# INTRODUCTION

## Overview

## Content

1. Extensions
2. Applications
3. Video Pipeline Latency Tool

## Changes Since Holoscan SDK 0.1.0

1. GXF Relative to DeepStream
2. Release Artifacts

## Core Technologies

1. The GXF Framework
2. GPUDirect RDMA
3. TensorRT Optimized Inference
4. CUDA and OpenGL Interoperability
5. Accelerated Image Transformations

## Install and Use the Holoscan SDK

1. Using the container from NGC
2. From source

## AJA Video Systems

1. Installing the AJA Hardware
2. Installing the AJA Software
3. Using AJA Devices in Containers
4. Troubleshooting

## SDK Sample Applications

1. Endoscopy Tool Tracking
2. Ultrasound Segmentation App & Customization

## SDK Extensions

1. GXF Standard Extensions
2. Holoscan SDK GXF Extensions

## SDK Development Guide

1. Holoscan Core Concepts
2. Developing Holoscan Extensions
3. Holoscan Applications by Example
NVIDIA Clara Holoscan is the AI computing platform for medical devices, consisting of Clara Developer Kits and the Clara Holoscan SDK. Clara Holoscan allows medical device developers to create the next-generation of AI-enabled medical devices and take it to the market using Clara Holoscan MGX.

The Clara Holoscan SDK version 0.2.0 provides the foundation to run streaming applications on Clara Developer Kits, enabling real-time AI inference and fast IO. These capabilities are showcased within the reference extensions and applications described below.
2.1 Extensions

The core of the Clara Holoscan SDK is implemented within extensions. The extensions packaged in the SDK cover tasks such as IO, inference, image processing, and visualization. They rely on a set of Core Technologies highlighted below.

This guide will provide more information on the existing extensions, and how to create your own.

2.2 Applications

This SDK includes two sample applications to show how users can implement their own end-to-end inference pipeline for streaming use cases. This guide provides detailed information on the inner-workings of those applications, and how to create your own.

See below for some information regarding the two sample applications:

2.2.1 Ultrasound Segmentation

Generic visualization of segmentation results based on a spinal scoliosis segmentation model of ultrasound videos. The model used is stateless, so this workflow could be configured to adapt to any vanilla DNN model. This guide will provide more details on the inner-workings of the Ultrasound Segmentation application and how to adjust it to use your own data.

The model is from a King’s College London research project, created by Richard Brown and released under the Apache 2.0 license.

The ultrasound dataset is released under the CC BY 4.0 license. When using this data, please cite the following paper:


Refer to the sample data resource on NGC for more information related to the model and video.
2.2.2 Endoscopy Tool Tracking

Leveraging a long-short term memory (LSTM) stateful model, this application demonstrates the use of custom components for surgical tool tracking and classification, as well as composition and rendering of text, tool position, and mask (as heatmap) overlayed on the original frames. This guide provides more details on the inner-workings of the Endoscopy Tool Tracking application.

The convolutional LSTM model and sample surgical video data were kindly provided by Research Group Camma, IHU Strasbourg & University of Strasbourg:


Refer to the sample data resource on NGC for more information related to the model and video.

2.3 Video Pipeline Latency Tool

To help developers make sense of the overall end-to-end latency that could be added to a video stream by augmenting it through a GPU-powered Holoscan platform such as the Clara Holoscan Developer Kit, the Holoscan SDK includes a Video Pipeline Latency Measurement Tool. This tool can be used 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 Clara Holoscan platforms. The measurements taken by this tool can then be displayed with a comprehensive and easy-to-read visualization of the data.
CHAPTER
THREE

CHANGES SINCE HOLOSCAN SDK 0.1.0

The following table outlines the component versions that have been upgraded or removed in version 0.2.0:

<table>
<thead>
<tr>
<th>Component</th>
<th>Holoscan 0.2.0</th>
<th>Holoscan 0.1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>JetPack</td>
<td>5.0 HP1</td>
<td>4.5</td>
</tr>
<tr>
<td>Jetson Linux</td>
<td>34.1.2</td>
<td>32.5</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>20.04</td>
<td>18.04</td>
</tr>
<tr>
<td>dGPU Drivers</td>
<td>510.73.08</td>
<td>465</td>
</tr>
<tr>
<td>CUDA</td>
<td>11.6.1</td>
<td>11.1</td>
</tr>
<tr>
<td>TensorRT</td>
<td>8.2.3</td>
<td>7.2.2</td>
</tr>
<tr>
<td>cuDNN</td>
<td>8.3.3.40</td>
<td>8.0</td>
</tr>
<tr>
<td>GXF</td>
<td>2.4.2</td>
<td>-</td>
</tr>
<tr>
<td>DeepStream</td>
<td>-</td>
<td>5.0</td>
</tr>
</tbody>
</table>

3.1 GXF Relative to DeepStream

The most significant change in Holoscan 0.2.0 is the shift of the core backend from Deepstream to NVIDIA’s Graphical eXecution Framework (GXF) for improved latency of real-time streaming applications in healthcare settings.

While DeepStream is built on top of GStreamer and is primarily focused on audio and video streams, GXF provides much more flexibility as it can support any type of streaming data and can be used in endoscopy, ultrasound, genomics, microscopy, robotics, and more. This user guide goes into more details on GXF’s core and how to leverage it.

3.2 Release Artifacts

The delivery and installation method used by Holoscan 0.2.0 has also changed. Components that were once installed as Debian packages by SDK Manager are now provided via open-source GitHub repositories and/or container images on the NVIDIA GPU Cloud (NGC). These repositories and container images will be documented in the relevant sections throughout this documentation.
Clara Holoscan accelerates streaming AI applications by leveraging both hardware and software. On the software side, the Clara Holoscan SDK 0.2.0 relies on multiple core technologies to achieve low latency and high throughput, including:

- **The GXF Framework**
- **GPUDirect RDMA**
- **CUDA and OpenGL Interoperability**
- **TensorRT Optimized Inference**
- **Accelerated Image Transformations**

### 4.1 The GXF Framework

At its core, GXF provides a very thin API with a plug-in model to load in custom extensions. Applications built on top of GXF are composed of components. The primary component is a Codelet that provides an interface for `start()`, `tick()`, and `stop()` functions. Configuration parameters are bound within the `registerInterface()` function.

In addition to the Codelet class, there are several others providing the underpinnings of Holoscan:

- **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 up-front allocating a large contiguous memory pool and then re-using 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.
- **Transmitters, Receivers, 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 that specify constants used by the Codelet loaded from the application yaml file modifiable without recompiling.
4.2 GPUDirect RDMA

Copying data between a third party PCIe device and a GPU traditionally requires two DMA operations: first the data is copied from the PCIe device to system memory, then it’s copied from system memory to the GPU.

![Fig. 4.1: Data Transfer Between PCIe Device and GPU Without GPUDirect RDMA](image)

For data that will be processed exclusively by the GPU, this additional data copy to system memory goes unused and wastes both time and system resources. GPUDirect RDMA optimizes this use case by enabling third party PCIe devices to DMA directly to or from GPU memory, bypassing the need to first copy to system memory.

![Fig. 4.2: Data Transfer Between PCIe Device and GPU With GPUDirect RDMA](image)

NVIDIA takes advantage of RDMA in many of its SDKs, including Rivermax for GPUDirect support with ConnectX network adapters, and GPUDirect Storage for transfers between a GPU and storage device. NVIDIA is also committed to supporting hardware vendors enable RDMA within their own drivers, an example of which is provided by AJA Video Systems as part of a partnership with NVIDIA for the Clara Holoscan SDK. The AJASource extension 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 userspace applications for the RHS Research PicoEVB and HiTech Global HTG-K800 FPGA boards.

4.3 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 Clara Developer Kits.

GXF comes with a TensorRT base extension which is extended in the Holoscan SDK: the updated TensorRT extension is able to selectively load a cached TensorRT model based on the system GPU specifications, making it ideal to interface with both the Clara AGX Developer Kit and the Clara Holoscan Developer Kit.

4.4 CUDA and OpenGL Interoperability

OpenGL is commonly used for realtime visualization, and like CUDA, is executed on the GPU. This provides an opportunity for efficient sharing of resources between CUDA and OpenGL.

The OpenGL and Segmentation Visualizer extensions use the OpenGL interoperability functions provided by the CUDA runtime API. This API is documented further in the CUDA Toolkit Documentation.

This concept can be extended to other rendering frameworks such as Vulkan.

4.5 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 extension of the Clara Holoscan SDK.
Clara Holoscan SDK can be installed from source or using a pre-built container runtime.

5.1 Using the container from NGC

The Clara Holoscan Sample Applications container is the simplest way to run the sample applications as it includes all necessary binaries and datasets, and allows for some customization of the application graph and its parameters.

Refer to the overview of the container on NGC for prerequisites, setup, and run instructions.

Note: The sample applications container from NGC does not include build dependencies to update or generate new extensions, or to build new applications with other extensions. Refer to the section below to do this from source.

5.2 From source

The Clara Holoscan SDK source code provides reference implementations for the GXF extensions and the sample applications, as well as infrastructure for building the current extensions and applications or your own.

Refer to the top-level README.md in the open-source repository on Github for prerequisites, setup, and run instructions.
AJA provides a wide range of proven, professional video I/O devices, and thanks to a partnership between NVIDIA and AJA, Clara Holoscan supports the AJA NTV2 SDK and device drivers as of the NTV2 SDK 16.1 release.

GPU compute performance is a key component of Clara Holoscan hardware platforms, and to optimize GPU based video processing applications the AJA drivers and SDK now 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 sysmem 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 Clara Holoscan platforms. Note that the AJA NTV2 SDK support for Clara Holoscan includes all of the AJA Developer Products, though the following instructions have only been verified for the Corvid 44 12G BNC and KONA HDMI products, specifically.

Note: The addition of an AJA device to the Clara Holoscan hardware platform 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 GXF reference 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 Installing the AJA Hardware**

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.
6.2 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 GXF 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

6.2.1 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 and checkout the correct branch as needed for Holoscan SDK.

```bash
$ git clone https://github.com/ibstewart/ntv2.git
$ export NTV2=$(pwd)/ntv2
$ cd $NTV2
$ git checkout holoscan-v0.2.0
```

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.
6.2.2 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}/bin directory.

```bash
$ cd ${NTV2}/ajadriver/linux
$ export AJA_RDMA=1
$ make -j
```

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

```bash
$ sudo sh ${NTV2}/bin/load_ajantv2
loaded ajantv2 driver module
created node /dev/ajantv20
```

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.

6.2.4 Building and Installing the AJA NTV2 SDK

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

```bash
$ 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
```

6.2.5 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:

```bash
$ 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'
```

(continues on next page)
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)
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 testrdma
   utility can be used:

   $ testrdma -t500

test device 0 start 0 end 7 size 8388608 count 500
frames/errors 500/0

6.3 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. For example:

   docker run -it --rm --runtime=nvidia \
   -v $(pwd):/workspace \ 
   --device /dev/ajantv20:/dev/ajantv20 \ 
   clara-holoscan-release:0.2

6.4 Troubleshooting

1. 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)
b. Make sure that the AJA drivers are loaded properly (see *Loading the AJA NTV2 Drivers*):

```bash
$ 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
```

2. **Problem:** The `testrdma` 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.
This section explains how the user can run the Clara Holoscan sample apps. The sample apps include

- tool tracking in endoscopy video using an LSTM model
- semantic segmentation bone contours with hyperechoic lines

Each app comes with support for an AJA capture card or replay from a video file included in the sample app container.

### 7.1 Endoscopy Tool Tracking

The Endoscopy tool tracking application provides an example of how an endoscopy data stream can be captured and processed using the GXF framework on multiple hardware platforms.

#### 7.1.1 Input source: Video Stream Replayer

The GXF pipeline in a graph form is defined at `apps/endoscopy_tool_tracking/tracking_replayer.yaml` in Holoscan Embedded SDK Github Repository.

![Fig. 7.1: Tool tracking app with replay from file](image)

The pipeline uses a recorded endoscopy video file (generated by `convert_video_to_gxf_entities` script) for input frames.

Each input frame in the file is loaded by Video Stream Replayer and Broadcast node passes the frame to the following two nodes (Entities):

- **Format Converter**: Convert image format from RGB888 (24-bit pixel) to RGBA8888 (32-bit pixel) for visualization (Tool Tracking Visualizer)
- **Format Converter**: Convert the data type of the image from `uint8` to `float32` for feeding into the tool tracking model (by Custom TensorRT Inference)
Then, **Tool Tracking Visualizer** uses outputs from the first Format Converter and Custom TensorRT Inference to render overlay frames (mask/point/text) on top of the original video frames.

**Tip:** To run the Endoscopy Tool Tracking Application with the recorded video as source, run the following commands after *setting up the Holoscan SDK*:

In the container runtime (from NGC):

```bash
cd /opt/holoscan_sdk/tracking_replayer
./apps/endoscopy_tool_tracking/run_tracking_replayer
```

In the development container (from source):

```bash
cd /workspace/holoscan-sdk/build
./apps/endoscopy_tool_tracking/tracking_replayer
```

### 7.1.2 Input source: AJA

The GXF pipeline in a graph form is defined at `apps/endoscopy_tool_tracking/tracking_aja.yaml` in Holoscan Embedded SDK Github Repository.

![Fig. 7.2: AJA tool tracking app](image)

The pipeline is similar with *Input source: Video Stream Replayer* but the input source is replaced with *AJA Source*.

The pipeline graph also defines an optional *Video Stream Recorder* that can be enabled to record the original video stream to disk. This stream recorder (and its associated *Format Converter*) are commented out in the graph definition and thus are disabled by default in order to maximize performance. To enable the stream recorder, uncomment all of the associated components in the graph definition.

- **AJA Source**: Get video frame from AJA HDMI capture card (pixel format is RGBA8888 with the resolution of 1920x1080)
- **Format Converter**: Convert image format from RGBA8888 (32-bit pixel) to RGBA888 (24-bit pixel) for recording
  - (Video Stream Recorder)
- **Video Stream Recorder**: Record input frames into a file

Please follow these steps to run the Endoscopy Tool Tracking Application:
Tip: To run the Endoscopy Tool Tracking Application with AJA capture, run the following commands after setting up the Holoscan SDK and your AJA system:

In the container runtime (from NGC):

```
cd /opt/holoscan_sdk/tracking_aja
./apps/endoscopy_tool_tracking/run_tracking_aja
```

In the development container (from source):

```
cd /workspace/holoscan-sdk/build
./apps/endoscopy_tool_tracking/tracking_aja
```

7.2 Ultrasound Segmentation App & Customization

This section will describe the details of the ultrasound segmentation example app as well as how to load a custom inference model into the app for some limited customization. Out of the box, the ultrasound segmentation app comes as a “video replayer” and “AJA source”, where the user can replay a pre-recorded ultrasound video file included in the runtime container or stream data from an AJA capture device directly through the GPU respectively.

7.2.1 Input source: Video Stream Replayer

The replayer pipeline is defined in `apps/ultrasound_segmentation/segmentation_replayer.yaml` in Holoscan Embedded SDK Github Repository.

![Segmentation app with replay from file](image)

Fig. 7.3: Segmentation app with replay from file

The pipeline uses a pre-recorded endoscopy video stream stored in `nvidia::gxf::Tensor` format as input. The tensor-formatted file is generated via `convert_video_to_gxf_entities` from a pre-recorded MP4 video file.

Input frames are loaded by `Video Stream Replayer` and `Broadcast` node passes the frame to two branches in the pipeline.

- In the `inference` branch the video frames are converted to floating-point precision using the `format converter`, pixel-wise segmentation is performed, and the segmentation result is post-processed for the visualizer.
- The `visualizer` receives the original frame as well as the result of the inference branch to show an overlay.

Tip: To run the Ultrasound Segmentation Application with the recorded video as source, run the following commands after setting up the Holoscan SDK:

In the container runtime (from NGC):

```
cd /opt/holoscan_sdk/segmentation_replayer
./apps/ultrasound_segmentation/run_segmentation_replayer
```

In the development container (from source):
7.2.2 Input source: AJA

The AJA pipeline is defined in `apps/ultrasound_segmentation/segmentation_aja.yaml` in Holoscan Embedded SDK Github Repository.

![Fig. 7.4: AJA segmentation app](image)

This pipeline is exactly the same as the pipeline described in the previous section except the Video Stream Replayer has been substituted with an AJA Video Source.

**Tip:** To run the Ultrasound Segmentation Application with AJA capture, run the following commands after setting up the Holoscan SDK and your AJA system:

In the container runtime (from NGC):

```
cd /opt/holoscan_sdk/segmentation_aja
./apps/ultrasound_segmentation/run_segmentation_aja
```

In the development container (from source):

```
cd /workspace/holoscan-sdk/build
./apps/ultrasound_segmentation/segmentation_aja
```

7.2.3 Customizing the Sample Ultrasound Segmentation App

This section will show how the user can trivially update the model in the sample ultrasound segmentation app. The runtime containers contain only binaries of the sample applications therefore the users may not modify the extensions, however, the users can substitute the ultrasound model with their own and add, remove, or replace the extensions used in the application.

**Tip:** For a comprehensive guide on building your own Holoscan extensions and apps please refer to Clara Holoscan Development Guide.

The sample ultrasound segmentation model expects a gray-scale image of 256 x 256 and outputs a semantic segmentation of the same size with two channels representing bone contours with hyperechoic lines (foreground) and hyperechoic acoustic shadow (background).
Warning: Currently, the sample apps are able to load ONNX models, or TensorRT engine files built for the architecture on which you will be running the model only. TRT engines are automatically generated from ONNX by the app when it is run.

Assuming a single-input/single-output custom model we can trivially substitute the model in the application as follows.

1. Enter the sample app container, but ensure to load your model from the host into the container. Assuming your model is in `${my_model_path_dir}` and your data is in `${my_data_path_dir}` then you can execute the following:

    ```bash
    docker run -it --rm --runtime=nvidia
    -e NVIDIA_DRIVER_CAPABILITIES=graphics,video,compute,utility
    -v ${my_model_path_dir}:/workspace/my_model
    -v ${my_data_path_dir}:/workspace/my_data
    -v /tmp/.X11-unix:/tmp/.X11-unix
    -e DISPLAY=${DISPLAY}
    nvcr.io/nvidia/clara-holoscan/clara_holoscan_sample_runtime:0.2.0-arm64
    ```

2. Check that the model and data correctly appear under `/workspace/my_model` and `/workspace/my_data`.

3. Now we are ready to make small modifications to the ultrasound sample app to have the new model load.

    ```bash
cd /opt/holoscan_sdk/segmentation_replayer/
vi ./apps/ultrasound_segmentation/segmentation_replayer.yaml
    ```

4. In the editor navigate to the `segmentation_inference` entity. In this entity, the component we will modify is `nvidia::gxf::TensorRtInference` where we want to specify the input and output names.

   a. In line 146 specify the names of the inputs specified in your model under `input_binding_names`. In the case of ONNX models converted from PyTorch inputs names take the form `INPUT_0`.

   b. In line 150 specify the names of the inputs specified in your model under `output_binding_names`. In the case of ONNX models converted from PyTorch inputs names take the form `OUTPUT_0`.

   Assuming the custom model input and output bindings are `MY_MODEL_INPUT_NAME` and `MY_MODEL_OUTPUT_NAME`, the `nvidia::gxf::TensorRtInference` component would result in:

   ```yaml
   - type: nvidia::gxf::TensorRtInference
     parameters:
     - input_binding_names:
       - MY_MODEL_INPUT_NAME
     - output_binding_names:
       - MY_MODEL_OUTPUT_NAME
   ```

Tip: The `nvidia::gxf::TensorRtInference` component binds the names of the Holoscan component inputs to the model inputs via the `input_tensor_names` and `input_binding_names` lists, where the first specifies the name of the tensor used by the Holoscan component `nvidia::gxf::TensorRtInference` and the latter specifies the name of the model input. Similarly, `output_tensor_names` and `output_binding_names` link the component output names to the model output (see extensions).

5. To be able to play the desired video through the custom model we first need to convert the video file into a GXF replayable tensor format. In the container perform the following actions.
apt update && DEBIAN_FRONTEND=noninteractive apt install -y ffmpeg
cd /workspace
git clone https://github.com/NVIDIA/clara-holoscan-embedded-sdk.git
cd clara-holoscan-embedded-sdk/scripts
ffmpeg -i /workspace/my_data/${my_video} -pix_fmt rgb24 -f rawvideo pipe:1 |
→ python3 convert_video_to_gxf_entities.py --width ${my_width} --height ${my_height}
→ --directory /workspace/my_data --basename my_video

The above commands should yield two Holoscan tensor replayer files in /workspace/my_data, namely my_video.gxf_index and my_video.gxf_entities. These will be used to run the app in the next step.

6. Update the source entity (line 3) to read from the newly generated tensor data. In the ./apps/ultrasound_segmentation/segmentation_replayer.yaml file, update lines 22, 23 with the path and basename used in the data generation.

   name: source
   components:
     - type: nvidia::holoscan::stream_playback::VideoStreamReplayer
       parameters:
         directory: "/workspace/my_data"
         basename: "my_video"

7. Run the application with the new model and data.

cd /opt/holoscan_sdk/segmentation_replayer/
./apps/ultrasound_segmentation/run_segmentation_replayer
8.1 GXF Standard Extensions

8.1.1 Std

The `GXF::std` extension provides the most commonly used interfaces and components in Gxf Core.

`nvidia::gxf::Broadcast`

Messages arrived on the input channel are distributed to all transmitters.

**Parameters**

- **source**: Source channel
  - type: `Handle<Receiver>`
- **mode**: The broadcast mode. Can be Broadcast or RoundRobin (default: 0)
  - type: `BroadcastMode`
  - value:
    - 0: Broadcast mode. Publishes income message to all transmitters
    - 1: RoundRobin mode. Publishes income message to one of the transmitters in round-robin fashion

8.1.2 Serialization

The `GXF::serialization` extension provides components for serializing messages.
nvidia::gxf::EntityRecorder

Serializes incoming messages and writes them to a file.

Parameters

- **receiver**: Receiver channel to log  
  - type: Handle<Receiver>
- **entity_serializer**: Serializer for serializing entities  
  - type: Handle<EntitySerializer>
- **directory**: Directory path for storing files  
  - type: std::string
- **basename**: User specified file name without extension (optional)  
  - type: std::string
- **flush_on_tick**: Flushes output buffer on every tick when true (default: false)  
  - type: bool

8.2 Holoscan SDK GXF Extensions

8.2.1 V4L2

The v4l2_source extension provides a codelet for a realtime Video for Linux 2 source supporting USB cameras and other media inputs. The output is a VideoBuffer object.

nvidia::holoscan::V4L2Source

V4L2 Source Codelet.

Parameters

- **signal**: Output channel  
  - type: gxf::Handle<gxf::Transmitter>
- **allocator**: Output Allocator  
  - type: gxf::Handle<gxf::Allocator>
- **device**: Path to the V4L2 device (default: /dev/video0)  
  - type: std::string
- **width**: Width of the V4L2 image (default: 640)  
  - type: uint32_t
- **height**: Height of the V4L2 image (default: 480)  
  - type: uint32_t
- **numBuffers**: Number of V4L2 buffers to use (default: 2)
  - type: `uint32_t`

### 8.2.2 AJA

The `aja_source` extension provides a codelet for supporting AJA capture card as a source. It offers support for GPUDirect-RDMA on Quadro GPUs. The output is a VideoBuffer object.

**nvidia::holoscan::AJASource**

AJA Source Codelet.

**Parameters**

- **signal**: Output signal
  - type: `gxf::Handle<gxf::Transmitter>`
- **device**: Device specifier (default: 0)
  - type: `std::string`
- **channel**: NTV2Channel to use (default: 0 (NTV2_CHANNEL1))
  - type: `NTV2Channel`
- **width**: Width of the stream (default: 1920)
  - type: `uint32_t`
- **height**: Height of the stream (default: 1080)
  - type: `uint32_t`
- **framerate**: Framerate of the stream (default: 60)
  - type: `uint32_t`
- **rdma**: Enable RDMA (default: false)
  - type: `bool`

### 8.2.3 Stream Playback

The `stream_playback` extension provides components for the video stream playback module to output video frames as a Tensor object.
VideoStreamReplayer codelet.

Parameters

- **transmitter**: Transmitter channel for replaying entities
  - type: gxf::Handle<gxf::Transmitter>
- **entity_serializer**: Serializer for serializing entities
  - type: gxf::Handle<gxf::EntitySerializer>
- **boolean_scheduling_term**: BooleanSchedulingTerm to stop the codelet from ticking after all messages are published
  - type: gxf::Handle<gxf::BooleanSchedulingTerm>
- **directory**: Directory path for storing files
  - type: std::string
- **basename**: User specified file name without extension (optional)
  - type: std::string
- **batch_size**: Number of entities to read and publish for one tick (default: 1)
  - type: size_t
- **ignore_corrupted_entities**: If an entity could not be deserialized, it is ignored by default; otherwise a failure is generated (default: true)
  - type: bool
- **frame_rate**: Frame rate to replay. If zero value is specified, it follows timings in timestamps (default: 0.f)
  - type: float
- **realtime**: Playback video in realtime, based on frame_rate or timestamps (default: true)
  - type: bool
- **repeat**: Repeat video stream (default: false)
  - type: bool
- **count**: Number of frame counts to playback. If zero value is specified, it is ignored. If the count is less than the number of frames in the video, it would be finished early (default: 0)
  - type: uint64_t
nvidia::holoscan::stream_playback::VideoStreamSerializer

The VideoStreamSerializer codelet is based on the nvidia::gxf::StdEntitySerializer with the addition of a repeat feature. (If the repeat parameter is true and the frame count is out of the maximum frame index, unnecessary warning messages are printed with nvidia::gxf::StdEntitySerializer.)

8.2.4 Format Converter

The format_converter extension includes a codelet that provides common video or tensor operations in inference pipelines to change datatypes, resize images, reorder channels, and normalize and scale values.

nvidia::holoscan::formatconverter::FormatConverter

This codelet executes the following processes:

- Resize the input image before converting data type
  - if resize_width > 0 && resize_height > 0
- Adjust output shape if the conversion involves the change in the channel dimension
  - if format conversion is one of the following:
    - rgb888 to rgba8888 (out channels: 4)
    - rgba8888 to rgb888 (out channels: 3)
    - rgba8888 to float32 (out channels: 3)
- Convert data type
  - The following conversions are supported:
    - "" (None): if in_dtype and out_dtype are the same
      - dst_order (default: [0,1,2] or [0,1,2,3] depending on out_dtype) needs to be set
    - uint8(rgb888) to float32
      - scale_min and scale_max need to be set
      - dst_order (default: [0,1,2]) needs to be set
    - float32 to uint8(rgb888)
      - scale_min and scale_max need to be set
      - dst_order (default: [0,1,2]) needs to be set
    - uint8(rgb888) to rgba8888
      - dst_order (default: [0,1,2,3]) and alpha_value (default: 255) need to be set
    - rgba8888 to uint8(rgb888)
      - dst_order (default: [0,1,2]) needs to be set
Parameters

- **in**: Input channel
  - type: gxf::Handle<gxf::Receiver>

- **in_tensor_name**: Name of the input tensor (default: "")
  - type: std::string

- **in_dtype**: Source data type (default: "")
  - type: std::string
  - If not specified, input data type is guessed from the input tensor.

- **out**: Output channel
  - type: gxf::Handle<gxf::Transmitter>

- **out_tensor_name**: Name of the output tensor (default: "")
  - type: std::string

- **out_dtype**: Destination data type
  - type: std::string

- **scale_min**: Minimum value of the scale (default: 0.f)
  - type: float

- **scale_max**: Maximum value of the scale (default: 1.f)
  - type: float

- **alpha_value**: Alpha value that can be used to fill the alpha channel when converting RGB888 to RGBA8888 (default: 255)
  - type: uint8_t

- **resize_width**: Width for resize. No actions if this value is zero (default: 0)
  - type: int32_t

- **resize_height**: Height for resize. No actions if this value is zero (default: 0)
  - type: int32_t

- **resize_mode**: Mode for resize. 4 (NPPI_INTER_CUBIC) if this value is zero (default: 0)
  - type: int32_t

```
0 = NPPI_INTER_CUBIC
1 = NPPI_INTER_NN       /**< Nearest neighbor */
   filtering.
2 = NPPI_INTER_LINEAR   /**< Linear interpolation. */
4 = NPPI_INTER_CUBIC    /**< Cubic interpolation. */
5 = NPPI_INTER_CUBIC2P_BSPLINE /**< Two-parameter cubic */
   filter (B=1, C=0)
6 = NPPI_INTER_CUBIC2P_CATMULLROM /**< Two-parameter cubic */
   filter (B=0, C=1/2)
7 = NPPI_INTER_CUBIC2P_B05C03, /**< Two-parameter cubic */
   filter (B=1/2, C=3/10)
8 = NPPI_INTER_SUPER    /**< Super sampling. */
```

(continues on next page)
Clara Holoscan SDK User Guide, Release 0.2.0

```
16 = NPPI_INTER_LANCZOS /**< Lanczos filtering. */
17 = NPPI_INTER_LANCZOS3_ADVANCED /**< Generic Lanczos filtering with order 3. */
(0x8000000 = NPPI_SMOOTH_EDGE /**< Smooth edge filtering. */
(0x8000000 = 134217728)
```

- **out_channel_order**: Host memory integer array describing how channel values are permuted (default: [])
  - type: std::vector<int>
- **pool**: Pool to allocate the output message
  - type: gxf::Handle<gxf::Allocator>

### 8.2.5 TensorRT

The `tensor_rt` extension provides the TensorRT inference codelet.

**nvidia::holoscan::TensorRtInference**

Codelet taking input tensors and feeding them into TensorRT for inference. Based on `nvidia::gxf::TensorRtInference`, with the addition of the `engine_cache_dir` to be able to provide a directory of engine files for multiple GPUs instead of a single one.

**Parameters**

- **model_file_path**: Path to ONNX model to be loaded
  - type: std::string
- **engine_cache_dir**: Path to a directory containing cached generated engines to be serialized and loaded from
  - type: std::string
- **plugins_lib_namespace**: Namespace used to register all the plugins in this library (default: "")
  - type: std::string
- **force_engine_update**: Always update engine regard less of existing engine file. Such conversion may take minutes (default: false)
  - type: bool
- **input_tensor_names**: Names of input tensors in the order to be fed into the model
  - type: std::vector<std::string>
- **input_binding_names**: Names of input bindings as in the model in the same order of what is provided in input_tensor_names
  - type: std::vector<std::string>
- **output_tensor_names**: Names of output tensors in the order to be retrieved from the model
  - type: std::vector<std::string>
- **output_binding_names**: Names of output bindings in the model in the same order of of what is provided in output_tensor_names

8.2. Holoscan SDK GXF Extensions
- type: std::vector<std::string>

- **pool**: Allocator instance for output tensors
  - type: gxf::Handle<gxf::Allocator>

- **cuda_stream_pool**: Instance of gxf::CudaStreamPool to allocate CUDA stream
  - type: gxf::Handle<gxf::CudaStreamPool>

- **max_workspace_size**: Size of working space in bytes (default: 67108864 (64MB))
  - type: int64_t

- **dla_core**: DLA Core to use. Fallback to GPU is always enabled. Default to use GPU only (optional)
  - type: int64_t

- **max_batch_size**: Maximum possible batch size in case the first dimension is dynamic and used as batch size (default: 1)
  - type: int32_t

- **enable_fp16_**: Enable inference with FP16 and FP32 fallback (default: false)
  - type: bool

- **verbose**: Enable verbose logging on console (default: false)
  - type: bool

- **relaxed_dimension_check**: Ignore dimensions of 1 for input tensor dimension check (default: true)
  - type: bool

- **clock**: Instance of clock for publish time (optional)
  - type: gxf::Handle<gxf::Clock>

- **rx**: List of receivers to take input tensors
  - type: std::vector<gxf::Handle<gxf::Receiver>>

- **tx**: Transmitter to publish output tensors
  - type: gxf::Handle<gxf::Transmitter>

### 8.2.6 OpenGL

The `opengl_renderer` extension provides a codelet that displays a VideoBuffer, leveraging OpenGL/CUDA interop.

**nvidia::holoscan::OpenGLRenderer**

OpenGL Renderer Codelet.
Parameters

- **signal**: Input Channel
  - type: gxf::Handle<gxf::Receiver>
- **width**: Width of the rendering window
  - type: unsigned int
- **height**: Height of the rendering window
  - type: unsigned int
- **window_close_scheduling_term**: BooleanSchedulingTerm to stop the codelet from ticking after all messages are published
  - type: gxf::Handle<gxf::BooleanSchedulingTerm>

### 8.2.7 Segmentation Post Processor

The `segmentation_postprocessor` extension provides a codelet that converts inference output to the highest-probability class index, including support for sigmoid, softmax, and activations.

```cpp
nvidia::holoscan::segmentation_postprocessor::Postprocessor
```

Segmentation Postprocessor codelet.

Parameters

- **in**: Input channel
  - type: gxf::Handle<gxf::Receiver>
- **in_tensor_name**: Name of the input tensor (default: "")
  - type: std::string
- **network_output_type**: Network output type (default: softmax)
  - type: std::string
- **out**: Output channel
  - type: gxf::Handle<gxf::Transmitter>
- **data_format**: Data format of network output (default: hwc)
  - type: std::string
- **allocator**: Output Allocator
  - type: gxf::Handle<gxf::Allocator>
8.2.8 Segmentation Visualizer

The segmentation_visualizer extension provides an OpenGL renderer codelet that combines segmentation output overlayed on video input, using CUDA/OpenGL interop.

nvidia::holoscan::segmentation_visualizer::Visualizer

OpenGL Segmentation Visualizer codelet.

Parameters

- **image_in**: Tensor input
  - type: gxf::Handle<gxf::Receiver>
- **image_width**: Width of the input image (default: 1920)
  - type: int32_t
- **image_height**: Height of the input image (default: 1080)
  - type: int32_t
- **class_index_in**: Tensor input
  - type: gxf::Handle<gxf::Receiver>
- **class_index_width**: Width of the segmentation class index tensor (default: 1920)
  - type: int32_t
- **class_index_height**: Height of the segmentation class index tensor (default: 1080)
  - type: int32_t
- **class_color_lut**: Overlay Image Segmentation Class Colormap
  - type: std::vector<std::vector<float>>
- **window_close_scheduling_term**: BooleanSchedulingTerm to stop the codelet from ticking after all messages are published
  - type: gxf::Handle<gxf::BooleanSchedulingTerm>

8.2.9 Custom LSTM Inference

The custom_lstm_inference extension provides LSTM (Long-Short Term Memory) stateful inference module using TensorRT.
nvidia::holoscan::custom_lstm_inference::TensorRtInference

Codelet, taking input tensors and feeding them into TensorRT for LSTM inference.
This implementation is based on nvidia::gxf::TensorRtInference.  
input_state_tensor_names and output_state_tensor_names parameters are added to specify tensor names for states in LSTM model.

Parameters

- **model_file_path**: Path to ONNX model to be loaded
  - type: std::string
- **engine_cache_dir**: Path to a directory containing cached generated engines to be serialized and loaded from
  - type: std::string
- **plugins_lib_namespace**: Namespace used to register all the plugins in this library (default: "")
  - type: std::string
- **force_engine_update**: Always update engine regard less of existing engine file. Such conversion may take minutes (default: false)
  - type: bool
- **input_tensor_names**: Names of input tensors in the order to be fed into the model
  - type: std::vector<std::string>
- **input_state_tensor_names**: Names of input state tensors that are used internally by TensorRT
  - type: std::vector<std::string>
- **input_binding_names**: Names of input bindings as in the model in the same order of what is provided in input_tensor_names
  - type: std::vector<std::string>
- **output_tensor_names**: Names of output tensors in the order to be retrieved from the model
  - type: std::vector<std::string>
- **output_state_tensor_names**: Names of output state tensors that are used internally by TensorRT
  - type: std::vector<std::string>
- **output_binding_names**: Names of output bindings in the model in the same order of the what is provided in output_tensor_names
  - type: std::vector<std::string>
- **pool**: Allocator instance for output tensors
  - type: gxf::Handle<gxf::Allocator>
- **cuda_stream_pool**: Instance of gxf::CudaStreamPool to allocate CUDA stream
  - type: gxf::Handle<gxf::CudaStreamPool>
- **max_workspace_size**: Size of working space in bytes (default: 67108864l (64MB))
  - type: int64_t
- **dla_core**: DLA Core to use. Fallback to GPU is always enabled. Default to use GPU only (optional)
  - type: int64_t
• **max_batch_size**: Maximum possible batch size in case the first dimension is dynamic and used as batch size (default: 1)
  – type: int32_t

• **enable_fp16**: Enable inference with FP16 and FP32 fallback (default: false)
  – type: bool

• **verbose**: Enable verbose logging on console (default: false)
  – type: bool

• **relaxed_dimension_check**: Ignore dimensions of 1 for input tensor dimension check (default: true)
  – type: bool

• **clock**: Instance of clock for publish time (optional)
  – type: gxf::Handle<gxf::Clock>

• **rx**: List of receivers to take input tensors
  – type: std::vector<gxf::Handle<gxf::Receiver>>

• **tx**: Transmitter to publish output tensors
  – type: gxf::Handle<gxf::Transmitter>

### 8.2.10 Visualizer Tool Tracking

The visualizer_tool_tracking extension provides a custom visualizer codelet that handles compositing, blending, and visualization of tool labels, tips, and masks given the output tensors of the custom_lstm_inference.

```cpp
nvidia::holoscan::visualizer_tool_tracking::Sink
```

Surgical Tool Tracking Viz codelet.

**Parameters**

• **videoframe_vertex_shader_path**: Path to vertex shader to be loaded
  – type: std::string

• **videoframe_fragment_shader_path**: Path to fragment shader to be loaded
  – type: std::string

• **tooltip_vertex_shader_path**: Path to vertex shader to be loaded
  – type: std::string

• **tooltip_fragment_shader_path**: Path to fragment shader to be loaded
  – type: std::string

• **num_tool_classes**: Number of different tool classes
  – type: int32_t

• **num_tool_pos_components**: Number of components of the tool position vector (default: 2)
  – type: int32_t
- **tool_tip_colors**: Color of the tool tips, a list of RGB values with components between 0 and 1 (default: 12 qualitative classes color scheme from colorbrewer2)
  - type: `std::vector<std::vector<float>>`
- **overlay_img_vertex_shader_path**: Path to vertex shader to be loaded
  - type: `std::string`
- **overlay_img_fragment_shader_path**: Path to fragment shader to be loaded
  - type: `std::string`
- **overlay_img_width**: Width of overlay image
  - type: `int32_t`
- **overlay_img_height**: Height of overlay image
  - type: `int32_t`
- **overlay_img_channels**: Number of Overlay Image Channels
  - type: `int32_t`
- **overlay_img_layers**: Number of Overlay Image Layers
  - type: `int32_t`
- **overlay_img_colors**: Color of the image overlays, a list of RGB values with components between 0 and 1 (default: 12 qualitative classes color scheme from colorbrewer2)
  - type: `std::vector<std::vector<float>>`
- **tool_labels**: List of tool names (default: `[]`
  - type: `std::vector<std::string>`
- **label_sans_font_path**: Path for sans font to be loaded
  - type: `std::string`
- **label_sans_bold_font_path**: Path for sans bold font to be loaded
  - type: `std::string`
- **in**: List of input channels
  - type: `std::vector<gxf::Handle<gxf::Receiver>>`
- **in_tensor_names**: Names of input tensors (default: "")
  - type: `std::vector<std::string>`
- **in_width**: Width of the image (default: 640)
  - type: `int32_t`
- **in_height**: Height of the image (default: 480)
  - type: `int32_t`
- **in_channels**: Number of channels (default: 3)
  - type: `int16_t`
- **in_bytes_per_pixel**: Number of bytes per pixel of the image (default: 1)
  - type: `uint8_t`
- **alpha_value**: Alpha value that can be used when converting RGB888 to RGBA8888 (default: 255)
– type: uint8_t

• **pool**: Pool to allocate the output message.
  – type: gxf::Handle<gxf::Allocator>

• **window_close_scheduling_term**: BooleanSchedulingTerm to stop the codelet from ticking after all messages are published.
  – type: gxf::Handle<gxf::BooleanSchedulingTerm>

### 8.2.11 Holoscan Test Mock

The mocks extension provides mock codelets that can be used for testing GXF applications.

**nvidia::holoscan::test::VideoBufferMock**

VideoBuffer Mock codelet. It creates RGB strips as an output message of `gxf::VideoBuffer` type to mimic the output of AJA extension.

**Parameters**

- **in_width**: Width of the image (default: 640)
  – type: int32_t

- **in_height**: Height of the image (default: 480)
  – type: int32_t

- **in_channels**: Number of input channels (default: 3)
  – type: int16_t

- **in_bytes_per_pixel**: Number of bytes per pixel of the image (default: 1)
  – type: int8_t

- **out_tensor_name**: Name of the output tensor (default: "")
  – type: std::string

- **out**: Output channel
  – type: gxf::Handle<gxf::Transmitter>

- **pool**: Pool to allocate the output message
  – type: gxf::Handle<gxf::Allocator>
Welcome to the Holoscan SDK development guide! Here you will learn the core concepts behind Holoscan, where we will first explore the Entity-Component-System pattern, and how it is used when building Holoscan extensions and apps.

### 9.1 Holoscan Core Concepts

Holoscan applications are built as compute graphs, based on GXF. This design provides modularity at the application level since existing entities can be swapped or updated without needing to recompile any extensions or application.

Those are the key terms used throughout this guide:

- Each node in the graph is known as an **entity**
- Each edge in the graph is known as a **connection**
- Each entity is a collection of **components**
- Each component performs a specific set of subtasks in that entity
- The implementation of a component’s task is known as a **codelet**
- Codelets are grouped in **extensions**

Similarly, the componentization of the entity itself allows for even more isolated changes. For example, if in an entity we have an input, an output, and a compute component, we can update the the compute component without changing the input and output.

#### 9.1.1 Holoscan Entities by Example

Let us look at an example of a Holoscan 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.

```yaml
---
# other entities declared
---
name: format_converter_entity
components:
  - name: in_tensor
    type: nvidia::gxf::DoubleBufferReceiver
  - type: nvidia::gxf::MessageAvailableSchedulingTerm
parameters:
(continues on next page)```
Clara Holoscan SDK User Guide, Release 0.2.0

---

```
    receiver: in_tensor
    min_size: 1
    - name: out_tensor
      type: nvidia::gxf::DoubleBufferTransmitter
      type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
      parameters:
        transmitter: out_tensor
        min_size: 1
    - name: pool
      type: nvidia::gxf::BlockMemoryPool
      parameters:
        storage_type: 1
        block_size: 4919040 # 854 * 480 * 3 (channel) * 4 (bytes per pixel)
        num_blocks: 2
    - name: format_converter_component
      type: nvidia::holoscan::formatconverter::FormatConverter
      parameters:
        in: in_tensor
        out: out_tensor
        out_tensor_name: source_video
        out_dtype: "float32"
        scale_min: 0.0
        scale_max: 255.0
        pool: pool
---

# other entities declared
---

components:
    - name: input_connection
      type: nvidia::gxf::Connection
      parameters:
        source: upstream_entity/output
        target: format_converter/in_tensor
---

components:
    - name: output_connection
      type: nvidia::gxf::Connection
      parameters:
        source: format_converter/out_tensor
        target: downstream_entity/input
---

name: scheduler
components:
    - 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_ent 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.

![Fig. 9.1: Arrangement of components and entities in a Holoscan application](image)

In the image, as in the YAML, you will notice the use of `MessageAvailableSchedulingTerm`, `DownstreamReceptiveSchedulingTerm`, and `BlockMemoryPool`. These are components which play a “supporting” role to `in_tensor`, `out_tensor`, and `format_converter_component` components respectively. Specifically:

- **MessageAvailableSchedulingTerm** is a component which 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 which takes a Transmitter (in this case `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 `formatConverted_ent` 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.

---

**9.1. Holoscan Core Concepts**


9.1.2 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`.

- **type**: `nvidia::gxf::MessageAvailableSchedulingTerm`
  - **parameters**:
    - `receiver`: `in_tensor`
    - `min_size`: 1

Similarly `DownStreamReceptiveSchedulingTerm` checks whether the `out_tensor` transmitter queue has at least one outgoing message in it. If there is 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 the downstream entity, however, has a full receiver queue the message is held in the `out_tensor` queue as a means to handle back-pressure.

- **type**: `nvidia::gxf::DownstreamReceptiveSchedulingTerm`
  - **parameters**:
    - `transmitter`: `out_tensor`
    - `min_size`: 1

If we were to draw the entity in Fig. 9.1 in greater detail it would look something like the following.

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.

9.2 Developing Holoscan Extensions

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 an “extension”. Holoscan extensions are typically concerned with application-specific sub-tasks such as data transformations, AI model inference, and the like.
9.2.1 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 light-weight initialization.
2. deinitialize - called only once before the codelet is destroyed, and used for light-weight de-initialisation.
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. 9.3.
9.2.2 Implementing an Extension

In this section we will implement a simple recorder which 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).

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 as follows.

```cpp
class MyRecorder : public nvidia::gxf::Codelet {
public:
    gxf_result_t registerInterface(nvidia::gxf::Registrar* registrar) override;
    gxf_result_t initialize() override;
    gxf_result_t deinitialize() override;
    gxf_result_t start() override;
    gxf_result_t tick() override;
    gxf_result_t stop() override;

private:
    nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::Receiver>> receiver_;
    nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::EntitySerializer>> my_serializer_;
    nvidia::gxf::Parameter<std::string> directory_;
    nvidia::gxf::Parameter<std::string> basename_;```

(continues on next page)
Declare the Parameters to Expose at the Application Level

Our recorder will need to expose the `nvidia::gxf::Parameter` variables to the application so the parameters can be modified by configuration.

```cpp
gxf_result_t MyRecorder::registerInterface(nvidia::gxf::Registrar* registrar) override {
    nvidia::gxf::Expected<void> result;
    result &= registrar->parameter(
        receiver_, "receiver", "Entity receiver",
        "Receiver channel to log");
    result &= registrar->parameter(
        my_serializer_, "serializer", "Entity serializer",
        "Serializer for serializing input data");
    result &= registrar->parameter(
        directory_, "out_directory", "Output directory path",
        "Directory path to store received output");
    result &= registrar->parameter(
        basename_, "basename", "File base name",
        "User specified file name without extension",
        nvidia::gxf::Registrar::NoDefaultParameter(), GXF_PARAMETER_FLAGS_OPTIONAL);
    result &= registrar->parameter(
        flush_on_tick_, "flush_on_tick", "Boolean to flush on tick",
        "Flushes output buffer on every 'tick' when true", false); // default value 'false'
    return nvidia::gxf::ToResultCode(result);
}
```

In the application YAML, our component's parameters can be specified as follows.

```yaml
name: my_recorder_entity
components:
  - name: my_recorder_component
type: MyRecorder
parameters:
  receiver: receiver
  serializer: my_serializer
  out_directory: /home/user/out_path
  basename: my_output_file  # optional
  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.

```c++
gxf_result_t MyRecorder::initialize() {
  // Create path by appending receiver name to directory path if basename is not provided
  std::string path = directory_.get() + '/';
  if (const auto& basename = basename_.try_get()) {
    path += basename.value();
  } else {
    path += receiver_->name();
  }

  // Initialize index file stream as write-only
  index_file_stream_ = nvidia::gxf::FileStream('', path + nvidia::gxf::FileStream::kIndexFileExtension);

  // Initialize binary file stream as write-only
  binary_file_stream_ = nvidia::gxf::FileStream('', path + nvidia::gxf::FileStream::kBinaryFileExtension);

  // Open index file stream
  nvidia::gxf::Expected<void> result = index_file_stream_.open();
  if (!result) {
    return nvidia::gxf::ToResultCode(result);
  }

  // Open binary file stream
  result = binary_file_stream_.open();
  if (!result) {
    return nvidia::gxf::ToResultCode(result);
  }
  binary_file_offset_ = 0;

  return GXF_SUCCESS;
}
```

When de-initializing, our component will take care of closing the file streams that were created at initialization.

```c++
gxf_result_t MyRecorder::deinitialize() {
  // Close binary file stream
  nvidia::gxf::Expected<void> result = binary_file_stream_.close();
  if (!result) {
    return nvidia::gxf::ToResultCode(result);
  }

  // Close index file stream
  result = index_file_stream_.close();
  if (!result) {
    return nvidia::gxf::ToResultCode(result);
  }
}
```

(continues on next page)
In our recorder, no heavy-weight initialization tasks are required so we implement the following, however, we would use `start()` and `stop()` methods for heavy-weight tasks such as memory allocation and deallocation.

```c++
gxf_result_t MyRecorder::start() {
    return GFX_SUCCESS;
}

gxf_result_t MyRecorder::stop() {
    return GFX_SUCCESS;
}
```

**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()`.

```c++
gxf_result_t MyRecorder::tick() {
    // Receive entity
    nvidia::gxf::Expected<nvidia::gxf::Entity> entity = receiver_ -> receive();
    if (!entity) {
        return nvidia::gxf::ToResultCode(entity);
    }

    // Write entity to binary file
    nvidia::gxf::Expected<size_t> size = my_serializer_ -> serializeEntity(entity.value(), &binary_file_stream_);
    if (!size) {
        return nvidia::gxf::ToResultCode(size);
    }

    // Create entity index
    nvidia::gxf::EntityIndex index;
    index.log_time = std::chrono::system_clock::now().time_since_epoch().count();
    index.data_size = size.value();
    index.data_offset = binary_file_offset_;

    // Write entity index to index file
    nvidia::gxf::Expected<size_t> result = index_file_stream_.writeTrivialType(&index);
    if (!result) {
        return nvidia::gxf::ToResultCode(result);
    }

    binary_file_offset_ += size.value();

    if (flush_on_tick_) {
        // Flush binary file output stream
        nvidia::gxf::Expected<void> result = binary_file_stream_.flush();
        if (!result) {
            return nvidia::gxf::ToResultCode(result);
        }
    }
}
```

(continues on next page)
At this point we have a functional extension, which records data coming into its receiver queue to the specified location on disk using the GXF-formatted binary/index-formatted files.

### 9.3 Holoscan Applications by Example

Now that we know how to write an extension, we can create a simple application consisting of a replayer, which reads contents from a file on disk, and our recorder (in the last section), which will store the output of the replayer exactly in the same format. This should allow us to see whether the output of the recorder matches the original input files.

Here is the complete application consisting of the mentioned entities:

```yaml
---
name: replayer
components:
  - name: output
type: nvidia::gxf::DoubleBufferTransmitter
  - name: allocator
type: nvidia::gxf::UnboundedAllocator
  - name: component_serializer
type: nvidia::gxf::StdComponentSerializer
parameters:
  allocator: allocator
  name: entity_serializer
type: nvidia::holoscan::stream_playback::VideoStreamSerializer  # inheriting from nvidia::gxf::EntitySerializer
parameters:
  component_serializers: [component_serializer]
  transmitter: output
  entity_serializer: entity_serializer
  boolean_scheduling_term: boolean_scheduling
directory: "/workspace/test_data/endoscopy/video"
basename: "surgical_video"
frame_rate: 0  # as specified in timestamps
repeat: true  # default: false
realtime: true  # default: true
count: 0  # default: 0 (no frame count restriction)
```

(continues on next page)
- name: boolean_scheduling
type: nvidia::gxf::BooleanSchedulingTerm
- type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
  parameters:
    transmitter: output
    min_size: 1

---
name: recorder
components:
- name: input
type: nvidia::gxf::DoubleBufferReceiver
- name: allocator
type: nvidia::gxf::UnboundedAllocator
- name: component_serializer
type: nvidia::gxf::StdComponentSerializer
  parameters:
    allocator: allocator
- name: entity_serializer
type: nvidia::gxf::StdEntitySerializer
  # inheriting from nvidia::gxf::EntitySerializer
  parameters:
    component_serializers: [component_serializer]
- type: MyRecorder
  parameters:
    receiver: input
    serializer: entity_serializer
    directory: '/tmp'
    basename: 'tensor_out'
- type: nvidia::gxf::MessageAvailableSchedulingTerm
  parameters:
    receiver: input
    min_size: 1

---
components:
- name: input_connection
type: nvidia::gxf::Connection
  parameters:
    source: replayer/output
    target: recorder/input

---
name: scheduler
components:
- type: nvidia::gxf::GreedyScheduler

Above:

- The replayer reads data from /workspace/test_data/endoscopy/video/surgical_video.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 `nvidia::gxf::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 entity

```yaml
- name: entity_serializer
type: nvidia::gxf::StdEntitySerializer  # inheriting from nvidia::gxf::EntitySerializer
parameters:
  component_serializers: [component_serializer]
```

and passed into `MyRecorder` via the `serializer` parameter which we exposed in the *last section*.

```yaml
- type: MyRecorder
parameters:
  receiver: input
  serializer: entity_serializer
directory: "/tmp"
  basename: "tensor_out"
```

We now have a minimal working application to test the integration of our newly built component.
CHAPTER
TEN

VIDEO PIPELINE LATENCY TOOL

The Clara AGX Developer Kit excels 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 Clara AGX Developer Kit and various video I/O hardware and software components, the Clara Holoscan SDK includes a sample latency measurement tool.

10.1 Requirements

10.1.1 Hardware

The latency measurement tool requires the use of a Clara AGX 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).
10.1.2 Software

The following additional software components are required and are installed either by the Clara 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).

10.2 Installation

10.2.1 Downloading the Source

The Video Pipeline Latency Tool can be found in the loopback-latency folder of the Clara Holoscan Performance Tools GitHub repository, which is cloned with the following:

$ git clone -b v0.2.0 https://github.com/NVIDIA/clara-holoscan-perf-tools.git

10.2.2 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
10.2.3 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 ..
```

10.3 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.

The following image shows an example of a loopback HDMI cable that is connected between the GPU and the HDMI capture card that is onboard the Clara AGX Developer Kit. This configuration can be used to measure the latency using any producer that outputs via the GPU and any consumer that captures from the onboard HDMI capture card. See Producers, Consumers, and Example Configurations for more details.
Fig. 10.1: HDMI Loopback Between GPU and HDMI Capture Card
10.3.1 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 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.

![Diagram of latency tool frame lifespan with RDMA enabled](image)

**Fig. 10.3: Latency Tool Frame Lifespan (RDMA Enabled)**

### 10.3.2 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
```
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 HW (consumer)
2. Copy to GPU (consumer)
3. Process Frame (producer)
4. Render Results to GPU (producer)
5. Copy to Host (producer)
6. Write to HW (producer)

![Image of frame lifespan]

**Fig. 10.4: 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)
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)

![Diagram of Final Estimated Latency with VSync and Physical Wire Time](image)

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)
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).

### 10.3.3 Reducing Latency With RMDA

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)**

```
Read  Copy  Process  Render  Copy  Write  VSync  Physical Wire
```

**RDMA Enabled (2 Frames)**

```
Read  Process  Render  Write  V  Physical Wire
```

**Fig. 10.6: 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
Write 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
10.3.4 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.

```
$ ./loopback-latency -p aja -c aja -s 2000
```

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
```

---

10.3. Operation Overview
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 = 28967
  Frames: avg = 1.23, min = 1.21, max = 1.26

Final Estimated Latencies (Processing + Vsync + Wire)

Microseconds: avg = 49998, min = 49990, 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.

## 10.4 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 16666us).

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.

10.4. Graphing Results
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.
10.5 Producers

There are currently 3 producer types supported by the Holoscan latency tool. See the following sections for a description of each supported producer.
10.5.1 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.

10.5.2 GStreamer GPU Rendering (HDMI)

This producer (gst) uses the `nveglglessink` GStreamer component that is included with JetPack in order 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 in order 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.

10.5.3 AJA Video Systems (SDI and HDMI)

This producer (aja) outputs video frames from an AJA Video Systems device that supports video playback. This can be either an SDI or an HDMI video source.

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 devices that have currently been verified to work with this producer are the KONA HDMI (for HDMI) and Corvid 44 12G BNC (for SDI).

10.6 Consumers

There are currently 3 consumer types supported by the Holoscan latency tool. See the following sections for a description of each supported consumer.

10.6.1 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 the Clara AGX Developer Kit.

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.

10.6.2 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.

10.6.3 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).

10.7 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):

```bash
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**:

![Ubuntu System Settings](image)

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*).

### 10.7.1 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.

For example, an *OpenGL producer* to *V4L2 consumer* can be measured using this configuration and the following command:

```bash
$ ./loopback-latency -p gl -c v4l2
```

### 10.7.2 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.

For example, an *OpenGL producer* to *AJA consumer* can be measured using this configuration and the following command:

```bash
$ ./loopback-latency -p gl -c aja -c.device 0 -c.channel 1
```

### 10.7.3 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.

```bash
$ ./loopback-latency -p aja -c aja
```
Fig. 10.7: DP-to-HDMI Cable Between GPU and Onboard HDMI Capture Card
Fig. 10.8: DP-to-HDMI Cable Between GPU and AJA KONA HDMI Capture Card (Channel 1)
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:

$ ./loopback-latency -p aja -p.device 0 -p.channel 1 -c aja -c.device 1 -c.

channel 1

10.8 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 Message]

   **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:

```plaintext
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:

```bash
$ xrandr --output DP-5 --mode 1920x1080 --panning 1920x1080 --rate 60
```

But using the `xrandr` command provided produces an error:

```bash
$ 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.

```bash
$ 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 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)
      cloned 1920x1080+0+0
   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.*
CHAPTER ELEVEN

GRAPH EXECUTION FRAMEWORK (GXF)

11.1 Overview

GXF is a modular and extensible framework to build high-performance AI applications.
- Enable developers to reuse components and app graphs between different products to build their own applications.
- Enable developers to use common data formats.
- Enable developers with tools to build and analyze their applications.

11.1.1 GXF Core

GXF Core implements basic framework for entity, component and parameters which enable developers to implement GXF extensions on top of it. A short description of GXF terms used throughout the document:

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Functional block. Defines the data and behavior aspects of an entity.</td>
</tr>
<tr>
<td>Entity</td>
<td>Composition of functional blocks. An entity is a lightweight, uniquely identifiable container of components</td>
</tr>
<tr>
<td>System</td>
<td>Governs a set of components across different nodes</td>
</tr>
<tr>
<td>Connection</td>
<td>Connection between two components</td>
</tr>
<tr>
<td>Extension</td>
<td>Collection of functional blocks, matched 1-1 with a file on disk (library)</td>
</tr>
<tr>
<td>Graph</td>
<td>Data-driven representation of an application using entities and connections</td>
</tr>
<tr>
<td>Sub-graph</td>
<td>Graph wrapped in entity as functional block</td>
</tr>
</tbody>
</table>

11.1.2 GXF Extensions

A GXF extension is a shared library and/or header file containing one or more components.
11.2 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. Developers implement custom components which can be added to entity to achieve the required functionality.

11.2.1 Concepts

The graph contains nodes which follow an entity-component design pattern implementing the “composition over inheritance” paradigm. A node itself is just a light-weight 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, the developer needs 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.

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 sub-graphs in parent with defined interface.

A subgraph interface can be defined as follows:

```yaml
---
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 light-weight 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 a developer does not derive from node as a base class, but instead composes 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 a developer to extend the runtime by injecting her 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.
11.2.2 Graph File Format

Graph file stores list of dependencies and 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. Dependencies describe extensions used in graph with version required. Dependencies information is used by registry to download the correct versions of all the required extensions.

```yaml
---
dependencies:
  - extension: StandardExtension
    uuid: 8ec2d5d6-b5df-48bf-8dee-0252606fdd7e
    version: 1.0.0
  - extension: test
    uuid: 346eeecbc-9039-37da-8456-44fe9ac6492c
    version: 1.0.0
---
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
```

11.3 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.

```bash
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=Idle, 1=Error, 2=Warning, 3=Info, 4=Debug. Default: Info) type: int32 default: 3
```
11.4 GXF Core C APIs

11.4.1 Context

Create context

```c
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 cannot 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

```c
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

```c
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.

11.4.2 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 com-
ponent can be added to a GXF entity the GXF extension shared library providing the component must be loaded. An
extensions 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 extensions may still be loaded only a single time.

:: # Example manifest YAML file extensions:
  • gxf/std/libgxf_std.so
  • gxf/npp/libgxf_npp.so

A manifest file is a YAML file with a singletop-level entry ‘extensions’ followed by a list of filenames of GXF extension
shared libraries.

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

```c
#include <gxfxincludetries.h>

#define GXF_SUCCESS 0

typedef enum {
    GXF_SUCCESS,
    GXF_ERROR,
} gxf_result_t;

bool 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 api’s on extension B and it’s components.

  parameter: `context` A valid GXF context.
  parameter: `filenames` absolute paths of metadata files generated by the registry during extension registration
  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

```c
#include <gxfxincludetries.h>

#define GXF_SUCCESS 0

typedef enum {
    GXF_SUCCESS,
    GXF_ERROR,
} gxf_result_t;

bool 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.
11.4.3 Graph Execution

**Loads a list of entities from YAML file**

```c
void GxfGraphLoadFile(gxf_context_t context, const char* filename, const char* parameters_override[], const uint32_t num_overrides);
```

- **context**: A valid GXF context
- **filename**: A valid YAML filename.
- **params_override**: An optional array of strings used for override parameters in yaml file.
- **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**

```c
void GxfGraphSetRootPath(gxf_context_t context, const char* path);
```

- **context**: A valid GXF context
- **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**

```c
void GxfGraphParseString(gxf_context_t context, const char* tex, const char* parameters_override[], const uint32_t num_overrides);
```

- **context**: A valid GXF context
- **text**: A valid YAML text.
- **params_override**: An optional array of strings used for override parameters in yaml file.
- **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**

```c
void GxfGraphActivate(gxf_context_t context);
```

- **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.

11.4.4 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 light-weight 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. parameter 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

Note: In case that the entity is currently executing this function will wait and block until the current execution is finished.

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.

Parameter: context A valid GXF context

Parameter: eid UID of a valid entity

Returns: GXF error code
## Destroy an entity

```c
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
  - `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

```c
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
  - `name` A C string with the name of the entity. Ownership is not transferred.
  - `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

```c
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
  - `num_entities` In/Out: the max number of entities that can fit in the buffer/the number of entities that exist in the application
  - `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 entites 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**

```c
void 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 “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.

### 11.4.5 Components

Maximum number of components in an entity or an extension can be up to 1024.

**Get component type identifier**

```c
void 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**

```c
void 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
  - `cid`: The unique object ID (UID) of the component
  - `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
  - `cid`: The unique object ID (UID) of the component
  - `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
  - `eid`: The unique object ID (UID) of the entity to which the component is added.
  - `tid`: The unique type ID (TID) of the component to be added to the entity.
  - `name`: The name of the new component. Ownership is not transferred.
  - `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

```c
#include <gxf_api.h>

// Add an existing component to the interface of an entity

int gxf_component_add_to_interface(const void* context, const void* eid, const void* cid, const char* name);
```

This function 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.

**Parameters:**
- `context` A valid GXF context
- `eid` The unique object ID (UID) of the entity to which the component is added.
- `cid` The unique object ID of the component.
- `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

```c
#include <gxf_api.h>

// Find a component in an entity

int gxf_component_find(const void* context, const void* eid, const void* tid, const char* name, int32_t* offset, const void* cid);
```

This function 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.

**Parameters:**
- `context` A valid GXF context
- `eid` The unique object ID (UID) of the entity which is searched.
- `tid` The component type ID (TID) of the component to find (optional)
- `name` The component name of the component to find (optional). Ownership not transferred.
- `offset` The index of the first component in the entity to search. Also contains the index of the component which was found.
- `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

```c
#include <gxf_api.h>

// Get the component type ID (TID) of a component

int gxf_component_type(const void* context, const void* cid, int32_t* tid);
```

This function gets the component type ID (TID) of a component.

**Parameters:**
- `context` A valid GXF context
- `cid` The component object ID (UID) for which the component type is requested.
- `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

\texttt{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: \texttt{context} A valid GXF context 
- parameter: \texttt{uid} The component object ID (UID). 
- parameter: \texttt{tid} The expected component type ID (TID) of the component 
- parameter: \texttt{pointer} The returned pointer to the component object. 

returns: GXF\_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

### 11.4.6 Primitive Parameters

64-bit floating point

**Set**

\texttt{gxf\_result\_t GxfParameterSetFloat64(gxf\_context\_t context, gxf\_uid\_t uid, const char* key, double value);} 

- parameter: \texttt{context} A valid GXF context. 
- parameter: \texttt{uid} A valid component identifier. 
- parameter: \texttt{key} A valid name of a component to set. 
- parameter: \texttt{value} a double value 

returns: GXF\_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

**Get**

\texttt{gxf\_result\_t GxfParameterGetFloat64(gxf\_context\_t context, gxf\_uid\_t uid, const char* key, double* value);} 

- parameter: \texttt{context} A valid GXF context. 
- parameter: \texttt{uid} A valid component identifier. 
- parameter: \texttt{key} A valid name of a component to set. 
- parameter: \texttt{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

```c
#include <gxf.h>

GxfResult GxfParameterSetInt64(gxfContext_t context, gxfUid_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

```c
#include <gxf.h>

GxfResult GxfParameterGetInt64(gxfContext_t context, gxfUid_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

```c
#include <gxf.h>

GxfResult GxfParameterSetUInt64(gxfContext_t context, gxfUid_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

```c
void 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

```c
void 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

```c
void 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

```c
void 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.

- **returns:** GXF_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.
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.
11.4.7 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 11.1: Supported Data Types to Set 1D Vector Parameters

<table>
<thead>
<tr>
<th>Function Name</th>
<th>data_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GxfParameterSet1DFloat64Vector(...)</td>
<td>double</td>
</tr>
<tr>
<td>GxfParameterSet1DInt64Vector(...)</td>
<td>int64_t</td>
</tr>
<tr>
<td>GxfParameterSet1DUInt64Vector(...)</td>
<td>uint64_t</td>
</tr>
<tr>
<td>GxfParameterSet1DInt32Vector(...)</td>
<td>int32_t</td>
</tr>
</tbody>
</table>

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 11.2: Supported Data Types to Set 2D Vector Parameters

<table>
<thead>
<tr>
<th>Function Name</th>
<th>data_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GxfParameterSet2DFloat64Vector(...)</td>
<td>double</td>
</tr>
<tr>
<td>GxfParameterSet2DInt64Vector(...)</td>
<td>int64_t</td>
</tr>
<tr>
<td>GxfParameterSet2DUInt64Vector(...)</td>
<td>uint64_t</td>
</tr>
<tr>
<td>GxfParameterSet2DInt32Vector(...)</td>
<td>int32_t</td>
</tr>
</tbody>
</table>

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 11.3: Supported Data Types to Get the Value of 1D Vector Parameters

<table>
<thead>
<tr>
<th>Function Name</th>
<th>data_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GxfParameterGet1DFloat64Vector(...)</td>
<td>double</td>
</tr>
<tr>
<td>GxfParameterGet1DInt64Vector(...)</td>
<td>int64_t</td>
</tr>
<tr>
<td>GxfParameterGet1DUInt64Vector(...)</td>
<td>uint64_t</td>
</tr>
<tr>
<td>GxfParameterGet1DInt32Vector(...)</td>
<td>int32_t</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>parameter</th>
<th>key The name of the parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter</td>
<td>value Allocated array to get the value of the parameter</td>
</tr>
<tr>
<td>parameter</td>
<td>height The height of the 2-D vector parameter obtained by calling GxfParameterGet2D&quot;DataType&quot;VectorInfo(...)</td>
</tr>
<tr>
<td>parameter</td>
<td>width The width of the 2-D vector parameter obtained by calling GxfParameterGet2D&quot;DataType&quot;VectorInfo(...)</td>
</tr>
</tbody>
</table>

Table 11.4: Supported Data Types to Get the Value of 2D Vector Parameters

<table>
<thead>
<tr>
<th>Function Name</th>
<th>data_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GxfParameterGet2DFloat64Vector(...)</td>
<td>double</td>
</tr>
<tr>
<td>GxfParameterGet2DInt64Vector(...)</td>
<td>int64_t</td>
</tr>
<tr>
<td>GxfParameterGet2DUInt64Vector(...)</td>
<td>uint64_t</td>
</tr>
<tr>
<td>GxfParameterGet2DInt32Vector(...)</td>
<td>int32_t</td>
</tr>
</tbody>
</table>

11.4.8 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

```
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

```
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

```
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.

11.4.9 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.

11.5 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.

Fig. 11.1: 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 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::gxf::SchedulingConditionType described using nvidia::gxf::SchedulingCondition.

<table>
<thead>
<tr>
<th>SchedulingConditionType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEVER</td>
<td>Entity will never execute again</td>
</tr>
<tr>
<td>READY</td>
<td>Entity is ready for execution</td>
</tr>
<tr>
<td>WAIT</td>
<td>Entity may execute in the future</td>
</tr>
<tr>
<td>WAIT_TIME</td>
<td>Entity will be ready for execution after specified duration</td>
</tr>
<tr>
<td>WAIT_EVENT</td>
<td>Entity is waiting on an asynchronous event with unknown time interval</td>
</tr>
</tbody>
</table>

Schedulers define deadlock as a condition when there are no entities which are in READY, WAIT_TIME or WAIT_EVENT state which guarantee execution at a future point in time. This implies all the entities are in 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 READY state. 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

11.5.1 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 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:
Fig. 11.2: Greedy Scheduler Activity Diagram
**Greedy Scheduler Configuration**

The greedy scheduler takes in the following parameters from the configuration file:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
<td>The clock used by the scheduler to define the flow of time. Typical choices are RealtimeClock or ManualClock</td>
</tr>
<tr>
<td>max_duration_ms</td>
<td>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</td>
</tr>
<tr>
<td>stop_on_deadlock</td>
<td>If stop_on_deadlock is disabled, the GreedyScheduler constantly polls for the status of all the waiting entities to check if any of them are ready for execution.</td>
</tr>
</tbody>
</table>

The following code snippet configures a Greedy scheduler with a ManualClock option specified.

```plaintext
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
```

### 11.5.2 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 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.

As depicted in the diagram, when an entity reaches 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.

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 WAIT state.
Fig. 11.3: MultiThread Scheduler Activity Diagram
Multithread Scheduler Configuration

The multithread scheduler takes in the following parameters from the configuration file:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
<td>The clock used by the scheduler to define the flow of time. Typical choices are RealtimeClock or ManualClock.</td>
</tr>
<tr>
<td>max_duration_ms</td>
<td>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.</td>
</tr>
<tr>
<td>check_recess_period_ms</td>
<td>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.</td>
</tr>
<tr>
<td>stop_on_deadlock</td>
<td>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.</td>
</tr>
<tr>
<td>worker_thread_number</td>
<td>Number of threads.</td>
</tr>
</tbody>
</table>

The following code snippet configures a Multithread scheduler with the number of worked threads and max duration specified:

```plaintext
name: scheduler
components:
- type: nvidia::gxf::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::gxf::RealtimeClock
```

11.5.3 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 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:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
<td>The clock used by the scheduler to define the flow of time. Typical choice is a RealtimeClock.</td>
</tr>
</tbody>
</table>

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.

11.5.4 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: 50000000
```

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.

```plaintext
- 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.

```plaintext
- 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.

Entities associated with this scheduling term most likely have an asynchronous thread which can update the state of the scheduling term outside of it’s 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
```

**DownsteamReceptiveSchedulingTerm**

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::DownsteamReceptiveSchedulingTerm
  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 10000000.

```
- name: target_st
  type: nvidia::gxf::ExpiringMessageAvailableSchedulingTerm
  parameters:
    receiver: signal
    max_batch_size: 5
    max_delay_ns: 10000000
    clock: misc/clock
```

**AND Combined**

An entity can be associated with multiple scheduling terms which define it’s 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
```
11.6 StandardExtension

Most commonly used interfaces and components in Gxf Core.

- UUID: 8ec2d5d6-b5df-48bf-8dee-0252606fdd7e
- Version: 2.0.0
- Author: NVIDIA

11.6.1 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: d1febc1a-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
11.6.2 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 time scale is changed manually.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_FLOAT64

initial_time_scale

The initial time scale used until 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-aef1-6bae85bddd13
- 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 times 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. It can for example 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: e07a0dc4-3908-4df8-8134-7ce38e60fbe
- Base Type: nvidia::gxf::SchedulingTerm

nvidia::gxf::AsynchronousSchedulingTerm

A component which is used to inform of 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 deadlock. 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_periods_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 deadlock. 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 pools 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 still a full block is 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 `operator new` 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, 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-d7b5-6e7cadc704e8

float

  • Component ID: a81bf295-421f-49ef-f24a-f59e9ea0d5d6

double

  • Component ID: d57cee59-686f-e26d-95be-659c126b02ea

11.6. StandardExtension
bool

- Component ID: c02f9e93-d01b-1d29-f523-78d2a9195128

### 11.7 CudaExtension

Extension for CUDA operations.

- UUID: d63a98fa-7882-11eb-a917-b38f664f399c
- Version: 2.0.0
- Author: NVIDIA

#### 11.7.1 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 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 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 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

11.8 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

11.8.1 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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>channels</td>
<td>Number of channels in an audio frame</td>
</tr>
<tr>
<td>samples</td>
<td>Number of samples in an audio frame</td>
</tr>
<tr>
<td>sampling_rate</td>
<td>sampling rate in Hz</td>
</tr>
<tr>
<td>bytes_per_sample</td>
<td>Number of bytes required per sample</td>
</tr>
<tr>
<td>audio_format</td>
<td>AudioFormat of an audio frame</td>
</tr>
<tr>
<td>audio_layout</td>
<td>AudioLayout of an audio frame</td>
</tr>
</tbody>
</table>

Supported AudioFormat types:

<table>
<thead>
<tr>
<th>AudioFormat</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GXF_AUDIO_FORMAT_S16LE</td>
<td>16-bit signed PCM audio</td>
</tr>
<tr>
<td>GXF_AUDIO_FORMAT_F32LE</td>
<td>32-bit floating-point audio</td>
</tr>
</tbody>
</table>

Supported AudioLayout types:

<table>
<thead>
<tr>
<th>AudioLayout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GXF_AUDIO_LAYOUT_INTERLEAVED</td>
<td>Data from all the channels to be interleaved - LRLRLR</td>
</tr>
<tr>
<td>GXF_AUDIO_LAYOUT_NON_INTERLEAVED</td>
<td>Data from all the channels not to be interleaved - LLLRRR</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>width of a video frame</td>
</tr>
<tr>
<td>height</td>
<td>height of a video frame</td>
</tr>
<tr>
<td>color_format</td>
<td>VideoFormat of a video frame</td>
</tr>
<tr>
<td>color_planes</td>
<td>ColorPlane(s) associated with the VideoFormat</td>
</tr>
<tr>
<td>surface_layout</td>
<td>SurfaceLayout of the video frame</td>
</tr>
</tbody>
</table>

Supported VideoFormat types:

<table>
<thead>
<tr>
<th>VideoFormat</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GXF_VIDEO_FORMAT_YUV420</td>
<td>BT.601 multi planar 4:2:0 YUV</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_YUV420_ER</td>
<td>BT.601 multi planar 4:2:0 YUV ER</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_YUV420_709</td>
<td>BT.709 multi planar 4:2:0 YUV</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_YUV420_709_ER</td>
<td>BT.709 multi planar 4:2:0 YUV ER</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_NV12</td>
<td>BT.601 multi planar 4:2:0 YUV with interleaved UV</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_NV12_ER</td>
<td>BT.601 multi planar 4:2:0 YUV ER with interleaved UV</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_NV12_709</td>
<td>BT.709 multi planar 4:2:0 YUV with interleaved UV</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_NV12_709_ER</td>
<td>BT.709 multi planar 4:2:0 YUV ER with interleaved UV</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_RGBA</td>
<td>RGBA-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_BGRA</td>
<td>BGRA-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_ARGB</td>
<td>ARGB-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_ABGR</td>
<td>ABGR-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_RGBX</td>
<td>RGBX-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_BGRX</td>
<td>BGRX-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_XRGB</td>
<td>XRGB-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_XBGR</td>
<td>XBGR-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_RGB</td>
<td>RGB-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_BGR</td>
<td>BGR-8-8-8-8 single plane</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_R8_G8_B8</td>
<td>RGB - unsigned 8 bit multiplanar</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_B8_G8_R8</td>
<td>BGR - unsigned 8 bit multiplanar</td>
</tr>
<tr>
<td>GXF_VIDEO_FORMAT_GRAY</td>
<td>8 bit GRAY scale single plane</td>
</tr>
</tbody>
</table>

Supported SurfaceLayout types:

<table>
<thead>
<tr>
<th>SurfaceLayout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GXF_SURFACE_LAYOUT_PITCH_LINEAR</td>
<td>pitch linear surface memory</td>
</tr>
<tr>
<td>GXF_SURFACE_LAYOUT_BLOCK_LINEAR</td>
<td>block linear surface memory</td>
</tr>
</tbody>
</table>
11.9 SerializationExtension

Extension for serializing messages.
- UUID: bc573c2f-89b3-d4b0-8061-2da8b11fe79a
- Version: 2.0.0
- Author: NVIDIA

11.9.1 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

11.9.2 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: GFX_PARAMETER_FLAGS_OPTIONAL
  • Type: GFX_PARAMETER_TYPE_HANDLE
  • Handle Type: nvidia::gxf::Allocator

11.10 TensorRTExtension

Components with TensorRT inference capability.
  • UUID: d43f23e4-b9bf-11eb-9d18-2b7be630552b
  • Version: 2.0.0
  • Author: NVIDIA

11.10.1 Components

nvidia::gxf::TensorRtInference

Codelet taking input tensors and feed them into TensorRT for inference.
  • Component ID: 06a7f0e0-b9c0-11eb-8cd6-23c9c2070107
  • Base Type: nvidia::gxf::Codelet

Parameters

model_file_path
Path to ONNX model to be loaded.
  • Flags: GFX_PARAMETER_FLAGS_NONE
  • Type: GFX_PARAMETER_TYPE_STRING

engine_file_path
Path to the generated engine to be serialized and loaded from.
  • Flags: GFX_PARAMETER_FLAGS_NONE
  • Type: GFX_PARAMETER_TYPE_STRING
**force_engine_update**

Always update engine regard less of existing engine file. Such conversion may take minutes. Default to false.

- **Flags:** GXF_PARAMETER_FLAGS_NONE
- **Type:** GXF_PARAMETER_TYPE_BOOL
- **Default:** False

**input_tensor_names**

Names of input tensors in the order to be fed into the model.

- **Flags:** GXF_PARAMETER_FLAGS_NONE
- **Type:** GXF_PARAMETER_TYPE_STRING

**input_binding_names**

Names of input bindings as in the model in the same order of what is provided in input_tensor_names.

- **Flags:** GXF_PARAMETER_FLAGS_NONE
- **Type:** GXF_PARAMETER_TYPE_STRING

**output_tensor_names**

Names of output tensors in the order to be retrieved from the model.

- **Flags:** GXF_PARAMETER_FLAGS_NONE
- **Type:** GXF_PARAMETER_TYPE_STRING

**output_binding_names**

Names of output bindings in the model in the same order of of what is provided in output_tensor_names.

- **Flags:** GXF_PARAMETER_FLAGS_NONE
- **Type:** GXF_PARAMETER_TYPE_STRING
### pool

Allocator instance for output tensors.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Allocator

### cuda_stream_pool

Instance of gxf::CudaStreamPool to allocate CUDA stream.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::CudaStreamPool

### max_workspace_size

Size of working space in bytes. Default to 64MB

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64
- Default: 67108864

### dla_core

DLA Core to use. Fallback to GPU is always enabled. Default to use GPU only.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_INT64

### max_batch_size

Maximum possible batch size in case the first dimension is dynamic and used as batch size.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default: 1
enable_fp16
Enable inference with FP16 and FP32 fallback.
- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL
- Default: False

verbose
Enable verbose logging on console. Default to false.
- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL
- Default: False

relaxed_dimension_check
Ignore dimensions of 1 for input tensor dimension check.
- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL
- Default: True

clock
Instance of clock for publish time.
- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

rx
List of receivers to take input tensors
- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Receiver

`tx`

Transmitter to publish output tensors
- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Transmitter