gxf_uid_t eid);
```

Increases the reference count for an entity by 1.

By default reference counting is disabled for an entity. This means that entities created with `GxfEntityCreate` are not automatically destroyed. If this function is called for an entity with disabled reference count, reference counting is enabled and the reference count is set to 1. Once reference counting is enabled an entity will be automatically destroyed if the reference count reaches zero, or if `GxfEntityCreate` is called explicitly.

parameter: `context` A valid GXF context.

parameter: `eid` The UID of a valid entity.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Decrease reference count of an entity.

```
gxf_result_t GxfEntityRefCountDec(gxf_context_t context, gxf_uid_t eid);
```

Decreases the reference count for an entity by 1.

See `GxfEntityRefCountInc` for more details on reference counting.

parameter: `context` A valid GXF context.

parameter: `eid` The UID of a valid entity.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Get status of an entity

```
gxf_result_t GxfEntityGetStatus(gxf_context_t context, gxf_uid_t eid,  
gxf_entity_status_t* entity_status);
```

Gets the status of the entity.

See `gxf_entity_status_t` for the various status.

parameter: `context` A valid GXF context.

parameter: `eid` The UID of a valid entity.

parameter: `entity_status` output; status of an entity `eid`.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Get state of an entity

```
gxf_result_t GxfEntityGetState(gxf_context_t context, gxf_uid_t eid, entity_state_t*  
entity_state);
```

Gets the state of the entity.

See `gxf_entity_status_t` for the various status.

parameter: `context` A valid GXF context.

parameter: `eid` The UID of a valid entity.

parameter: `entity_state` output; behavior status of an entity `eid` used by the behavior tree parent codelet.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Notify entity of an event

```
gxf_result_t GxfEntityEventNotify(gxf_context_t context, gxf_uid_t eid);
```

Notifies the occurrence of an event and inform the scheduler to check the status of the entity

The entity must have an `AsynchronousSchedulingTerm` scheduling term component and it must be in the `EVENT_WAITING` state for the notification to be acknowledged.

See `AsynchronousEventState` for various states.

parameter: `context` A valid GXF context.

parameter: `eid` The UID of a valid entity.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

31.6.5 Components

Maximum number of components in an entity or an extension can be up to 1024.

Get component type identifier

```
gxf_result_t GxfComponentTypeId(gxf_context_t context, const char* name, gxf_tid_t* tid);
```

Gets the GXF unique type ID (TID) of a component.

Get the unique type ID which was used to register the component with GXF. The function expects the fully qualified C++ type name of the component including namespaces.

Example of a valid component type name: `nvidia::gxf::test::PingTx`

parameter: `context` A valid GXF context.

parameter: `name` The fully qualified C++ type name of the component.

parameter: `tid` The returned TID of the component.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get component type name

```
gxf_result_t GxfComponentTypeName(gxf_context_t context, gxf_tid_t tid, const char** name);
```

Gets the fully qualified C++ type name GXF component typename

Get the unique typename of the component with which it was registered using one of the `GXF_EXT_FACTORY_ADD*()` macros

parameter: `context` A valid GXF context.

parameter: `tid` The unique type ID (TID) of the component with which the component was registered.

parameter: `name` The returned name of the component.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get component name

```
gxf_result_t GxfComponentName(gxf_context_t context, gxf_uid_t cid, const char** name);
```

Gets the name of a component.

Each component has a user-defined name which was used in the call to `GxfComponentAdd`. Usually the name is specified in the GXF application file.

parameter: `context` A valid GXF context.

parameter: `cid` The unique object ID (UID) of the component.

parameter: `name` The returned name of the component.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Get unique identifier of the entity of given component

```
gxf_result_t GxfComponentEntity(gxf_context_t context, gxf_uid_t cid, gxf_uid_t* eid);
```

Gets the unique object ID of the entity of a component

Each component has a unique ID with respect to the context and is stored in one entity. This function can be used to retrieve the ID of the entity to which a given component belongs.

parameter: `context` A valid GXF context.

parameter: `cid` The unique object ID (UID) of the component.

parameter: `eid` The returned UID of the entity.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Add a new component

```
gxf_result_t GxfComponentAdd(gxf_context_t context, gxf_uid_t eid, gxf_tid_t tid, const char* name, gxf_uid_t* cid);
```

Adds a new component to an entity.

An entity can contain multiple components and this function can be used to add a new component to an entity. A component must be added before an entity is activated, or after it was deactivated. Components must not be added to active entities. The order of components is stable and identical to the order in which components are added (see `GxfComponentFind`).

parameter: `context` A valid GXF context.

parameter: `eid` The unique object ID (UID) of the entity to which the component is added.

parameter: `tid` The unique type ID (TID) of the component to be added to the entity.

parameter: `name` The name of the new component. Ownership is not transferred.

parameter: `cid` The returned UID of the created component.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Add component to entity interface

```
gxf_result_t GxfComponentAddToInterface(gxf_context_t context, gxf_uid_t eid, gxf_uid_t
cid, const char* name);
```

Adds an existing component to the interface of an entity.

An entity can hold references to other components in its interface, so that when finding a component in an entity, both the component this entity holds and those it refers to will be returned. This supports the case when an entity contains a subgraph, then those components that have been declared in the subgraph interface will be put to the interface of the parent entity.

parameter: `context` A valid GXF context.

parameter: `eid` The unique object ID (UID) of the entity to which the component is added.

parameter: `cid` The unique object ID of the component.

parameter: `name` The name of the new component. Ownership is not transferred.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Find a component in an entity

```
gxf_result_t GxfComponentFind(gxf_context_t context, gxf_uid_t eid, gxf_tid_t tid, const
char* name, int32_t* offset, gxf_uid_t* cid);
```

Finds a component in an entity.

Searches components in an entity which satisfy certain criteria: component type, component name, and component min index. All three criteria are optional; in case no criteria is given the first component is returned. The main use case for `component_min_index` is a repeated search which continues at the index which was returned by a previous search.

In case no entity with the given criteria was found `GXF_ENTITY_NOT_FOUND` is returned.

parameter: `context` A valid GXF context.

parameter: `eid` The unique object ID (UID) of the entity which is searched.

parameter: `tid` The component type ID (TID) of the component to find (optional).

parameter: `name` The component name of the component to find (optional). Ownership not transferred.

parameter: `offset` The index of the first component in the entity to search. Also contains the index of the component which was found.

parameter: `cid` The returned UID of the searched component.

returns: `GXF_SUCCESS` if a component matching the criteria was found, `GXF_ENTITY_NOT_FOUND` if no component matching the criteria was found, or otherwise one of the GXF error codes.

Get type identifier for a component

```
gxf_result_t GxfComponentType(gxf_context_t context, gxf_uid_t cid, gxf_tid_t* tid);
```

Gets the component type ID (TID) of a component.

parameter: `context` A valid GXF context.

parameter: `cid` The component object ID (UID) for which the component type is requested.

parameter: `tid` The returned TID of the component.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Gets pointer to component

```
gxf_result_t GxfComponentPointer(gxf_context_t context, gxf_uid_t uid, gxf_tid_t tid, void** pointer);
```

Verifies that a component exists, has the given type, gets a pointer to it.

parameter: `context` A valid GXF context.

parameter: `uid` The component object ID (UID).

parameter: `tid` The expected component type ID (TID) of the component.

parameter: `pointer` The returned pointer to the component object.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

31.6.6 Primitive Parameters

64-bit floating point

Set

```
gxf_result_t GxfParameterSetFloat64(gxf_context_t context, gxf_uid_t uid, const char* key, double value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` a double value.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Get

```
gxf_result_t GxfParameterGetFloat64(gxf_context_t context, gxf_uid_t uid, const char*
key, double* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the double value.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

64-bit signed integer**Set**

```
gxf_result_t GxfParameterSetInt64(gxf_context_t context, gxf_uid_t uid, const char* key,
int64_t value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value 64-bit integer value to set.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get

```
gxf_result_t GxfParameterGetInt64(gxf_context_t context, gxf_uid_t uid, const char* key,
int64_t* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the 64-bit integer value.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

64-bit unsigned integer**Set**

```
gxf_result_t GxfParameterSetUInt64(gxf_context_t context, gxf_uid_t uid, const char* key,
uint64_t value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value unsigned 64-bit integer value to set.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get

```
gxf_result_t GxfParameterGetUInt64(gxf_context_t context, gxf_uid_t uid, const char* key,
uint64_t* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the unsigned 64-bit integer value.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

32-bit signed integer

Set

```
gxf_result_t GxfParameterSetInt32(gxf_context_t context, gxf_uid_t uid, const char* key,
int32_t value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value 32-bit integer value to set.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get

```
gxf_result_t GxfParameterGetInt32(gxf_context_t context, gxf_uid_t uid, const char* key,
int32_t* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the 32-bit integer value.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

String parameter

Set

```
gxf_result_t GxfParameterSetStr(gxf_context_t context, gxf_uid_t uid, const char* key,
const char* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value A char array containing value to set.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get

```
gxf_result_t GxfParameterGetStr(gxf_context_t context, gxf_uid_t uid, const char* key,
const char** value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to a char* array to get the value.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Boolean

Set

```
gxf_result_t GxfParameterSetBool(gxf_context_t context, gxf_uid_t uid, const char* key,
bool value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value A boolean value to set.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get

```
gxf_result_t GxfParameterGetBool(gxf_context_t context, gxf_uid_t uid, const char* key,
bool* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the boolean value.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Handle

Set

```
gxf_result_t GxfParameterSetHandle(gxf_context_t context, gxf_uid_t uid, const char* key,
gxf_uid_t cid);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: cid Unique identifier to set.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get

```
gxf_result_t GxfParameterGetHandle(gxf_context_t context, gxf_uid_t uid, const char* key,
gxf_uid_t* cid);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value Pointer to a unique identifier to get the value.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

31.6.7 Vector Parameters

To set or get the vector parameters of a component, users can use the following C-APIs for various data types:

Set 1-D Vector Parameters

Users can call `gxf_result_t GxfParameterSet1D(DataType)Vector(gxf_context_t context, gxf_uid_t uid, const char* key, data_type* value, uint64_t length)`

`value` should point to an array of the data to be set of the corresponding type. The size of the stored array should match the `length` argument passed.

See the table below for all the supported data types and their corresponding function signatures.

parameter: `key` The name of the parameter.

parameter: `value` The value to set of the parameter.

parameter: `length` The length of the vector parameter.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Table 31.1: Supported Data Types to Set 1D Vector Parameters

Function Name	data_type
<code>GxfParameterSet1DFloat64Vector(...)</code>	<code>double</code>
<code>GxfParameterSet1DInt64Vector(...)</code>	<code>int64_t</code>
<code>GxfParameterSet1DUInt64Vector(...)</code>	<code>uint64_t</code>
<code>GxfParameterSet1DInt32Vector(...)</code>	<code>int32_t</code>

Set 2-D Vector Parameters

Users can call `gxf_result_t GxfParameterSet2D(DataType)Vector(gxf_context_t context, gxf_uid_t uid, const char* key, data_type** value, uint64_t height, uint64_t width)`

`value` should point to an array of array (and not to the address of a contiguous array of data) of the data to be set of the corresponding type. The length of the first dimension of the array should match the `height` argument passed, and similarly the length of the second dimension of the array should match the `width` passed.

See the table below for all the supported data types and their corresponding function signatures.

parameter: `key` The name of the parameter.

parameter: `value` The value to set of the parameter.

parameter: `height` The height of the 2-D vector parameter.

parameter: `width` The width of the 2-D vector parameter.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Table 31.2: Supported Data Types to Set 2D Vector Parameters

Function Name	data_type
<code>GxfParameterSet2DFloat64Vector(...)</code>	<code>double</code>
<code>GxfParameterSet2DInt64Vector(...)</code>	<code>int64_t</code>
<code>GxfParameterSet2DUInt64Vector(...)</code>	<code>uint64_t</code>
<code>GxfParameterSet2DInt32Vector(...)</code>	<code>int32_t</code>

Get 1-D Vector Parameters

Users can call `gxf_result_t GxfParameterGet1D"DataType"Vector(gxf_context_t context, gxf_uid_t uid, const char* key, data_type** value, uint64_t* length)` to get the value of a 1-D vector.

Before calling this method, users should call `GxfParameterGet1D"DataType"VectorInfo(gxf_context_t context, gxf_uid_t uid, const char* key, uint64_t* length)` to obtain the length of the vector parameter and then allocate at least that much memory to retrieve the value.

`value` should point to an array of size greater than or equal to `length` allocated by user of the corresponding type to retrieve the data. If the `length` doesn't match the size of stored vector then it will be updated with the expected size.

See the table below for all the supported data types and their corresponding function signatures.

parameter: `key` The name of the parameter.

parameter: `value` The value to set of the parameter.

parameter: `length` The length of the 1-D vector parameter obtained by calling `GxfParameterGet1D"DataType"VectorInfo(...)`.

Table 31.3: Supported Data Types to Get the Value of 1D Vector Parameters

Function Name	data_type
<code>GxfParameterGet1DFloat64Vector(...)</code>	<code>double</code>
<code>GxfParameterGet1DInt64Vector(...)</code>	<code>int64_t</code>
<code>GxfParameterGet1DUInt64Vector(...)</code>	<code>uint64_t</code>
<code>GxfParameterGet1DInt32Vector(...)</code>	<code>int32_t</code>

Get 2-D Vector Parameters

Users can call `gxf_result_t GxfParameterGet2D"DataType"Vector(gxf_context_t context, gxf_uid_t uid, const char* key, data_type** value, uint64_t* height, uint64_t* width)` to get the value of a 2D vector.

Before calling this method, users should call `GxfParameterGet1D"DataType"VectorInfo(gxf_context_t context, gxf_uid_t uid, const char* key, uint64_t* height, uint64_t* width)` to obtain the height and width of the 2D-vector parameter and then allocate at least that much memory to retrieve the value.

`value` should point to an array of array of height (size of first dimension) greater than or equal to `height` and width (size of the second dimension) greater than or equal to `width` allocated by user of the corresponding type to get the data. If the `height` or `width` don't match the height and width of the stored vector then they will be updated with the expected values.

See the table below for all the supported data types and their corresponding function signatures.

parameter": `key` The name of the parameter.

parameter": `value` Allocated array to get the value of the parameter.

parameter": `height` The height of the 2-D vector parameter obtained by calling `GxfParameterGet2D"DataType"VectorInfo(...)`.

parameter": `width` The width of the 2-D vector parameter obtained by calling `GxfParameterGet2D"DataType"VectorInfo(...)`.

Table 31.4: Supported Data Types to Get the Value of 2D Vector Parameters

Function Name	data_type
GxfParameterGet2DFloat64Vector(...)	double
GxfParameterGet2DInt64Vector(...)	int64_t
GxfParameterGet2DUInt64Vector(...)	uint64_t
GxfParameterGet2DInt32Vector(...)	int32_t

31.6.8 Information Queries

Get Meta Data about the GXF Runtime

```
gxf_result_t GxfRuntimeInfo(gxf_context_t context, gxf_runtime_info* info);
```

parameter: context A valid GXF context.

parameter: info pointer to gxf_runtime_info object to get the meta data.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get description and list of components in loaded Extension

```
gxf_result_t GxfExtensionInfo(gxf_context_t context, gxf_tid_t tid, gxf_extension_info_t* info);
```

parameter: context A valid GXF context.

parameter: tid The unique identifier of the extension.

parameter: info pointer to gxf_extension_info_t object to get the meta data.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get description and list of parameters of Component

```
gxf_result_t GxfComponentInfo(gxf_context_t context, gxf_tid_t tid, gxf_component_info_t* info);
```

Note: Parameters are only available after at least one instance is created for the Component.

parameter: context A valid GXF context.

parameter: tid The unique identifier of the component.

parameter: info pointer to gxf_component_info_t object to get the meta data.

returns: GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

Get parameter type description

Gets a string describing the parameter type.

```
const char* GxfParameterTypeStr(gxf_parameter_type_t param_type);
```

parameter: `param_type` Type of parameter to get info about.

returns: C-style string description of the parameter type.

Get flag type description

Gets a string describing the flag type.

```
const char* GxfParameterFlagTypeStr(gxf_parameter_flags_t flag_type);
```

parameter: `flag_type` Type of flag to get info about.

returns: C-style string description of the flag type.

Get parameter description

Gets description of specific parameter. Fails if the component is not instantiated yet.

```
gxf_result_t GxfGetParameterInfo(gxf_context_t context, gxf_tid_t cid, const char* key,
gxf_parameter_info_t* info);
```

parameter: `context` A valid GXF context.

parameter: `cid` The unique identifier of the component.

parameter: `key` The name of the parameter.

parameter: `info` Pointer to a `gxf_parameter_info_t` object to get the value.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

Redirect logs to a file

Redirect console logs to the provided file.

```
gxf_result_t GxfGetParameterInfo(gxf_context_t context, FILE* fp);
```

parameter: `context` A valid GXF context.

parameter: `fp` File path for the redirected logs.

returns: `GXF_SUCCESS` if the operation was successful, or otherwise one of the GXF error codes.

31.6.9 Miscellaneous

Get string description of error

```
const char* GxfResultStr(gxf_result_t result);
```

Gets a string describing an GXF error code.

The caller does not get ownership of the return C string and must not delete it.

parameter: `result` A GXF error code.

returns: A pointer to a C string with the error code description.

31.7 CudaExtension

Extension for CUDA operations.

- UUID: d63a98fa-7882-11eb-a917-b38f664f399c
- Version: 2.0.0
- Author: NVIDIA
- License: LICENSE

31.7.1 Components

`nvidia::gxf::CudaStream`

Holds and provides access to native `cudaStream_t`.

`nvidia::gxf::CudaStream` handle must be allocated by `nvidia::gxf::CudaStreamPool`. Its lifecycle is valid until explicitly recycled through `nvidia::gxf::CudaStreamPool.releaseStream()` or implicitly until `nvidia::gxf::CudaStreamPool` is deactivated.

You may call `stream()` to get the native `cudaStream_t` handle, and to submit GPU operations. After the submission, GPU takes over the input tensors/buffers and keeps them in use. To prevent the host carelessly releasing these in-use buffers, CUDA Codelet needs to call `record(event, input_entity, sync_cb)` to extend `input_entity`'s lifecycle until the GPU completely consumes it. Alternatively, you may call `record(event, event_destroy_cb)` for native `cudaEvent_t` operations and free in-use resource via `event_destroy_cb`.

It is required to have a `nvidia::gxf::CudaStreamSync` in the graph pipeline after all the CUDA operations. See more details in `nvidia::gxf::CudaStreamSync`.

- Component ID: 5683d692-7884-11eb-9338-c3be62d576be
- Defined in: `gxf/cuda/cuda_stream.hpp`

`nvidia::gxf::CudaStreamId`

Holds CUDA stream Id to deduce `nvidia::gxf::CudaStream` handle.

`stream_cid` should be `nvidia::gxf::CudaStream` component id.

- Component ID: 7982aeac-37f1-41be-ade8-6f00b4b5d47c
- Defined in: `gxf/cuda/cuda_stream_id.hpp`

nvidia::gxf::CudaEvent

Holds and provides access to native `cudaEvent_t` handle.

When a `nvidia::gxf::CudaEvent` is created, you'll need to initialize a native `cudaEvent_t` through `init(flags, dev_id)`, or set third party event through `initWithEvent(event, dev_id, free_fnc)`. The event keeps valid until `deinit` is called explicitly otherwise gets recycled in destructor.

- Component ID: f5388d5c-a709-47e7-86c4-171779bc64f3
- Defined in: `gxf/cuda/cuda_event.hpp`

nvidia::gxf::CudaStreamPool

CudaStream allocation.

You must explicitly call `allocateStream()` to get a valid `nvidia::gxf::CudaStream` handle. This component would hold all the its allocated `nvidia::gxf::CudaStream` entities until `releaseStream(stream)` is called explicitly or the `CudaStreamPool` component is deactivated.

- Component ID: 6733bf8b-ba5e-4fae-b596-af2d1269d0e7
- Base Type: `nvidia::gxf::Allocator`

Parameters

dev_id

GPU device id.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_INT32`
- Default Value: 0

stream_flags

Flag values to create CUDA streams.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_INT32`
- Default Value: 0

stream_priority

Priority values to create CUDA streams.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_INT32`

- Default Value: 0

reserved_size

User-specified file name without extension.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 1

max_size

Maximum Stream Size.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 0, no limitation.

nvidia::gxf::CudaStreamSync

Synchronize all CUDA streams which are carried by message entities.

This codelet is required to get connected in the graph pipeline after all CUDA ops codelets. When a message entity is received, it would find all of the `nvidia::gxf::CudaStreamId` in that message, and extract out each `nvidia::gxf::CudaStream`. With each `CudaStream` handle, it synchronizes all previous `nvidia::gxf::CudaStream.record()` events, along with all submitted GPU operations before this point.

Note: `CudaStreamSync` must be set in the graph when `nvidia::gxf::CudaStream.record()` is used, otherwise it may cause a memory leak.

- Component ID: 0d1d8142-6648-485d-97d5-277eed00129c
- Base Type: `nvidia::gxf::Codelet`

Parameters**rx**

Receiver to receive all messages carrying `nvidia::gxf::CudaStreamId`.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: `nvidia::gxf::Receiver`

tx

Transmitter to send messages to downstream.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: `nvidia::gxf::Transmitter`

31.8 MultimediaExtension

Extension for multimedia related data types, interfaces and components in GXF Core.

- UUID: 6f2d1afc-1057-481a-9da6-a5f61fed178e
- Version: 2.0.0
- Author: NVIDIA
- License: LICENSE

31.8.1 Components

`nvidia::gxf::AudioBuffer`

`AudioBuffer` is similar to `Tensor` component in the standard extension and holds memory and metadata corresponding to an audio buffer.

- Component ID: a914cac6-5f19-449d-9ade-8c5cdcebe7c3

`AudioBufferInfo` structure captures the following metadata:

Field	Description
<code>channels</code>	Number of channels in an audio frame
<code>samples</code>	Number of samples in an audio frame
<code>sampling_rate</code>	sampling rate in Hz
<code>bytes_per_sample</code>	Number of bytes required per sample
<code>audio_format</code>	<code>AudioFormat</code> of an audio frame
<code>audio_layout</code>	<code>AudioLayout</code> of an audio frame

Supported `AudioFormat` types:

AudioFormat	Description
<code>GXF_AUDIO_FORMAT_S16LE</code>	16-bit signed PCM audio
<code>GXF_AUDIO_FORMAT_F32LE</code>	32-bit floating-point audio

Supported `AudioLayout` types:

AudioLayout	Description
<code>GXF_AUDIO_LAYOUT_INTERLEAVED</code>	Data from all the channels to be interleaved - LRLRLR
<code>GXF_AUDIO_LAYOUT_NON_INTERLEAVED</code>	Data from all the channels not to be interleaved - LLLRRR

nvidia::gxf::VideoBuffer

VideoBuffer is similar to Tensor component in the standard extension and holds memory and metadata corresponding to a video buffer.

- Component ID: 16ad58c8-b463-422c-b097-61a9acc5050e

VideoBufferInfo structure captures the following metadata:

Field	Description
width	width of a video frame
height	height of a video frame
color_format	VideoFormat of a video frame
color_planes	ColorPlane(s) associated with the VideoFormat
surface_layout	SurfaceLayout of the video frame

Supported VideoFormat types:

VideoFormat	Description
GXF_VIDEO_FORMAT_YUV420	BT.601 multi planar 4:2:0 YUV
GXF_VIDEO_FORMAT_YUV420_ER	BT.601 multi planar 4:2:0 YUV ER
GXF_VIDEO_FORMAT_YUV420_709	BT.709 multi planar 4:2:0 YUV
GXF_VIDEO_FORMAT_YUV420_709_ER	BT.709 multi planar 4:2:0 YUV ER
GXF_VIDEO_FORMAT_NV12	BT.601 multi planar 4:2:0 YUV with interleaved UV
GXF_VIDEO_FORMAT_NV12_ER	BT.601 multi planar 4:2:0 YUV ER with interleaved UV
GXF_VIDEO_FORMAT_NV12_709	BT.709 multi planar 4:2:0 YUV with interleaved UV
GXF_VIDEO_FORMAT_NV12_709_ER	BT.709 multi planar 4:2:0 YUV ER with interleaved UV
GXF_VIDEO_FORMAT_RGBA	RGBA-8-8-8-8 single plane
GXF_VIDEO_FORMAT_BGRA	BGRA-8-8-8-8 single plane
GXF_VIDEO_FORMAT_ARGB	ARGB-8-8-8-8 single plane
GXF_VIDEO_FORMAT_ABGR	ABGR-8-8-8-8 single plane
GXF_VIDEO_FORMAT_RGBX	RGBX-8-8-8-8 single plane
GXF_VIDEO_FORMAT_BGRX	BGRX-8-8-8-8 single plane
GXF_VIDEO_FORMAT_XRGB	XRGB-8-8-8-8 single plane
GXF_VIDEO_FORMAT_XBGR	XBGR-8-8-8-8 single plane
GXF_VIDEO_FORMAT_RGB	RGB-8-8-8 single plane
GXF_VIDEO_FORMAT_BGR	BGR-8-8-8 single plane
GXF_VIDEO_FORMAT_R8_G8_B8	RGB - unsigned 8 bit multiplanar
GXF_VIDEO_FORMAT_B8_G8_R8	BGR - unsigned 8 bit multiplanar
GXF_VIDEO_FORMAT_GRAY	8 bit GRAY scale single plane

Supported SurfaceLayout types:

SurfaceLayout	Description
GXF_SURFACE_LAYOUT_PITCH_LINEAR	pitch linear surface memory
GXF_SURFACE_LAYOUT_BLOCK_LINEAR	block linear surface memory

31.9 NetworkExtension

Extension for communications external to a computation graph.

- UUID: f50665e5-ade2-f71b-de2a-2380614b1725
- Version: 1.0.0
- Author: NVIDIA
- License: LICENSE

31.9.1 Interfaces

31.9.2 Components

nvidia::gxf::TcpClient

Codelet that functions as a client in a TCP connection.

- Component ID: 9d5955c7-8fda-22c7-f18f-ea5e2d195be9
- Base Type: nvidia::gxf::Codelet

Parameters

receivers

List of receivers to receive entities from.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::Receiver>>

transmitters

List of transmitters to publish entities to.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::Transmitter>>

serializers

List of component serializers to serialize and de-serialize entities.

- Flags: GXF_PARAMETER_FLAGS_NONE

- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: `std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>`

address

Address of TCP server.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

port

Port of TCP server.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32

timeout_ms

Time in milliseconds to wait before retrying connection to TCP server.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

maximum_attempts

Maximum number of attempts for I/O operations before failing.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

nvidia::gxf::TcpServer

Codelet that functions as a server in a TCP connection.

- Component ID: a3e0e42d-e32e-73ab-ef83-fbb311310759
- Base Type: nvidia::gxf::Codelet

Parameters

receivers

List of receivers to receive entities from.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::Receiver>>

transmitters

List of transmitters to publish entities to.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::Transmitter>>

serializers

List of component serializers to serialize and de-serialize entities.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>

address

Address of TCP server.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

port

Port of TCP server.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32

timeout_ms

Time in milliseconds to wait before retrying connection to TCP client.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

maximum_attempts

Maximum number of attempts for I/O operations before failing.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

31.10 SerializationExtension

Extension for serializing messages.

- UUID: bc573c2f-89b3-d4b0-8061-2da8b11fe79a
- Version: 2.0.0
- Author: NVIDIA
- License: LICENSE

31.10.1 Interfaces

nvidia::gxf::ComponentSerializer

Interface for serializing components.

- Component ID: 8c76a828-2177-1484-f841-d39c3fa47613
- Base Type: nvidia::gxf::Component
- Defined in: gxf/serialization/component_serializer.hpp

31.10.2 Components

nvidia::gxf::EntityRecorder

Serializes incoming messages and writes them to a file.

- Component ID: 9d5955c7-8fda-22c7-f18f-ea5e2d195be9
- Base Type: nvidia::gxf::Codelet

Parameters

receiver

Receiver channel to log.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

serializers

List of component serializers to serialize entities.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>

directory

Directory path for storing files.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

basename

User specified file name without extension.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_STRING

flush_on_tick

Flushes output buffer on every tick when true.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

nvidia::gxf::EntityReplayer

De-serializes and publishes messages from a file.

- Component ID: fe827c12-d360-c63c-8094-32b9244d83b6
- Base Type: nvidia::gxf::Codelet

Parameters**transmitter**

Transmitter channel for replaying entities.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Transmitter

serializers

List of component serializers to serialize entities.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>

directory

Directory path for storing files.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

batch_size

Number of entities to read and publish for one tick.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

ignore_corrupted_entities

If an entity could not be de-serialized, it is ignored by default; otherwise a failure is generated.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

nvidia::gxf::StdComponentSerializer

Serializer for Timestamp and Tensor components.

- Component ID: c0e6b36c-39ac-50ac-ce8d-702e18d8bff7
- Base Type: nvidia::gxf::ComponentSerializer

Parameters

allocator

Memory allocator for tensor components.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Allocator

31.11 StandardExtension

Most commonly used interfaces and components in Gxf Core.

- UUID: 8ec2d5d6-b5df-48bf-8dee-0252606fdd7e
- Version: 2.1.0
- Author: NVIDIA
- License: LICENSE

31.11.1 Interfaces

nvidia::gxf::Codelet

Interface for a component which can be executed to run custom code.

- Component ID: 5c6166fa-6eed-41e7-bbf0-bd48cd6e1014
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/codelet.hpp

nvidia::gxf::Clock

Interface for clock components which provide time.

- Component ID: 779e61c2-ae70-441d-a26c-8ca64b39f8e7
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/clock.hpp

nvidia::gxf::System

Component interface for systems which are run as part of the application run cycle.

- Component ID: d1febca1-80df-454e-a3f2-715f2b3c6e69
- Base Type: nvidia::gxf::Component

nvidia::gxf::Queue

Interface for storing entities in a queue.

- Component ID: 792151bf-3138-4603-a912-5ca91828dea8
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/queue.hpp

nvidia::gxf::Router

Interface for classes which are routing messages in and out of entities.

- Component ID: 8b317aad-f55c-4c07-8520-8f66db92a19e
- Defined in: gxf/std/router.hpp

nvidia::gxf::Transmitter

Interface for publishing entities.

- Component ID: c30cc60f-0db2-409d-92b6-b2db92e02cce
- Base Type: nvidia::gxf::Queue
- Defined in: gxf/std/transmitter.hpp

nvidia::gxf::Receiver

Interface for receiving entities.

- Component ID: a47d2f62-245f-40fc-90b7-5dc78ff2437e
- Base Type: nvidia::gxf::Queue
- Defined in: gxf/std/receiver.hpp

nvidia::gxf::Scheduler

A simple poll-based single-threaded scheduler which executes codelets.

- Component ID: f0103b75-d2e1-4d70-9b13-3fe5b40209be
- Base Type: nvidia::gxf::System
- Defined in: nvidia/gxf/system.hpp

nvidia::gxf::SchedulingTerm

Interface for terms used by a scheduler to determine if codelets in an entity are ready to step.

- Component ID: 184d8e4e-086c-475a-903a-69d723f95d19
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/scheduling_term.hpp

nvidia::gxf::Allocator

Provides allocation and deallocation of memory.

- Component ID: 3cdd82d0-2326-4867-8de2-d565dbe28e03
- Base Type: nvidia::gxf::Component
- Defined in: nvidia/gxf/allocator.hpp

nvidia::gxf::Monitor

Monitors entities during execution.

- Component ID: 9ccf9421-b35b-8c79-e1f0-97dc23bd38ea
- Base Type: nvidia::gxf::Component
- Defined in: nvidia/gxf/monitor.hpp

31.11.2 Components**nvidia::gxf::RealtimeClock**

A real-time clock which runs based off a system steady clock.

- Component ID: 7b170b7b-cf1a-4f3f-997c-bfea25342381
- Base Type: nvidia::gxf::Clock

Parameters**initial_time_offset**

The initial time offset used, until the time scale is changed manually.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_FLOAT64

initial_time_scale

The initial time scale used, until the time scale is changed manually.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_FLOAT64

use_time_since_epoch

If true, clock time is time since epoch + initial_time_offset at initialize(). Otherwise, clock time is initial_time_offset at initialize().

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

nvidia::gxf::ManualClock

A manual clock which is instrumented manually.

- Component ID: 52fa1f97-eba8-472a-a8ca-4cff1a2c440f
- Base Type: nvidia::gxf::Clock

Parameters

initial_timestamp

The initial timestamp on the clock (in nanoseconds).

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

nvidia::gxf::SystemGroup

A group of systems.

- Component ID: 3d23d470-0aed-41c6-ac92-685c1b5469a0
- Base Type: nvidia::gxf::System

nvidia::gxf::MessageRouter

A router which sends transmitted messages to receivers.

- Component ID: 84fd5d56-fda6-4937-0b3c-c283252553d8
- Base Type: nvidia::gxf::Router

nvidia::gxf::RouterGroup

A group of routers.

- Component ID: ca64ee14-2280-4099-9f10-d4b501e09117
- Base Type: nvidia::gxf::Router

nvidia::gxf::DoubleBufferTransmitter

A transmitter which uses a double-buffered queue where messages are pushed to a backstage after they are published.

- Component ID: 0c3c0ec7-77f1-4389-aef1-6bae85bddc13
- Base Type: nvidia::gxf::Transmitter

Parameters

capacity

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64
- Default: 1

policy

0: pop, 1: reject, 2: fault.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64
- Default: 2

nvidia::gxf::DoubleBufferReceiver

A receiver which uses a double-buffered queue where new messages are first pushed to a backstage.

- Component ID: ee45883d-bf84-4f99-8419-7c5e9deac6a5
- Base Type: nvidia::gxf::Receiver

Parameters

capacity

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64
- Default: 1

policy

0: pop, 1: reject, 2: fault

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64
- Default: 2

nvidia::gxf::Connection

A component which establishes a connection between two other components.

- Component ID: cc71afae-5ede-47e9-b267-60a5c750a89a
- Base Type: nvidia::gxf::Component

Parameters

source

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Transmitter

target

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

nvidia::gxf::PeriodicSchedulingTerm

A component which specifies that an entity shall be executed periodically.

- Component ID: d392c98a-9b08-49b4-a422-d5fe6cd72e3e
- Base Type: nvidia::gxf::SchedulingTerm

Parameters

recess_period

The recess period indicates the minimum amount of time which has to pass before the entity is permitted to execute again. The period is specified as a string containing of a number and an (optional) unit. If no unit is given, the value is assumed to be in nanoseconds. Supported units are: Hz, s, ms. Example: 10ms, 10000000, 0.2s, 50Hz.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

nvidia::gxf::CountSchedulingTerm

A component which specifies that an entity shall be executed exactly a given number of times.

- Component ID: f89da2e4-fddf-4aa2-9a80-1119ba3fde05
- Base Type: nvidia::gxf::SchedulingTerm

Parameters**count**

The total number of time this term will permit execution.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

nvidia::gxf::TargetTimeSchedulingTerm

A component where the next execution time of the entity needs to be specified after every tick.

- Component ID: e4aaf5c3-2b10-4c9a-c463-ebf6084149bf
- Base Type: nvidia::gxf::SchedulingTerm

Parameters**clock**

The clock used to define target time.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

nvidia::gxf::DownstreamReceptiveSchedulingTerm

A component which specifies that an entity shall be executed if receivers for a certain transmitter can accept new messages.

- Component ID: 9de75119-8d0f-4819-9a71-2aeaefd23f71
- Base Type: nvidia::gxf::SchedulingTerm

Parameters

min_size

The term permits execution if the receiver connected to the transmitter has at least the specified number of free slots in its back buffer.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

transmitter

The term permits execution if this transmitter can publish a message; i.e., if the receiver which is connected to this transmitter can receive messages.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: `nvidia::gxf::Transmitter`

nvidia::gxf::MessageAvailableSchedulingTerm

A scheduling term which specifies that an entity can be executed when the total number of messages over a set of input channels is at least a given number of messages.

- Component ID: `fe799e65-f78b-48eb-beb6-e73083a12d5b`
- Base Type: `nvidia::gxf::SchedulingTerm`

Parameters

front_stage_max_size

If set, the scheduling term will only allow execution if the number of messages in the front stage does not exceed this count. For example, it can be used in combination with codelets which do not clear the front stage in every tick.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_UINT64

min_size

The scheduling term permits execution if the given receiver has at least the given number of messages available.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

receiver

The scheduling term permits execution if this channel has at least a given number of messages available.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

nvidia::gxf::MultiMessageAvailableSchedulingTerm

A component which specifies that an entity shall be executed when a queue has at least a certain number of elements.

- Component ID: f15dbeaa-afd6-47a6-9ffc-7afd7e1b4c52
- Base Type: nvidia::gxf::SchedulingTerm

Parameters**min_size**

The scheduling term permits execution if all given receivers together have at least the given number of messages available.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

receivers

The scheduling term permits execution if the given channels have at least a given number of messages available.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

nvidia::gxf::ExpiringMessageAvailableSchedulingTerm

A component which tries to wait for specified number of messages in queue for at most specified time.

- Component ID: eb22280c-76ff-11eb-b341-cf6b417c95c9
- Base Type: nvidia::gxf::SchedulingTerm

Parameters

clock

Clock to get time from.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

max_batch_size

The maximum number of messages to be batched together.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

max_delay_ns

The maximum delay from first message to wait before submitting workload anyway.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

receiver

Receiver to watch on.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

nvidia::gxf::BooleanSchedulingTerm

A component which acts as a Boolean AND term that can be used to control the execution of the entity.

- Component ID: e07a0dc4-3908-4df8-8134-7ce38e60fbef
- Base Type: nvidia::gxf::SchedulingTerm

nvidia::gxf::AsynchronousSchedulingTerm

A component which is used to inform that an entity is dependent upon an async event for its execution.

- Component ID: 56be1662-ff63-4179-9200-3fcd8dc38673
- Base Type: nvidia::gxf::SchedulingTerm

nvidia::gxf::GreedyScheduler

A simple poll-based single-threaded scheduler which executes codelets.

- Component ID: 869d30ca-a443-4619-b988-7a52e657f39b
- Base Type: nvidia::gxf::Scheduler

Parameters**clock**

The clock used by the scheduler to define flow of time. Typical choices are a `RealtimeClock` or a `ManualClock`.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

max_duration_ms

The maximum duration for which the scheduler will execute (in ms). If not specified the scheduler will run until all work is done. If periodic terms are present, this means the application will run indefinitely.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_INT64

realtime

This parameter is deprecated. Assign a clock directly.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL

- Type: GXF_PARAMETER_TYPE_BOOL

stop_on_deadlock

If enabled, the scheduler will stop when all entities are in a waiting state, but no periodic entity exists to break the dead end. Should be disabled when scheduling conditions can be changed by external actors; for example, by clearing queues manually.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

nvidia::gxf::MultiThreadScheduler

A multi-thread scheduler that executes codelets for maximum throughput.

- Component ID: de5e0646-7fa5-11eb-a5c4-330ebfa81bbf
- Base Type: nvidia::gxf::Scheduler

Parameters**check_recession_perios_ms**

The maximum duration for which the scheduler would wait (in ms) when an entity is not ready to run yet.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

clock

The clock used by the scheduler to define flow of time. Typical choices are a `RealtimeClock` or a `ManualClock`.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

max_duration_ms

The maximum duration for which the scheduler will execute (in ms). If not specified, the scheduler will run until all work is done. If periodic terms are present, this means the application will run indefinitely.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_INT64

stop_on_deadlock

If enabled, the scheduler will stop when all entities are in a waiting state, but no periodic entity exists to break the dead end. Should be disabled when scheduling conditions can be changed by external actors; for example, by clearing queues manually.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

worker_thread_number

Number of threads.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64
- Default: 1

nvidia::gxf::BlockMemoryPool

A memory pool which provides a maximum number of equally sized blocks of memory.

- Component ID: 92b627a3-5dd3-4c3c-976c-4700e8a3b96a
- Base Type: nvidia::gxf::Allocator

Parameters**block_size**

The size of one block of memory in byte. Allocation requests can only be fulfilled if they fit into one block. If less memory is requested, a full block is still issued.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

do_not_use_cuda_malloc_host

If enabled operator new will be used to allocate host memory instead of cudaMallocHost.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL
- Default: True

num_blocks

The total number of blocks which are allocated by the pool. If more blocks are requested, allocation requests will fail.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

storage_type

The memory storage type used by this allocator. Can be kHost (0) or kDevice (1) or kSystem (2).

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default: 0

nvidia::gxf::UnboundedAllocator

Allocator that uses dynamic memory allocation without an upper bound.

- Component ID: c3951b16-a01c-539f-d87e-1dc18d911ea0
- Base Type: nvidia::gxf::Allocator

Parameters**do_not_use_cuda_malloc_host**

If enabled, a new operator will be used to allocate host memory instead of `cudaMallocHost`.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL
- Default: True

nvidia::gxf::Tensor

A component which holds a single tensor.

- Component ID: 377501d6-9abf-447c-a617-0114d4f33ab8
- Defined in: `gxf/std/tensor.hpp`

nvidia::gxf::Timestamp

Holds message publishing and acquisition related timing information.

- Component ID: d1095b10-5c90-4bbc-bc89-601134cb4e03
- Defined in: gxf/std/timestamp.hpp

nvidia::gxf::Metric

Collects, aggregates, and evaluates metric data.

- Component ID: f7cef803-5beb-46f1-186a-05d3919842ac
- Base Type: nvidia::gxf::Component

Parameters

aggregation_policy

Aggregation policy used to aggregate individual metric samples. Choices: {mean, min, max}.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_STRING

lower_threshold

Lower threshold of the metric's expected range.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_FLOAT64

upper_threshold

Upper threshold of the metric's expected range.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_FLOAT64

nvidia::gxf::JobStatistics

Collects runtime statistics.

- Component ID: 2093b91a-7c82-11eb-a92b-3f1304ecc959
- Base Type: nvidia::gxf::Component

Parameters

clock

The clock component instance to retrieve time from.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

codelet_statistics

If set to true, the JobStatistics component will collect performance statistics related to codelets.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_BOOL

json_file_path

If provided, all the collected performance statistics data will be dumped into a json file.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_STRING

nvidia::gxf::Broadcast

Messages arrived on the input channel are distributed to all transmitters.

- Component ID: 3daadb31-0bca-47e5-9924-342b9984a014
- Base Type: nvidia::gxf::Codelet

Parameters

mode

The broadcast mode. Can be Broadcast or RoundRobin.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM

source

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

nvidia::gxf::Gather

All messages arriving on any input channel are published on the single output channel.

- Component ID: 85f64c84-8236-4035-9b9a-3843a6a2026f
- Base Type: nvidia::gxf::Codelet

Parameters

sink

The output channel for gathered messages.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Transmitter

tick_source_limit

Maximum number of messages to take from each source in one tick. 0 means no limit.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

nvidia::gxf::TensorCopier

Copies tensor either from host to device or from device to host.

- Component ID: c07680f4-75b3-189b-8886-4b5e448e7bb6
- Base Type: nvidia::gxf::Codelet

Parameters

allocator

Memory allocator for tensor data.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Allocator

mode

Configuration to select what tensors to copy:

1. kCopyToDevice (0) - copies to device memory, ignores device allocation.
 2. kCopyToHost (1) - copies to pinned host memory, ignores host allocation.
 3. kCopyToSystem (2) - copies to system memory, ignores system allocation.
- Flags: GXF_PARAMETER_FLAGS_NONE
 - Type: GXF_PARAMETER_TYPE_INT32

receiver

Receiver for incoming entities.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

transmitter

Transmitter for outgoing entities.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Transmitter

nvidia::gxf::TimedThrottler

Publishes the received entity respecting the timestamp within the entity.

- Component ID: ccf7729c-f62c-4250-5cf7-f4f3ec80454b
- Base Type: nvidia::gxf::Codelet

Parameters

execution_clock

Clock on which the codelet is executed by the scheduler.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

receiver

Channel to receive messages that need to be synchronized.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

scheduling_term

Scheduling term for executing the codelet.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::TargetTimeSchedulingTerm

throttling_clock

Clock which the received entity timestamps are based on.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

transmitter

Transmitter channel publishing messages at appropriate timesteps.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Transmitter

nvidia::gxf::Vault

Safely stores received entities for further processing.

- Component ID: 1108cb8d-85e4-4303-ba02-d27406ee9e65
- Base Type: nvidia::gxf::Codelet

Parameters

drop_waiting

If too many messages are waiting, the oldest ones are dropped.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

max_waiting_count

The maximum number of waiting messages. If exceeded, the codelet will stop pulling messages out of the input queue.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

source

Receiver from which messages are taken and transferred to the vault.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

nvidia::gxf::Subgraph

Helper component to import a subgraph.

- Component ID: 576eedd7-7c3f-4d2f-8c38-8baa79a3d231
- Base Type: nvidia::gxf::Component

Parameters

location

Yaml source of the subgraph.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

nvidia::gxf::EndOfStream

A component which represents end-of-stream notification.

- Component ID: 8c42f7bf-7041-4626-9792-9eb20ce33cce
- Defined in: gxf/std/eos.hpp

nvidia::gxf::Synchronization

Component to synchronize messages from multiple receivers based on the `acq_time`.

- Component ID: f1cb80d6-e5ec-4dba-9f9e-b06b0def4443
- Base Type: nvidia::gxf::Codelet

Parameters

inputs

All the inputs for synchronization. Number of inputs must match that of the outputs.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

outputs

All the outputs for synchronization. Number of outputs must match that of the inputs.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Transmitter

signed char

- Component ID: 83905c6a-ca34-4f40-b474-cf2cde8274de

unsigned char

- Component ID: d4299e15-0006-d0bf-8cbd-9b743575e155

short int

- Component ID: 9e1dde79-3550-307d-e81a-b864890b3685

short unsigned int

- Component ID: 958cbdef-b505-bcc7-8a43-dc4b23f8cead

int

- Component ID: b557ec7f-49a5-08f7-a35e-086e9d1ea767

unsigned int

- Component ID: d5506b68-5c86-fedb-a2a2-a7bae38ff3ef

long int

- Component ID: c611627b-6393-365f-d234-1f26bfa8d28f

long unsigned int

- Component ID: c4385f5b-6e25-01d9-d7b5-6e7cad704e8

float

- Component ID: a81bf295-421f-49ef-f24a-f59e9ea0d5d6

double

- Component ID: d57cee59-686f-e26d-95be-659c126b02ea

bool

- Component ID: c02f9e93-d01b-1d29-f523-78d2a9195128

DATA FLOW TRACKING

Warning: Data Flow Tracking is currently only supported between multiple fragments in a *distributed application* in a single machine.

The Holoscan SDK provides the Data Flow Tracking APIs as a mechanism to profile your application and analyze the fine-grained timing properties and data flow between operators in the graph of a fragment.

Currently, data flow tracking is only supported between the root operators and leaf operators of a graph and in simple cycles in a graph (support for tracking data flow between any pair of operators in a graph is planned for the future).

- A *root operator* is an operator without any predecessor nodes.
- A *leaf operator* (also known as a *sink operator*) is an operator without any successor nodes.

When data flow tracking is enabled, every message is tracked from the root operators to the leaf operators and in cycles. Then, the maximum (worst-case), average, and minimum end-to-end latencies of one or more paths can be retrieved using the Data Flow Tracking APIs.

Tip:

- The end-to-end latency between a root operator and a leaf operator is the time taken between the start of a root operator and the end of a leaf operator. Data Flow Tracking enables the support to track the end-to-end latency of every message being passed between a root operator and a leaf operator.
- The reported end-to-end latency for a cyclic path is the time taken between the start of the first operator of a cycle and the time when a message is again received by the first operator of the cycle.

The API also provides the ability to retrieve the number of messages sent from the root operators.

Tip:

- The Data Flow Tracking feature is also illustrated in the [flow_tracker](#)
 - Look at the C++ and python API documentation for exhaustive definitions
-

32.1 Enabling Data Flow Tracking

Before an application (C++/python) is run with the `run()` method, data flow tracking can be enabled. For single fragment applications, this can be done by calling the `track()` method in C++ and using the `Tracker` class in python.

C++

```
auto app = holoscan::make_application<MyPingApp>();
auto& tracker = app->track(); // Enable Data Flow Tracking
// Change tracker and application configurations
...
app->run();
```

Python

```
from holoscan.core import Tracker
...
app = MyPingApp()
with Tracker(app) as tracker:
    # Change tracker and application configurations
    ...
    app.run()
```

32.2 Enabling Data Flow Tracking for Distributed Applications

For distributed (multi-fragment) applications, a separate tracker object is used for each Fragment so the API is slightly different than in the single fragment case.

C++

```
auto app = holoscan::make_application<MyPingApp>();
auto trackers = app->track_distributed(); // Enable data flow tracking for a distributed_
↪ app
// Change tracker and application configurations
...
app->run();
```

Note that instead of a returning a single `DataFlowTracker*` like `track`, the `track_distributed` method returns a `std::unordered_map<std::string, DataFlowTracker*>` where the keys are the names of the fragments.

Python

```
with Tracker(app) as trackers:
    app.run()
```

The Tracker context manager detects whether the app is distributed and returns a `dict[str, DataFlowTracker]` as `trackers` in the distributed case. For a single fragment application, the returned value is just a single `DataFlowTracker` object.

32.3 Retrieving Data Flow Tracking Results

After an application has been run, data flow tracking results can be accessed by various methods on the `DataFlowTracker` (C++/python) class.

1. `print()` (C++/python)

- Prints all data flow tracking results including end-to-end latencies and the number of source messages to the standard output.

2. `get_num_paths()` (C++/python)

- Returns the number of paths between the root operators and the leaf operators.

3. `get_path_strings()` (C++/python)

- Returns a vector of strings, where each string represents a path between the root operators and the leaf operators. A path is a comma-separated list of operator names.

4. `get_metric()` (C++/python)

- Returns the value of different metrics based on the arguments.
- `get_metric(std::string pathstring, holoscan::DataFlowMetric metric)` returns the value of a metric `metric` for a path `pathstring`. The metric can be one of the following:
 - `holoscan::DataFlowMetric::kMaxE2ELatency` (python): the maximum end-to-end latency in the path.
 - `holoscan::DataFlowMetric::kAvgE2ELatency` (python): the average end-to-end latency in the path.
 - `holoscan::DataFlowMetric::kMinE2ELatency` (python): the minimum end-to-end latency in the path.
 - `holoscan::DataFlowMetric::kMaxMessageID` (python): the message number or ID which resulted in the maximum end-to-end latency.
 - `holoscan::DataFlowMetric::kMinMessageID` (python): the message number or ID which resulted in the minimum end-to-end latency.
- `get_metric(holoscan::DataFlowMetric metric = DataFlowMetric::kNumSrcMessages)` returns a map of source operator and its edge, and the number of messages sent from the source operator to the edge.

In the above example, the data flow tracking results can be printed to the standard output like the following:

C++

```
auto app = holoscan::make_application<MyPingApp>();
auto& tracker = app->track(); // Enable Data Flow Tracking
// Change application configurations
...
app->run();
tracker.print();
```

Python

```
from holoscan.core import Tracker
...
app = MyPingApp()
with Tracker(app) as trackers:
    # Change tracker and application configurations
    ...
app.run()
tracker.print()
```

If this was a distributed application, there would instead be a separate `DataFlowTracker` for each fragment. The overall flow tracking results for all fragments can be printed as in the following:

C++

```
auto app = holoscan::make_application<MyPingApp>();
auto trackers = app->track_distributed(); // Enable data flow tracking for a distributed...
↪ app
// Change application configurations
...
app->run();
// print the data flow tracking results
for (const auto& [name, tracker] : trackers) {
    std::cout << "Fragment: " << name << std::endl;
    tracker->print();
}
```

Python

```
from holoscan.core import Tracker
...
app = MyPingApp()
with Tracker(app) as trackers:
    # Change tracker and application configurations
    ...
app.run()
# print the data flow tracking results
for fragment_name, tracker in trackers.items():
```

(continues on next page)

(continued from previous page)

```
print(f"Fragment: {fragment_name}")
tracker.print()
```

32.4 Customizing Data Flow Tracking

Data flow tracking can be customized using a few, optional configuration parameters. The `track()` method (C++/Python) (or `track_distributed` method (C++/Python) for distributed apps) can be configured to skip a few messages at the beginning of an application's execution as a *warm-up* period. It is also possible to discard a few messages at the end of an application's run as a *wrap-up* period. Additionally, outlier end-to-end latencies can be ignored by setting a latency threshold value (in ms) which is the minimum latency below which the observed latencies are ignored.

For Python, it is recommended to use the `Tracker` context manager class instead of the `track` or `track_distributed` methods. This class will autodetect if the application is a single fragment or distributed app, using the appropriate method for each.

Tip: For effective benchmarking, it is common practice to include warm-up and cool-down periods by skipping the initial and final messages.

C++

Listing 32.1: Optional parameters to `track()`

```
Fragment::track(uint64_t num_start_messages_to_skip = kDefaultNumStartMessagesToSkip,
                uint64_t num_last_messages_to_discard =
↳ kDefaultNumLastMessagesToDiscard,
                int latency_threshold = kDefaultLatencyThreshold);
```

Python

Listing 32.2: Optional parameters to `Tracker`

```
Tracker(num_start_messages_to_skip=num_start_messages_to_skip,
        num_last_messages_to_discard=num_last_messages_to_discard,
        latency_threshold=latency_threshold)
```

The default values of these parameters of `track()` are as follows:

- `kDefaultNumStartMessagesToSkip`: 10
- `kDefaultNumLastMessagesToDiscard`: 10
- `kDefaultLatencyThreshold`: 0 (do not filter out any latency values)

These parameters can also be configured using the helper functions: `set_skip_starting_messages`, `set_discard_last_messages` and `set_skip_latencies`.

32.5 Logging

The Data Flow Tracking API provides the ability to log every message's graph-traversal information to a file. This enables you to analyze the data flow at a granular level. When logging is enabled, every message's received and sent timestamps at every operator between the root and the leaf operators are logged after a message has been processed at the leaf operator.

The logging is enabled by calling the `enable_logging` method in C++ and by providing the `filename` parameter to `Tracker` in python.

C++

```
auto app = holoscan::make_application<MyPingApp>();
auto& tracker = app->track(); // Enable Data Flow Tracking
tracker.enable_logging("logging_file_name.log");
...
app->run();
```

Python

```
from holoscan.core import Tracker
...
app = MyPingApp()
with Tracker(app, filename="logger.log") as tracker:
    ...
    app.run()
```

The logger file logs the paths of the messages after a leaf operator has finished its `compute` method. Every path in the logfile includes an array of tuples of the form:

“(root operator name, message receive timestamp, message publish timestamp) -> ... -> (leaf operator name, message receive timestamp, message publish timestamp)”.

This log file can further be analyzed to understand latency distributions, bottlenecks, data flow, and other characteristics of an application.

GXF JOB STATISTICS

Holoscan can have the underlying graph execution framework (GXF) collect job statistics during application execution. Collection of these statistics causes a small amount of runtime overhead, so they are disabled by default, but can be enabled on request via the environment variables documented below. The job statistics will appear in the console on application shutdown, but can optionally also be saved to a JSON file.

The statistics collected via this method correspond to individual entities (operators) in isolation. To track execution times along specific paths through the computation graph, see the documentation on [flow tracking](#) instead.

Note: The job statistics will be collected by the underlying Graph Execution Framework (GXF) runtime. Given that, the terms used in the report correspond to GXF concepts (entity and codelet) rather than Holoscan classes.

From the GXF perspective, each Holoscan Operator is a unique entity which contains a single codelet as well as its associated components (corresponding to Holoscan Condition or Resource classes). Any additional entities and codelets that get implicitly created by Holoscan will also appear in the report. For example, if an output port of an operator connects to multiple downstream operators, you will see a corresponding implicit “broadcast” codelet appearing in the report).

33.1 Holoscan SDK environment variables related to GXF job statistics

Collection of GXF job statistics can be enabled by setting `HOLOSCAN_ENABLE_GXF_JOB_STATISTICS`.

- **HOLOSCAN_ENABLE_GXF_JOB_STATISTICS** : Determines if job statistics should be collected. Interprets values like “true”, “1”, or “on” (case-insensitive) as true (to enable job statistics). It defaults to false if left unspecified.
- **HOLOSCAN_GXF_JOB_STATISTICS_CODELET** : Determines if a codelet statistics summary table should be created in addition to the entity statistics. Interprets values like “true”, “1”, or “on” (case-insensitive) as true (to enable codelet statistics). It defaults to false if left unspecified.
- **HOLOSCAN_GXF_JOB_STATISTICS_COUNT** : Count of the number of events to be maintained in history per entity. Statistics such as median and maximum correspond to a history of this length. If unspecified, it defaults to 100.
- **HOLOSCAN_GXF_JOB_STATISTICS_PATH** : Output JSON file name where statistics should be stored. The default if unspecified (or given an empty string) is to output the statistics only to the console. Statistics will still be shown in the console when a file path is specified.

NSIGHT SYSTEMS PROFILING

The Holoscan SDK has been annotated using the [NVTX](#) API to provide runtime tracing and profiling of key application calls such as the `start`, `compute`, and `stop` callbacks made to the operators used by the application. This profiling can be captured and visualized using the tools provided by [NSight Systems](#).

To enable profiling and output the profile results of running an application, enable the `HOLOSCAN_ENABLE_PROFILE` environment variable and use the `nsys` runtime provided with NSight Systems to run the application. For example, the following command will profile the first 3 seconds of the `bring_your_own_model` example application and write the results to `byom_profile.nsys-rep`:

```
export HOLOSCAN_ENABLE_PROFILE=1
nsys profile -t cuda,nvtx,osrt -o byom_profile -f true -d 3 python3 ./examples/bring_
↪your_own_model/python/byom.py
```

The written profile can then be opened with the NSight Systems UI (`nsys-ui`) to visualize the results. This is a sample profile of the `bring your own model` example application, zoomed in to show the details of the CPU and CUDA runtime of the application's operators:

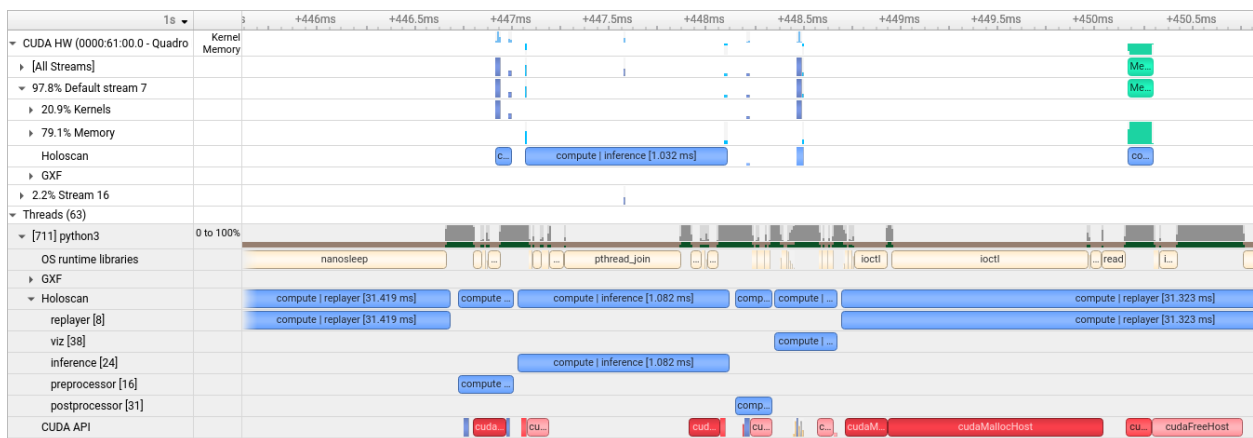


Fig. 34.1: Sample profile of the `bring_your_own_model` example application

VIDEO PIPELINE LATENCY TOOL

The NVIDIA Developer Kits excel as a high-performance computing platform by combining high-bandwidth video I/O components and the compute capabilities of an NVIDIA GPU to meet the needs of the most demanding video processing and inference applications.

For many video processing applications located at the edge—especially those designed to augment medical instruments and aid live medical procedures—minimizing the latency added between image capture and display, often referred to as the end-to-end latency, is of the utmost importance.

While it is generally easy to measure the individual processing time of an isolated compute or inference algorithm by simply measuring the time that it takes for a single frame (or a sequence of frames) to be processed, it is not always so easy to measure the complete end-to-end latency when the video capture and display is incorporated as this usually involves external capture hardware (e.g., cameras and other sensors) and displays.

In order to establish a baseline measurement of the minimal end-to-end latency that can be achieved with the NVIDIA Developer Kits and various video I/O hardware and software components, the Holoscan SDK includes a sample latency measurement tool.

35.1 Requirements

35.1.1 Hardware

The latency measurement tool requires the use of a NVIDIA Developer Kit in dGPU mode, and operates by having an output component generate a sequence of known video frames that are then transferred back to an input component using a physical loopback cable.

Testing the latency of any of the HDMI modes that output from the GPU requires a DisplayPort to HDMI adapter or cable (see *Example Configurations*, below). Note that this cable must support the mode that is being tested; for example, the UHD mode will only be available if the cable is advertised to support “4K Ultra HD (3840 x 2160) at 60 Hz.”

Testing the latency of an optional AJA Video Systems device requires a supported AJA SDI or HDMI capture device (see *AJA Video Systems* for the list of supported devices), along with the HDMI or SDI cable that is required for the configuration that is being tested (see *Example Configurations*, below).

35.1.2 Software

The following additional software components are required and are installed either by the Holoscan SDK installation or in the *Installation* steps below:

- CUDA 11.1 or newer (<https://developer.nvidia.com/cuda-toolkit>)
- CMake 3.10 or newer (<https://cmake.org/>)
- GLFW 3.2 or newer (<https://www.glfw.org/>)
- GStreamer 1.14 or newer (<https://gstreamer.freedesktop.org/>)
- GTK 3.22 or newer (<https://www.gtk.org/>)
- pkg-config 0.29 or newer (<https://www.freedesktop.org/wiki/Software/pkg-config/>)

The following is optional to enable DeepStream support (for RDMA support from the *GStreamer Producer*):

- DeepStream 5.1 or newer (<https://developer.nvidia.com/deepstream-sdk>)

The following is optional to enable AJA Video Systems support:

- AJA NTV2 SDK 16.1 or newer (See *AJA Video Systems* for details on installing the AJA NTV2 SDK and drivers).

35.2 Installation

35.2.1 Downloading the Source

The Video Pipeline Latency Tool can be found in the `loopback-latency` folder of the *Holoscan Performance Tools* GitHub repository, which is cloned with the following:

```
$ git clone https://github.com/nvidia-holoscan/holoscan-perf-tools.git
```

35.2.2 Installing Software Requirements

CUDA is installed automatically during the dGPU setup. The rest of the software requirements are installed with the following:

```
$ sudo apt-get update && sudo apt-get install -y \  
  cmake \  
  libglfw3-dev \  
  libgstreamer1.0-dev \  
  libgstreamer-plugins-base1.0-dev \  
  libgtk-3-dev \  
  pkg-config
```

35.2.3 Building

Start by creating a build folder within the `loopback-latency` directory:

```
$ cd clara-holoscan-perf-tools/loopback-latency
$ mkdir build
$ cd build
```

CMake is then used to build the tool and output the `loopback-latency` binary to the current directory:

```
$ cmake ..
$ make -j
```

Note: If the error `No CMAKE_CUDA_COMPILER could be found` is encountered, make sure that the `nvcc` executable can be found by adding the CUDA runtime location to your `PATH` variable.

```
$ export PATH=$PATH:/usr/local/cuda/bin
```

Enabling DeepStream Support

DeepStream support enables RDMA when using the *GStreamer Producer*. To enable DeepStream support, the `DEEPSTREAM_SDK` path must be appended to the `cmake` command with the location of the DeepStream SDK. For example, when building against DeepStream 5.1, replace the `cmake` command above with the following:

```
$ cmake -DDEEPSTREAM_SDK=/opt/nvidia/deepstream/deepstream-5.1 ..
```

Enabling AJA Support

To enable AJA support, the `NTV2_SDK` path must be appended to the `cmake` command with the location of the NTV2 SDK in which both the headers and compiled libraries (i.e., `libajantv2`) exist. For example, if the NTV2 SDK is in `/home/nvidia/ntv2`, replace the `cmake` command above with the following:

```
$ cmake -DNTV2_SDK=/home/nvidia/ntv2 ..
```

35.3 Example Configurations

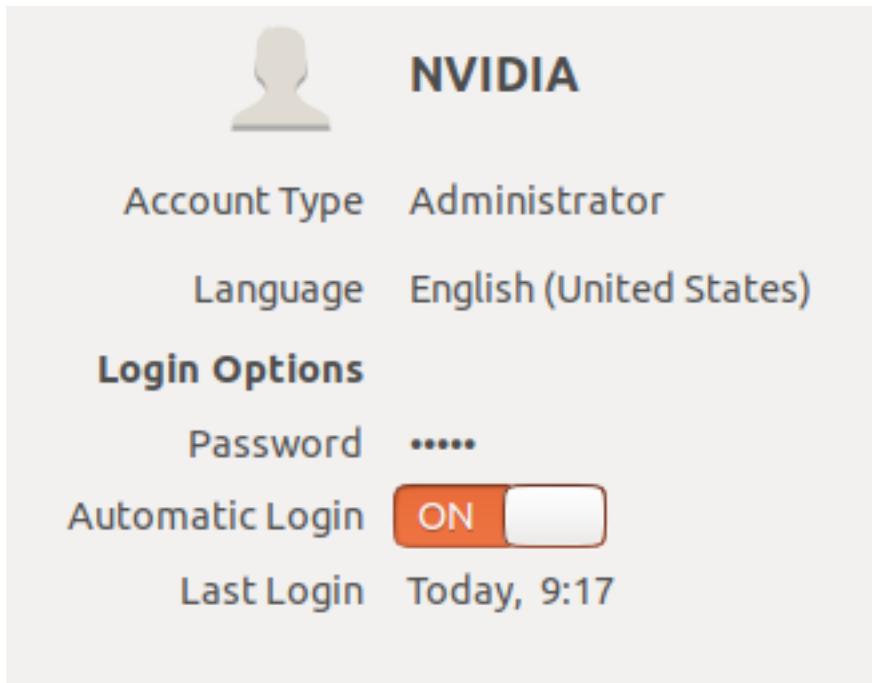
Note: When testing a configuration that outputs from the GPU, the tool currently only supports a display-less environment in which the loopback cable is the only cable attached to the GPU. Because of this, any tests that output from the GPU must be performed using a remote connection such as SSH from another machine. When this is the case, make sure that the `DISPLAY` environment variable is set to the ID of the X11 display you are using (e.g., in `~/ .bashrc`):

```
export DISPLAY=:0
```

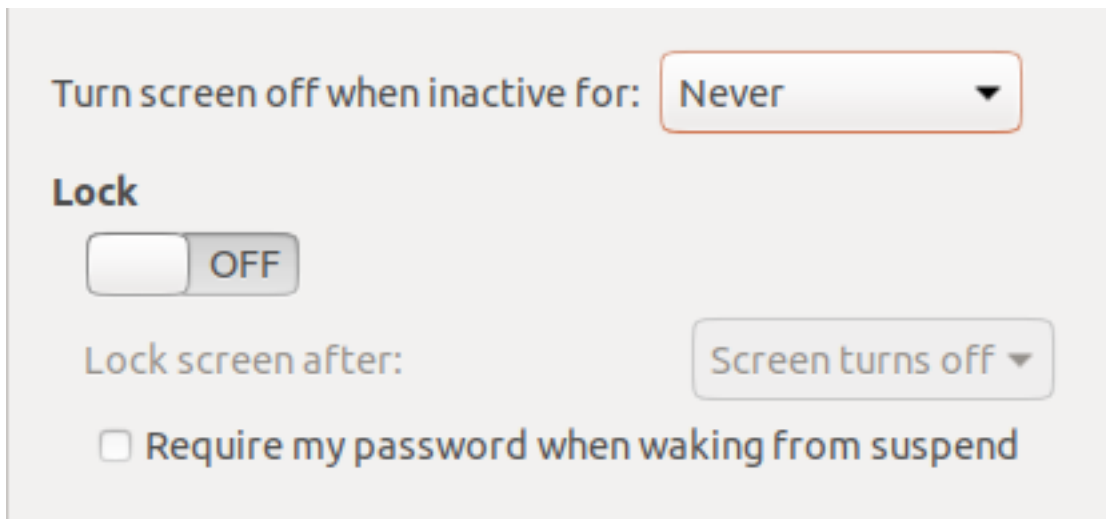
It is also required that the system is logged into the desktop and that the system does not sleep or lock when the latency tool is being used. This can be done by temporarily attaching a display to the system to do the following:

1. Open the **Ubuntu System Settings**

2. Open **User Accounts**, click **Unlock** at the top right, and enable **Automatic Login**:



3. Return to **All Settings** (top left), open **Brightness & Lock**, and disable sleep and lock as pictured:



Make sure that the display is detached again after making these changes.

See the [Producers](#) section for more details about GPU-based producers (i.e., [OpenGL](#) and [GStreamer](#)).

35.3.1 GPU To Onboard HDMI Capture Card

In this configuration, a DisplayPort to HDMI cable is connected from the GPU to the onboard HDMI capture card. This configuration supports the *OpenGL* and *GStreamer* producers, and the *V4L2* and *GStreamer* consumers.



Fig. 35.1: DP-to-HDMI Cable Between GPU and Onboard HDMI Capture Card

For example, an *OpenGL producer* to *V4L2 consumer* can be measured using this configuration and the following command:

```
$ ./loopback-latency -p gl -c v4l2
```

35.3.2 GPU to AJA HDMI Capture Card

In this configuration, a DisplayPort to HDMI cable is connected from the GPU to an HDMI input channel on an AJA capture card. This configuration supports the *OpenGL* and *GStreamer* producers, and the *AJA consumer* using an AJA HDMI capture card.



Fig. 35.2: DP-to-HDMI Cable Between GPU and AJA KONA HDMI Capture Card (Channel 1)

For example, an *OpenGL producer* to *AJA consumer* can be measured using this configuration and the following command:

```
$ ./loopback-latency -p gl -c aja -c.device 0 -c.channel 1
```

35.3.3 AJA SDI to AJA SDI

In this configuration, an SDI cable is attached between either two channels on the same device or between two separate devices (pictured is a loopback between two channels of a single device). This configuration must use the *AJA producer* and *AJA consumer*.

For example, the following can be used to measure the pictured configuration using a single device with a loopback between channels 1 and 2. Note that the tool defaults to use channel 1 for the producer and channel 2 for the consumer, so the `channel` parameters can be omitted.

```
$ ./loopback-latency -p aja -c aja
```

If instead there are two AJA devices being connected, the following can be used to measure a configuration in which they are both connected to channel 1:



Fig. 35.3: SDI Cable Between Channel 1 and 2 of a Single AJA Corvid 44 Capture Card

```
$ ./loopback-latency -p aja -p.device 0 -p.channel 1 -c aja -c.device 1 -c.  
channel 1
```

35.4 Operation Overview

The latency measurement tool operates by having a **producer** component generate a sequence of known video frames that are output and then transferred back to an input **consumer** component using a physical loopback cable. Timestamps are compared throughout the life of the frame to measure the overall latency that the frame sees during this process, and these results are summarized when all of the frames have been received and the measurement completes. See [Producers](#), [Consumers](#), and [Example Configurations](#) for more details.

35.4.1 Frame Measurements

Each frame that is generated by the tool goes through the following steps in order, each of which has its time measured and then reported when all frames complete.

1. CUDA Processing

In order to simulate a real-world GPU workload, the tool first runs a CUDA kernel for a user-specified amount of loops (defaults to zero). This step is described below in [Simulating GPU Workload](#).

2. Render on GPU

After optionally simulating a GPU workload, every producer then generates its frames using the GPU, either by a common CUDA kernel or by another method that is available to the producer's API (such as the OpenGL

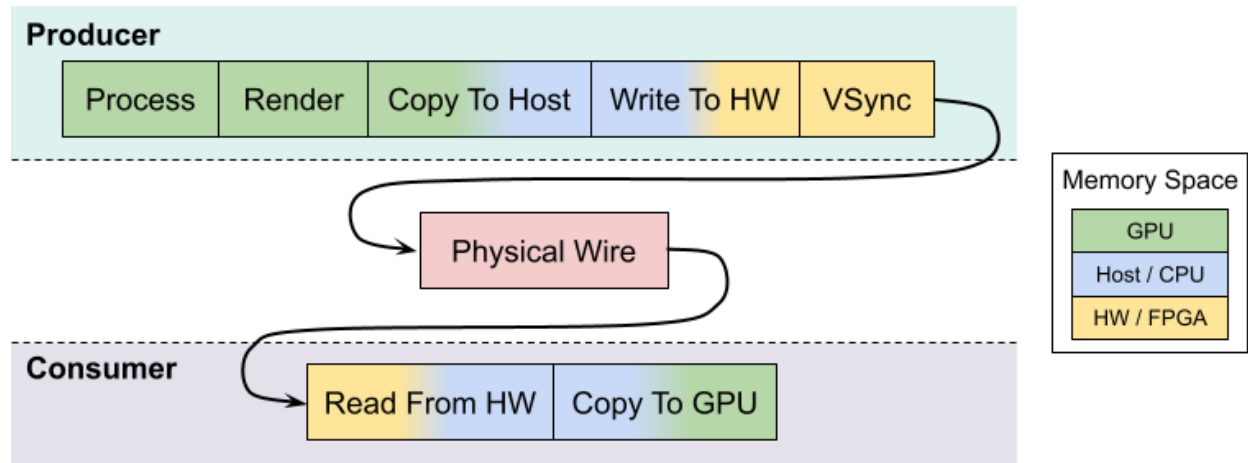


Fig. 35.4: Latency Tool Frame Lifespan (RDMA Disabled)

producer).

This step is expected to be very fast (<100us), but higher times may be seen if overall system load is high.

3. Copy To Host

Once the frame has been generated on the GPU, it may be necessary to copy the frame to host memory in order for the frame to be output by the producer component (for example, an AJA producer with RDMA disabled).

If a host copy is not required (i.e., RDMA is enabled for the producer), this time should be zero.

4. Write to HW

Some producer components require frames to be copied to peripheral memory before they can be output (for example, an AJA producer requires frames to be copied to the external frame stores on the AJA device). This copy may originate from host memory if RDMA is disabled for the producer, or from GPU memory if RDMA is enabled.

If this copy is not required (e.g., the producer outputs directly from the GPU), this time should be zero.

5. VSync Wait

Once the frame is ready to be output, the producer hardware must wait for the next VSync interval before the frame can be output.

The sum of this VSync wait and all of the preceding steps is expected to be near a multiple of the frame interval. For example, if the frame rate is 60Hz then the sum of the times for steps 1 through 5 should be near a multiple of 16666us.

6. Wire Time

The wire time is the amount of time that it takes for the frame to transfer across the physical loopback cable. This should be near the time for a single frame interval.

7. Read From HW

Once the frame has been transferred across the wire and is available to the consumer, some consumer components require frames to be copied from peripheral memory into host (RDMA disabled) or GPU (RDMA enable) memory. For example, an AJA consumer requires frames to be copied from the external frame store of the AJA device.

If this copy is not required (e.g., the consumer component writes received frames directly to host/GPU memory), this time should be zero.

8. Copy to GPU

If the consumer received the frame into host memory, the final step required for processing the frame with the GPU is to copy the frame into GPU memory.

If RDMA is enabled for the consumer and the frame was previously written directly to GPU memory, this time should be zero.

Note that if RDMA is enabled on the producer and consumer sides then the GPU/host copy steps above, 3 and 8 respectively, are effectively removed since RDMA will copy directly between the video HW and the GPU. The following shows the same diagram as above but with RDMA enabled for both the producer and consumer.

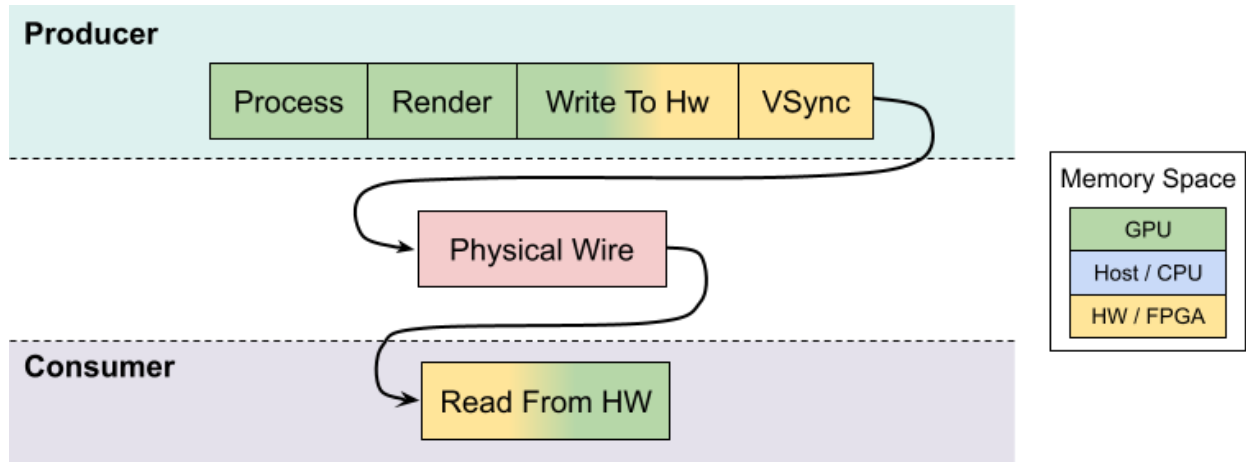


Fig. 35.5: Latency Tool Frame Lifespan (RDMA Enabled)

35.4.2 Interpreting The Results

The following shows example output of the above measurements from the tool when testing a 4K stream at 60Hz from an AJA producer to an AJA consumer, both with RDMA disabled, and no GPU/CUDA workload simulation. Note that all time values are given in microseconds.

```
$ ./loopback-latency -p aja -p.rdma 0 -c aja -c.rdma 0 -f 4k
```

```

Format: 4096x2160 RGBA @ 60Hz

Producer: AJA
  Device: 0
  Channel: NTV2_CHANNEL1
  RDMA: 0

Consumer: AJA
  Device: 0
  Channel: NTV2_CHANNEL2
  RDMA: 0

Measuring 600 frames...Done!

CUDA Processing: avg =      0, min =      0, max =      64
Render on GPU:   avg =    144, min =     94, max =    386
Copy To Host:    avg =   5788, min =   4145, max =   7024
Write To HW:     avg =   9468, min =   8219, max =   9916
Vsync Wait:      avg =   1245, min =    126, max =   2608
Wire Time:       avg =  16745, min =  16547, max =  17379
Read From HW:    avg =   7086, min =   6983, max =   7357
Copy To GPU:     avg =   4282, min =   3805, max =   6304
=====
Total:           avg =  44764, min =  44122, max =  46680

```

While this tool measures the producer times followed by the consumer times, the expectation for real-world video processing applications is that this order would be reversed. That is to say, the expectation for a real-world application is that it would capture, process, and output frames in the following order (with the component responsible for measuring that time within this tool given in parentheses):

1. **Read from Hardware** (consumer)
2. **Copy to GPU** (consumer)
3. **Process Frame** (producer)
4. **Render Results to GPU** (producer)
5. **Copy to Host** (producer)
6. **Write to Hardware** (producer)

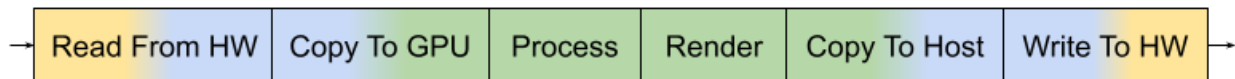


Fig. 35.6: Real Application Frame Lifespan

To illustrate this, the tool sums and displays the total producer and consumer times, then provides the **Estimated Application Times** as the total sum of all of these steps (i.e., steps 1 through 6, above).

(continued from above)

```

Producer (Process and Write to HW)
=====
Microseconds: avg = 15403, min = 14074, max = 16495
Frames: avg = 0.924, min = 0.844, max = 0.99

Consumer (Read from HW and Copy to GPU)
=====
Microseconds: avg = 11369, min = 10856, max = 13381
Frames: avg = 0.682, min = 0.651, max = 0.803

Estimated Application Times (Read + Process + Write)
=====
Microseconds: avg = 26772, min = 25101, max = 29204
Frames: avg = 1.61, min = 1.51, max = 1.75

```

Once a real-world application captures, processes, and outputs a frame, it would still be required that this final output waits for the next VSync interval before it is actually sent across the physical wire to the display hardware. Using this assumption, the tool then estimates one final value for the **Final Estimated Latencies** by doing the following:

1. Take the **Estimated Application Time** (from above)
2. Round it up to the next VSync interval
3. Add the physical wire time (i.e., a frame interval)

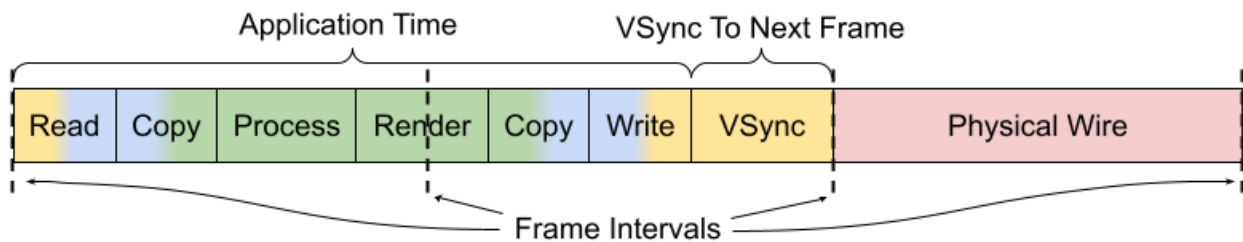


Fig. 35.7: Final Estimated Latency with VSync and Physical Wire Time

Continuing this example using a frame interval of 16666us (60Hz), this means that the average **Final Estimated Latency** is determined by:

1. Average application time = **26772**
2. Round up to next VSync interval = **33332**
3. Add physical wire time (+16666) = **49998**

These times are also reported as a multiple of frame intervals.

(continued from above)

```

Final Estimated Latencies (Processing + Vsync + Wire)
=====
Microseconds: avg = 49998, min = 49998, max = 49998
Frames: avg = 3, min = 3, max = 3

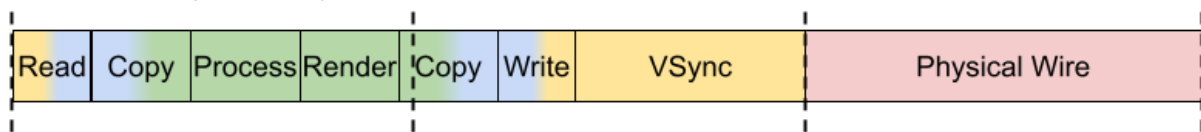
```

Using this example, we should then expect that the total end-to-end latency that is seen by running this pipeline using these components and configuration is 3 frame intervals (49998us).

35.4.3 Reducing Latency With RDMA

The previous example uses an AJA producer and consumer for a 4K @ 60Hz stream; however, RDMA was disabled for both components. Because of this, the additional copies between the GPU and host memory added more than 10000us of latency to the pipeline, causing the application to exceed one frame interval of processing time per frame and therefore a total frame latency of 3 frames. If RDMA is enabled, these GPU and host copies can be avoided so the processing latency is reduced by more than 10000us. More importantly, however, this also allows the total processing time to fit within a single frame interval so that the total end-to-end latency can be reduced to just 2 frames.

RDMA Disabled (3 Frames)



RDMA Enabled (2 Frames)

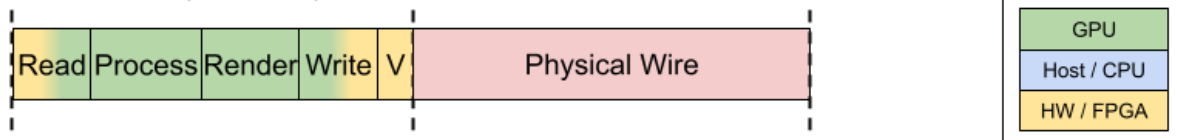


Fig. 35.8: Reducing Latency With RDMA

The following shows the above example repeated with RDMA enabled.

```
$ ./loopback-latency -p aja -p.rdma 1 -c aja -c.rdma 1 -f 4k
```

```

Format: 4096x2160 RGBA @ 60Hz

Producer: AJA
  Device: 0
  Channel: NTV2_CHANNEL1
  RDMA: 1

Consumer: AJA
  Device: 0
  Channel: NTV2_CHANNEL2
  RDMA: 1

Measuring 600 frames...Done!

CUDA Processing: avg =      0, min =      0, max =      74
Render on GPU:   avg =    122, min =     94, max =    356
Copy To Host:    avg =      0, min =      0, max =     35
Write To HW:     avg =   8209, min =   7453, max =   8856
Vsync Wait:     avg =   8314, min =   6338, max =  10036
Wire Time:       avg =  16650, min =  14814, max =  18391
Read From HW:    avg =   6041, min =   5962, max =   6931
Copy To GPU:     avg =      0, min =      0, max =     30
=====
Total:           avg =  39343, min =  37668, max =  41081

Producer (Process and Write to HW)
=====
  Microseconds: avg =   8334, min =   7580, max =   8988
    Frames: avg =    0.5, min =   0.455, max =   0.539

Consumer (Read from HW and Copy to GPU)
=====
  Microseconds: avg =   6042, min =   5962, max =   6932
    Frames: avg =   0.363, min =   0.358, max =   0.416

Estimated Application Times (Read + Process + Write)
=====
  Microseconds: avg =  14377, min =  13627, max =  15233
    Frames: avg =   0.863, min =   0.818, max =   0.914

Final Estimated Latencies (Processing + Vsync + Wire)
=====
  Microseconds: avg =  33332, min =  33332, max =  33332
    Frames: avg =      2, min =      2, max =      2

```

35.4.4 Simulating GPU Workload

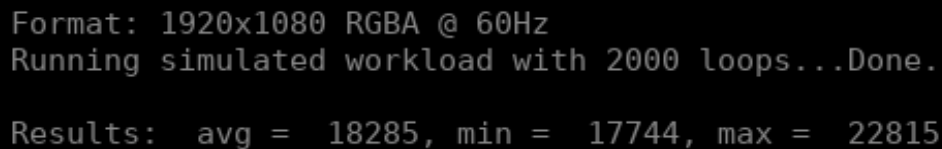
By default, the tool measures what is essentially a pass-through video pipeline; that is, no processing of the video frames is performed by the system. While this is useful for measuring the minimum latency that can be achieved by the video input and output components, it's not very indicative of a real-world use case in which the GPU is used for compute-intensive processing operations on the video frames between the input and output (for example, an object detection algorithm that applies an overlay to the output frames).

While it may be relatively simple to measure the runtime latency of the processing algorithms that are to be applied to the video frames – by simply measuring the runtime of running the algorithm on a single or stream of frames – this may not be indicative of the effects that such processing might have on the overall system load, which may further increase the latency of the video input and output components.

In order to estimate the total latency when an additional GPU workload is added to the system, the latency tool has an `-s {count}` option that can be used to run an arbitrary CUDA loop the specified number of times before the producer actually generates a frame. The expected usage for this option is as follows:

1. The per-frame runtime of the actual GPU processing algorithm is measured outside of the latency measurement tool.
2. The latency tool is repeatedly run with just the `-s {count}` option, adjusting the `{count}` parameter until the time that it takes to run the simulated loop approximately matches the actual processing time that was measured in the previous step.

```
$ ./loopback-latency -s 2000
```



```
Format: 1920x1080 RGBA @ 60Hz
Running simulated workload with 2000 loops...Done.

Results:  avg = 18285, min = 17744, max = 22815
```

3. The latency tool is run with the full producer (`-p`) and consumer (`-c`) options used for the video I/O, along with the `-s {count}` option using the loop count that was determined in the previous step.

Note: The following example shows that approximately half of the frames received by the consumer were duplicate/repeated frames. This is due to the fact that the additional processing latency of the producer causes it to exceed a single frame interval, and so the producer is only able to output a new frame every second frame interval.

```
$ ./loopback-latency -p aja -c aja -s 2000
```



```

Format: 1920x1080 RGBA @ 60Hz

Producer: AJA
  Device: 0
  Channel: NTV2_CHANNEL1
  RDMA: 1

Consumer: AJA
  Device: 0
  Channel: NTV2_CHANNEL2
  RDMA: 1

Simulating processing with 2000 CUDA loops per frame.

Measuring 600 frames...Done!

WARNING: Frames were skipped or repeated!
Frames received: 301
Frames skipped: 0
Frames repeated: 299

CUDA Processing: avg = 17153, min = 16877, max = 17569
Render on GPU:   avg = 50, min = 34, max = 116
Copy To Host:    avg = 0, min = 0, max = 19
Write To HW:     avg = 1785, min = 1721, max = 1849
Vsync Wait:      avg = 14321, min = 13782, max = 14718
Wire Time:       avg = 16723, min = 16360, max = 33470
Read From HW:    avg = 1502, min = 1442, max = 1726
Copy To GPU:     avg = 0, min = 0, max = 0
=====
Total:           avg = 51541, min = 51164, max = 68238

Producer (Process and Write to HW)
=====
  Microseconds: avg = 18991, min = 18689, max = 19405
  Frames: avg = 1.14, min = 1.12, max = 1.16

Consumer (Read from HW and Copy to GPU)
=====
  Microseconds: avg = 1502, min = 1443, max = 1726
  Frames: avg = 0.0901, min = 0.0866, max = 0.104

Estimated Application Times (Read + Process + Write)
=====
  Microseconds: avg = 20493, min = 20191, max = 20967
  Frames: avg = 1.23, min = 1.21, max = 1.26

Final Estimated Latencies (Processing + Vsync + Wire)
=====
  Microseconds: avg = 49998, min = 49998, max = 49998
  Frames: avg = 3, min = 3, max = 3

WARNING: Frames were skipped or repeated. These times only
include frames that were actually received, and the times
include only the first instance each frame was received.

```

Tip: To get the most accurate estimation of the latency that would be seen by a real world application, the best thing to do would be to run the actual frame processing algorithm used by the application during the latency measurement. This could be done by modifying the `SimulateProcessing` function in the latency tool source code.

35.5 Graphing Results

The latency tool includes a `-o {file}` option that can be used to output a CSV file with all of the measured times for every frame. This file can then be used with the `graph_results.py` script that is included with the tool in order to generate a graph of the measurements.

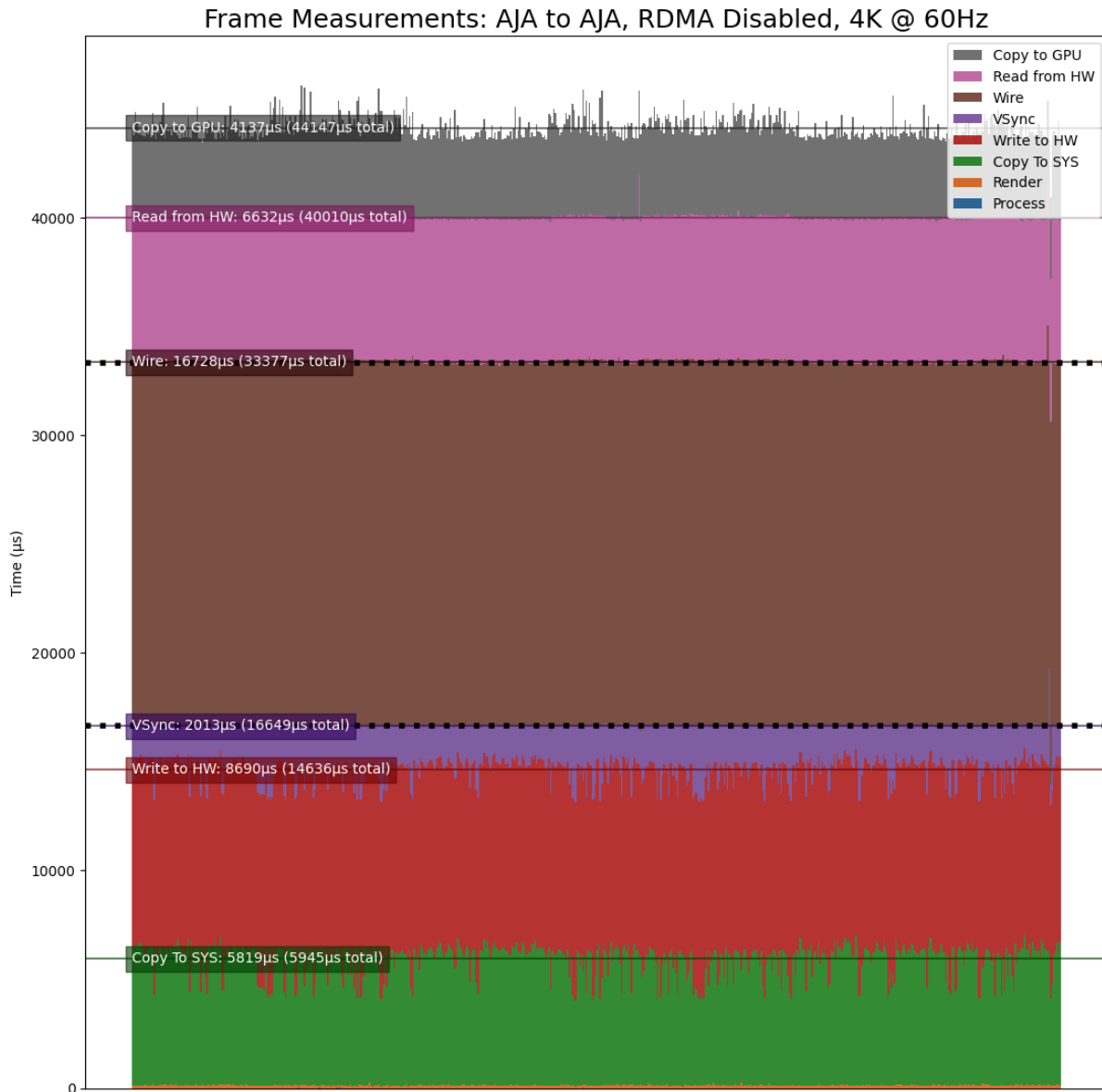
For example, if the latencies are measured using:

```
$ ./loopback-latency -p aja -c aja -o latencies.csv
```

The graph can then be generated using the following, which will open a window on the desktop to display the graph:

```
$ ./graph_results.py --file latencies.csv
```

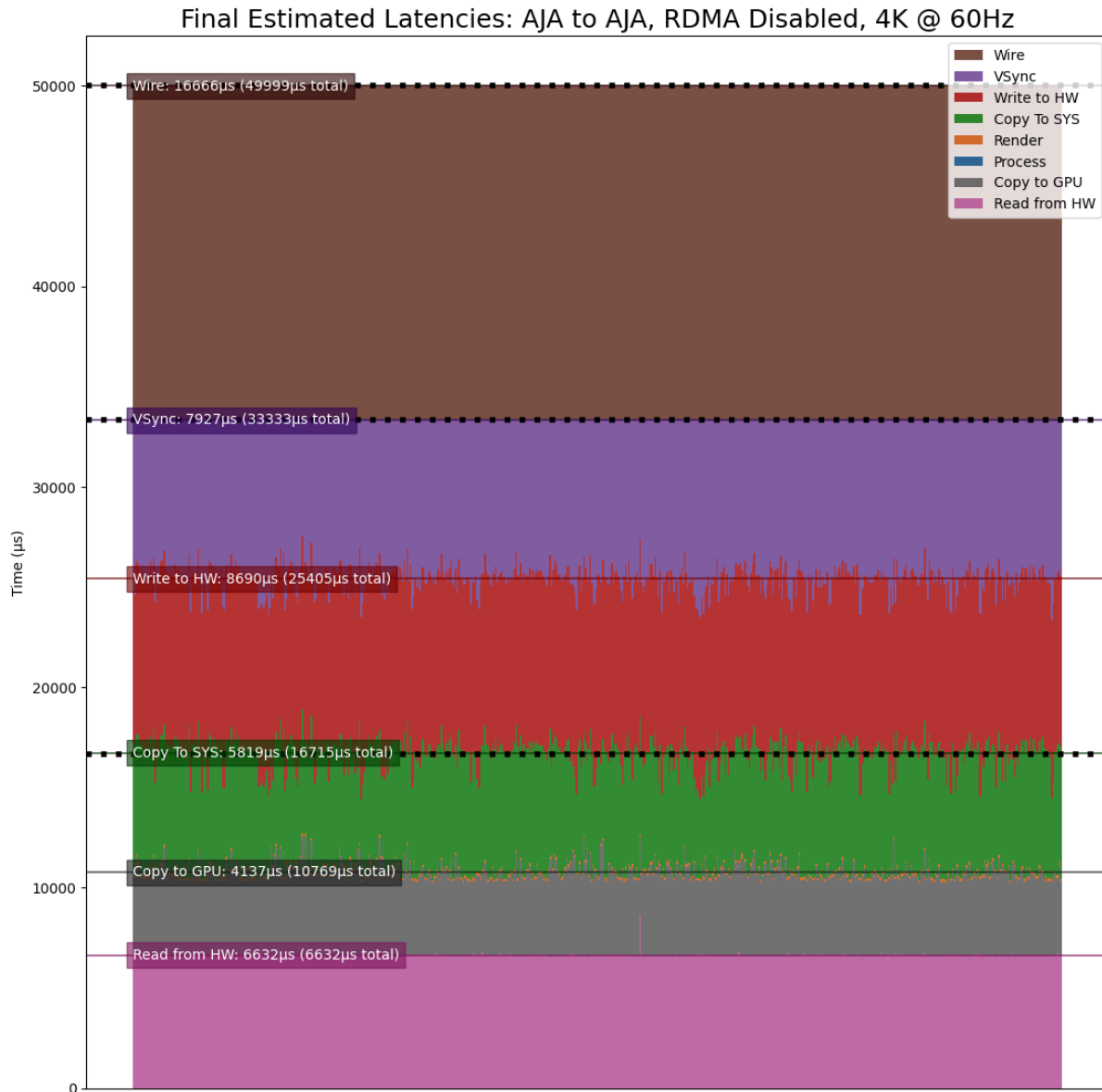
The graph can also be output to a PNG image file instead of opening a window on the desktop by providing the `--png {file}` option to the script. The following shows an example graph for an AJA to AJA measurement of a 4K @ 60Hz stream with RDMA disabled (as shown as an example in [Interpreting The Results](#), above).



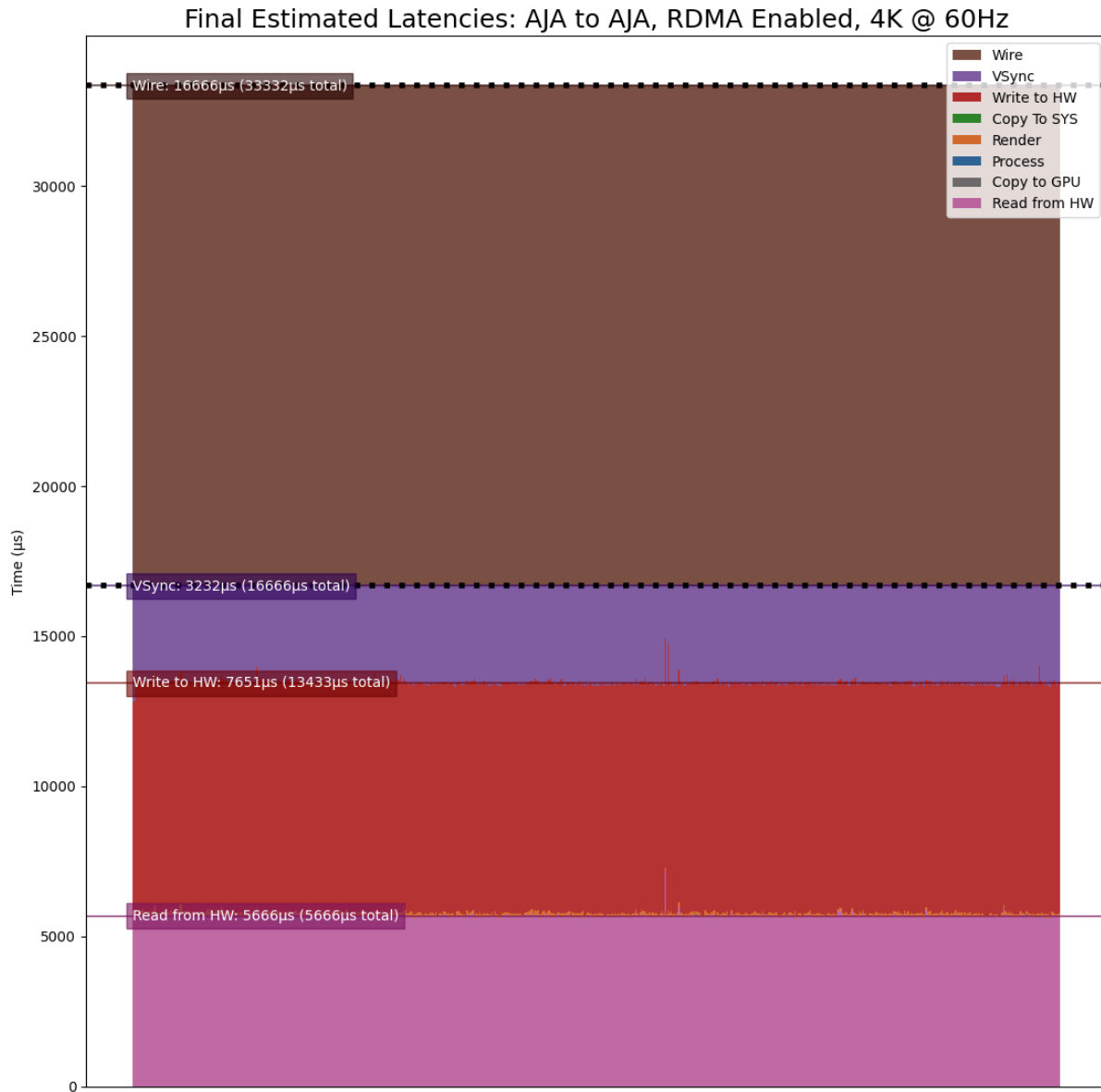
Note that this is showing the times for 600 frames, from left to right, with the life of each frame beginning at the bottom and ending at the top. The dotted black lines represent frame VSync intervals (every 16666μs).

The above example graphs the times directly as measured by the tool. To instead generate a graph for the **Final Estimated Latencies** as described above in *Interpreting The Results*, the `--estimate` flag can be provided to the script. As is done by the latency tool when it reports the estimated latencies, this reorders the producer and consumer steps then adds a VSync interval followed by the physical wire latency.

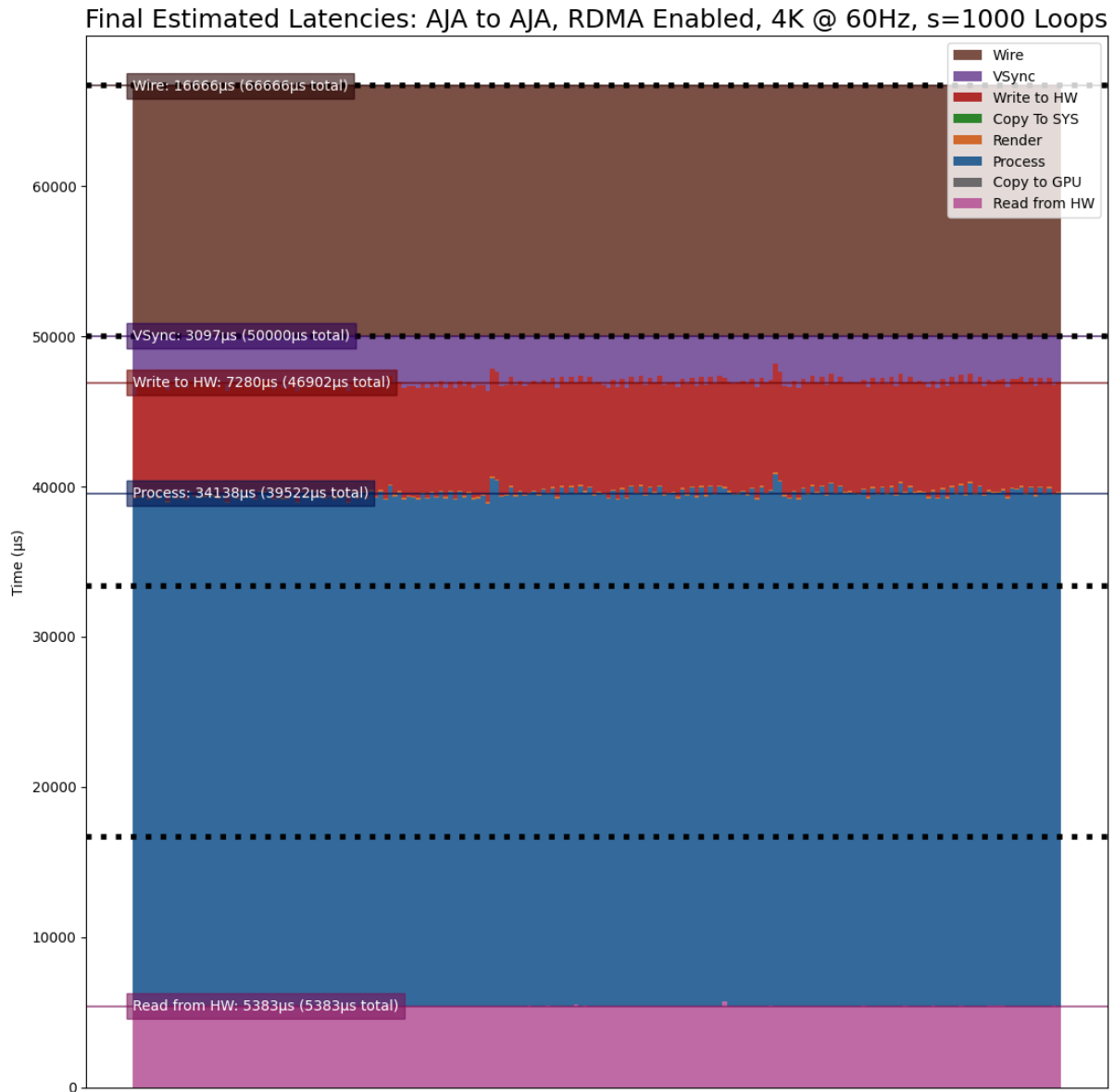
The following graphs the **Final Estimated Latencies** using the same data file as the graph above. Note that this shows a total of 3 frames of expected latency.



For the sake of comparison, the following graph shows the same test but with RDMA enabled. Note that the **Copy To GPU** and **Copy To SYS** times are now zero due to the use of RDMA, and this now shows just 2 frames of expected latency.



As a final example, the following graph duplicates the above test with RDMA enabled, but adds roughly 34ms of additional GPU processing time (`-s 1000`) to the pipeline to produce a final estimated latency of 4 frames.



35.6 Producers

There are currently 3 producer types supported by the Holoscan latency tool. See the following sections for a description of each supported producer.

35.6.1 OpenGL GPU Direct Rendering (HDMI)

This producer (gl) uses OpenGL to render frames directly on the GPU for output via the HDMI connectors on the GPU. This is currently expected to be the lowest latency path for GPU video output.

OpenGL Producer Notes:

- The video generated by this producer is rendered full-screen to the primary display. As of this version, this component has only been tested in a display-less environment in which the loop-back HDMI cable is the only cable attached to the GPU (and thus is the primary display). It may also be required to use the `xrandr` tool to configure the HDMI output; the tool will provide the `xrandr` commands needed if this is the case.
- Since OpenGL renders directly to the GPU, the `p.rdma` flag is not supported and RDMA is always considered to be enabled for this producer.

35.6.2 GStreamer GPU Rendering (HDMI)

This producer (gst) uses the `nveglglessink` GStreamer component that is included with Holopack to render frames that originate from a GStreamer pipeline to the HDMI connectors on the GPU.

GStreamer Producer Notes:

- The tool must be built with DeepStream support for this producer to support RDMA (see [Enabling DeepStream Support](#) for details).
- The video generated by this producer is rendered full-screen to the primary display. As of this version, this component has only been tested in a display-less environment in which the loop-back HDMI cable is the only cable attached to the GPU (and thus is the primary display). It may also be required to use the `xrandr` tool to configure the HDMI output; the tool will provide the `xrandr` commands needed if this is the case.
- Since the output of the generated frames is handled internally by the `nveglglessink` plugin, the timing of when the frames are output from the GPU are not known. Because of this, the *Wire Time* that is reported by this producer includes all of the time that the frame spends between being passed to the `nveglglessink` and when it is finally received by the consumer.

35.6.3 AJA Video Systems (SDI)

This producer (aja) outputs video frames from an AJA Video Systems device that supports video playback.

AJA Producer Notes:

- The latency tool must be built with AJA Video Systems support in order for this producer to be available (see [Building](#) for details).
- The following parameters can be used to configure the AJA device and channel that are used to output the frames:
 - `-p.device {index}`
Integer specifying the device index (i.e. 0 or 1). Defaults to 0.
 - `-p.channel {channel}`
Integer specifying the channel number, starting at 1 (i.e. 1 specifies NTV2_CHANNEL_1). Defaults to 1.
- The `p.rdma` flag can be used to enable (1) or disable (0) the use of RDMA with the producer. If RDMA is to be used, the AJA drivers loaded on the system must also support RDMA.
- The only AJA device that have currently been verified to work with this producer is the [Corvid 44 12G BNC \(SDI\)](#).

35.7 Consumers

There are currently 3 consumer types supported by the Holoscan latency tool. See the following sections for a description of each supported consumer.

35.7.1 V4L2 (Onboard HDMI Capture Card)

This consumer (`v4l2`) uses the V4L2 API directly in order to capture frames using the HDMI capture card that is onboard some of the NVIDIA Developer Kits.

V4L2 Consumer Notes:

- The onboard HDMI capture card is locked to a specific frame resolution and frame rate (1080p @ 60Hz), and so **1080** is the only supported format when using this consumer.
- The `-c.device {device}` parameter can be used to specify the path to the device that is being used to capture the frames (defaults to `/dev/video0`).
- The V4L2 API does not support RDMA, and so the `c.rdma` option is ignored.

35.7.2 GStreamer (Onboard HDMI Capture Card)

This consumer (`gst`) also captures frames from the onboard HDMI capture card, but uses the `v4l2src` GStreamer plugin that wraps the V4L2 API to support capturing frames for using within a GStreamer pipeline.

GStreamer Consumer Notes:

- The onboard HDMI capture card is locked to a specific frame resolution and frame rate (1080p @ 60Hz), and so **1080** is the only supported format when using this consumer.
- The `-c.device {device}` parameter can be used to specify the path to the device that is being used to capture the frames (defaults to `/dev/video0`).
- The `v4l2src` GStreamer plugin does not support RDMA, and so the `c.rdma` option is ignored.

35.7.3 AJA Video Systems (SDI and HDMI)

This consumer (`aja`) captures video frames from an AJA Video Systems device that supports video capture. This can be either an SDI or an HDMI video capture card.

AJA Consumer Notes:

- The latency tool must be built with AJA Video Systems support in order for this producer to be available (see [Building](#) for details).
- The following parameters can be used to configure the AJA device and channel that are used to capture the frames:
 - `-c.device {index}`
Integer specifying the device index (i.e. 0 or 1). Defaults to 0.
 - `-c.channel {channel}`
Integer specifying the channel number, starting at 1 (i.e. 1 specifies `NTV2_CHANNEL_1`). Defaults to 2.
- The `c.rdma` flag can be used to enable (1) or disable (0) the use of RDMA with the consumer. If RDMA is to be used, the AJA drivers loaded on the system must also support RDMA.

- The only AJA devices that have currently been verified to work with this consumer are the [KONA HDMI](#) (for HDMI) and [Corvid 44 12G BNC](#) (for SDI).

35.8 Troubleshooting

If any of the loopback-latency commands described above fail with errors, the following steps may help resolve the issue.

1. **Problem:** The following error is output:

```
ERROR: Failed to get a handle to the display (is the DISPLAY environment variable
↪set?)
```

Solution: Ensure that the DISPLAY environment variable is set with the ID of the X11 display you are using; e.g. for display ID 0:

```
$ export DISPLAY=:0
```

If the error persists, try changing the display ID; e.g., replacing 0 with 1:

```
$ export DISPLAY=:1
```

It might also be convenient to set this variable in your ~/.bashrc file so that it is set automatically whenever you login.

2. **Problem:** An error like the following is output:

```
ERROR: The requested format (1920x1080 @ 60Hz) does not match
       the current display mode (1024x768 @ 60Hz)
       Please set the display mode with the xrandr tool using
       the following command:

       $ xrandr --output DP-5 --mode 1920x1080 --panning 1920x1080 --rate 60
```

But using the xrandr command provided produces an error:

```
$ xrandr --output DP-5 --mode 1920x1080 --panning 1920x1080 --rate 60
xrandr: cannot find mode 1920x1080
```

Solution: Try the following:

1. Ensure that no other displays are connected to the GPU.
2. Check the output of an xrandr command to see that the requested format is supported. The following shows an example of what the onboard HDMI capture card should support. Note that each row of the supported modes shows the resolution on the left followed by all of the supported frame rates for that resolution to the right.

```
$ xrandr
Screen 0: minimum 8 x 8, current 1920 x 1080, maximum 32767 x 32767
DP-0 disconnected (normal left inverted right x axis y axis)
DP-1 disconnected (normal left inverted right x axis y axis)
DP-2 disconnected (normal left inverted right x axis y axis)
DP-3 disconnected (normal left inverted right x axis y axis)
DP-4 disconnected (normal left inverted right x axis y axis)
```

(continues on next page)

(continued from previous page)

```

DP-5 connected primary 1920x1080+0+0 (normal left inverted right x axis y axis)
↪1872mm x 1053mm
  1920x1080    60.00*+  59.94   50.00   29.97   25.00   23.98
  1680x1050    59.95
  1600x900     60.00
  1440x900     59.89
  1366x768     59.79
  1280x1024    75.02   60.02
  1280x800     59.81
  1280x720     60.00   59.94   50.00
  1152x864     75.00
  1024x768     75.03   70.07   60.00
   800x600     75.00   72.19   60.32
   720x576     50.00
   720x480     59.94
   640x480     75.00   72.81   59.94
DP-6 disconnected (normal left inverted right x axis y axis)
DP-7 disconnected (normal left inverted right x axis y axis)
USB-C-0 disconnected (normal left inverted right x axis y axis)

```

3. If a UHD or 4K mode is being requested, ensure that the DisplayPort to HDMI cable that is being used supports that mode.
 4. If the `xrandr` output still does not show the mode that is being requested but it should be supported by the cable and capture device, try rebooting the device.
3. **Problem:** One of the following errors is output:

```
ERROR: Select timeout on /dev/video0
```

```
ERROR: Failed to get the monitor mode (is the display cable attached?)
```

```
ERROR: Could not find frame color (0,0,0) in producer records.
```

These errors mean that either the capture device is not receiving frames, or the frames are empty (the producer will never output black frames, (0,0,0)).

Solution: Check the output of `xrandr` to ensure that the loopback cable is connected and the capture device is recognized as a display. If the following is output, showing no displays attached, this could mean that the loopback cable is either not connected properly or is faulty. Try connecting the cable again and/or replacing the cable.

```

$ xrandr
Screen 0: minimum 8 x 8, current 1920 x 1080, maximum 32767 x 32767
DP-0 disconnected (normal left inverted right x axis y axis)
DP-1 disconnected (normal left inverted right x axis y axis)
DP-2 disconnected (normal left inverted right x axis y axis)
DP-3 disconnected (normal left inverted right x axis y axis)
DP-4 disconnected (normal left inverted right x axis y axis)
DP-5 disconnected primary 1920x1080+0+0 (normal left inverted right x axis y axis)
↪0mm x 0mm
DP-6 disconnected (normal left inverted right x axis y axis)
DP-7 disconnected (normal left inverted right x axis y axis)

```

4. **Problem:** An error like the following is output:

ERROR: Could **not** find frame color (27,28,26) in producer records.

Colors near this particular value (27,28,26) are displayed on the Ubuntu lock screen, which prevents the latency tool from rendering frames properly. Note that the color value may differ slightly from (27,28,26).

Solution:

Follow the steps provided in the note at the top of the Example Configurations section to *enable automatic login and disable the Ubuntu lock screen*.

HOLOSCAN SDK FAQs

36.1 General

Q1: What is the Holoscan SDK?

A1: The Holoscan SDK is a comprehensive software development kit from NVIDIA designed for developing real-time AI applications, particularly in the healthcare sector. It includes acceleration libraries, pre-trained AI models, and reference applications for various medical imaging modalities like ultrasound, endoscopy, surgical robotics, and more.

Q2: What are the core components of the Holoscan SDK?

A2: The core components include:

- **Application:** A collection of Fragments that acquire and process streaming data. A single fragment application executes in a single process while multi-fragment (distributed) applications can span multiple processes and/or physical nodes.
- **Fragments:** Directed graphs of Operators, which can be allocated to physical nodes in a Holoscan cluster.
- **Operators:** Basic units of work that process streaming data.
- **Conditions:** Components that determine conditions under which a given Operator will be considered ready to execute.
- **Resources:** Components that provide shared functionality which can be reused across operators. Examples are device memory pools, CUDA stream pools and components for serialization/deserialization of data.
- **Ports:** An operator's input ports are used to receive data from upstream operators. Input ports consist of a receiver and any associated conditions. An operator's output ports are used to emit data to downstream operators. An output port consists of a transmitter and any associated Conditions.

Q3: How is the Holoscan SDK different from other SDKs?

A3: The Holoscan SDK is a domain and sensor agnostic SDK optimized for the easy construction and deployment of high-performance, high bandwidth, and real-time AI applications. By marrying high speed instruments to NVIDIA software, Holoscan is the platform for a future of self-driving, software defined, and scalable sensor processing solutions, touching industries from scientific computing and instrumentation to medical devices.

36.2 Installation and Setup

Q1: How do I install the Holoscan SDK?

A1: There are multiple ways to install the Holoscan SDK:

- Using NGC containers :
 - For **dGPU** (x86_64, IGX Orin dGPU, Clara AGX dGPU, GH200)

```
docker pull nvcr.io/nvidia/clara-holoscan/holoscan:v2.6.0-dgpu
```

- For **iGPU** (Jetson, IGX Orin iGPU, Clara AGX iGPU)

```
docker pull nvcr.io/nvidia/clara-holoscan/holoscan:v2.6.0-igpu
```

For more information, please refer to details and usage instructions on [NGC](#).

- Using Debian packages

```
sudo apt update
sudo apt install holoscan
```

If holoscan is not found, try the following before repeating the steps above:

- **IGX Orin**: Ensure the **compute stack is properly installed** which should configure the L4T repository source. If you still cannot install the Holoscan SDK, use the [arm64-sbsa](#) from the CUDA repository.
- **Jetson**: Ensure **JetPack is properly installed** which should configure the L4T repository source. If you still cannot install the Holoscan SDK, use the [aarch64-jetson](#) from the CUDA repository.
- **GH200**: Use the [arm64-sbsa](#) from the CUDA repository.
- **x86_64**: Use the [x86_64](#) from the CUDA repository.

Please note that , to leverage the python module included in the debian package (instead of installing the python wheel), include the path below to your python path.

```
export PYTHONPATH="/opt/nvidia/holoscan/python/lib"
```

- Using Python wheels

```
pip install holoscan
```

For more details and troubleshooting , please refer to [PyPI](#). For x86_64, ensure that the [CUDA Runtime is installed](#).

If you are unsure of which installation option to use, please refer to the considerations below:

- The **Holoscan container image on NGC** is the safest way to ensure all the dependencies are present with the expected versions (including Torch and ONNX Runtime), and should work on most Linux distributions. It is the simplest way to run the embedded examples, while still allowing you to create your own C++ and Python Holoscan applications on top of it. These benefits come at a cost:
 - Large image size due to the numerous (some of them optional) dependencies. If you need a lean runtime image, see the **section below**.
 - Standard inconveniences that exist when using Docker, such as more complex run instructions for proper configuration.

- If you are confident in your ability to manage dependencies on your own in your host environment, the **Holoscan Debian package** should provide all the capabilities needed to use the Holoscan SDK, assuming you are using Ubuntu 22.04.
- If you are not interested in the C++ API but just need to work in Python, or want to use a different version than Python 3.10, you can use the **Holoscan Python wheels** on PyPI. While they are the easiest solution for installing the SDK, they might require the most work to set up your environment with extra dependencies based on your needs. Finally, they are only formally supported on Ubuntu 22.04, though they should support other Linux distributions with glibc 2.35 or above.

Q2: What are the prerequisites for installing the Holoscan SDK?

A2: The prerequisites include:

- If you are installing Holoscan SDK on a Developer kit, please refer to the details below

Developer Kit	User Guide	OS	GPU mode
NVIDIA IGX Orin	Link to User Guide	IGX Software 1.0 Production Release	iGPU or * dGPU
NVIDIA Jetson AGX Orin and Orin Nano	Link to User Guide	JetPack 6.0	iGPU
NVIDIA Clara AGX	Link to User Guide	Holopack 1.2	iGPU or * dGPU

- If you are installing Holoscan SDK on NVIDIA SuperChips, please note that HSDK 2.2 has only been tested with the Grace-Hopper SuperChip (GH200) with Ubuntu 22.04. Follow setup instructions [here](#).
- If you are installing Holoscan SDK on Linux x86_64 workstations, please refer to the details below for supported distributions

OS	NGC Container	Debian/RPM Package	Python wheel	Build from source
Ubuntu 22.04	Yes	Yes	Yes	No
RHEL 9.x	Yes	No	No	No
Other Linux distros	No	No	No	No

For specific NVIDIA discrete GPU (dGPU) requirements, please check below :

- Ampere or above recommended for best performance
- [Quadro/NVIDIA RTX](#) necessary for GPUDirect RDMA support
- Tested with [NVIDIA Quadro RTX 6000](#) and [NVIDIA RTX A6000](#)
- [NVIDIA dGPU drivers](#): 535 or above

Q3: Are there any additional setup steps required?

A3: Additional setup steps to achieve peak performance may include:

- [Enabling RDMA](#)
- [Enabling G-SYNC](#)
- [Disabling Variable Backlight](#)
- [Enabling Exclusive Display Mode](#)
- [Use both Integrated and Discrete GPUs on NVIDIA Developer Kits](#)
- [Deployment Software Stack](#)

36.3 Getting Started

Q1: How do I get started with developing applications using the Holoscan SDK?

A1: To get started:

1. Set up the SDK and your development environment.
2. Follow the “Getting Started” [guide](#) and tutorials provided in the SDK documentation.
3. Explore the [example applications](#) in Holohub to understand the framework and its capabilities.

Q2: Are there any pre-trained models available in the SDK?

A2: Yes, the SDK includes pre-trained AI models for various medical imaging tasks such as segmentation, classification, and object detection. These models can be fine-tuned or used directly in your applications. For more details, please refer to the endoscopy tool tracking [example](#) in Holohub and the body pose estimation [example](#).

36.4 Development

Q1: How do I create a new application with the Holoscan SDK?

A1: To create a new application:

1. Define the core concepts such as Operators, Fragments, and their interactions.
2. Use the provided templates and examples as a starting point.

For more details, please refer to the [Holoscan by Example section](#).

Q2: What are some example applications provided in the SDK?

A2: Example applications include:

- Hello World: Basic introduction to the SDK.
- Ping Simple and Ping Custom Op: Demonstrates simple data processing.
- Video Replayer: Shows how to process and display video streams.
- Medical imaging examples like ultrasound and endoscopy processing

For a list of example applications, please visit the [Holoscan by Example section](#).

Q3: How can I integrate my own AI models into the Holoscan SDK?

A3: Integrating your own AI models involves:

1. Converting your model to a compatible format (e.g., TensorRT, ONNX).
2. Ensuring that your data preprocessing and postprocessing steps align with the model’s requirements.

For more information on how to bring your own model in Holsocan SDK and build an inference example, please refer to [this example](#) .

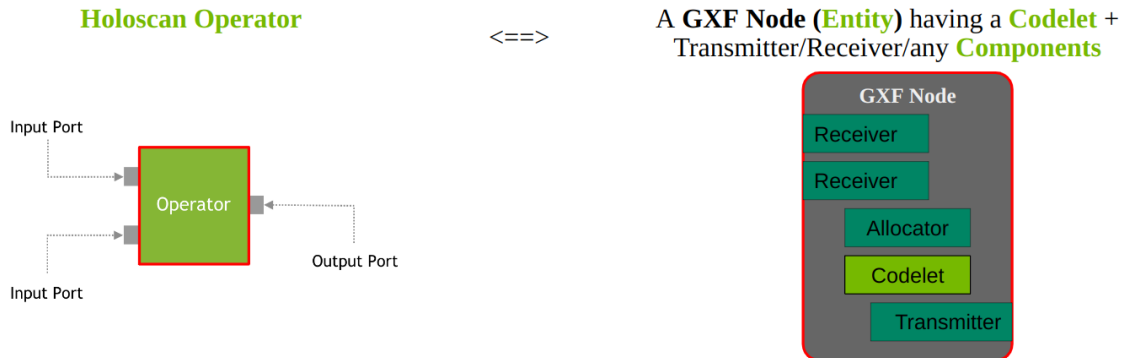
Q4: How can I update the VideoStreamReplayerOp and the VideoStreamRecorderOp to use a custom file format ?

A4: Holoscan SDK depends on GXF.GXF is using [Entity-Component-System](#) paradigm. Holoscan uses GXF as an execution engine and Holoscan’s API abstracts Entity-Component-System and abstracts GXF node as Operator with input/output ports.

Concepts

Holoscan Operator/Resource and GXF Codelet/Component

- **Holoscan Operator** encapsulates a **GXF Entity**.
 - Has only one **GXF Codelet**.
 - Has the concept of input/output ports. (Input port is the GXF Receiver, output port is the GXF Transmitter component)
- **Holoscan Resource** encapsulates a **GXF Component**.



Most messages between Codelets (Operator in Holoscan) are also an entity object. An entity can hold multiple components/types. In `VideoStreamReplayerOp` and `VideoStreamRecorderOp`, an entity object that holds one or more GXF Tensor objects (as a component) is sent/received - you can think that an entity as a dictionary of objects – `<key,object>` map. For `VideoStreamReplayerOp` and `VideoStreamRecorderOp`, it currently uses a custom file type (`.gxf_entities` and `.gxf_index` files) to load and store a sequence of entity objects (in this case, an entity has one GXF Tensor object). `.gxf_index` file include a file offset/timestamp information for each entity and `.gxf_entities` includes a series of (serialized) entity data. Serializing/deserializing an entity object is done by using `nvidia::gxf::EntitySerializer` class (with `StdEntitySerializer` implementation), and `nvidia::gxf::FileStream` endpoint. The official way to support GDS in GXF would be to extend `nvidia::gxf::FileStream` class so it uses `cufile(GDS)` internally. However, setting the development environment wouldn't be straightforward. This are the steps you would need to follow:

- Update `VideoStreamRecorderOp::compute()` to use your own implementation to save an entity (as a single tensor) to the file system.
 - Example

```
auto in_message = op_input.receive<nvidia::gxf::TensorMap>("in").value();
```

- Update `VideoStreamReplayerOp::compute()` to use your own implementation to read the file (with the custom format) and emit it as an entity (holding tensor(s) as a component – it is called `TensorMap`).
 - Example:

```
nvidia::gxf::Expected<nvidia::gxf::Entity> out_message =
    CreateTensorMap(context.context(),
                    pool.value(),
                    {{out_tensor_name_.get(),
                     nvidia::gxf::MemoryStorageType::kDevice,
                     out_shape,
                     out_primitive_type_,
                     0,
                     nvidia::gxf::ComputeTrivialStrides(out_shape, dst_typesize)}}},
                    false);
```

- You need to update initialize() and other methods to get rid of `nvidia::gxf::FileStream` and `nvidia::gxf::FileStream` endpoint. For testing `VideoStreamReplayerOp`, you can just use `VideoReplayerApp` example. You can develop/test/create operator (release) binaries by following the user guide: <https://github.com/nvidia-holoscan/holoscan-sdk/blob/main/DEVELOP.md>

```
class VideoReplayerApp : public holoscan::Application {
public:
    void compose() override {
        using namespace holoscan;

        // Sets the data directory to use from the environment variable if it is set
        ArgList args;
        auto data_directory = std::getenv("HOLOSCAN_INPUT_PATH");
        if (data_directory != nullptr && data_directory[0] != '\0') {
            auto video_directory = std::filesystem::path(data_directory);
            video_directory /= "racetrx";
            args.add(Arg("directory", video_directory.string()));
        }

        // Define the replayer and holoviz operators and configure using yaml configuration
        auto replayer =
            make_operator<ops::VideoStreamReplayerOp>("replayer", from_config("replayer"),
            ↪args);
        auto visualizer = make_operator<ops::HolovizOp>("holoviz", from_config("holoviz"));

        // Define the workflow: replayer -> holoviz
        add_flow(replayer, visualizer, {"output", "receivers"});
    }
};
```

```
./run build
./run launch
# inside the container
./examples/video_replayer/cpp/video_replayer
```

- As an alternative, you can create a separate Holoscan Operator and apply it with other sample applications (such as endoscopy tool tracking app) by following HoloHub's guide (<https://github.com/nvidia-holoscan/holohub>). You can also use Holoscan SDK's installation binary with `holoscan install dir` created by `./run build` with Holoscan SDK repo.

Q5: How can I use the Inference Operator with Python tensor?

A5: The Inference Operator accepts `holoscan::Entity` or `holoscan::TensorMap` (similar to the dictionary of Array-like objects in Python) as an input message.

For example, you can define an operator processing input video (as a tensor). You can find a more detailed example of this type of operator together with an example by referencing the tensor interop [example](#).

This operator has:

```
inputs: "input_tensor"
outputs: "output_tensor"
```

The data from each input is processed by a CuPy gaussian filter and

(continues on next page)

(continued from previous page)

```

the result is sent to the output.
...
def compute(self, op_input, op_output, context):
    # in_message is of dict
    in_message = op_input.receive("input_tensor")

    # smooth along first two axes, but not the color channels
    sigma = (self.sigma, self.sigma, 0)

    # out_message is of dict
    out_message = dict()

    for key, value in in_message.items():
        print(f"message received (count: {self.count})")
        self.count += 1

        cp_array = cp.asarray(value)

        # process cp_array
        cp_array = ndi.gaussian_filter(cp_array, sigma)

        out_message[key] = cp_array

    op_output.emit(out_message, "output_tensor")
...

```

Q6: Is there support in the Holoscan SDK, particularly for models written as Triton Python backends like NVIDIA's FoundationPose?

A6: Triton backends are not currently supported. The Inference Operator supports TensorRT (trt), ONNX Runtime (onnxrt), and Torch backends.

For more information on the Inference Operator please refer to the [section](#) in the User Guide regarding the Inference Operator .

Q7: Can I directly use a .pth (PyTorch) model file with the Holoscan SDK's inference operator? A7: No, you cannot use a .pth model file directly with the Holoscan SDK. Here's why and what you can do instead:

1. Holoscan SDK's Torch backend is based on libtorch, which requires models to be in TorchScript format.
2. Converting a .pth model to TorchScript is a manual process and cannot be done automatically within the SDK.
3. For the best performance and ease of use, it's recommended to: a) Use a TensorRT (TRT) model if available. b) If you have an ONNX model, you can convert it to TRT automatically within the SDK.
4. Using a TRT model (or converting from ONNX to TRT) will likely provide the fastest inference and be the easiest to set up with the Holoscan SDK.

In summary, while direct .pth file usage isn't supported, converting to TensorRT or using ONNX with automatic TRT conversion are the recommended approaches for optimal performance and compatibility with the Holoscan SDK.

Q8: Can I use multiple models with the Inference Operator?

A8: Yes, you can use multiple models by specifying them in the `model_path_map` parameter. For more information, please refer to the [Parameters section](#) of the Inference Operator in the Holoscan User Guide.

Q9: How can I enable parallel inference for multiple models?

A9: Parallel inference is enabled by default. To disable it, set `parallel_inference: false` in the parameter set. For more information, please refer to the [Parameters section](#) of the Inference Operator in the Holoscan User Guide.

Q9: Can I use different backends for different models in the same application?

A9: Yes, you can specify different backends for different models using the `backend_map` parameter. For more information, please refer to the [Parameters section](#) of the Inference Operator in the Holoscan User Guide.

Q10: Can I perform inference on the CPU?

A10: Yes, you can perform inference on the CPU by setting `infer_on_cpu: true` and use either the ONNX Runtime or PyTorch backend. For more information, please refer to the [Parameters section](#) of the Inference Operator in the Holoscan User Guide.

Q11: Can I control where the input and output data is stored (CPU vs GPU memory)?

A11: Yes, use the `input_on_cuda`, `output_on_cuda`, and `transmit_on_cuda` parameters to control data location. For more information, please refer to the [Parameters section](#) of the Inference Operator in the Holoscan User Guide.

Q12: How can I use the Optional flag?

A12: In Python, there are two ways to define parameter:

- Using `spec.param()` method in Python's `setup()` method of the operator, usually done when wrapping the existing C++ operator.
- Parameters are passed to the Constructor (`__init__()`) directly. In Python there is no `try_get()` method in the parameter. Instead, the default value is set to `None`, allowing us to check whether the parameter is set by users by verifying if the parameter value is `None`.

Q13: How can I define an Operator's creator for passing custom arguments?

A13: Feeding custom data to the constructor of an Operator in the `compose()` method is crucial. When you use the `make_operator<>()` template method in C++ or the Python Operator constructor, the `setup()` method is called internally, which prevents you from passing custom data (such as configuration values) after `make_operator<>()` is called. In C++, to pass non-condition/argument data to the constructor of a C++ Operator class, you need to define an additional constructor to accept your custom data. For example, you can define a constructor that accepts a `std::vector<std::string>` argument for the list of output port names as a second parameter.

Q14: How can I stop an application?

A14: There are two approaches to stopping an application:

- using `BooleanCondition` on replayer operator

```
std::string op_name = "replayer";
std::string param_name = "boolean_scheduling_term";

// get the operator
holoscan::Operator* op = nullptr;
auto& app_graph = fragment()->graph();
if (!app_graph.is_empty()) { op = app_graph.find_node(op_name).get(); }
if (!op) {
    HOLOSCAN_LOG_ERROR("Operator '{}' is not defined", op_name);
    return;
}

// Stop executing compute() for 'replayer' operator

auto boolean_condition = op->condition<holoscan::BooleanCondition>(param_name);
```

(continues on next page)

(continued from previous page)

```
boolean_condition->disable_tick();

// Stop executing compute() for this operator
boolean_condition = condition<holoscan::BooleanCondition>(param_name);
boolean_condition->disable_tick();
return;
```

To terminate the application smoothly, it is recommended to rely on the stop-on-deadlock feature in the scheduler. By default, the `stop_on_deadlock` parameter of `GreedyScheduler` is set to true. In case the `VideoReplayer` Operator stops, the entire pipeline will stop.

- using `interrupt()`

```
fragment()->executor().interrupt();
```

Please note that using `interrupt()` forces to terminate the execution and can cause error messages, and the recommendation is using deadlock-based approach.

As an alternative, you can also use the `CountCondition`. Please refer to the [section](#). At a high level, this is how attaching a `CountCondition` to an operator works:

The operator starts in a READY state. Each time the operator executes, the count decreases by 1. When the count reaches 0, the operator's state changes to NEVER. In the NEVER state, the operator stops executing.

For example, if you want to run the application 100 times and then stop it:

```
auto my_operator = make_operator<MyOperator>("my_operator", make_condition
-><CountCondition>(100));
```

Q15:How can I loop an `output.emit()` call within the operator?

A15: Each input or output port has its own queue. Internally, the process works as follows:

1. Before the `compute()` method of an operator A is triggered, for each input port (usually backed by `DoubleBufferReceiver`), data (messages) in the backstage of the input port's queue are moved to the main stage of the queue. This is done using `router->syncInbox(entity)`.
2. The `compute()` method of operator A is triggered.
3. For each output port of operator A, data in the output port's queue are moved to the queue (backstage) of the downstream operator's input port using `router->syncOutbox(entity)`.

By default, the queue capacity of the input/output port is set to 1, although this can be configured in the `setup()` method. This is why we cannot call `output.emit()` multiple times in a `compute()` method, as doing so can cause a `GXF_EXCEEDING_PREALLOCATED_SIZE` error.

With the `GreedyScheduler`, which is the default scheduler using a single thread to trigger an operator's `compute()` method, no other operator can be scheduled until the `compute()` method of the current operator returns.

To address this challenge, we might consider creating a utility method or class designed to accept a generator or iterator object. This approach would be particularly effective within a `compute()` method, especially if the operator is a source operator without input ports. It would enable the method to preserve the state of the input and either call `output.emit()` for each yielded value in a single `compute()` invocation or return without blocking the thread.

The Python API code to override the connector would be something like this if we wanted a queue with capacity 20 and policy of "reject" (discard) the item if the queue is full:

```

from holoscan.core import IOSpec

# and then within the setup method define the output using the connector method like this

spec.output("out1").connector(
    IOSpec.ConnectorType.DOUBLE_BUFFER, capacity=20, policy=1
)

```

For the policy options:

- 0 = pop (if the queue is full, remove an item from the queue to make room for the incoming one)
- 1 = reject (if the queue is full, reject the new item)
- 2 = fault (terminate the application if the queue is full and a new item is added)

For completeness, to explicitly specify both the connector and its conditions, the syntax should be:

```

# The default setting for an output should be equivalent to explicitly specifying
spec.output("out1").connector(
    IOSpec.ConnectorType.DOUBLE_BUFFER, capacity=1, policy=2
).condition(
    ConditionType.DOWNSTREAM_MESSAGE_AFFORDABLE, min_size=1, front_stage_max_size=1
)

```

Q16: How can I add a green border and a small image to a corner to a Holoviz Operator?

A16: You can follow the Holoviz examples here:

- Holoviz geometry example : https://github.com/nvidia-holoscan/holoscan-sdk/blob/main/examples/holoviz/python/holoviz_geometry.py for border examples
- Holoviz views example https://github.com/nvidia-holoscan/holoscan-sdk/blob/main/examples/holoviz/python/holoviz_views.py for view/image example

Q17: What is the difference between setup vs initialize vs __init__ ?

A17: Since v0.6 release, Holoscan Operator does “lazy initialization” and Operator instance creation (`super().__init__(*args, **kwargs)`) doesn’t initialize (calling `Operator.initialize(self)`) the corresponding GXF entity anymore. Currently, [setting the class members in Python](#) is done when Operator is [initialized by GXF Executor](#). The purpose of setup method is for getting “operator’s spec” by providing OperatorSpec object (spec param) to the method. When `__init__` is called, it calls C++’s `Operator::spec(const std::shared_ptr<OperatorSpec>& spec)` method (and also sets `self.spec` class member), and call setup method so that Operator’s `spec()` method hold the operator’s specification. Since setup method can be called multiple times with other OperatorSpec object (e.g., to enumerate the description of the operator), in the setup method, user shouldn’t initialize something in the Operator object. Such initialization needs to be done in initialize method. `__init__` method is for creating Operator object. it can be used for initializing operator object itself by passing miscellaneous arguments, but it doesn’t ‘initialize’ corresponding GXF entity object.

Q18: I’d like to use a CUDA stream allocated by the Holoscan SDK in a non-Holoscan library (OpenCV, CuPy, PyTorch). All these 3rd party libraries support CUDA streams, allocators etc. but they have different objects to represent that CUDA Stream (such as a `cupy.cuda.Stream`). I need to get the Holoscan CUDA stream and convert it to a `cupy.cuda.Stream` in a similar way a Holoscan Tensor is converted to a CuPy array with memory pointers. Please propose a solution.

A18: There is a `CudaStreamHandler` utility that works via GXF APIs in the C++ layer. We have not currently created a Python API to allow users to use it from the compute methods of native Python operators. In general, the underlying GXF library is currently refactoring how CUDA streams are handled and we plan to then improve the stream handling

on Holoscan SDK after that. You can use CuPy or other 3rd party stream APIs within their own native operators and pass the stream objects as a Python object between your own native operators. I think this doesn't help with the issue you are facing as you want to reuse a stream allocated by some upstream wrapped C++ operator provided by the SDK there is currently no proper way to do that from Python.

Q19: What is the purpose of the `activation_map` parameter in the Holoscan `HoloInfer` operator? A19: The `activation_map` parameter allows users to enable or disable model inferences dynamically at runtime. It can be used to decide on which frames to run inference for each model.

Q20: Is there an existing example or template that demonstrates the simultaneous use of integrated GPU (iGPU) and discrete GPU (dGPU) in a Holoscan application pipeline? Specifically, I am looking for a sample workflow that includes:

1. Receiving and processing data on the iGPU of an AGX Orin
2. Transferring the processed data to a dGPU
3. Running multiple AI models on the dGPU
4. Displaying results using the dGPU

A20: To leverage both the integrated GPU (iGPU) and discrete GPU (dGPU) on your IGX system with Holoscan, please refer to the [IGX user guide](#). This guide provides detailed instructions on utilizing the iGPU in containers when the IGX developer kit is configured in dGPU mode.

For Holoscan applications, there are two primary approaches to utilize both GPUs:

1. Concurrent Application Execution: Run separate applications simultaneously, as outlined in the IGX documentation. The iGPU application must be executed within the Holoscan iGPU container, while the dGPU application can be run either natively or within the Holoscan dGPU container.
2. Distributed Application: Develop a single distributed application that utilizes both GPUs by executing distinct fragments on the iGPU and dGPU respectively.

To illustrate the second approach, consider the following example using the 'ping' distributed application. This demonstrates communication between the iGPU and dGPU using Holoscan containers:

```
COMMON_DOCKER_FLAGS="--rm -i --init --net=host
--runtime=nvidia -e NVIDIA_DRIVER_CAPABILITIES=all
--cap-add CAP_SYS_PTRACE --ipc=host --ulimit memlock=-1 --ulimit stack=67108864
"
HOLOSCAN_VERSION=2.2.0
HOLOSCAN_IMG="nvcr.io/nvidia/clara-holoscan/holoscan:v$HOLOSCAN_VERSION"
HOLOSCAN_DGPU_IMG="$HOLOSCAN_IMG-dgpu"
HOLOSCAN_IGPU_IMG="$HOLOSCAN_IMG-igpu"

# Pull necessary images
docker pull $HOLOSCAN_DGPU_IMG
docker pull $HOLOSCAN_IGPU_IMG

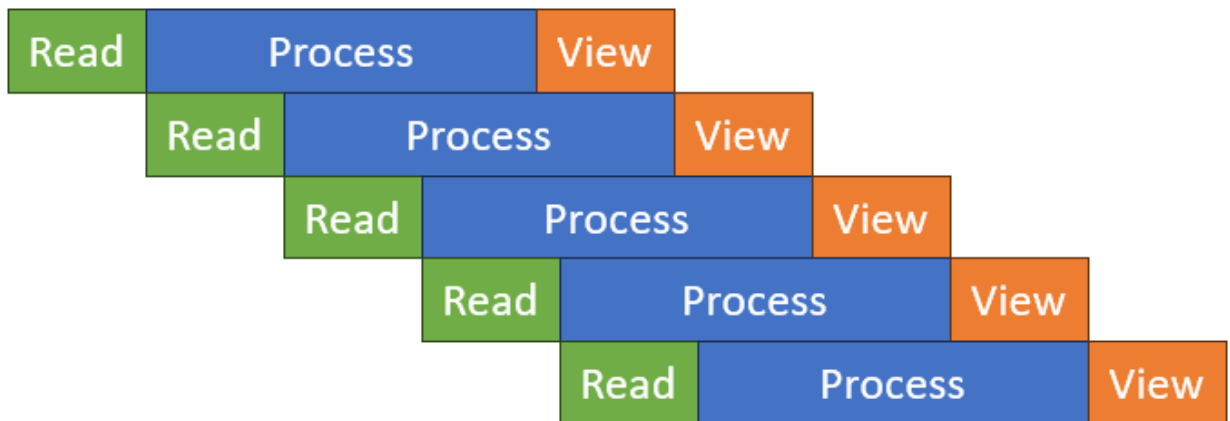
# Execute ping distributed (Python) in dGPU container
# Note: This instance serves as the 'driver', but the iGPU could also fulfill this role
# The '&' allows for non-blocking execution, enabling subsequent iGPU command
docker run \
  $COMMON_DOCKER_FLAGS \
  $HOLOSCAN_DGPU_IMG \
  bash -c "python3 ./examples/ping_distributed/python/ping_distributed.py --gpu --worker -
↪-driver" &
```

(continues on next page)

(continued from previous page)

```
# Execute ping distributed (C++) in iGPU container
docker run \
  $COMMON_DOCKER_FLAGS \
  -e NVIDIA_VISIBLE_DEVICES=nvidia.com/igpu=0 \
  $HOLOSCAN_IMG-igpu \
  bash -c "./examples/ping_distributed/cpp/ping_distributed --gpu --worker"
```

Q21: Is there an efficient method to configure Holoscan to enable concurrent processing pipelines? My objective is to implement a system where frame acquisition and processing can occur simultaneously. Specifically, I aim to initiate the reading of a subsequent frame while the current frame is still undergoing processing through the InferenceOp. To illustrate:



1. Is it possible to begin reading Frame N+1 while Frame N is still being processed by the InferenceOp?
2. Or does Holoscan require the completion of all operations on Frame N before initiating any operations on Frame N+1?

If concurrent processing is achievable, what would be the recommended approach to implement such a system within the Holoscan framework?

A21: The NVIDIA GXF framework provides a `nvidia::gxf::BroadcastCodelet` with a “round robin” mode that offers an alternative to the standard broadcast behavior. This mode sequentially directs input messages to different output ports in rotation. While this functionality was accessible in Holoscan 2.1 through the `GXFCodeletOp`, we could also develop a native operator that provides equivalent utility.

The GXF source defines the modes as follows:

```
enum struct BroadcastMode {
  kBroadcast = 0, // publishes incoming message to all transmitters
  kRoundRobin = 1, // publishes incoming message to one transmitter in round-robin
  ↪ fashion
};
```

For the gathering operation, GXF implements the `nvidia::gxf::Gather` codelet. This codelet transfers any messages from the receive ports to the output port. The `tick_source_limit` parameter can be configured to cease checking for additional received messages on other ports once a specified maximum number of messages to output has been reached.

It's important to note that the GXF Gather codelet may not inherently preserve the order in which inference operations were called. While messages might naturally be processed in the order they were received if inference operations complete sequentially, this behavior is not guaranteed.

To ensure strict ordering, we could develop a custom native operator. This operator would sequentially check for messages on each port, beginning with port 1, and only move to the next port once a message has been received on the current port. This approach would guarantee that frames are processed and displayed in the correct order.

This custom implementation would provide greater control over the message flow and ensure the integrity of the processing pipeline, particularly in scenarios where maintaining the original frame order is critical.

Q22: How can I use other libraries in my Holoscan SDK application pipeline? A22: Refer to the dedicated [HoloHub tutorial](#) for an overview of how to use external C++ or Python libraries in your custom Holoscan SDK application.

Q23: How can I ensure proper data flow and handling in a Holoscan pipeline with branching paths, especially when inline updates are performed on shared data?

A23: In a Holoscan pipeline with branching paths, such as:

```
A -> B -> C -> D
      \
      -> E
```

There are several considerations and potential solutions to ensure proper data flow and handling, especially when operators like C or D perform inline updates to the data.

1. Data Access Guarantee: E is guaranteed to access the data egressing from B. However, in a multithreaded scheduler, careful attention must be paid to potential data race conditions.
2. Execution Order: The current implementation maintains the order of root nodes, but the creation of GXF connections between B -> C and B -> E is randomly determined due to the use of `std::unordered_map` as the graph data structure. This randomness can affect which connection is prioritized in GXF.
3. Potential Solutions: a. PeriodicCondition: One approach is to use a `PeriodicCondition` to control the execution timing of operators. Here's an example:

```
from holoscan.conditions import CountCondition, PeriodicCondition
from holoscan.core import Application
from holoscan.operators import PingRxOp, PingTxOp

class MyPingApp(Application):
    def compose(self):
        b = PingTxOp(self, CountCondition(self, 10), name="B")
        c = PingRxOp(self, PeriodicCondition(self, 20_000_000), name="C")
        e = PingRxOp(self, name="E")

        self.add_flow(b, c)
        self.add_flow(b, e)
```

In this example, the `PeriodicCondition` is used to ensure that C executes only after a specified period (20 milliseconds in this case) has elapsed. This can help control the timing of data processing between different branches. b. Custom Operator: Developing a custom native operator could provide more control over the message flow. This operator could sequentially check for messages on each port, ensuring that frames are processed and displayed in the correct order. c. Data Copying: To avoid issues with inline updates affecting shared data, consider implementing a mechanism to create copies of the data for each branch. This ensures that modifications in one branch don't unintentionally affect the other.

1. Limitations and Considerations:
 - The GXF Gather codelet may not inherently preserve the order in which inference operations were called.
 - When using mock objects for testing (as mentioned in the background information), modifying the original pipeline structure might be challenging. In such cases, focusing on data copying or careful timing control might be more feasible.

1. Future Improvements: Updating the graph structure to use `std::map` instead of `std::unordered_map` for `succ_` and `pred_` could potentially provide more predictable behavior in terms of connection creation order.

Q24: I'm inquiring about the availability of a Holoscan example that demonstrates distributed processing across two networked computers. Specifically, I'm interested in a scenario where:

1. One computer captures a frame, potentially using an AJA capture card.
2. The captured frame is then transferred over a local network to a second computer.
3. The second computer receives and displays the frame.

Additionally, I have some questions regarding the networking aspects of such a setup:

1. Does Holoscan provide flexibility in selecting the transport layer for this inter-computer communication?
2. Is it possible to utilize WebRTC as the transport protocol in this scenario?

A24: There are two relevant approaches:

1. WebRTC Implementation: A reference application demonstrating WebRTC-based video streaming is available in the HoloHub repository. You can find this example at: https://github.com/nvidia-holoscan/holohub/tree/main/applications/webrtc_video_server This application showcases how WebRTC can be utilized for inter-computer communication within the Holoscan framework.
2. Distributed Application Approach: An alternative method involves creating a distributed application with separate fragments running on each node. For more information, please refer to the [section](#) in the User guide on Creating Distributed applications .

Q25: How can I use `run_async()` to launch an application in a separate thread and stop the application?

A25: We can set the event state to `EVENT_WAITING` (request sent to an async service, pending event done notification) and then `EVENT_DONE` (event done notification received, entity ready to be ticked) to allow a specific operator to wait/resume its operation. In the example, it calls `AsynchronousCondition::event_state(AsynchronousEventState::EVENT_NEVER)` to set the status of the condition to NEVER. (`BooleanCondition` does the same by setting the status of the condition to NEVER when `BooleanCondition::disable_tick()` is called). This means the operator does not want to be ticked again (end of execution). Once the state of the condition goes to NEVER (internally, `SchedulingConditionType::NEVER`), it marks the end of execution and cannot be undone.

Q26: Are there any existing applications or examples in the Holoscan ecosystem that demonstrate the ability to utilize multiple GPUs concurrently within a single application?

A26: The multi ai ultrasound application has [settings](#) for multi GPU in a different YAML file. It can be controlled by the inference parameters.

Q27: What is the role of a scheduler in Holoscan?

A27: The scheduler is responsible for determining when each operator in an application will execute.

Q28: How many types of schedulers are available in the Holoscan SDK?

A28: There are three [schedulers](#) available: Greedy Scheduler, Multi-Thread Scheduler, and Event-Based Scheduler.

Q29: Which scheduler is used by default for non-distributed applications?

A29: Non-distributed applications use the Greedy Scheduler by default.

Q30: What is the main characteristic of the Greedy Scheduler?

A30: The Greedy Scheduler has only a single worker thread that executes operators sequentially in a deterministic order.

Q31: How does the Multi-Thread Scheduler work?

A31: It's a polling-based scheduler with a user-defined number of worker threads and a dedicated thread that polls operators at a user-defined interval.

Q32: What is unique about the Event-Based Scheduler?

A32: The Event-Based Scheduler waits for events indicating changes in operator readiness, rather than constantly polling.

Q33: How can the Event-Based Scheduler reduce CPU overhead?

A33: By eliminating the need for constant polling, it can significantly reduce CPU usage compared to the Multi-Thread Scheduler in certain scenarios.

Q34: In what situations do Multi-thread and Event-Based Schedulers show benefits?

A34: They show benefits in scenarios with multiple operators that can run simultaneously, potentially providing significant speedup compared to the Greedy Scheduler.

Q35: How do the Multi-Thread and Event-Based Schedulers compare in terms of performance?

A35: They often have similar runtime performance, but the Event-Based Scheduler tends to have lower CPU overhead on average.

Q36: Are there scenarios where using multi-thread schedulers might not be beneficial?

A36: Yes, for linear inference pipelines or applications with minimal computation per operator, multi-thread schedulers might not provide significant benefits and could even introduce overhead.

Q37: How does the number of worker threads affect performance in multi-thread schedulers?

A37: Increasing the number of worker threads can improve performance up to a point, but it also increases CPU usage.

Q38: Is there any memory pool (allocator) that supports both host and device memory?

Please use `RMMAllocator` for this purpose. It supports simultaneous memory pools for CUDA device memory and pinned host memory. A `BlockMemoryPool` can be used on either host or device memory, but cannot support both types at the same time. `UnboundedAllocator` also supports both host and device memory, but is not a memory pool (it allocates and frees new memory each time).

36.5 Performance

Q1: What performance tools are available in Holoscan SDK?

A1: Holoscan SDK provides several performance tools, including Data Flow Tracking, GXF job statistics, and a Video Pipeline Latency Tool.

Q2: What is Data Flow Tracking in Holoscan SDK?

A2: Data Flow Tracking is a mechanism to profile applications and analyze fine-grained timing properties and data flow between operators in a fragment's graph. For more detailed information, please refer to the [Data Flow Tracking section](#) in the Holoscan User Guide.

Q3: How do I enable Data Flow Tracking in my Holoscan application?

A3: You can enable Data Flow Tracking by calling the `track()` method in C++ or using the `Tracker` class in Python before running your application.

Q4: What metrics can I retrieve using Data Flow Tracking?

A4: You can retrieve metrics such as maximum, average, and minimum end-to-end latencies, as well as the number of messages sent from root operators.

Q5: How can I customize Data Flow Tracking?

A5: You can customize Data Flow Tracking by configuring parameters such as the number of messages to skip at the start and end, and setting a latency threshold to ignore outliers.

Q6: How do I enable GXF job statistics?

A6: You can enable GXF job statistics by setting the environment variable `HOLOSCAN_ENABLE_GXF_JOB_STATISTICS` to true.

Q7: Can I save GXF job statistics to a file?

A7: Yes, you can save GXF job statistics to a JSON file by setting the `HOLOSCAN_GXF_JOB_STATISTICS_PATH` environment variable. For more information on the GXF job statistics, please refer to [this](#) section in the User Guide.

Q8: How does the Video Pipeline Latency Tool work?

A8: It generates a sequence of known video frames, transfers them back to an input component using a physical loopback cable, and measures the latency at various stages of the process.

Q9: Can the Video Pipeline Latency Tool simulate GPU workload?

A9: Yes, the tool has an option to simulate GPU workload by running an arbitrary CUDA loop a specified number of times before frame generation. For more information on the Video Pipeline Latency Tool, please refer to [this](#) section in the User Guide.

Q10: What types of producers and consumers are supported by the Video Pipeline Latency Tool?

A10: The tool supports various producers (OpenGL GPU Direct Rendering, GStreamer GPU Rendering, AJA Video Systems) and consumers (V4L2, GStreamer, AJA Video Systems) for different video input/output scenarios.

Q11: How do NVTX markers work in Holoscan SDK?

A11: This is how NVTX markers added to a Holoscan application work:

- The multithreaded scheduler starts a worker thread, checks the status of entities (also known as Holoscan Operators), and executes each entity using `EntityExecutor::executeEntity()` method in GXF.
- `EntityExecutor::executeEntity()` calls the `EntityExecutor::EntityItem::execute()` GXF method for the given entity ID
- The `EntityExecutor::EntityItem::execute()` method checks the scheduling status of the entity (Holoscan Operator) and then calls the `EntityExecutor::EntityItem::tick()` method in GXF
- In the `EntityExecutor::EntityItem::tick()` method it is where the annotation happens, the following steps occur:
 1. `router->syncInbox(entity);` is called to synchronize the inbox. For example, for the given Holoscan operator, `UCXReceiver` (input port) receives data from the network and pushes it into the queue in the `UCXReceiver` object. Data in the queue can be retrieved by calling the `receive()` method within the `Operator::compute()` method.
 2. For each codelet in the entity (in Holoscan, an entity can have only one codelet), `EntityExecutor::EntityItem::tickCodelet()` is called, which in turn calls `Codelet::tick()` (in Holoscan, this is the `Operator::compute()` method) (Figure 5).
 3. `router->syncOutbox(entity);` is called to synchronize the outbox. For example, for the given Holoscan operator, the data pushed to the queue in the `UCXTransmitter` object (output port) via `emit()` method calls in the `Operator::compute()` method is sent to the network using UCX.

During these calls, the system measures statistics, executes monitors (if any), and executes controllers (if any).

It is important to note that the tick codelet NVTX annotation doesn't cover `router->syncInbox(entity);` and `router->syncOutbox(entity);`. This means that the time range captured by NVTX measures only the duration for executing the `Codelet::tick()`. The time for sending and receiving data via `UCXTransmitter/UCXReceiver` is not measured by looking at the annotation range.

Q12:During the performance analysis of my Holoscan application, I've observed significant latency issues that are negatively impacting real-time performance. I've compiled a timing breakdown for each operator in the pipeline, which I've included below for reference.

Initially, I had assumed that each Holoscan Operator processed frames independently and concurrently. However, my observations suggest that the entire pipeline is processing each frame sequentially, which appears to be suboptimal for my real-time requirements.

Currently, my visualization component is only achieving approximately 15 fps, which falls short of my performance target. Given that my pipeline has a total execution time of approximately 70ms, I'm concerned that it may only be capable of processing one frame every 70ms.

Could you provide more detailed information about the implementation and potential benefits of Schedulers in Holoscan (as referenced in the NVIDIA documentation on Schedulers)? Specifically, I'm interested in understanding if and how Schedulers could be leveraged to address my performance concerns.

Here's the timing breakdown for each operator in my pipeline:

- Replayer: 24.145 ms
- ImageProcessing: 18.289 ms
- Preprocessor: 1.213 ms
- Inference: 23.861 ms
- Postprocessor: 0.275 ms
- PostImageProcessing: 2.695 ms
- Viz: 1.575 ms

A12: The following scheduler mechanisms can potentially impact the performance of your application:

1. Frame Processing Parallelization: The Multi-threaded Scheduler (MTS) and Event-based Scheduler (EBS) are designed to enable concurrent processing of multiple frames. These schedulers allow the initiation of processing for subsequent frames while preceding frames are still being processed by other operators in the pipeline.
2. Latency vs. Throughput Considerations: It's important to distinguish between end-to-end latency and throughput in the context of application performance. While MTS and EBS can potentially enhance the overall throughput of an application (defined as the number of frames processed per unit time), they do not necessarily reduce the end-to-end latency. End-to-end latency refers to the time required for a single frame to traverse the entire pipeline from source to sink.
3. Current Scheduler Implementation Status: Please note that the MTS and EBS are currently undergoing optimization. In their present state, they may exhibit higher latency compared to the greedy scheduler.
4. Inter-Operator Dependencies: It's crucial to understand that operators within a pipeline do not function in complete isolation. The pipeline architecture incorporates double-buffer queues between operators, with scheduling conditions applied to these queues. This design introduces data dependencies between adjacent nodes in the application graph, which influences the overall execution flow and timing.

Q13:How can I improve the performance of VideoStreamRecorderOp by reusing GPU memory?

A13:The VideoStreamReplayerOp can reuse the CUDA device buffers and avoid alloc/free for each frame by using [BlockMemoryPool](#).

The VideoStreamReplayerOp does not have a parameter (such as allocator) to use a custom allocator, even though the user can specify `entity_serializer`.

- The current implementation always uses `holoscan::StdEntitySerializer` and `holoscan::StdComponentSerializer` with `UnboundedAllocator`, regardless of the user-specified `entity_serializer` parameter.

- The storage type of the tensor (GPU or CPU) created by the VideoStreamReplayerOp depends on the input video file to which the tensor object is serialized. Therefore, without updating the `holoscan::StdComponentSerializer` implementation, VideoStreamReplayerOp cannot blindly use a specific memory pool allocator that requires memory storage type, memory pool size, etc.

Q14: I've observed some CUPVA-CUDA interop-related latencies in this application that are not present in our CUPVA test applications. One notable difference between the Holohub application and the CUPVA test application lies in the method of CUDA stream creation.

In the CUPVA test application, we create the CUDA stream as follows:

```
cudaStreamCreateWithFlags(&cuStream, cudaStreamNonBlocking)
```

In contrast, the Holohub application utilizes a CudaStreamPool, created in this manner:

```
const std::shared_ptr<CudaStreamPool> cuda_stream_pool = make_resource<CudaStreamPool>(
    ↪ "cuda_stream", 0, 0, 0, 1, 5);
// or
const std::shared_ptr<CudaStreamPool> cuda_stream_pool = make_resource<CudaStreamPool>(
    ↪ "cuda_stream", 0, cudaStreamNonBlocking, 0, 1, 5);
```

The CUDA stream is then retrieved in the compute API using:

```
auto cudaStream = cuda_stream_handler_.get_cuda_stream(context.context());
```

I would like to clarify the following points:

1. When calling `get_cuda_stream(...)`, will the default stream be utilized, or will it be a non-default stream?
2. Is there a method to ensure that the default CUDA stream is not used in the pool, given that we currently lack support for it in CUPVA?

A14: It is important to know that CUDA stream management in both GXF and Holoscan is currently in a state of evolution.

In the current Holoscan implementation, CUDA streams are managed through the `holoscan::CudaStreamHandler` class. This class offers utility methods to define the operator specification with `CudaStreamPool`.

To utilize non-default streams, the application's `compose()` method should create a `cuda_stream_pool` as follows:

```
const std::shared_ptr<CudaStreamPool> cuda_stream_pool = make_resource<CudaStreamPool>(
    ↪ "cuda_stream", 0, cudaStreamNonBlocking, 0, 1, 5);
```

Note that using 1 or `cudaStreamNonBlocking` for the flag parameter ensures the use of non-default streams.

For proper CUDA stream creation and sharing across operators in the application workflow:

1. In the Application's `compose()` method:
 - Create `CudaMemoryPool` and pass it as a parameter to the operators in the workflow graph.
2. For each operator in the workflow:
 - Define a `holoscan::CudaStreamHandler` member variable in the operator class (e.g., `cuda_stream_handler_`).
 - Call `cuda_stream_handler_.define_params(spec);` in the `setup(OperatorSpec& spec)` method.
 - In the `compute()` method, use the following calls:

- `cuda_stream_handler_.from_message(context.context(), in_message);` to retrieve CUDA stream information from the input message.
- `cuda_stream_handler_.get_cuda_stream(context.context());` to obtain the CUDA stream.
- `cuda_stream_handler_.to_message(out_message);` to set the currently used CUDA stream to the output message for subsequent operators.

Regarding your specific concern about forcing non-use of the default CUDA stream in the pool due to lack of support in CUPVA, there are two potential approaches:

1. Ensure that your application uses an appropriate CUDA stream pool configuration.
2. Implement error handling or exceptions at the application/operator level to prevent the use of the default CUDA stream.

It's worth noting that the `VideoReplayerOp` currently doesn't support this CUDA stream handling. Consideration is being given to supporting it alongside CUDA memory pool support in future updates.

Q15: I'm seeking to understand the memory allocation strategies employed within the Holoscan framework. Specifically, I'd like clarification on the following points:

1. **How is memory allocated across different components of Holoscan?**
2. **What is the timing of memory allocation in relation to the `compute()` method execution? Is memory allocated: a) Each time `compute()` is called, or b) Once at the beginning of execution and then reused?**
3. **What types of CUDA memory are utilized in Holoscan operations? Specifically: a) Is pinned memory used? b) Is CUDA managed memory employed? c) Do all memory exchanges remain within device (GPU) memory?**

A15: Memory allocation can be done either once and reused or separately on each compute call. It depends on how the user writes the compute method. We provide a `BlockMemoryPool` allocator class that allows reusing the same memory blocks on each call. Similarly there is ability to use CUDA streams and asynchronous memory allocation calls (`CudaStreamPool`). We hope to refactor over the coming months to make these easier to use than they are currently, but the capability is there now. `BlockMemoryPool` currently uses on device memory only. There is an `UnboundedAllocator` that can allocate on one of three places

- system memory (i.e. C++ new/delete)
- pinned host memory (`cudaMallocHost` / `cudaFreeHost`)
- device memory (`cudaMalloc` / `cudaFree`)

Q16: I'm running the endoscopy tool tracking application with a configuration that separates compute and graphics operations onto two distinct GPUs. I have a query about the data transfer mechanism between these GPUs:

1. **Is there an explicit use of `memcpy` for transferring data from the compute GPU to the graphics GPU?**
2. **In my analysis of the `nsys` profiler report, I've observed `MemcpyDtoH` and `MemcpyHToD` operations. This leads me to question whether the inter-GPU data transfer is actually being routed through the host system.**

A16: The tool tracking post process is doing device to host copies [here](#) and [here](#). These operations are also executed when the app is running on a single GPU. Holoviz is not doing any device to host operations, neither single nor multi GPU.

36.6 Troubleshooting

Q1: How can I debug Holoscan SDK examples and tests using Visual Studio Code?

A1: You can use the

```
./run vscode
```

command to launch VSCode in a development container. Configure CMake, build the source code, and use the Run and Debug view to start debugging sessions.

Q2: How can I get started with debugging my Holoscan application?

For debugging applications in Holoscan repo, refer to the [Debugging Section](#). For debugging applications in HoloHub, refer to HoloHub [tutorials](#) for strategies to set up debugging with Visual Studio Code or other tools such as GDB.

Q3: Is it possible to debug both C++ and Python components simultaneously in Holoscan SDK?

A3: Yes, you can use the Python C++ Debugger extension in VSCode to debug both C++ and Python components simultaneously. For more information, please refer to the [Debugging section](#) in the Holoscan SDK User Guide.

Q4: How do I analyze a core dump file when my application crashes?

A4: Use the gdb command with your application and core dump file, e.g.,

```
gdb <application> <coredump_file>
```

This will allow you to examine the stack trace and other debugging information.

Q5: What should I do if core dumps are not being generated?

A5: Enable core dumps by setting

```
ulimit -c unlimited
```

and configuring the `core_pattern` value. You may need to do this on the host system if working in a Docker container. For more information, please refer to the [Debugging section](#) in the [Holoscan User Guide](#).

Q6: How can I debug a distributed application using UCX?

A6: Set the `UCX_HANDLE_ERRORS` environment variable to control UCX's behavior during crashes. Options include printing backtraces, attaching a debugger, or freezing the application. For more information, please refer to the [Debugging section](#) in the [Holoscan User Guide](#).

The `UCX_LOG_LEVEL` environment variable can be set to “info” or higher level to see more detailed information about UCX transports used (the default level for UCX logging is “warn”). The full set of available UCX logging levels correspond to the list [here](#).

Q7: What tools are available for profiling Python applications in Holoscan?

A7: You can use tools like `pyinstrument`, `pprofile`, `yappi`, `cProfile`, or `line_profiler`. Each has different strengths and may be more suitable depending on your specific needs. For more information, please refer to the [Python debugging section](#) in the Holoscan User Guide.

Each profiler has its strengths and is suitable for different debugging scenarios, from high-level overviews to detailed line-by-line analysis. Please find below more details:

1. `pyinstrument`:

- Call stack profiler that highlights performance bottlenecks
- Provides easily understandable output directly in the terminal

- Multithreading-aware, suitable for use with multithreaded schedulers
- Visualizes the execution tree, showing time spent in each function

2. pprofile:

- Line-granularity profiler
- Thread-aware and deterministic
- Provides detailed statistics for each line of code
- Shows hit counts, time per hit, and percentage of total time for each line

3. yappi:

- Tracing profiler that is multithreading, asyncio, and gevent aware
- Can handle complex threading scenarios in Holoscan applications
- Provides hit counts for methods across different threads
- Requires setting a context ID callback for accurate thread identification

4. cProfile:

- Deterministic profiling module built into Python
- Provides a high-level overview of function calls and time spent
- Easy to use with minimal setup
- Good for identifying which functions are taking the most time overall

5. line_profiler:

- Offers line-by-line profiling of specific functions
- Provides detailed timing information for each line within a function
- Useful for pinpointing exact lines causing performance issues
- Requires adding @profile decorators to functions of interest

Q8: How do I measure code coverage for my Holoscan Python application?

A8: You can use

```
Coverage.py
```

to measure code coverage.

- install it with pip
- run your application with coverage
- generate reports using commands like coverage report or coverage html

For more detailed information, please refer to [Measuring Code Coverage](#) section in the Holoscan User Guide.

Q9: How can I trace function calls in my Python application?

A9: You can use the trace module to track function calls. Run your application with

```
python -m trace --trackcalls
```

or use the trace module programmatically in your code.

Q10: How can I leverage a large language model (LLM) to assist in debugging a complex and lengthy task implemented with the Holoscan SDK?

A10: Holoscan SDK offers to do the task piecemeal and get feedback from you as it completes each part of the task. This allows for step-by-step debugging of complex, lengthy processes. For Holochat links, please refer to [Holochat-GPT](#) to ask questions and receive feedback on building and debugging Holoscan SDK applications. Note that Holochat-GPT is not confidential and may not be trained on the latest Holoscan SDK release. Please refer to the [Holoscan SDK User Guide](#) for latest APIs.

Q11: How can I use Python's `coverage.py` with the Holoscan SDK?

A4: In Python, both coverage measurement and Python debugger is using `sys.settrace()` or `PyEval_SetTrace()` method to register a trace method for the current thread. When `coverage.py` or Python debugger is running, it calls those method (and `threading.settrace` for newly-created threads) for tracing Python code execution. However, when Python methods (such as `compute()`) are called by the threads (worker(s) of `GreedyScheduler/MultiThreadScheduler`) in Holoscan SDK, which is not derived from the Python's main thread, `sys.settrace()` (or `PyEval_SetTrace()`) is not called properly for those threads. The resolution is:

1. Capture the current trace method (by using `sys.gettrace()`, let's say `CURR_TRACE_METHOD`- if it exists when `Application.run()` method is called.
2. When Python methods (such as `Operator.compute()/Fragment.compose()/Operator.initialize()/Operator.start()/Operator.stop()`) are called, get current trace method (by using `sys.gettrace()`) and call `sys.settrace(CURR_TRACE_METHOD)` and set current stack frame's `f_trace` to `CURR_TRACE_METHOD` (current stack frame is available through `inspect.currentframe()`) if no trace method was set before.

- This process can be sped up by storing thread id-> <trace method> map (or using thread local variable) and checking if trace method is already registered to the current thread.

Python's `cProfile` module is using `sys.setprofile()` instead of `sys.settrace()` (because the profile method is called per method, which is more effective), and we can apply similar approach for enabling profiler on Holoscan's Python Operator methods.

Q12: How does Holoscan SDK reconcile command line arguments for multi-fragment applications?

A12: The CLI arguments (such as `--driver`, `--worker`, `--fragments`) are parsed by the `Application` class and the remaining arguments are available as `app.argv` property.

```
import argparse
import sys
from holoscan.core import Application

class MyApp(Application):
    def compose(self):
        pass

if __name__ == "__main__":
    app = MyApp()

    print("sys.argv:", sys.argv)
    print("app.argv:", app.argv)

    parser = argparse.ArgumentParser()
    parser.add_argument("--input")
    args = parser.parse_args(app.argv[1:])
    print("args:", args)
```

(continues on next page)

(continued from previous page)

```

app.run()

# $ python cli_test.py --address 12.3.0 --input a
# sys.argv: ['cli_test.py', '--address', '12.3.0', '--input', 'a']
# app.argv: ['cli_test.py', '--input', 'a']
args: Namespace(input='a')

```

Q13: Why is the Inference Operator rejecting the input shape for a CNN-LSTM model with a 5-dimensions input (batch, temporal_dim, channels, width, height) ? A13: In Holoscan SDK v2.4 and earlier, the InferenceOp supports rank only between 2 and 4.

Q14: I am attempting to profile a Holoscan application in a container using NVIDIA NSight Systems. I'm following the documentation available at <https://github.com/nvidia-holoscan/holohub/blob/main/doc/developer.md> and using a recent checkout of Holohub with the Holoscan v2.1 NGC image.

My process is as follows:

1. Initiate the development container with NSight profiling enabled:

```
./dev_container launch --nsys_profile
```

1. Launch the endoscopy tool tracking application with NSight profiling:

```
./run launch endoscopy_tool_tracking python --nsys_profile
```

However, I encounter the following error:

```

ERROR: For Nsight Systems profiling the Linux operating system's perf_event_paranoid_
↪level must be 2 or less.
See https://docs.nvidia.com/nsight-systems/InstallationGuide/index.html#linux-
↪requirements for more information.

```

How can I fix it ?

A14: The

```
sudo sh -c 'echo 2 >/proc/sys/kernel/perf_event_paranoid'
```

command needs to be executed on the host system, not inside the container.

- Check the current value

```
cat /proc/sys/kernel/perf_event_paranoid
```

- To temporarily change the value to 2 (which allows kernel profiling by unprivileged users):

```
sudo sh -c 'echo 2 >/proc/sys/kernel/perf_event_paranoid'
```

- To make the change permanent, edit /etc/sysctl.conf: Add or modify the line:

```
update /etc/sysctl.conf
```

- Then apply the changes:

```
sudo sysctl -p
```

- If you need to allow use of almost all events by all users, you can set the value to -1 instead of 2.
- The values and their meanings:
 - -1: Allow use of (almost) all events by all users
 - 0 or higher: Disallow ftrace function tracepoint by users without CAP_SYS_ADMIN
 - 1 or higher: Also disallow CPU event access by users without CAP_SYS_ADMIN
 - 2 or higher: Also disallow kernel profiling by users without CAP_SYS_ADMIN
- After making changes, you may need to restart your application or services that depend on these performance events.
- Refer to the [perf_event_open manpage](#) for more detailed information.

Remember that lowering this value increases the access unprivileged users have to performance data, which could potentially be a security concern in some environments. Always consider the security implications before making such changes. Refer to the [NVIDIA Nsight Systems installation guide](#) for more information.

Q15: I am developing an application utilizing two distinct processing graphs:

1. **VideoReplayerOp -> HolovizOp**
2. **CameraOp -> HolovizOp**

The first graph displays source video for camera observation, while the second applies AI processing to enhance camera input. Currently, I am employing two separate Holoviz instances, with the instance for source video (graph 1) exclusively utilizing the second monitor (DP-4).

I've encountered an issue during application termination: when pressing 'Esc' to exit, the main Holoviz instance on the primary screen closes as expected, but the source video continues playing on the secondary screen. While I can force quit the application using 'Ctrl+C', I am seeking a more elegant solution for proper termination.

Is there a recommended method for gracefully closing the entire application?

A15: To address your concerns about graceful application termination with multiple Holoviz instances, let's first understand how the ESC key functions in Holoviz:

1. ESC Key Behavior:
 - Holoviz monitors for window closure requests via the ESC key.
 - When pressed, it deactivates the associated HolovizOp by setting a boolean scheduling term to false.
2. Termination Scenarios:
 - Single HolovizOp: ESC key press closes the entire application.
 - Multiple HolovizOps: ESC only terminates the specific Holoviz instance, leaving others running.

Proposed Solution:

To achieve synchronized termination across all Holoviz instances:

1. Create a shared boolean scheduling condition.
2. For each HolovizOp in your application:
 - Set this condition as a general execution condition.
 - Importantly, also set it as the `window_close_scheduling_term` parameter.

Q16: I'm trying to use the `render_buffer_output` from Holoviz Python operator, but I get the following error :

```
[error] [entity.hpp:90] Unable to find component from the name 'render_buffer_output'
↳(error code: 24)
```

A16:The reason why you are getting this error is because HolovizOp returns a `gxf::VideoBuffer` which is not yet supported by Python in Holoscan SDK.

Q17:I am using an AGX Orin with an MST board, with 2 display monitors - DP-0.1 (touch screen) and DP-0.2 (an external main display). I am trying to force Holoviz to display on monitor DP-0.2, by setting `use_exclusive_display = True, display_name = "DP-0.2"`. But this results in error below:

```
[info] [exclusive_window.cpp:125] _____
[info] [exclusive_window.cpp:126] Available displays :
[info] [exclusive_window.cpp:129] ADA (DP-0.1)
[info] [exclusive_window.cpp:129] LG Electronics LG ULTRAWIDE (DP-0.2)
[info] [exclusive_window.cpp:134]
[info] [exclusive_window.cpp:135] Using display "LG Electronics LG ULTRAWIDE (DP-0.2)"
[info] [exclusive_window.cpp:148] X server is running, trying to acquire display
[error] [context.cpp:56] VkResult -13 - unknown
[error] [gxf_wrapper.cpp:57] Exception occurred when starting operator: 'holoviz' -
↳Failed to acquire display from X-Server.
```

As a potential fix, I have disabled the main display in `nvidia-settings` for the compositor, but the application still crashes with the following error:

```
[error] [context.cpp:56] /workspace/holoscan-sdk/modules/holoviz/thirdparty/nvpro_core/
↳nvvk/swapchain_vk.cpp(172): Vulkan Error : unknown
[error] [context.cpp:56] /workspace/holoscan-sdk/modules/holoviz/thirdparty/nvpro_core/
↳nvvk/swapchain_vk.cpp(172): Vulkan Error : unknown
[error] [context.cpp:56] /workspace/holoscan-sdk/modules/holoviz/thirdparty/nvpro_core/
↳nvvk/swapchain_vk.cpp(172): Vulkan Error : unknown
[error] [gxf_wrapper.cpp:57] Exception occurred when starting operator: 'holoviz' -
↳Failed to update swap chain.
```

A17:The progress indicated by the message “[info] [exclusive_window.cpp:161] Using display mode 1920x1080 60.000 Hz” is a positive sign. However, the failure of Vulkan swap chain creation with an unknown error is concerning. While we regularly test exclusive display with discrete GPUs (dGPUs), this is seldom tested with integrated GPUs (iGPUs). In the past, we’ve encountered issues with Vulkan features on iGPUs.

An alternative option would be to use fullscreen mode, which is known to work on iGPUs. It’s important to note that Holoviz always opens in fullscreen mode on the primary display, and display name selection is not currently supported.

Given that the display they want to use for Holoviz appears to be the primary display, you could try setting `fullscreen = True` instead of `use_exclusive_display = True`.

Q18: How can I use CuPy arrays with Holoscan SDK’s InferenceOp? A18: Both CuPy and Holoscan SDK support `_cuda_array_interface` to facilitate seamless array integration among libraries. Refer to the [HoloHub library integration tutorial CuPy section](#) for details on using CuPy arrays in a Holoscan SDK application.

Q19:I am running into these errors when using the Holoscan Packager in my networking environment:

```
curl: (6) Could not resolve host: github.com.
Failed to establish a new connection: [Errno -3] Temporary failure in name solution...
```

A19: To resolve these errors, edit the `/etc/docker/daemon.json` file to include `dns` and `dns-search` fields as follows:

```
{
  "default-runtime": "nvidia",
  "runtimes": {
    "nvidia": {
      "args": [],
      "path": "nvidia-container-runtime"
    }
  },
  "dns": ["IP-1", "IP-n"],
  "dns-search": ["DNS-SERVER-1", "DNS-SERVER-n"]
}
```

You may need to consult your IT team and replace `IP-x` and `DNS-SERVER-x` with the provided values.

Q20: I am seeing the following error when trying to use the `RMMAllocator`

When running the application, if it fails to start with an error like the following being logged:

```
[error] [rmm_allocator.cpp:74] Unexpected error while initializing RMM Allocator rmm_
↪allocator: std::bad_alloc: out_of_memory: RMM failure at:bazel-out/k8-opt/bin/external/
↪rmm/_virtual_includes/rmm/rmm/mr/device/pool_memory_resource.hpp:424: Maximum pool_
↪size exceeded
```

This indicates that the requested memory sizes on host and/or device exceed the available memory. Please make sure that your device supports the specified memory size. Also check that the specified the values for `device_memory_initial_size`, `device_memory_max_size`, `host_memory_initial_size` and `host_memory_max_size` were specified using the intended units (B, KB, MB, GB or TB).

36.7 Miscellaneous

Q1: Can I use DLA cores with the Holoscan SDK?

A1: There are 2 situations in which Holoscan SDK can support Deep Learning Accelerator (DLA) cores. You can configure your application to offload certain inference tasks to DLA cores to improve performance and efficiency.

- User created engine file: If the TensorRT engine file is created with `-useDLACore=0` and the engine file is used in the Holoscan SDK inference framework, then it will use DLA (core 0) as HoloInfer just guides the execution to TensorRT and it will automatically pick it up. The user must give the path to engine file in `model_path_map` and enable the flag `is_engine_path=true` in the inference parameter set.
- If user is creating engine file via HoloInfer: HoloInfer currently does not support using any DLA cores for engine creation (even if it's available) and it defaults to GPU.

Q2: How can I generate HSDK applications from Graph Composer?

A2: In Graph Composer, graph nodes (entities) are based on GXF Codelets/Components (from GXF extensions) that are registered in the Graph Composer registry. Currently, none of the Holoscan operators are registered in this registry. However, we have a method to convert Holoscan Operators into GXF extensions. Please find example code below :

- https://github.com/nvidia-holoscan/holoscan-sdk/tree/main/examples/wrap_operator_as_gxf_extensio
- https://github.com/nvidia-holoscan/holoscan-sdk/tree/main/gxf_extensions/gxf_holoscan_wrapper

To generate or run an application graph (pipeline) described in Graph Composer, we need a method to import GXF Codelets/Components as Holoscan Operators/Resources. Currently, users need to manually wrap GXF Codelets/Resources to convert them into Holoscan Operators/Resources.

Q3: How can I support external events like React?

A3: Regarding parameter flags, we support GXF's optional/dynamic parameter flag (Figure 1) and can set it in the `Operator::setup(OperatorSpec& spec)` method using `spec.param(..., ParameterFlag::kDynamic);` (Figure 2). However, this parameter flag is primarily for wrapping GXF Codelets. When implementing a Holoscan Native operator, there is no runtime check for parameter value changes, allowing us to update parameter values and use them without setting the parameter flag (`holoscan::Parameter<T>` simply acts as a value holder using `std::any`).

To support external events/controls (such as for Qt), we can implement a Holoscan Resource (GXF Component) that handles events and state management. This would allow a Holoscan operator to update the state based on events and use it, similar to React's state management approach. `QtHoloscanVideo` (Figure 3) could be a good candidate for such a Holoscan Resource.

For reference, `CudaStreamPool` is an example of a Holoscan Resource (wrapping the GXF `CudaStreamPool` component), and the object is passed to the operator as an argument (Figure 4). However, it's not necessary to pass Holoscan resources as arguments. Instead, we can provide them to the operator (Figure 5) and access them through the `resource<T>()` method inside the `compute()` method (Figure 6). In Python, resources can be accessed via the `op.resources` property (which is of type `dict[str, Resource]`).

Holoscan supports Native Resources for C++ (with an example demonstrating this feature), but Native Python resources have not been implemented yet.

Given this situation, it might be preferable to implement it as a GXF Component and expose it as a C++/Python Holoscan Resource. This approach would allow `QtVideoOp` (in Figure 3) to define a public method for accessing the event/control Holoscan resource object from the Qt app logic (possibly through a `HoloscanVideo*` object in `QtVideoOp`).

```
public > include > holoscan > core > parameter.hpp > {} holoscan
31
32 You, 5 months ago | 3 authors (You and others)
33 namespace holoscan {
34
35 /**
36  * @brief Enum class to define the type of a parameter.
37  *
38  * The parameter flag can be used to control the behavior of the parameter.
39  */
40 enum class ParameterFlag {
41     /// The parameter is mandatory and static. It cannot be changed at runtime.
42     kNone = 0,
43     /// The parameter is optional and might not be available at runtime. Use `Parameter::try_get()` to
44     /// get the value.
45     kOptional = 1,
46     /// The parameter is dynamic and might change at runtime.
47     kDynamic = 2,
48 };
```

Figure 1

```

63  /**
64   * @brief Define a parameter for this component.
65   *
66   * @tparam typeT The type of the parameter.
67   * @param parameter The parameter to define.
68   * @param key The key (name) of the parameter.
69   * @param flag The flag of the parameter (default: ParameterFlag::kNone).
70   */
71  template <typename typeT>
72  void param(Parameter<typeT>& parameter, const char* key,
73           ParameterFlag flag = ParameterFlag::kNone) {
74      param(parameter, key, "N/A", "N/A", flag);
75  }
76
77  /**
78   * @brief Define a parameter for this component.
79   *
80   * @tparam typeT The type of the parameter.
81   * @param parameter The parameter to define.
82   * @param key The key (name) of the parameter.
83   * @param headline The headline of the parameter.
84   * @param flag The flag of the parameter (default: ParameterFlag::kNone).
85   */
86  template <typename typeT>
87  void param(Parameter<typeT>& parameter, const char* key, const char* headline,
88           ParameterFlag flag = ParameterFlag::kNone) {
89      param(parameter, key, headline, "N/A", flag);
90  }

```

Figure 2

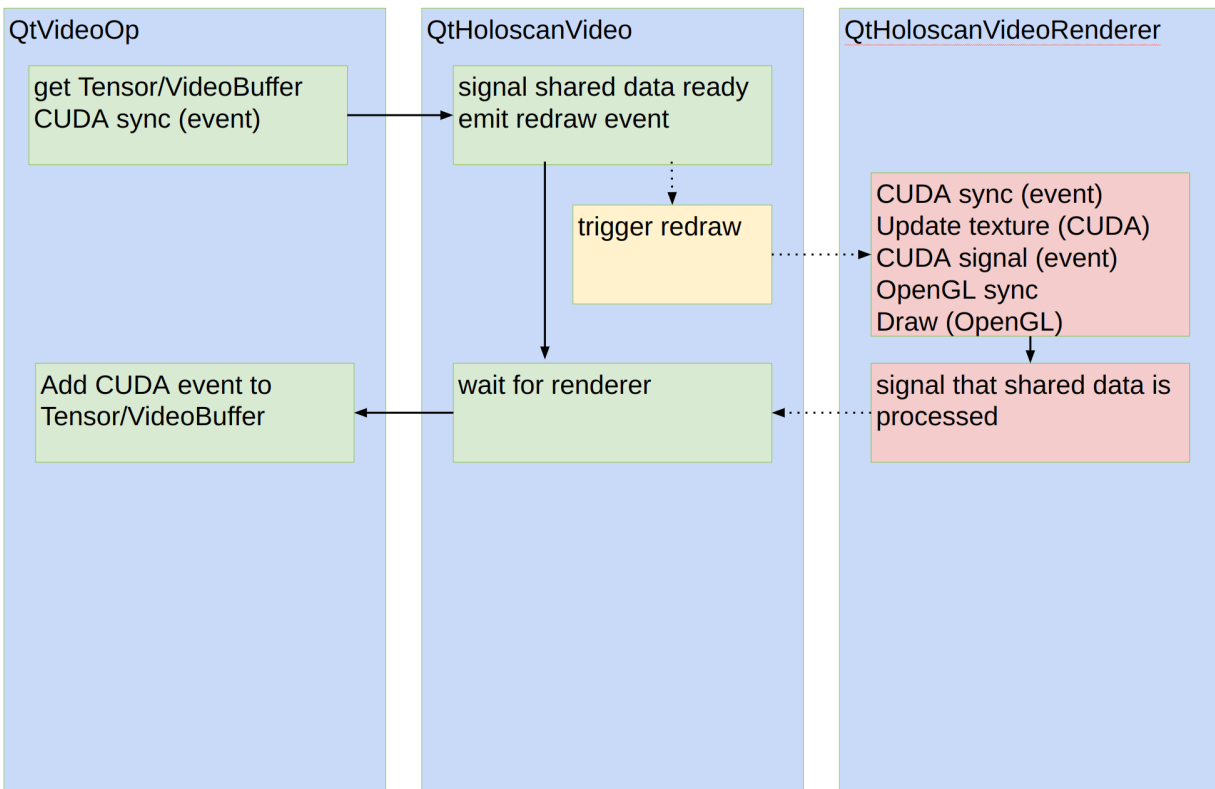


Figure 3


```

holohub-internal > applications > endoscopy_tool_tracking > cpp > main.cpp > App > compose()
89     }
90
91     const std::shared_ptr<CudaStreamPool> cuda_stream_pool =
92         make_resource<CudaStreamPool>("cuda_stream", 0, 0, 0, 1, 5);
93
94     auto format_converter = make_operator<ops::FormatConverterOp>(
95         "format_converter",
96         from_config(is_aja_source ? "format_converter_aja" : "format_converter_replayer"),
97         Arg("pool") =
98             make_resource<BlockMemoryPool>("pool", 1, source_block_size, source_num_blocks),
99         Arg("cuda_stream_pool") = cuda_stream_pool);
100

```

Figure 4

```

You, 7 months ago | 1 author (You)
74 class MinimalNativeResourceApp : public holoscan::Application {
75 public:
76     void compose() override {
77         using namespace holoscan;
78
79         auto res =
80             make_resource<MinimalNativeResource>("string_native_resource", from_config("str_value"));
81         auto res2 = make_resource<MinimalNativeResource>(
82             "hardcoded_native_resource", Arg("str_value") = std::string("hardcoded_string"));
83         auto res3 = make_resource<MinimalNativeResource>("empty_native_resource");
84
85         auto op = make_operator<ops::MinimalNativeResourceOp>(
86             "min_op", make_condition<CountCondition>(1), res, res2, res3);
87         add_operator(op);
88     }
89 };

```

Figure 5

```

53 void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {
54     auto res = resource<MinimalNativeResource>("string_native_resource");
55     if (res) {
56         HOLOSCAN_LOG_INFO("MinimalNativeResource - string_native_resource.string_param: {}",
57             res->string_param());
58     }
59     auto res2 = resource<MinimalNativeResource>("hardcoded_native_resource");
60     if (res2) {
61         HOLOSCAN_LOG_INFO("MinimalNativeResource - hardcoded_native_resource.string_param: {}",
62             res2->string_param());
63     }
64     auto res3 = resource<MinimalNativeResource>("empty_native_resource");
65     if (res3) {
66         HOLOSCAN_LOG_INFO("MinimalNativeResource - empty_native_resource.string_param: '{}'",
67             res3->string_param());
68     }
69 };

```

Figure 6

Q5: I'm encountering difficulties implementing the `PYBIND11_OVERRIDE_PURE` mechanism in `pybind11`. Specifically, I'm trying to override a pure virtual C++ method with a Python subclass, but it's not working as expected. The Python subclass doesn't seem to be successfully overriding the C++ method. **A5:** A potential fix is to keep a global reference to the Python object. Please refer to the fix provide [here](#).

This code addresses a potential issue in `pybind11`'s handling of class inheritance and instance management. The problem arises when mocking the C++ `RoceReceiverOp` class with a Python `InstrumentedReceiverOperator` class. Here's a breakdown of the situation:

1. Issue: When the Holoscan app runs, it calls methods from `RoceReceiverOp` instead of the intended

InstrumentedReceiverOperator methods.

2. Cause: The InstrumentedReceiverOperator instance seems to disappear from pybind11's internal registry during app execution. This occurs despite expectations that passing it to C++ should maintain its lifecycle.
3. Debugging attempts: Efforts to track the instance deregistration (via breakpoints in pybind11's `deregister_instance()`) were unsuccessful, leaving the exact cause unclear.
4. Workaround: To prevent the premature destruction of the InstrumentedReceiverOperator instance, the code maintains a global reference to it.

This solution ensures the mocked class instance remains available throughout the app's lifecycle, allowing proper method overriding and execution.

36.8 Additional Resources

Q1: Where can I find additional resources and support for the Holoscan SDK?

A1: Additional resources and support can be found on:

- The NVIDIA developer page. For more information, please refer to [this link](#).
- Holoscan SDK GitHub repository. Please refer to [this link](#).
- NVIDIA Developer Forums for community support and discussions. For the Holoscan SDK forum, please refer to [this link](#).

Q2: How can I contribute to the Holoscan SDK?

A2: The Holoscan SDK is open-source. You can contribute by:

- Submitting pull requests for bug fixes or new features. For more detailed information on how to contribute, please refer to [this link](#).
- Participating in community discussions on the [Holoscan SDK forum](#)