Introduction

1 Overview

2 Clara Holoscan SDK Components
   2.1 DeepStream Sample Applications
   2.2 DeepStream GPU Video Source Plugin
   2.3 Video Pipeline Latency Tool

3 Prerequisites
   3.1 JetPack Software
   3.2 CSI Camera Module
   3.3 USB Camera
   3.4 Linux x86 Host

4 SDK Installation

5 Storage Setup (m2 SSD)
   5.1 Create Partition
   5.2 Mount Partition
   5.3 Move Docker Storage to m2 Partition

6 Camera Setup and Verification
   6.1 AGX Camera Setup
   6.2 Linux x86 Camera Setup
   6.3 Streaming Camera Setup Check

7 Switching Between iGPU and dGPU
   7.1 Reinstalling Clara Holoscan SDK Packages

8 Rivermax SDK
   8.1 Installing Mellanox Drivers (OFED)
   8.2 Installing GPUDirect
   8.3 Installing Rivermax SDK
   8.4 Testing Rivermax and GPUDirect
   8.5 GPUDirect and CUDA Sample
   8.6 Troubleshooting

9 AJA Video Systems
   9.1 Installing the AJA Hardware
   9.2 Installing the AJA NTV2 SDK and Drivers
   9.3 Installing the AJA GStreamer Plugin
   9.4 Testing the AJA GStreamer Plugin
   9.5 Using DeepStream with the AJA GStreamer Plugin
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Troubleshooting</td>
<td>32</td>
</tr>
<tr>
<td>10 DeepStream and Endoscopy Sample Apps</td>
<td>36</td>
</tr>
<tr>
<td>10.1 Clara AGX Developer Kit and Jetson AGX</td>
<td>38</td>
</tr>
<tr>
<td>10.2 Linux x86 DeepStream Sample</td>
<td>41</td>
</tr>
<tr>
<td>10.3 Rebuilding TensorRT Engines</td>
<td>41</td>
</tr>
<tr>
<td>10.4 Using Docker for the DeepStream or Endoscopy Sample Apps</td>
<td>43</td>
</tr>
<tr>
<td>11 DeepStream GPU Video Source Plugin</td>
<td>46</td>
</tr>
<tr>
<td>11.1 Running the GPU Video Test Source</td>
<td>46</td>
</tr>
<tr>
<td>11.2 Accessing the GPU Video Test Source Code</td>
<td>48</td>
</tr>
<tr>
<td>11.3 Custom GPU Source with RDMA Support</td>
<td>48</td>
</tr>
<tr>
<td>12 Video Pipeline Latency Tool</td>
<td>51</td>
</tr>
<tr>
<td>12.1 Building</td>
<td>51</td>
</tr>
<tr>
<td>12.2 Operation Overview</td>
<td>53</td>
</tr>
<tr>
<td>12.3 Graphing Results</td>
<td>63</td>
</tr>
<tr>
<td>12.4 Producers</td>
<td>67</td>
</tr>
<tr>
<td>12.5 Consumers</td>
<td>69</td>
</tr>
<tr>
<td>12.6 Example Configurations</td>
<td>70</td>
</tr>
<tr>
<td>12.7 Troubleshooting</td>
<td>74</td>
</tr>
<tr>
<td>13 NGC Containers</td>
<td>77</td>
</tr>
</tbody>
</table>
Chapter 1

Overview

Clara Holoscan SDK is a collection of documentation, drivers, tools, and reference applications that has been designed by NVIDIA to help developers design and build end-to-end streaming workflows and hardware platforms for medical imaging applications. From building a streaming inference application using DeepStream, to minimizing end-to-end latency using latency measurement tools and GPUDirect RDMA-optimized drivers, to deployment on a Clara AGX embedded device; the Clara Holoscan SDK takes an incremental approach to introduce concepts and NVIDIA technology in a way that can be applied irrespective of previous development experience in the field of medical imaging.

While the ultimate goal of the Clara Holoscan SDK is to deploy applications on a Clara AGX embedded device, the SDK includes x86 Linux versions of the sample applications in order to provide a cross-platform development environment that can be used even without access to a Clara AGX device. In any case, an NVIDIA GPU will be required in order to utilize DeepStream, TensorRT, CUDA, and any other NVIDIA software components that are used by the Clara Holoscan SDK (see Prerequisites).
Chapter 2

Clara Holoscan SDK Components

The Clara Holoscan SDK has been divided into a number of separate installation packages, as follows.

2.1 DeepStream Sample Applications

Building a powerful end-to-end inference application using DeepStream and TensorRT requires the use of a number of different software libraries that can be overwhelming if starting with no knowledge of these libraries. To help alleviate this problem, the Holoscan SDK includes three different sample applications that build on top of one another to incrementally introduce new components and features.

All of the sample applications included with the Holoscan SDK are written in C and/or C++.

2.1.1 GStreamer Sample

The GStreamer sample application is the most basic sample which demonstrates the use of the GStreamer framework in order to simply capture image frames from a camera and then render the captured video stream onto the display. This sample does not make use of DeepStream, but since DeepStream functionality is exposed to applications via GStreamer framework plugins, a basic understanding of GStreamer is essential to write DeepStream applications.

The GStreamer sample is available for the Clara AGX and Linux host, and are located at /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/GStreamerJetsonSample on AGX, and /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/GStreamerLinuxHostSample on the host machine.

2.1.2 DeepStream Sample

The DeepStream sample application builds on top of the GStreamer sample by including the DeepStream inference plugin into the pipeline such that the stomach and intestines regions of a sample model are identified by bounding boxes rendered on top of the video stream. Alternatively, the sample model can be replaced with a user-provided model (by changing a configuration file used by the sample) to identify regions specified by the custom model.

This sample also expands on the GStreamer sample by providing multi-camera support, as well as the ability to send the captured and annotated video across a UDP H.264 video stream such that it can be viewed by another remote system.
The DeepStream sample is available for Clara AGX and the Linux host, and are located at /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/DeepstreamJetsonAGXSample on AGX, and /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/DeepstreamLinuxHostSample on the host machine.

The DeepStream sample is explored in more detail here: DeepStream Sample Application.

2.1.3 Endoscopy Sample

The Endoscopy sample builds on top of the DeepStream sample by adding runtime controls that are specific to the model and inferencing that is performed by DeepStream, such as being able to enable or disable the annotation of individual inference regions that are provided by the model. The UI is also modified to provide a more streamlined experience that might be desirable for an application that is deployed into medical imaging environments.

The Endoscopy sample is available only for Clara AGX, and is located here: /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/EndoscopyJetsonAGXSample.

The Endoscopy sample is explored in more detail here: Endoscopy Sample Application.
2.2 DeepStream GPU Video Source Plugin

To help developers get started with writing GPU-enabled GStreamer plugins that interface with DeepStream and/or use GPUDirect RDMA for efficient data transfer between the GPU and other peripheral devices, the Clara Holoscan SDK includes the DeepStream GPU Video Source Plugin. This sample plugin produces a stream of GPU buffers that are allocated, filled with a test pattern using CUDA, and then output by the plugin such that they can be consumed by other DeepStream or GPU-enabled components. Passing GPU buffers through the pipeline means that data can flow between components directly in GPU memory, avoiding any copies to or from system memory throughout the pipeline.

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 AGX 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.
2.3. Video Pipeline Latency Tool
Chapter 3

Prerequisites

The Clara Holoscan SDK has been optimized to work with either the Jetson AGX Xavier or Clara AGX Developer Kits. The following additional components are required to make use of various feature or samples included with the Holoscan SDK.

3.1 JetPack Software

The following components are required and will be installed onto the AGX device using the NVIDIA SDK Manager.

- JetPack 4.5.1
- DeepStream 6.0 EA

3.2 CSI Camera Module

A CSI camera module is used for the execution of the DeepStream and endoscopy sample applications on the target AGX device. While many CSI camera modules should work with the samples, the following have been tested and are recommended.

- Clara AGX HDMI Input Module (Included with Clara AGX Developer Kit)
- Leopard IMX-274
- Leopard IMX-390 with the GMSL kit

**Note:** The Leopard IMX-390 requires a custom kernel, which can be built using the L4T kernel build instructions.

**Important:** Camera modules with CSI connectors are not supported in dGPU mode on Clara AGX Developer kit.
3.3 USB Camera

The DeepStream Sample application may also be run on either the NVIDIA Xavier or the Linux x86 host machine using a USB camera. Many USB cameras should be compatible, but the following have been tested with the Holoscan SDK.

- Logitech C270
- Logitech C310

3.4 Linux x86 Host

During setup, the Clara Holoscan SDK requires a Linux x86 PC with the following hardware and software:

- An NVIDIA Quadro® or Tesla® GPU that supports CUDA 11.1.
- At least 8GB of RAM
- Ubuntu 18.04
- NVIDIA Container Runtime

The following software is also required and will be installed by SDK Manager as part of the Clara Holoscan SDK installation.

- CUDA 11.1
- TensorRT 7.2
- DeepStream 6.0 EA

After setup is complete the host machine is no longer needed to operate the AGX developer kit, but it may be required for certain tasks such as measuring HDMI latency using the latency measurement tool.
Chapter 4

SDK Installation

Installation of the Clara Holoscan SDK is automated via the NVIDIA SDK Manager. See the SDK Manager documentation for details on how to install and use the SDK Manager to install the Clara Holoscan SDK.

For complete instructions on how to set up and flash your Clara AGX Developer Kit, see the Clara AGX Developer Kit User Guide.

Note: The Clara Holoscan SDK includes DeepStream 6.0 EA, but DeepStream 5.1 may also appear as a stand-alone Additional SDK in SDK Manager. Only one version of DeepStream may be installed at the same time, so be sure to unselect DeepStream 5.1 during the Clara Holoscan SDK installation.
Chapter 5

Storage Setup (m2 SSD)

Note: If the Clara AGX Developer Kit is reflashed with a new JetPack image, the partition table of the m2 drive will not be modified and the contents of the partition will be retained. In this case the Create Partition steps can be skipped, however the Mount Partition steps should be followed again in order to remount the partition.

Also note that any state, binaries, or docker images that persist on the m2 drive after flashing the system may be made incompatible with new libraries or components that are flashed onto the system. It may be required to recompile or rebuild these persistent objects to restore runtime compatibility with the system.

The Clara AGX Developer Kit includes a pre-installed 250GB m2 solid-state drive (SSD), but this drive is not partitioned or mounted by default. This page outlines the steps that should followed after the initial SDK installation in order to partition and format the drive for use.

Note: The following steps assume that the m2 drive is identified by the Clara AGX Developer Kit as /dev/sda. This is the case if no additional drives have been attached, but if other drives have been attached (such as USB drives) then the disk identifier may change. This can be verified by looking at the symlink to the drive that is created for the m2 hardware address on the system. If the symlink below shows something other than ../../../sda, replace all instances of sda in the instruction below with the identifier that is being used by your system:

```
$ ls -l /dev/disk/by-path/platform-14100000.pcie-pci-0001\:01\:00.0-ata-1
lrwxrwxrwx 1 root root 9 Jan 28 12:24 /dev/disk/by-path/platform-14100000.pcie-pci-0001:01:00.0-ata-1 -> ../../../sda
```

5.1 Create Partition

1. Launch fdisk utility:

```
$ sudo fdisk /dev/sda
```

2. Create a new primary partition. Use the command ‘n’, then accept the defaults (press enter) for the next 4 questions to create a single partition that uses the entire drive:
Command (m for help): n
Partition type
    p  primary (0 primary, 0 extended, 4 free)
    e  extended (container for logical partitions)
Select (default p):

Using default response p.
Partition number (1-4, default 1):
First sector (2048-488397167, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-488397167, default → 488397167):

Created a new partition 1 of type 'Linux' and of size 232.9 GiB.

3. Write the new partition table and exit. Use the ‘w’ command:

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.

4. Initialize ext4 filesystem on the new partition:

$ sudo mkfs -t ext4 /dev/sda1
mke2fs 1.44.1 (24-Mar-2018)
Creating filesystem with 486400 4k blocks and 121680 inodes
Filesystem UUID: c3817b9c-eaa9-4423-ad5b-d6bae8aa44ea
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912
Allocating group tables: done
Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

5.2 Mount Partition

1. Create a directory for the mount point. These instructions will use the path /media/m2, but any path may be used if preferred.

   $ sudo mkdir /media/m2

2. Determine the UUID of the new partition. The UUID will be displayed as a symlink to the /dev/sda1 partition within the /dev/disk/by-uuid directory. For example, the following output shows that the UUID of the /dev/sda1 partition is 4b2bb292-a4d8-4b7e-a8cc-bb799dfeb925:

   $ ls -l /dev/disk/by-uuid/ | grep sda1
   lrwxrwxrwx 1 root root 10 Jan 28 10:05 4b2bb292-a4d8-4b7e-a8cc-bb799dfeb925 -> ../../../sda1
   (continues on next page)
3. **Add the fstab entry.** Using the mount path and the UUID from the previous steps, add the following line to the end of `/etc/fstab`:

```
UUID=4b2bb292-a4d8-4b7e-a8cc-bb799dfeb925 /media/m2 ext4 defaults 0 2
```

4. **Mount the partition.** The `/etc/fstab` entry above will mount the partition automatically at boot time. To instead mount the partition immediately without rebooting, use the `mount` command (and `df` to verify the mount):

```
$ sudo mount -a
$ df -h /dev/sda1
```

<table>
<thead>
<tr>
<th>Filesystem</th>
<th>Size</th>
<th>Used</th>
<th>Avail</th>
<th>Use%</th>
<th>Mounted on</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/sda1</td>
<td>229G</td>
<td>5.6M</td>
<td>229G</td>
<td>0%</td>
<td>/media/m2</td>
</tr>
</tbody>
</table>

### 5.3 Move Docker Storage to m2 Partition

A complete installation of the Clara SDK leaves only about 10GB of storage remaining in the root 32GB filesystem (`/dev/mmcblk0p1`). When using Docker it is often the case that individual images will be many GB in size, and so this remaining disk space is generally insufficient for the storage needs of Docker images. For this reason it is highly recommended that the Docker daemon data directory be moved to a location on the new m2 partition. This can be done with the following steps:

1. **Create a new Docker data directory.** This is where Docker will store all of its data including build cache and container images. These instructions use the path `/media/m2/docker-data`, but another directory name can be used if preferred:

   ```
   $ sudo mkdir /media/m2/docker-data
   ```

2. **Configure the Docker Daemon.** Add the following `data-root` configuration option to the `/etc/docker/daemon.json` file, pointing to the new data directory created above. Create the `/etc/docker/daemon.json` file if it does not already exist.

   ```
   {
       "data-root": "/media/m2/docker-data"
   }
   ```

   **Note:** If existing configuration already exists in the `daemon.json` file, make sure to add a comma to the preceding line before the `data-root` configuration, e.g.

   ```
   {
       "default-runtime": "nvidia",
       "data-root": "/media/m2/docker-data"
   }
   ```

3. **Restart the Docker Daemon.**
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
Chapter 6

Camera Setup and Verification

This page outlines how to verify the camera setup used with the Clara Holoscan target and x86 host devices to ensure they are functioning properly for use with the sample applications.

6.1 AGX Camera Setup

1. Connect a CSI or USB camera module to your Clara AGX Developer Kit or Jetson AGX module. Once connected, you may see what device(s) you have by the following:

   $ ls /dev/video*

   The format(s) of your device(s) may be seen by:

   $ v4l2-ctl -d /dev/video0 --list-formats

   If the above command fails (which may occur if the DeepStream example was not installed), then run:

   $ sudo apt install -y v4l-utils

   Please see the Prerequisites section for the list of compatible camera devices.

   **Note:** The CSI camera module, including the CSI HDMI input board included with the Clara AGX Developer Kit, will enumerate as /dev/video0 and any additional USB cameras will start to enumerate with /dev/video1. If there is no CSI module attached, USB cameras will start to enumerate with /dev/video0.

2. Run one of the following commands to prove that capture is working correctly. In all cases, a window should be rendered onto the display with a live stream from the capture.

   • CSI Camera

   $ gst-launch-1.0 nvarguscamerasrc ! 'video/x-raw(memory:NVMM), width=(int)1920, height=(int)1080, format=(string)NV12, framerate=(fraction)30/1' ! nvoverlaysink -e
• CSI HDMI Input Board

```
$ gst-launch-1.0 v4l2src io-mode=mmap device=/dev/video0 ! video/x-raw, format=(string)BGRA, width=(int)1920, height=(int)1080, framerate=(fraction)60/1 ! capssetter join=true caps='video/x-raw, format=(string)RGBA' ! nvvideoconvert ! video/x-raw(memory:NVMM), format=(string)RGBA ! nveglglessink sync=0
```

• USB Camera

```
$ gst-launch-1.0 v4l2src device=/dev/video1 ! xvimagesink
```

Note: Replace /dev/video0 in the commands above with the path that corresponds to the device being tested. See note about device enumeration in step 1, above.

3. Press Ctrl+C to exit the camera capture.

### 6.2 Linux x86 Camera Setup

1. Connect a USB camera to your Linux x86 host.

2. Run the following command to prove that capture is working correctly. A window should be rendered onto the display with a live stream from the camera.

```
$ gst-launch-1.0 v4l2src device=/dev/video1 ! xvimagesink
```

Note: Replace /dev/video1 in the command above with the path that corresponds to the USB camera device being tested.

3. Press Ctrl+C to exit the camera capture.

### 6.3 Streaming Camera Setup Check

The DeepStream sample application offers the ability to stream the camera image from the target device to a remote host system using a UDP data stream. These steps can be followed to ensure that both the target and host systems are setup correctly for this streaming to work.

#### 6.3.1 AGX Target

1. Ensure the camera capture is working as described in the AGX Camera Setup section, above.
2. Run the following command to start streaming the camera capture, replacing `<IP_ADDRESS>` with the IP address of the host system that will be receiving the stream.

- **CSI Camera**

```bash
$ gst-launch-1.0 nvarguscamerasrc ! 'video/x-raw(memory:NVMM), width=(int)1920, height=(int)1080, format=(string)NV12, framerate=(fraction)30/1' ! omxh264enc control-rate=2 bitrate=4000000 ! 'video/x-h264, stream-format=(string)byte-stream' ! h264parse ! rtph264pay mtu=1400 ! udpsink host=<IP_ADDRESS> port=5000 sync=false async=false
```

- **CSI HDMI Input Board**

```bash
$ gst-launch-1.0 v4l2src io-mode=mmap device=/dev/video0 ! 'video/x-raw, format=(string)BGRA, width=(int)1920, height=(int)1080, framerate=(fraction)60/1' ! capssetter join=true caps='video/x-raw, format=(string)RGBA' ! nvvideoconvert ! 'video/x-raw(memory:NVMM), format=(string)I420' ! nv412h264enc ! h264parse ! rtph264pay ! udpsink host=<IP_ADDRESS> port=5000
```

- **USB Camera**

```bash
$ gst-launch-1.0 v4l2src device=/dev/video0 ! nvvideoconvert ! 'video/x-raw(memory:NVMM), format=(string)I420' ! nv412h264enc ! h264parse ! rtph264pay ! udpsink host=<IP_ADDRESS> port=5000
```

**Note:** Replace `/dev/video0` in the commands above with the path that corresponds to the device being tested. See note about device enumeration in step 1 of the AGX Camera Setup, above.

### 6.3.2 Linux Host

Use the following command to start streaming the camera capture on a Linux x86 host (close with Ctrl+C):

```bash
$ gst-launch-1.0 udpsrc port=5000 ! application/x-rtp,encoding-name=H264, payload=96 ! rtph264depay ! queue ! avdec_h264 ! autovideosink sync=false
```

---

6.3. Streaming Camera Setup Check 15
Chapter 7

Switching Between iGPU and dGPU

The Clara AGX Developer Kit can use either the Xavier AGX module GPU (iGPU, integrated GPU) or the RTX6000 add-in card GPU (dGPU, discrete GPU). Only one GPU can be used at a time, with the iGPU being the default after flashing the Clara Holoscan SDK with JetPack. Switching between the iGPU and dGPU is performed using the `nvgpuswitch.py` script contained in `/usr/local/bin`.

**Warning:** The Clara Holoscan SDK components, such as the DeepStream sample applications, will be uninstalled during the process of switching GPUs. If you have made any changes to the source code installed by these packages (i.e. in `/opt/nvidia/clara-holoscan-sdk`), make sure to back up these changes before switching GPUs. See *Reinstalling Clara Holoscan SDK Packages* for more details.

To view the currently installed drivers and their version, use the query command:

```
$ nvgpuswitch.py query
iGPU (nvidia-l4t-cuda, 32.5.0-20201012161040)
```

To install the dGPU drivers, use the install command with the dGPU parameter (note that `sudo` must be used to install drivers):

```
$ sudo nvgpuswitch.py install dGPU
```

The install command will begin by printing out the list of commands that will be executed as part of the driver install, then will continue to execute those commands. This aids with debugging if any of the commands fail to execute for any reason. The following **debug arguments** may also be provided with the install command:

- `-d` Does a dry run, showing the commands that would be executed by the install but does not execute them.

- `-v` Enable verbose output (used with `-d` to describe each of the commands that would be run).

- `-i` Run commands interactively (asks before running each command).

- `-l [LOG]` Writes a log of the install to the file `LOG`.
The dGPU driver install may be verified once again using the query command:

```
$ nvgpuswitch.py query
dGPU (cuda-drivers, 455.32.00-1)
```

After the dGPU drivers have been installed, rebooting the system will complete the switch to the dGPU. At this point the Ubuntu desktop will be output via DisplayPort on the dGPU, and so the display cable must be switched from the onboard HDMI to DisplayPort on the dGPU.

**Note:** CUDA installs its runtime binaries such as `nvcc` into its own versioned path that is not included by the default `$PATH` environment variable. Because of this, attempts to run commands like `nvcc` will fail on dGPU unless the CUDA 11.1 path is added to the `$PATH` variable. This can be done for the current user after the switch to dGPU by adding the following line to `$HOME/.bashrc`:

```
export PATH=/usr/local/cuda-11.1/bin:$PATH
```

If at any time you want to switch back to **iGPU**, use the install command with the `iGPU` parameter:

```
$ sudo nvgpuswitch.py install iGPU
```

After the iGPU drivers have been installed, rebooting the system will complete the switch to the iGPU. At this point the Ubuntu desktop will be output via the onboard HDMI, and so the display cable must be switched from the DisplayPort on the dGPU to the onboard HDMI.

**Note:** The GPU settings will persist through reboots until it is changed again with nvgpuswitch.py.

### 7.1 Reinstalling Clara Holoscan SDK Packages

When switching between GPUs, CUDA is first uninstalled and then reinstalled by the script in order to provide the correct versions used by iGPU or dGPU (CUDA 10.2 and 11.1, respectively). Since many of the Clara Holoscan SDK installed packages depend on CUDA, this means that these Clara Holoscan SDK packages are also uninstalled when the active GPU is switched.

To reinstall the Clara Holoscan SDK packages after switching GPUs, the corresponding `*.deb` packages that were downloaded by SDK Manager during the initial installation can be copied to the Clara AGX Developer Kit and installed using `apt`. By default, SDK Manager downloads the `*.deb` packages to the following location:

```
~/Downloads/nvidia/sdkm_downloads
```

The current list of packages that must be reinstalled is as follows. Note that the version numbers may differ – if this is the case, use the latest version of the `arm64` package that exists in the download directory.

- **VisionWorks**
  - `libvisionworks` and `libvisionworks-dev`
- **DeepStream 6.0**
  - `deepstream-6.0_6.0.0-1_arm64.deb`
- **Clara Holoscan DeepStream Samples**
  
  clara-holoscan-deepstream-sample_1.4.0-0_arm64.deb

- **DeepStream GPU Video Test Source Plugin**
  
  clara-holoscan-gstnvvideotestsRC_1.1.0-0_arm64.deb

Use `apt` to reinstall these packages:

```bash
$ sudo apt install -y libvisionworks libvisionworks-dev
$ sudo apt install -y ./deepstream-6.0_6.0.0-1_arm64.deb
$ sudo apt install -y ./clara-holoscan-deepstream-sample_1.4.0-0_arm64.deb
$ sudo apt install -y ./clara-holoscan-gstnvvideotestsRC_1.1.0-0_arm64.deb
```
Chapter 8

Rivermax SDK

The Clara AGX Developer Kit can be used along with the NVIDIA Rivermax SDK to provide an extremely efficient network connection using the onboard ConnectX-6 network adapter that is further optimized for GPU workloads by using GPUDirect. This technology avoids unnecessary memory copies and CPU overhead by copying data directly to or from pinned GPU memory, and supports both the integrated GPU or the RTX6000 add-in dGPU.

The instructions below describe the steps required to install and test the Rivermax SDK with the Clara AGX Developer Kit. The test applications used by these instructions, `generic_sender` and `generic_receiver`, can then be used as samples in order to develop custom applications that use the Rivermax SDK to optimize data transfers using GPUDirect.

Note: The Rivermax SDK may also be installed onto the Clara AGX Developer Kit via SDK Manager by selecting it as an additional SDK during the JetPack installation. If Rivermax SDK was previously installed by SDK Manager, many of these instructions can be skipped (see additional notes in the steps below).

Note: Access to the Rivermax SDK Developer Program as well as a valid Rivermax software license is required to use the Rivermax SDK.

8.1 Installing Mellanox Drivers (OFED)

The Mellanox OpenFabrics Enterprise Distribution Drivers for Linux (OFED) must be installed in order to use the ConnectX-6 network adapter that is onboard the Clara AGX Developer Kit.

Note: If Rivermax SDK was previously installed via SDK Manager, OFED will already be installed and these steps can be skipped.

1. Download OFED version 5.4-1.0.3.0:
   
   MLNX_OFED_LINUX-5.4-1.0.3.0-ubuntu18.04-aarch64.tgz

   If the above link does not work, navigate to the Downloads section on the main OFED page, select either Current Versions or Archive Versions to find version 5.4-1.0.3.0, select Ubuntu, Ubuntu...
18.04, aarch64, then download the tgz file.

Note: Newer versions of OFED have not been tested and may not work.

2. Install OFED:

```
$ sudo apt install -y apt-utils
$ tar -xvf MLNX_OFED_LINUX-5.4-1.0.3.0-ubuntu18.04-aarch64.tgz
$ cd MLNX_OFED_LINUX-5.4-1.0.3.0-ubuntu18.04-aarch64
$ sudo ./mlnxofedinstall --force --force-fw-update --vma --add-kernel-support
$ sudo /etc/init.d/openibd restart
```

8.2 Installing GPUDirect

The GPUDirect drivers must be installed to enable the use of GPUDirect when using an RTX6000 add-in dGPU. When using the iGPU the CPU and GPU share the unified memory and the GPUDirect drivers are not required, so this step may be skipped when using the iGPU.

Note: The GPUDirect drivers are not installed by SDK Manager, even when Rivermax SDK is installed, so these steps must always be followed to enable GPUDirect support when using the dGPU.

1. Download GPUDirect Drivers for OFED:

   nvidia-peer-memory_1.1.tar.gz

   If the above link does not work, navigate to the Downloads section on the GPUDirect page.

2. Install GPUDirect:

```
$ mv nvidia-peer-memory_1.1.tar.gz nvidia-peer-memory_1.1.orig.tar.gz
$ tar -xvf nvidia-peer-memory_1.1.orig.tar.gz
$ cd nvidia-peer-memory-1.1
$ dpkg-buildpackage -us -uc
$ sudo dpkg -i ../nvidia-peer-memory_1.1-0_all.deb
$ sudo dpkg -i ../nvidia-peer-memory-dkms_1.1-0_all.deb
$ sudo service nv_peer_mem start
```

Verify the nv_peer_mem service is running:

```
$ sudo service nv_peer_mem status
```

Enable the nv_peer_mem service at boot time:

```
$ sudo systemctl enable nv_peer_mem
$ sudo /lib/systemd/systemd-sysv-install enable nv_peer_mem
```
8.3 Installing Rivermax SDK

Note: If Rivermax SDK was previously installed via SDK Manager, the download and install steps (1 and 2) can be skipped. The Rivermax license must still be installed, however, so step 3 must still be followed.

1. Download version 1.8.21 or newer of the Rivermax SDK from the NVIDIA Rivermax SDK developer page.
   a. Click Get Started and login using your NVIDIA developer account.
   b. Scroll down to Downloads and click I Agree To the Terms of the NVIDIA Rivermax Software Licence Agreement
   c. Select Rivermax SDK 1.8.21, Linux, then download rivermax_ubuntu1804_1.8.21.tar.gz.
      If a newer version is available, replace 1.8.21 in this and all following steps with the newer version that is available.

2. Install Rivermax SDK:
   ```
   $ tar -xvf rivermax_ubuntu1804_1.8.21.tar.gz
   $ sudo dpkg -i 1.8.21/Ubuntu.18.04/deb-dist/aarch64/rivermax_11.3.9.21--arm64.deb
   ```

3. Install Rivermax License

Using Rivermax requires a valid license, which can be purchased from the Rivermax Licenses page. Once the license file has been obtained, it must be placed onto the system using the following path:

```
/opt/mellanox/rivermax/rivermax.lic
```

8.4 Testing Rivermax and GPUDirect

Running the Rivermax sample applications requires two systems, a sender and a receiver, connected via ConnectX network adapters. If two Clara AGX Developer Kits are used then the onboard ConnectX-6 can be used on each system, but if only one Clara AGX is available then it’s expected that another system with an add-in ConnectX network adapter will need to be used. Rivermax supports a wide array of platforms, including both Linux and Windows, but these instructions assume that another Linux based platform will be used as the sender device while the Clara AGX is used as the receiver.

1. Determine the logical name for the ConnectX devices that are used by each system. This can be done by using the lshw -class network command, finding the product: entry for the ConnectX device, and making note of the logical name: that corresponds to that device. For example, this output on a Clara AGX shows the onboard ConnectX-6 device using the enp9s0f01 logical name (lshw output shortened for demonstration purposes).

```
$ sudo lshw -class network
*-network:0
   description: Ethernet interface
   product: MT28908 Family [ConnectX-6]
```
2. Run the `generic_sender` application on the sending system.

   a. Bring up the network:

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

   b. Build the sample apps:

   ```
   $ cd 1.8.21/apps
   $ make
   ```

   **Note:** The `1.8.21` path above corresponds to the path where the Rivermax SDK package was extracted in step 2 of the `Installing Rivermax SDK` section, above. If the Rivermax SDK was installed via SDK Manager, this path will be `$HOME/Documents/Rivermax/1.8.21`.

   e. Launch the `generic_sender` application:

   ```
   $ sudo ./generic_sender -l 10.0.0.1 -d 10.0.0.2 -p 5001 -y 1462 -k 8192 -z 500 -v
   ...
   +*********************************************************************************
   | Sender index: 0
   | Thread ID: 0x7fa1ffb1c0
   | CPU core affinity: -1
   | Number of streams in this thread: 1
   | Memory address: 0x7f986e3010
   | Memory length: 59883520[B]
   | Memory key: 40308
   +*********************************************************************************
   | Stream index: 0
   ```
3. Run the `generic_receiver` application on the receiving system.

a. Bring up the network:

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

b. Build the sample apps with GPUDirect support (CUDA=y):

```bash
$ cd 1.8.21/apps
$ make CUDA=y
```

**Note:** The `1.8.21` path above corresponds to the path where the Rivermax SDK package was extracted in step 2 of the *Installing Rivermax SDK* section, above. If the Rivermax SDK was installed via SDK Manager, this path will be `$HOME/Documents/Rivermax/1.8.21`.

c. Launch the `generic_receiver` application:

```bash
$ sudo ./generic_receiver -i 10.0.0.2 -m 10.0.0.2 -s 10.0.0.1 -p 5001 -g 0
```

... 

Attached flow 1 to stream.

Running main receive loop...

Got 5877704 GPU packets | 68.75 Gbps during 1.00 sec
Got 5878240 GPU packets | 68.75 Gbps during 1.00 sec
Got 5878240 GPU packets | 68.75 Gbps during 1.00 sec
Got 5877704 GPU packets | 68.75 Gbps during 1.00 sec

... 

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

GPUDirect is ideal for applications which receive data from the network adapter and then use the GPU to process the received data directly in GPU memory. The generic_sender and generic_receiver demo applications include a simple demonstration of the use of CUDA with received packets by using a CUDA kernel to compute and then compare a checksum of the packet against an expected checksum as provided by the sender. This additional checksum packet included by the sender also includes a packet sequence number that is used by the receiver to detect when any packets are lost during transmission.

In order to enable the CUDA checksum sample, append the -x parameter to the generic_sender and generic_receiver commands that are run above.

Due to the increased workload by the receiver when the checksum calculation is enabled, you will begin to see dropped packets and/or checksum errors if you try to maintain the same data rate from the sender as you did when the checksum was disabled (i.e. when all received packet data was simply discarded). Because of this the sleep parameter used by the sender, -z, should be increased until there are no more dropped packets or checksum errors. In this example, the sleep parameter was increased from 500 to 40000 in order to ensure the receiver can receive and process the sent packets without any errors or loss:

```
[Sender]
$ sudo ./generic_sender -l 10.0.0.1 -d 10.0.0.2 -p 5001 -y 1462 -k 8192 -z 40000 -v -x

[Receiver]
$ sudo ./generic_receiver -i 10.0.0.2 -m 10.0.0.2 -s 10.0.0.1 -p 5001 -g 0 -x ...
Got 203968 GPU packets | 2.40 Gbps during 1.02 sec | 0 dropped packets | 0 checksum errors
Got 200632 GPU packets | 2.36 Gbps during 1.00 sec | 0 dropped packets | 0 checksum errors
Got 203968 GPU packets | 2.40 Gbps during 1.01 sec | 0 dropped packets | 0 checksum errors
Got 201608 GPU packets | 2.37 Gbps during 1.01 sec | 0 dropped packets | 0 checksum errors
```

If you would like to write an application that uses Rivermax and GPUDirect for CUDA data processing, refer to the source code for the generic_sender and generic_receiver applications included with the Rivermax SDK in generic_sender.cpp and generic_receiver.cpp, respectively.

**Note:** The CUDA checksum calculation in the generic_receiver is included only to show how the data received through GPUDirect can be processed through CUDA. This example is not optimized in any way, and should not be used as an example of how to write a high-performance CUDA application. Please refer to the CUDA Best Practices Guide for an introduction to optimizing CUDA applications.

8.6 Troubleshooting

If running the driver installation or sample applications do not work, check the following.

1. The ConnectX network adapter is recognized by the system. For example, on a Linux system using a ConnectX-6 Dx add-in PCI card:
Clara Holoscan SDK User Guide

$ lspci
...
0000:05:00.0 Ethernet controller: Mellanox Technologies MT28841
0000:05:00.1 Ethernet controller: Mellanox Technologies MT28841
...

If the network adapter is not recognized, try rebooting the system and/or reseating the card in the PCI slot.

2. The ConnectX network adapter is recognized by the OFED driver. For example, on a Linux system using a ConnectX-6 Dx add-in PCI card:

```
$ sudo mlxfwmanager
...
Device Type: ConnectX6DX
Part Number: MCX623106AC-CDA_Ax
Description: ConnectX-6 Dx EN adapter card; 100GbE; Dual-port...
→ QSF56; PCIe 4.0 x16; Crypto and Secure Boot
PSID: MT_0000000436
PCI Device Name: /dev/mst/mt4125_pciconf0
Base GUID: 0c42a1030024053a
Base MAC: 0c42a124053a
Versions: Current Available
  FW 22.31.1014 N/A
  FW (Running) 22.30.1004 N/A
  PXE 3.6.0301 N/A
  UEFI 14.23.0017 N/A
```

If the device does not appear, first try rebooting and then :ref:`reinstalling OFED <installing_ofed>` as described above.

3. The sender and receiver systems can ping each other:

```
$ ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.205 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.206 ms
...
```

If the systems cannot ping each other, try bringing up the network interfaces again using the `ifconfig` commands.

4. The `nv_peer_mem` service is running:

```
$ sudo service nv_peer_mem status
* nv_peer_mem.service - LSB: Activates/Deactivates nv_peer_mem to \ start...
→ at boot time.
Loaded: loaded (/etc/init.d/nv_peer_mem; generated)
Active: active (exited) since Mon 2021-01-25 16:45:08 MST; 9min ago
Docs: man:systemd-sysv-generator(8)
Process: 6847 ExecStart=/etc/init.d/nv_peer_mem start (code=exited, ...
→ status=0/SUCCESS)
```

(continues on next page)
If the service is not running, try starting it again using `sudo service nv_peer_mem start`. 
Chapter 9

AJA Video Systems

AJA provides a wide range of proven, professional video I/O devices, and a partnership between NVIDIA and AJA has led to the addition of Clara Holoscan support to 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.

RDMA support from AJA devices has also been incorporated into the AJA GStreamer plugin in order to enable zero-copy GPU buffer integration with the DeepStream SDK. This support allows DeepStream applications to process video data along the entire pipeline, from the initial capture to final display, without ever leaving GPU memory.

The following instructions describe the steps required to setup and use an AJA device with Clara Holoscan platforms, including RDMA and DeepStream integration. 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.

9.1 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.
9.2 Installing the AJA NTV2 SDK and Drivers

1. Download the AJA NTV2 SDK
   
a. Login to your AJA developer account at https://sdksupport.aja.com.

   **Note:** Access to the AJA NTV2 SDK requires access to the AJA developer program, which can be requested from https://aja.com/developer/request.

   b. Under Knowledgebase on the left side, go to NTV2 SDK Downloads.

   c. Under Current NTV2 SDKs, go to NTV2 SDK 16.1 for Linux, MacOS and Windows.

   d. At the bottom of the page, under Attachments, download ntv2sdklinux_16.1.0.3.zip and ntv2sdklinux_16.1.0.3_arm.zip. The first zip file contains the NTV2 SDK itself, while the second zip file contains libraries and binaries that have been built for the arm64 architecture as required by Clara Holoscan.

   **Note:** If a newer version of the 16.1 SDK has been released, replace 16.1.0.3 above and in all following steps with the version that is available for download.

   e. Move the downloaded zip files to a new directory, e.g. ${HOME}/aja. In all following steps, this path will be referred to as AJA_BASE.

   ```
   $ export AJA_BASE=${HOME}/aja
   $ mkdir ${AJA_BASE}
   $ mv ~/Downloads/ntv2sdklinux_16.1.0.3.zip ${AJA_BASE}
   $ mv ~/Downloads/ntv2sdklinux_16.1.0.3_arm.zip ${AJA_BASE}
   ```

2. Build and install the AJA drivers with RDMA support
   
a. Extract the NTV2 SDK zip files.

   ```
   $ cd ${AJA_BASE}
   $ export NTV2_SDK=ntv2sdklinux_16.1.0.3
   $ unzip ${NTV2_SDK}.zip
   $ unzip ${NTV2_SDK}_arm.zip
   ```

   b. Copy the arm64 libraries to the main SDK directory.

   ```
   $ cp ${AJA_BASE}/${NTV2_SDK}_arm/lib/* ${AJA_BASE}/${NTV2_SDK}/lib/
   ```

   c. Build the AJA drivers with RDMA support.

   ```
   $ cd ${AJA_BASE}/${NTV2_SDK}/ajadriver/linux
   $ export AJA_RDMA=1
   $ make
   ```

   d. Load the AJA drivers.
$ sudo ${AJA_BASE}/${NTV2_SDK}/bin/load_ajantv2
loaded ajantv2 driver module
created node /dev/ajantv20

e. Ensure that the AJA driver and RDMA are working by using the rdmawhacker utility included with the NTV2 SDK.

$ cd ${AJA_BASE}/${NTV2_SDK}/ajaapps/crossplatform/rdmawhacker
$ make
$ ${AJA_BASE}/${NTV2_SDK}/bin/rdmawhacker

DMA engine 1 WRITE 8388608 bytes rate: 4678.72 MB/sec 584.84 → xfers/sec

<Ctrl-C to exit>

9.3 Installing the AJA GStreamer Plugin

1. Clone the ntv2-gst git repo.

   $ cd ${AJA_BASE}
   $ git clone https://github.com/ibstewart/ntv2-gst.git

   Note: The ibstewart/ntv2-gst repo is a fork of the aja-video/ntv2-gst repo that adds some changes required for RDMA support.

2. Setup the build environment and paths.

   $ sudo apt install libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev
   $ sudo ln -sf /usr/lib/aarch64-linux-gnu/tegra/libnvdsbufferpool.so /opt/nvidia/deepstream/deepstream-6.0/lib/
   $ export GST_CUDA=/usr/local/cuda-11.1/targets/sbsa-linux
   $ export GST_DEEPSTREAM=/opt/nvidia/deepstream/deepstream-6.0
   $ export GST_NTV2=${AJA_BASE}/${NTV2_SDK}

   Note: If the following error is output when creating the symlink using the ln command above, make sure that DeepStream has been installed as described in Reinstalling Clara Holoscan SDK Packages.

   ln: target '/opt/nvidia/deepstream/deepstream-6.0/lib/' is not a directory

3. Build and install the plugin.

   $ cd ${AJA_BASE}/ntv2-gst/gst-plugin
   $ ./autogen.sh
   $ make

(continues on next page)
9.4 Testing the AJA GStreamer Plugin

The following `gst-launch-1.0` command can be used to test the AJA GStreamer plugin with an SDI device.

```
$ gst-launch-1.0 ajavideosrc mode=720p5994-rgba nvmm=true ! nv3dsink
```

The above assumes that a 720p @ 59.94 FPS video stream is being broadcast to SDI channel 1 (i.e. plugin index 0) of the AJA SDI video capture card. This video stream may be generated by any SDI source, but for testing purposes it may be useful to use the `ntv2player` utility that is included with the NTV2 SDK in order to generate a known test pattern. This test pattern can even be generated by the same AJA card that is receiving the stream by connecting a loopback between two channels on the same AJA card.

For example, if a loopback is created by connecting an SDI cable between channels 1 and 2 on the same AJA card, the following command can be used to output a test signal on channel 2 that is then received on channel 1 using the `gst-launch-1.0` command above.

```
$ ${AJA_BASE}/${NTV2_SDK}_arm/bin/ntv2player -v=720p -c=2
```

To use an AJA KONA HDMI card, `input-mode=hdmi` needs to be set in the `gst-launch-1.0` command. For reference, see the following command.

```
$ gst-launch-1.0 ajavideosrc mode=UHDp30-rgba input-mode=hdmi nvmm=true ! nv3dsink
```

**Note:** KONA HDMI is a capture-only device, and therefore is not capable of outputting a test signal via a loopback cable as described in the SDI case above. To capture from a KONA HDMI, a suitable signal must be produced from another HDMI device.

If issues are encountered when running these commands, see *Troubleshooting*, below.

**Note:** The `nvmm=true` parameter that is provided to the `ajavideosrc` plugin is what enables the use of RDMA so that the buffers written by the plugin are output directly to GPU memory via NVMM buffers (i.e. using the GStreamer caps `video/x-raw(memory:NVMM)`).

Enabling RDMA also requires the use of an `rgba` mode due to GPU support limitations. See Problem 3 in the *Troubleshooting* section below for more details.

9.5 Using DeepStream with the AJA GStreamer Plugin

Since the AJA plugin using RDMA outputs to GPU (`video/x-raw(memory:NVMM)`) buffers directly, an RDMA stream from the AJA plugin may be passed directly into DeepStream without requiring any
additional buffer conversions or transfers between sysmem and GPU. Assuming that the Clara Holoscan DeepStream sample is installed, the following command will run a GStreamer pipeline that captures a stream from the AJA card using RDMA and passes it through DeepStream using the sample model files included with the DeepStream sample.

```
$ gst-launch-1.0 ajavideosrc mode=720p5994-rgba nvmm=true \
  ! m.sink_0 nvstreammux name=m width=1280 height=720 batch-size=1 \
  ! nvinfer config-file-path=/opt/nvidia/clara-holoscan-sdk/clara-holoscan-\n ˓→deepstream-sample/build/DeepstreamJetsonAGXSample/dsjas_nvinfer_config.txt \
  ! nvdsosd ! nv3dsink sync=false
```

If issues are encountered when running this commands, see Troubleshooting, below.

### 9.6 Troubleshooting

If the `gst-launch-1.0` commands described in steps above fail to render a video stream onto the display, the following steps may help resolve the issue.

1. **Problem**: The command fails with the error:

   ```
   ERROR: from element /GstPipeline:pipeline0/GstNv3dSink:nv3dsink0: 
   GStreamer error: state change failed and some element failed to post 
   a proper error message with the reason for the failure.
   ```

   **Solution**: Make sure the DISPLAY environment variable is set (replace :0 with the ID of the X11 display you are using):
$ export DISPLAY=:0

2. **Problem:** The command fails with the error:

```plaintext
WARNING: from element /GstPipeline:pipeline0/
   → GstAjaVideoSrc:ajavideosrc0: Signal lost
Additional debug info:
gstajavideosrc.cpp(1325): gst_aja_video_src_create (): /
   → GstPipeline:pipeline0/GstAjaVideoSrc:ajavideosrc0:
No input source was detected - video frames invalid
```

**Solution:** This error occurs when the mode parameter provided to the ajavideosrc plugin does not match the video format that is being received by the AJA card.

The video format that is being received by the AJA card as well as the format that is being requested by the plugin can be checked by using the ntv2watcher utility that is included with the NTV2 SDK:

```plaintext
$ sudo apt install libqt5multimedia5
$ ${AJA_BASE}/${NTV2_SDK}_arm/bin/ntv2watcher
```

In the app, select *Routing* on the left navigation bar to see the current status and pipeline setup of the AJA card:

In this example, we can see that SDI channel 1 is currently receiving a **1080p60** video signal while the pipeline setup by the plugin is expecting a **720p59.94** signal (Frame Store 1). To resolve this, the mode parameter for the plugin should be changed to match the signal that is being received.

To view this list of modes that are supported by the plugin, use the **gst-inspect-1.9** utility (the
following output is shortened for illustration purposes):

```bash
$ gst-inspect-1.0 ajavideosrc
... mode : Video Mode to use for playback flags: readable, writable
Enum "GstAjaRawModes" Default: 15, "HD720 8Bit 59.94p"
  (0): NTSC 8Bit 23.98i - ntsc-2398
  (1): NTSC 8Bit 24i - ntsc-24
  (2): NTSC 8Bit 59.94i - ntsc
  (3): NTSC 8Bit RGBA 23.98i - ntsc-2398-rgba
  (4): NTSC 8Bit RGBA 24i - ntsc-24-rgba
  (5): NTSC 8Bit RGBA 59.94i - ntsc-rgba
  ...
  (38): HD1080 8Bit RGBA 23.98p - 1080p2398-rgba
  (39): HD1080 8Bit RGBA 24p - 1080p24-rgba
  (40): HD1080 8Bit RGBA 25p - 1080p25-rgba
  (41): HD1080 8Bit RGBA 29.97p - 1080p2997-rgba
  (42): HD1080 8Bit RGBA 30p - 1080p30-rgba
  (43): HD1080 8Bit RGBA 50i - 1080i50-rgba
  (44): HD1080 8Bit RGBA 50p - 1080p50-rgba
  (45): HD1080 8Bit RGBA 59.94i - 1080i5994-rgba
  (46): HD1080 8Bit RGBA 59.94p - 1080p5994-rgba
  (47): HD1080 8Bit RGBA 60i - 1080i60-rgba
  (48): HD1080 8Bit RGBA 60p - 1080p60-rgba
  ...
```

In this particular example, the 1080p60-rgba mode can be used for the plugin in order to match
the signal that is being received:

```
$ gst-launch-1.0 ajavideosrc mode=1080p60-rgba nvmm=true ! nv3dsink
```

3. **Problem:** The command fails with the error:

```
WARNING: erroneous pipeline: could not link ajavideosrc0 to nv3dsink0
```

**Solution:** The reason for this error is that the video format that is being requested by the ajavideosrc plugin (i.e. the `mode` parameter) is not supported by the nv3dsink plugin.

The most likely reason for this incompatibility is that the nv3dsink – and all NVIDIA GStreamer/DeepStream plugins – do not support the packed UYVY (8-bit) or v210 (10-bit) YUV 4:2:2 formats that are output by the ajavideosrc plugin. Because of this, an RGBA mode must be used with the ajavideosrc plugin when it is used with NVIDIA plugins.

In order to select an RGBA mode, ensure that the `mode` parameter that is given to the ajavideosrc plugin has the `-rgba` suffix, e.g. `1080p60-rgba`.

4. **Problem:** The command fails with the error:

```
ERROR: [TRT]: INVALID_CONFIG: Deserialize the cuda engine failed.
```

**Solution:** The reason for this error is that the DeepStream engine files for the DeepStream sample application have not been generated. See *Rebuilding TensorRT Engines* for the steps to resolve this issue.

5. **Problem:** After loading the AJA drivers with `load_ajantv2`, the display briefly flashes and then the desktop applications, UI, and cursor no longer appear.

**Solution:** If a loopback cable between the GPU and AJA card is connected, such as is done for the *Video Pipeline Latency Tool*, loading the AJA drivers may cause the system to detect the AJA card as another display and adjust the desktop to use the loopback output as the primary display. To resolve this, remove any loopback cable between the GPU and AJA device.
Chapter 10

DeepStream and Endoscopy Sample Apps

The DeepStream and Endoscopy sample applications provide examples of how to use the DeepStream SDK from a native C++ application in order to perform inference on a live video stream from a camera or video capture card. Both of these applications use a sample inference model that is able to detect the stomach and intestine regions of a Learning Resources Anatomy Model.
If the Learning Resources Anatomy Model is not available, the inference functionality of these sample applications can be tested using a subset of the original training images that are installed alongside the sample applications in the following directory:

```
/opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/model/organs/...
```

**Note:** The inference model that is included with the DeepStream and Endoscopy sample applications was trained using the NVIDIA TAO Toolkit. This toolkit uses transfer learning to accelerate development time by enabling developers to fine-tune on high-quality NVIDIA pre-trained models with only a fraction of the data as training from scratch. For more information, see the NVIDIA TAO Toolkit webpage.
10.1 Clara AGX Developer Kit and Jetson AGX

The DeepStream and Endoscopy sample apps work in either iGPU or dGPU mode on Clara AGX Developer Kit; however, the TensorRT engine files that are included with the sample apps are only compatible with the dGPU. Running the apps with the iGPU requires new TensorRT engine files to be generated. To do so, follow instructions given under Rebuilding TensorRT Engines.

**Important:** Camera modules with CSI connectors are not supported in dGPU mode on Clara AGX Developer Kit.

### 10.1.1 Endoscopy Sample Application

Follow these steps to run the Endoscopy Sample Application:

1. Navigate to the installation directory:

   ```
   $ cd /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/
   .../build/EndoscopyJetsonAGXSample
   ```

2. Run the application:

   ```
   $ ./EndoscopyJetsonAGXSample
   ```

   The application window should appear.

   - All connected cameras should be visible on the top right.
   - Select a source by clicking one of the buttons named "Scope <N>" , where <N> corresponds to the video source.
• The main window area should show “Loading...” on the bottom while the camera is being initialized. When ready, the main window area will start displaying the live video stream from the camera.

• Click any of the "Class Control" buttons to dynamically enable or disable the bounding boxes and labels of detected objects.

• Click the button on the bottom right to quit the application.

10.1.2 DeepStream Sample Application

Follow these steps to run the DeepStream Sample Application:

1. Navigate to the installation directory:

   ```
   $ cd /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/build/DeepstreamJetsonAGXSample
   ```

2. Run the application:

   ```
   $ ./DeepstreamJetsonAGXSample
   ```

3. The application window should appear.
• All connected cameras should be visible on the right.

• Use Enable to select which camera streams to process. This only works before you play the stream the first time after starting the application.

• If desired, check the Enable Streaming box on the bottom to stream the processed output as a raw h.264 UDP stream. Once this box is checked, you can define the Host Port and Host IP address of the streaming destination (see step 4, below).

• Click the green arrow to start video capture and object detection.
Important: If streaming video from multiple camera sensors at the same time, ensure that they have compatible capture mode. To check capture modes, use:

$ v4l2-ctl -d /dev/video<N> --list-formats-ext

where N corresponds to the video source.

4. If streaming was enabled above, the following command can be run on the destination host in order to receive the video stream output by the application:

$ gst-launch-1.0 udpsrc port=5000 ! application/x-rtp,encoding-name=H264, payload=96 ! rtph264depay ! queue ! avdec_h264 ! autovideosink

10.2 Linux x86 DeepStream Sample

The Holoscan DeepStream sample application can also be run on a Linux x86 host using one or more identical USB cameras. To avoid bandwidth issues, ensure that each USB camera is connected to its own USB controller. Depending on the motherboard in use, it is typically sufficient to connect one camera to the front and one to the back of the PC.

Note: Unlike with Tegra, it is difficult to predict the exact version of the GPU and software running on your PC, so it is likely that the engine files will need to be rebuilt for your machine. If errors are encountered when streaming of the applications is started, see Rebuilding TensorRT Engines.

1. Navigate to the installation directory:

$ cd /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/build/DeepstreamLinuxHostSample

2. Run the application:

$ ./DeepstreamLinuxHostSample

The application window should appear. See Step 3 of the DeepStream Sample Application section above for more details about the application.

10.3 Rebuilding TensorRT Engines

The DeepStream sample application includes a sample model and prebuilt engine files that are compatible with the Clara AGX dGPU. These engine files are specific to the platform, GPU, and TensorRT version that is being used, and so these engine files must be rebuilt if any of these change. The following shows an example error message that may be seen if the engine files are not compatible with your current platform.
ERROR: [TRT]: coreReadArchive.cpp (38) - Serialization Error in verifyHeader:
  -> 0 (Version tag does not match)
ERROR: [TRT]: INVALID_STATE: std::exception
ERROR: [TRT]: INVALID_CONFIG: Deserialize the cuda engine failed.
ERROR: Deserialize engine failed from file: /opt/nvidia/clara-holoscan-sdk/
  clara-holoscan-deepstream-sample/build/DeepstreamJetsonAGXSample/model/
  resnet18_detector_fp16.trt
0:00:30.9554959 10922 0x559324d190 WARN nvinfer gstnvinfer.cpp:616:gst_nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]:
  Warning from NvDsInferContextImpl::deserializeEngineAndBackend() <nvsinfer_context_impl.cpp:1793> [UID = 1]: deserialize engine from file :
0:00:30.9559243 10922 0x559324d190 WARN nvinfer gstnvinfer.cpp:616:gst_nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]:
  Warning from NvDsInferContextImpl::generateBackendContext() <nvsinfer_context_impl.cpp:1900> [UID = 1]: deserialize backend context from engine
  from file :
  /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/build/DeepstreamJetsonAGXSample/model/resnet18_detector_fp16.trt failed,
  try rebuild
0:00:30.9562261 10922 0x559324d190 INFO nvinfer gstnvinfer.cpp:619:gst_nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]: Info
  from NvDsInferContextImpl::buildModel() <nvsinfer_context_impl.cpp:1818> [UID = 1]: Trying to create engine from model files
ERROR: failed to build network since there is no model file matched.
ERROR: failed to build network.
0:00:30.9562451 10922 0x559324d190 ERROR nvinfer gstnvinfer.cpp:613:gst_nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]: Error
  in NvDsInferContextImpl::buildModel() <nvsinfer_context_impl.cpp:1838> [UID = 1]: build engine file failed
0:00:30.9565763 10922 0x559324d190 ERROR nvinfer gstnvinfer.cpp:613:gst_nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]: Error
  in NvDsInferContextImpl::generateBackendContext() <nvsinfer_context_impl.cpp:1924> [UID = 1]: build backend context failed
0:00:30.9568128 10922 0x559324d190 ERROR nvinfer gstnvinfer.cpp:613:gst_nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]: Error
  in NvDsInferContextImpl::initialize() <nvsinfer_context_impl.cpp:1243> [UID = 1]: generate backend failed, check config file settings
0:00:30.9568516 10922 0x559324d190 WARN nvinfer gstnvinfer.cpp:816:gst_nvinfer_start:<primary-nvinference-engine> error: Failed to create
  NvDsInferContext instance
0:00:30.9569183 10922 0x559324d190 WARN nvinfer gstnvinfer.cpp:816:gst_nvinfer_start:<primary-nvinference-engine> error: Config file path: dsjas_nvinfer_config.txt, NvDsInfer Error: NVDSINFERENCE_CONFIG_FAILED
Playing...
ERROR from element primary-nvinference-engine: Failed to create
  NvDsInferContext instance
Error details: /dvs/git/dirty/git-master_linux/deepstream/sdk/src/gst-plugins/
  gst-nvinfer/gstnvinfer.cpp(816): gst_nvinfer_start (): /GstPipeline:camera-
  player/GstBin:Inference Bin/GstNvInfer:primary-nvinference-engine:

(continues on next page)
Config file path: dsjas_nvinfer_config.txt, NvDsInfer Error: NVDSINFER_CONFIG_→FAILED
Returned, stopping playback

If this is the case, follow these steps in order to rebuild the engine files on the device that will be running the DeepStream application.

1. Download the TensorRT tlt-converter binary for your platform:
   - Jetson and Clara AGX with iGPU or dGPU (TensorRT 7.2):
     https://developer.nvidia.com/tlt-converter-
   - Linux x86 (TensorRT 7.2):

2. Make the binary executable and move it to a location accessible through $PATH (e.g. /usr/bin):

   ```
   $ chmod a+x tlt-converter
   $ sudo mv tlt-converter /usr/bin
   ```

3. Navigate to the model directory of the DeepStream sample:

   ```
   $ cd /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/
   →model/organs
   ```

4. Run the detectnet_convert.sh script to generate the engine files:

   ```
   $ sudo ./detectnet_convert.sh
   ```

5. The previous step creates the TensorRT engine files (*.trt) and outputs them alongside the model files in the current directory. To use these new engines, copy them to the DeepStream application’s model directory, e.g.:

   ```
   $ sudo cp resnet18_detector_* .trt /opt/nvidia/clara-holoscan-sdk/clara-
   →holoscan-deepstream-sample/build/DeepstreamLinuxHostSample/model
   ```

### 10.4 Using Docker for the DeepStream or Endoscopy Sample Apps

**Note:** Running DeepStream applications using Docker is currently only supported when using the iGPU. The following section only applies to an iGPU configuration.

Docker provides lightweight reusable containers for deploying and sharing your applications.

Due to the way containers are used with Jetson, with host-side libraries mounted into the container in order to force library compatibility with the host drivers, there are two ways to build a container image containing the Holoscan samples:

1. Build the sample applications on the Jetson, outside of a container, then copy the executables into an image for use in a container.

2. Build the sample applications within a container.
These instructions document the process for #2, since it should be relatively straightforward to adapt the instructions to use method #1.

10.4.1 Configure the Container Runtime

The DeepStream libraries are not mounted into containers by default, and so an additional configuration file must be added to tell the container runtime to mount these components into containers.

1. Add the following to a new file, `/etc/nvidia-container-runtime/host-files-for-container.d/deepstream.csv`:

   ```
   dir, /opt/nvidia/deepstream/deepstream-6.0
   sym, /opt/nvidia/deepstream/deepstream
   ```

2. Configure Docker to default to use nvidia-docker with "default-runtime": "nvidia" in `/etc/docker/daemon.json`:

   ```
   {
   "runtimes": {
       "nvidia": {
           "path": "nvidia-container-runtime",
           "runtimeArgs": []
       }
   },
   "default-runtime": "nvidia"
   }
   ```

3. Restart the Docker daemon:

   ```
   $ sudo systemctl daemon-reload
   $ sudo systemctl restart docker
   ```

10.4.2 Build the Container Image

1. Follow steps 1 through 4 in Rebuilding TensorRT Engines to build the engine files for the iGPU, then copy these engines to the `model/organs/JetsonAGX` directory:

   ```
   $ sudo cp /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/model/organs/*.trt
   /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/model/organs/JetsonAGX
   ```

2. Build the clara-holoscan-samples Docker image:

   ```
   $ cd /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample
   $ sudo docker build -t clara-holoscan-samples -f Dockerfile.igpu .
   ```
10.4.3 Running the Samples in a Container

Since the samples are graphical applications and need permissions to access the X server, the DISPLAY environment variable must be set to the ID of the X11 display you are using and root (the user running the container) must have permissions to access the X server:

```bash
$ export DISPLAY=:0
$ xhost +si:localuser:root
```

Then, using the clara-holoscan-samples image that was previously built, the samples can be run using the following:

```bash
$ sudo docker run --rm \
  --device /dev/video0:/dev/video0 \
  --env DISPLAY=$DISPLAY --volume /tmp/.X11-unix:/tmp/.X11-unix \
  --network host --volume /tmp:/tmp \
  --workdir /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/ \
  build/DeepstreamJetsonAGXSample clara-holoscan-samples ./DeepstreamJetsonAGXSample
```

If you have no /dev/video0 device this means you do not have a camera attached. Please see AGX Camera Setup.

The various arguments to docker run do the following:

- Give the container access to the camera device /dev/video0:
  ```bash
  --device /dev/video0:/dev/video0
  ```

- Give the container access to the X desktop:
  ```bash
  --env DISPLAY=$DISPLAY --volume /tmp/.X11-unix:/tmp/.X11-unix
  ```

- Give the container access to the Argus socket used by the Camera API:
  ```bash
  --network host --volume /tmp:/tmp
  ```

- Run the DeepstreamJetsonAGXSample from the clara-holoscan-samples image:
  ```bash
  --workdir /opt/nvidia/clara-holoscan-sdk/clara-holoscan-deepstream-sample/build/DeepstreamJetsonAGXSample clara-holoscan-samples ./DeepstreamJetsonAGXSample
  ```
Chapter 11

DeepStream GPU Video Source Plugin

DeepStream SDK is based on the GStreamer framework, and provides a collection of proprietary GStreamer plugins that provide various GPU accelerated features for optimizing video analytic applications.

While DeepStream provides many plugins to meet the needs of a wide variety of use cases, it may be the case that a new plugin may be required in order to allow DeepStream integration with custom data input, output, or processing algorithms. In the case of Clara Holoscan, which provides high performance edge computing platforms for medical video analytics applications, one of the most common requirements will be the ability to capture video data from an external hardware device such that the data can be efficiently processed on the GPU using DeepStream or some other proprietary algorithms. For example, an ultrasound device may need to input raw data directly into NVIDIA GPU memory so that it can be beamformed on the GPU before being passed through a model in DeepStream for AI inference and annotation.

Due to the proprietary nature of third party hardware devices or processing algorithms it is not possible for NVIDIA to provide a generic GStreamer plugin that will work with all third party products, and it will generally be the partner’s responsibility to develop their own plugins. That being said, we also recognize that writing GStreamer plugins may be difficult for developers without any GStreamer experience, and writing these plugins such that they can work with GPU buffers and integrate with DeepStream complicates this process even further.

To help partners get started with writing GPU and DeepStream-enabled GStreamer plugins, the Clara Holoscan SDK includes a GPU Video Test Source GStreamer plugin, `nvvideotestsrc`. This sample plugin produces a stream of GPU buffers that are allocated, filled with a test pattern using CUDA, and then output by the plugin such that they can be consumed by downstream DeepStream components. When consumed only by other DeepStream or GPU-enabled plugins, this means that the buffers can go from end-to-end directly in GPU memory, avoiding any copies to/from system memory throughout the pipeline.

11.1 Running the GPU Video Test Source

The DeepStream GPU Video Test Source Plugin is automatically installed with the Clara Holoscan SDK. To test this plugin, the following command can be used.

```bash
$ gst-launch-1.0 nvvideotestsrc ! nveglglessink
```
This command, using the default `nvvideotestsrc` plugin parameters, will bring up a window showing an SMPTE color bars test pattern.

![Fig. 11.1: nvvideotestsrc plugin SMPTE color bars pattern](image)

Additional test patterns are available, some of which include animation. The pattern that is generated by the plugin can be configured using the `pattern` parameter. For example, to generate a Mandelbrot set pattern, which animates to zoom in and out of the image, the following command can be used.

```bash
$ gst-launch-1.0 nvvideotestsrc pattern=mandelbrot ! nveglglessink
```

Unless a downstream component specifically requests a specific video resolution or framerate, the plugin will default to a 1280x720 stream at 60FPS. To request a specific resolution or framerate, the GStreamer caps can also be provided to the `gst-launch-1.0` command between the `nvvideotestsrc` and downstream plugin. For example, the following command will tell the plugin to generate a 1920x1080 stream at 30FPS.

```bash
$ gst-launch-1.0 nvvideotestsrc ! 'video/x-raw(memory:NVMM), format=NV12, width=1920, height=1080, framerate=30/1' ! nveglglessink
```

**Note:** If any of these commands fail, make sure that the DISPLAY environment variable is set to the ID of the X11 display you are using, e.g.: `$ export DISPLAY=:0`
11.2 Accessing the GPU Video Test Source Code

The source code for the GPU Video Test Source plugin is installed to the following path.

```
/opt/nvidia/clara-holoscan-sdk/clara-holoscan-gstnvvideotestsrc
```

The plugin is compiled and installed from this location during the Clara Holoscan SDK installation. If any changes to the plugin source code are made, the plugin can be recompiled and installed with the following.

```
$ cd /opt/nvidia/clara-holoscan-sdk/clara-holoscan-gstnvvideotestsrc/build
$ sudo make && sudo make install
```

11.3 Custom GPU Source with RDMA Support

A primary use for the Clara Holoscan SDK is to develop edge devices that captures video frames from an external device into GPU memory for further processing and analytics. In this case, it is recommended that the device drivers for the external device take advantage of GPUDirect RDMA such that data from the device is copied directly into GPU memory so that it doesn’t need to first pass through system memory. Assuming the device drivers support RDMA, a GStreamer plugin can then use this RDMA support to accelerate DeepStream and GPU-enabled pipelines.

There are a number of resources that can be helpful for developers to add GPUDirect RDMA support to their DeepStream and GPU-enabled Clara Holoscan products.
1. The **GPUDirect RDMA Documentation**.

2. The **jetson-rdma-picoevb** GPUDirect RDMA sample drivers and applications.
   
   This sample provides a minimal hardware-based demonstration of GPUDirect RDMA including both the kernel drivers and userspace applications required to interface with an inexpensive and off-the-shelf FPGA board (PicoEVB). This is a good starting point for learning how to add RDMA support to your device drivers.

3. The **nvvideotestsrc** plugin. The source code for this plugin provides some basic placeholders and documentation that help point developers in the right direction when wanting to add RDMA support to a GStreamer source plugin. Specifically, the `gst_nv_video_test_src_fill` method – which is the GStreamer callback responsible for filling the GPU buffers – contains the following.

   ```c
   gst_buffer_map(buffer, &map, GST_MAP_READWRITE);
   // The memory of a GstBuffer allocated by an NvDsBufferPool contains
   // the NvBufSurface descriptor which then describes the actual GPU
   // buffer allocation(s) in its surfaceList.
   NvBufSurface *surf = (NvBufSurface*)map.data;
   // Use CUDA to fill the GPU buffer with a test pattern.
   //
   // NOTE: In this source, we currently only fill the GPU buffer using CUDA.
   // This source could be modified to fill the buffer instead with
   // other
   // mechanisms such as mapped CPU writes or RDMA transfers:
   //
   // 1) To use mapped CPU writes, the GPU buffer could be mapped into
   // the CPU address space using NvBufSurfaceMap.
   //
   // 2) To use RDMA transfers from another hardware device, the GPU
   // address for the buffer(s) (i.e. the `dataPtr` member of the
   // NvBufSurfaceParams) could be passed here to the device driver
   // that is responsible for performing the RDMA transfer into the
   // buffer. Details on how RDMA to NVIDIA GPUs can be performed
   // by
   // device drivers is provided in a demonstration application
   //
   // For more details on the NvBufSurface API, see nvbufsurface.h
   //
   gst_nv_video_test_src_cuda_prepare(self, surf->surfaceList);
   self->cuda_fill_image(self);
   // Set the numFilled field of the surface with the number of surfaces
   // that have been filled (always 1 in this example).
   // This metadata is required by other downstream DeepStream plugins.
   surf->numFilled = 1;
   gst_buffer_unmap(buffer, &map);
   ```

4. The **AJA Video Systems NTV2 SDK** and **GStreamer plugin**.
The use of AJA Video Systems hardware with Clara Holoscan platforms is documented in the AJA Video Systems section. This setup requires the installation of the AJA NTV2 SDK and Drivers and the AJA GStreamer plugin. These components are both open source and provide real world examples of video capture device drivers and a GStreamer plugin that use GPU Direct RDMA, respectively.

In the case of the device driver support within the NTV2 SDK, all of the RDMA support is enabled at compile time using the AJA_RDMA flag, and so the RDMA specifics within the driver can be located using the #ifdef AJA_RDMA directives within the source code.

In the case of the GStreamer plugin, the RDMA support was added to the plugin by this change: Add NVMM RDMA support for NVIDIA GPU buffer output.
Chapter 12

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 amount of latency added between the 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.

12.1 Building

The Holoscan latency tool source code is extracted to the target device during the SDK Manager installation of the Holoscan SDK, but the tool is not built automatically due to the optional AJA Video Systems support. The source code for the tool can be found in the following path:

```
/opt/nvidia/clara-holoscan-sdk/clara-holoscan-latency-tool_1.0.0
```

12.1.1 Requirements

The latency tool currently requires the use of a Clara AGX Developer Kit in dGPU mode. Since the RTX6000 GPU only has DisplayPort connectors, testing the latency of any of the HDMI modes that output from the GPU will require 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”.

The following additional software components are required:

- CUDA 11.1 or newer (https://developer.nvidia.com/cuda-toolkit)
• DeepStream 5.1 or newer (https://developer.nvidia.com/deepstream-sdk)
• 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 required for the optional AJA Video Systems Support:

• An AJA Video Systems SDI or HDMI capture device
• AJA NTV2 SDK 16.1 or newer (See AJA Video Systems for details on installing the AJA NTV2 SDK and drivers).

**Note:** Only the KONA HDMI and Corvid 44 12G BNC cards have currently been verified to work with the latency tool.

### 12.1.2 Installing Additional Requirements

The following steps assume that CUDA and DeepStream have already been installed by the Clara Holoscan SDK. Only the additional packages will be installed here.

```bash
$ sudo apt-get install -y \
cmake \ 
libglfw3-dev \ 
libgstreamer1.0-dev \ 
libgstreamer-plugins-base1.0-dev \ 
libgtk-3-dev \ 
pkg-config
```

### 12.1.3 Building Without AJA Support

```bash
$ cd /opt/nvidia/clara-holoscan-sdk/clara-holoscan-latency-tool_1.0.0
$ sudo mkdir build
$ sudo chown `whoami` build
$ cd build
$ cmake ..
$ make -j
```

The above will build the project using CMake and will output the holoscan-latency binary to the current build directory.

**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:
12.1.4 Building With AJA Support

When building with AJA support, the NTV2_SDK path must point to the location of the NTV2 SDK in which both the headers and compiled libraries (i.e. libajatv2) exist. For example, if the NTV2 SDK is in /home/nvidia/ntv2sdklinux_16.1.0.3 then the following is used to build the latency tool with AJA support enabled:

```
$ cd /opt/nvidia/clara-holoscan-sdk/clara-holoscan-latency-tool_1.0.0
$ sudo mkdir build
$ sudo chown `whoami` build
$ cd build
$ cmake -DNTV2_SDK=/home/nvidia/ntv2sdklinux_16.1.0.3 ..
$ make -j
```

The above will build the project using CMake and will output the holoscan-latency binary to the current build directory.

**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
```

12.2 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.

12.2.1 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.
12.2. Operation Overview

Fig. 12.1: HDMI Loopback Between GPU and HDMI Capture Card

Fig. 12.2: Latency Tool Frame Lifespan (RDMA Disabled)
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 (RDMA Enabled)](image)

Fig. 12.3: Latency Tool Frame Lifespan (RDMA Enabled)

### 12.2.2 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.

```shell
$ ./holoscan-latency -p aja -p.rdma 0 -c aja -c.rdma 0 -f 4k
```
Clara Holoscan SDK User Guide

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)

---

![Real Application Frame Lifespan](image)

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)

![Fig. 12.5: 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)

![Final Estimated Latencies (Processing + Vsync + Wire)](image)

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).

12.2. Operation Overview 58
12.2.3 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)](image)

![RDMA Enabled (2 Frames)](image)

Fig. 12.6: Reducing Latency With RDMA

The following shows the above example repeated with RDMA enabled.

```
$ ./holoscan-latency -p aja -p.rdma 1 -c aja -c.rdma 1 -f 4k
```
12.2.4 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 

```scalable北京赛车```

-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 

```scalable北京赛车```

-option, adjusting the 

```scalable北京赛车```

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.

```bash
$ ./holoscan-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 

```scalable北京赛车```

-option using the loop count that was determined in the previous step.

```
$ ./holoscan-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.
Format: 1920x1080 RGBA @ 60Hz

Producer: AJA
  Device: 0
  Channel: NTV2_CHANNEL1
  ROMA: 1

Consumer: AJA
  Device: 0
  Channel: NTV2_CHANNEL2
  ROMA: 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 = 59, min = 34, max = 116
Copy To Host: avg = 0, min = 0, max = 19
Write To HW: avg = 1785, min = 1721, max = 1849
Vsycn Wait: avg = 14321, min = 13782, max = 14718
Wire Time: avg = 16723, min = 16360, max = 33470
Road 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 = 19465
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 = 29493, min = 20191, max = 20967
Frames: avg = 1.23, min = 1.21, max = 1.26

Final Estimated Latencies (Processing + Vsync + Wire)

Microseconds: avg = 49998, min = 49998, max = 49998
Frames: avg = 3, min = 3, max = 3

WARNING: Frames were skipped or repeated. These times only include frames that were actually received, and the times include only the first instance each frame was received.
Tip: To get the most accurate estimation of the latency that would be seen by a real world application, the best thing to do would be to run the actual frame processing algorithm used by the application during the latency measurement. This could be done by modifying the SimulateProcessing function in the latency tool source code.

12.3 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:

```
$ ./holoscan-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.
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.
12.4 Producers

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

12.4.1 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.

### 12.4.2 GStreamer GPU Rendering (HDMI)

This producer (gst) uses the `nveglglessink` GStreamer component that is included with DeepStream in order to render frames that originate from a GStreamer pipeline to the HDMI connectors on the GPU.

GStreamer 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 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.

### 12.4.3 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).
12.5 Consumers

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

12.5.1 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.

12.5.2 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.

12.5.3 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).

### 12.6 Example Configurations

**Note:** When testing a configuration that outputs from the GPU, the tool currently only supports a displayless 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**:

   ![User Accounts](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*).

### 12.6.1 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:

```
$ ./holoscan-latency -p gl -c v4l2
```

### 12.6.2 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:

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

### 12.6.3 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.

```
$ ./holoscan-latency -p aja -c aja
```
Fig. 12.7: DP-to-HDMI Cable Between GPU and Onboard HDMI Capture Card
12.6. Example Configurations

Fig. 12.8: DP-to-HDMI Cable Between GPU and AJA KONA HDMI Capture Card (Channel 1)

Fig. 12.9: SDI Cable Between Channel 1 and 2 of a Single AJA Corvid 44 Capture Card
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:

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

### 12.7 Troubleshooting

If any of the `holoscan-latency` commands described above fail with errors, the following steps may help resolve the issue.

1. **Problem:** The following error is output:

   ```
   ERROR: Failed to get a handle to the display (is the DISPLAY environment variable set?)
   ```

   **Solution:** Ensure that the `DISPLAY` environment variable is set with the ID of the X11 display you are using; e.g. for display ID 0:

   ```
   $ export DISPLAY=:0
   ```

   If the error persists, try changing the display ID; e.g. replacing 0 with 1:

   ```
   $ export DISPLAY=:1
   ```

   It might also be convenient to set this variable in your `~/.bashrc` file so that it is set automatically whenever you login.

2. **Problem:** An error like the following is output:

   ```
   ERROR: The requested format (1920x1080 @ 60Hz) does not match the current display mode (1024x768 @ 60Hz)
   Please set the display mode with the xrandr tool using the following command:
   $$ xrandr --output DP-5 --mode 1920x1080 --panning 1920x1080 --rate 60
   ```

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

   ```
   $ xrandr --output DP-5 --mode 1920x1080 --panning 1920x1080 --rate 60
   xrandr: cannot find mode 1920x1080
   ```

   **Solution:** Try the following:

   1. Ensure that no other displays are connected to the GPU.

   2. Check the output of an `xrandr` command to see that the requested format is supported. The following shows an example of what the onboard HDMI capture card should support. Note that each row of the supported modes shows the resolution on the left followed by all of the supported frame rates for that resolution to the right.
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 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)
```

(continues on next page)
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 13

NGC Containers

In addition to the samples and packages that are installed locally as part of the Clara Holoscan SDK, containerized samples are also provided via the NVIDIA GPU Cloud (NGC).

In order to access the Clara Holoscan containers, you must first create an account and login to NGC via https://ngc.nvidia.com/signin. Once logged into NGC, the Catalog will provide access to all of the NVIDIA-provided containers, models, and other resources. In order to narrow this down to display just the containers provided as part of Clara Holoscan, the Clara Holoscan label can be used as a search query by typing label: Clara Holoscan into the search bar, or by using the following link:

Clara Holoscan Containers on NGC

To pull these Docker images to your system, generate an API key and set up the NGC CLI via the instructions at https://ngc.nvidia.com/setup.

The Clara Holoscan containers that are posted to NGC may be updated separately from the Clara Holoscan SDK releases, and so these containers may not be documented here. Please refer to the description page for the individual containers on NGC for any additional documentation related to these containers.
The NGC catalog hosts containers for the top AI and data science software, trained, tested and optimized by NVIDIA, as well as fully tested containers for HPC applications and data analytics. NGC catalog containers provide powerful and easy-to-deploy software proven to deliver the fastest results, allowing users to build solutions from a tested framework, with complete control.