



# **Clara Holoscan SDK User Guide**

*Release 0.3.0*

**NVIDIA Corporation**

**Oct 20, 2022**



# INTRODUCTION

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	Content . . . . .	1
1.1.1	Extensions . . . . .	1
1.1.2	Applications . . . . .	1
1.1.3	Video Pipeline Latency Tool . . . . .	2
1.2	Changes Since Holoscan SDK 0.2.0 . . . . .	2
1.2.1	Holoscan C++ API . . . . .	3
<b>2</b>	<b>Software Stack Installation</b>	<b>5</b>
2.1	Development Software Stack with Holopack on Clara Developer Kits . . . . .	5
2.2	Development Software Stack on x86 . . . . .	5
2.3	Deployment Software Stack with OpenEmbedded on Clara Developer Kits . . . . .	5
<b>3</b>	<b>Install and Use the Clara Holoscan SDK</b>	<b>7</b>
3.1	Using the container from NGC . . . . .	7
3.2	From source . . . . .	7
<b>4</b>	<b>Third Party Hardware Setup</b>	<b>9</b>
4.1	AJA Video Systems . . . . .	9
4.1.1	Installing the AJA Hardware . . . . .	9
4.1.2	Installing the AJA Software . . . . .	11
4.1.3	Using AJA Devices in Containers . . . . .	13
4.1.4	Troubleshooting . . . . .	13
4.2	Emergent Vision Technologies (EVT) . . . . .	14
4.2.1	Installing EVT Hardware . . . . .	14
4.2.2	Installing EVT Software . . . . .	15
4.2.3	Post EVT Software Installation Steps . . . . .	15
4.2.4	Testing the EVT Camera . . . . .	16
4.2.5	Troubleshooting . . . . .	16
<b>5</b>	<b>Clara Holoscan Development Guide</b>	<b>17</b>
5.1	Holoscan Core Concepts . . . . .	17
5.2	Getting Started . . . . .	18
5.2.1	Code Example . . . . .	19
5.2.2	Build and Run the Application . . . . .	22
5.3	Developing Holoscan GXF Extensions . . . . .	23
5.3.1	Extension Lifecycle . . . . .	23
5.3.2	Implementing an Extension . . . . .	24
5.4	Wrapping a GXF Codelet as a Holoscan Operator (C++ API) . . . . .	30
5.5	Creating the Holoscan Application (C++ API) . . . . .	35

5.6	Running the Holoscan MyRecorder Application (C++ API)	39
<b>6</b>	<b>Clara Holoscan Sample Applications</b>	<b>43</b>
6.1	Endoscopy Tool Tracking Application	43
6.1.1	Input source: Video Stream Replayer	43
6.1.2	Input source: AJA	45
6.2	Hi-Speed Endoscopy Application	47
6.2.1	Enable G-SYNC for Display	48
6.2.2	Installing and Enabling GPUDirect RDMA	48
6.2.3	Enabling Exclusive Display Mode	51
6.3	Ultrasound Segmentation Application & Customization	53
6.3.1	Input source: Video Stream Replayer	53
6.3.2	Input source: AJA	54
6.3.3	Bring Your Own Model (BYOM) - Customizing the Ultrasound Segmentation Application For Your Model	55
<b>7</b>	<b>Clara Holoscan GXF Extensions</b>	<b>59</b>
7.1	GXF Built-in Extensions	59
7.1.1	Std	59
7.1.2	Serialization	59
7.2	Holoscan SDK GXF Extensions	60
7.2.1	V4L2	60
7.2.2	AJA	61
7.2.3	Stream Playback	61
7.2.4	Format Converter	63
7.2.5	TensorRT	65
7.2.6	OpenGL	66
7.2.7	Segmentation Post Processor	67
7.2.8	Segmentation Visualizer	68
7.2.9	Custom LSTM Inference	68
7.2.10	Visualizer Tool Tracking	70
7.2.11	Holoscan Test Mock	72
7.2.12	Emergent	73
7.2.13	Bayer Demosaic	73
7.2.14	Holoviz Viewer	74
<b>8</b>	<b>Clara Holoviz</b>	<b>77</b>
8.1	Overview	77
8.2	Concepts	77
8.3	Usage	77
8.4	Layers	78
<b>9</b>	<b>Video Pipeline Latency Tool</b>	<b>79</b>
9.1	Requirements	79
9.1.1	Hardware	79
9.1.2	Software	80
9.2	Installation	80
9.2.1	Downloading the Source	80
9.2.2	Installing Software Requirements	80
9.2.3	Building	81
9.3	Example Configurations	81
9.3.1	GPU To Onboard HDMI Capture Card	83
9.3.2	GPU to AJA HDMI Capture Card	84
9.3.3	AJA SDI to AJA SDI	84
9.4	Operation Overview	85

9.4.1	Frame Measurements . . . . .	85
9.4.2	Interpreting The Results . . . . .	87
9.4.3	Reducing Latency With RMDA . . . . .	90
9.4.4	Simulating GPU Workload . . . . .	92
9.5	Graphing Results . . . . .	94
9.6	Producers . . . . .	98
9.6.1	OpenGL GPU Direct Rendering (HDMI) . . . . .	99
9.6.2	GStreamer GPU Rendering (HDMI) . . . . .	99
9.6.3	AJA Video Systems (SDI) . . . . .	99
9.7	Consumers . . . . .	100
9.7.1	V4L2 (Onboard HDMI Capture Card) . . . . .	100
9.7.2	GStreamer (Onboard HDMI Capture Card) . . . . .	100
9.7.3	AJA Video Systems (SDI and HDMI) . . . . .	100
9.8	Troubleshooting . . . . .	101
<b>10</b>	<b>NGC Containers</b>	<b>105</b>
10.1	ARM Container . . . . .	105
10.2	x86 Container . . . . .	105
<b>11</b>	<b>Relevant Technologies</b>	<b>107</b>
11.1	Graph Execution Framework (GXF) . . . . .	107
11.1.1	GXF Entities by Example . . . . .	108
11.1.2	Data Flow and Triggering Rules . . . . .	110
11.1.3	Creating the GXF Application Definition . . . . .	111
11.1.4	Running the GXF Recorder Application . . . . .	115
11.1.5	GXF User Guide . . . . .	116
11.2	Rivermax SDK . . . . .	184
11.2.1	Testing Rivermax and GPUDirect . . . . .	184
11.3	GPUDirect RDMA . . . . .	186
11.4	TensorRT Optimized Inference . . . . .	188
11.5	CUDA and OpenGL Interoperability . . . . .	188
11.6	Accelerated Image Transformations . . . . .	188



## OVERVIEW

NVIDIA Clara Holoscan is the AI computing platform for medical devices, consisting of Clara Developer Kits and the Clara Holoscan SDK. Clara Holoscan allows medical device developers to create the next-generation of AI-enabled medical devices.

The Clara Holoscan SDK version 0.3.0 provides the foundation to run streaming applications on Clara Developer Kits, enabling real-time AI inference and fast IO. These capabilities are showcased within the reference *extensions* and *applications* described below.

### 1.1 Content

#### 1.1.1 Extensions

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

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

#### 1.1.2 Applications

This SDK includes two core sample applications to show how users can implement their own end-to-end inference pipeline for streaming use cases, as well as an additional “bring your own model” (BYOM) segmentation ability which is modality agnostic. This guide provides detailed information on the *inner-workings of those applications*, and *how to create your own*.

See below for some information regarding the sample applications:

#### Endoscopy Tool Tracking

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

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

Nwoye, C.I., Mutter, D., Marescaux, J. and Padoy, N., 2019. Weakly supervised convolutional LSTM approach for tool tracking in laparoscopic videos. *International journal of computer assisted radiology and surgery*, 14(6), pp.1059-1067

Refer to the [sample data resource on NGC](#) for more information related to the model and video.

## Ultrasound Segmentation

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

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

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

Ungi *et al.*, “Automatic Spine Ultrasound Segmentation for Scoliosis Visualization and Measurement,” in *IEEE Transactions on Biomedical Engineering*, vol. 67, no. 11, pp. 3234-3241, Nov. 2020, doi: 10.1109/TBME.2020.2980540.

Refer to the [sample data resource on NGC](#) for more information related to the model and video.

## Colonoscopy Polyp Segmentation

As an example of the BYOM ability mentioned above, we show how the same code used for ultrasound segmentation may be used for a polyp segmentation application.

This model was trained on the Kvasir-SEG dataset<sup>1</sup>, using the [ColonSegNet model architecture](#)<sup>2</sup>.

Refer to the [sample data resource on NGC](#) for more information related to the model and video.

### 1.1.3 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 NVIDIA IGX Orin 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.

## 1.2 Changes Since Holoscan SDK 0.2.0

The following table outlines the component versions that have been upgraded or removed in version 0.3.0:

Component	Holoscan 0.3.0	Holoscan 0.2.0
<b>Jetpack</b>	<b>Holopack 1.1</b>	JP5 HP1
<b>GXF</b> <sup>3</sup>	<b>2.4.3</b>	2.4.2

---

<sup>1</sup> Jha, Debesh, Pia H. Smedsrud, Michael A. Riegler, Pål Halvorsen, Thomas de Lange, Dag Johansen, and Håvard D. Johansen, “Kvasir-seg: A segmented polyp dataset” Proceedings of the *International Conference on Multimedia Modeling*, pp. 451-462, 2020.

<sup>2</sup> Jha D, Ali S, Tomar NK, Johansen HD, Johansen D, Rittscher J, Riegler MA, Halvorsen P. Real-Time Polyp Detection, Localization and Segmentation in Colonoscopy Using Deep Learning. *IEEE Access*. 2021 Mar 4;9:40496-40510. doi: 10.1109/ACCESS.2021.3063716. PMID: 33747684; PMCID: PMC7968127.

<sup>3</sup> NVIDIA Graph eXecution Framework (GXF)



### 1.2.1 Holoscan C++ API

The most significant change in Holoscan 0.3.0 is the addition of a new C++ API for the creation of GXF extensions, giving developers an additional pathway to building their desired applications.

---



## SOFTWARE STACK INSTALLATION

The Clara Holoscan SDK requires a specific software stack to build and run. The SDK can either run on a **Development Stack** for Clara Developer Kits based on Holopack and for x86 Linux compute platform or on a **Deployment Stack** for Clara Developer Kits based on OpenEmbedded.

The Deployment Stack is recommended if you want to include just software components that are actually needed for your application. Furthermore, the runtime BSP can be easily optimized with respect to memory usage, speed, security and power requirements.

### 2.1 Development Software Stack with Holopack on Clara Developer Kits

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] (<https://docs.nvidia.com/sdk-manager/install-with-sdkm-clara>). Make sure you have joined the [Clara Holoscan SDK Program](#) and, if needed, the [RiverMax SDK Program](#) before using the NVIDIA SDK Manager.

For complete instructions on how to set up and flash your Clara Developer Kit, see the [Clara AGX Developer Kit User Guide](#) or [NVIDIA IGX Orin Developer Kit User Guide](#)

### 2.2 Development Software Stack on x86

Since v0.3.0, the Clara Holoscan SDK is also available for x86\_64 (amd64) Linux platform and has been exclusively tested on Ubuntu 20.04. Please refer to the [prerequisites](#) located in the [Clara Holoscan SDK source code](#)

### 2.3 Deployment Software Stack with OpenEmbedded on Clara Developer Kits

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

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

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

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

## INSTALL AND USE THE CLARA HOLOSCAN SDK

Once the software stack has been installed on your device, Clara Holoscan SDK can be installed from source or using a pre-built container runtime.

### 3.1 Using the container from NGC

The [Clara Holoscan Sample Applications](#) multi-arch container (for arm64 and amd64/x86\_64) is the simplest way to run the sample applications as it includes all necessary binaries and datasets, and allows for some customization of the application graph and its parameters.

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

---

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

---

### 3.2 From source

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

Refer to the top-level `README.md` in the open-source repository on Github for prerequisites, setup, and run instructions.



## THIRD PARTY HARDWARE SETUP

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

### 4.1 AJA Video Systems

AJA provides a wide range of proven, professional video I/O devices, and thanks to a partnership between NVIDIA and AJA, Clara Holoscan supports the AJA NTV2 SDK and device drivers as of the NTV2 SDK 16.1 release.

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 system to GPU copies are eliminated from the processing pipeline.

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

---

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

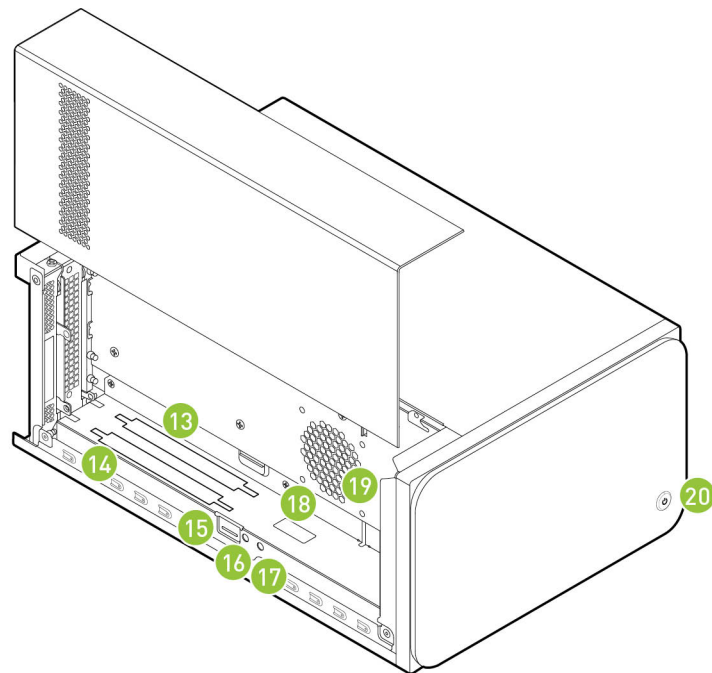
---

#### 4.1.1 Installing the AJA Hardware

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

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

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





### 4.1.2 Installing the AJA Software

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

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

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

---

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

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

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

- *Loading the AJA NTV2 Drivers*

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

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

### Downloading the AJA NTV2 SDK Source

Navigate to a directory where you would like the source code to be downloaded, then perform the following to clone the NTV2 SDK source code and checkout the correct branch as needed for Holoscan SDK.

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

---

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

---

## Building the AJA NTV2 Drivers

The following will build the AJA NTV2 drivers with RDMA support enabled. Once built, the kernel module (**ajantv2.ko**) and load/unload scripts (**load\_ajantv2** and **unload\_ajantv2**) will be output to the `${NTV2}/bin` directory.

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

## Loading the AJA NTV2 Drivers

Running any application that uses an AJA device requires the AJA kernel drivers to be loaded, even if the application is being run from within a container. The drivers must be manually loaded every time the machine is rebooted using the **load\_ajantv2** script:

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

---

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

---

## Building and Installing the AJA NTV2 SDK

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

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

## Testing the AJA Device

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

```

2. To ensure that RDMA support has been compiled into the AJA driver and is functioning correctly, the `testrdma` utility can be used:

```

$ testrdma -t500

test device 0 start 0 end 7 size 8388608 count 500

frames/errors 500/0

```

### 4.1.3 Using AJA Devices in Containers

Accessing an AJA device from a container requires the drivers to be loaded natively on the host (see *Loading the AJA NTV2 Drivers*), then the device that is created by the `load_ajantv2` script must be shared with the container using the `--device` docker argument. For example, when running the runtime container on NGC:

```

docker run -it --rm \
  --runtime=nvidia -e NVIDIA_DRIVER_CAPABILITIES=graphics,video,compute,
  ↪ utility \
  -e DISPLAY=${DISPLAY} -v /tmp/.X11-unix:/tmp/.X11-unix \
  --device /dev/ajantv20:/dev/ajantv20 \
  nvcr.io/nvidia/clara-holoscan/clara_holoscan_sample_runtime:v0.2.0-arm64

```

### 4.1.4 Troubleshooting

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

#### Solutions:

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

```

$ lspci
0000:00:00.0 PCI bridge: NVIDIA Corporation Device 1ad0 (rev a1)

```

(continues on next page)

(continued from previous page)

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

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

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

2. **Problem:** The `testrdma` command outputs the following error:

```
error - GPU buffer lock failed
```

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

## 4.2 Emergent Vision Technologies (EVT)

Thanks to the collaboration with [Emergent Vision Technologies](#), Clara Holoscan SDK now supports EVT high-speed cameras. Clara Holoscan SDK application has been developed and verified using [HB-9000-G-C: 25GigE camera with Gpixel GMAX2509](#)

**Note:** The addition of an EVT camera to the Clara Holoscan hardware platform is optional. The Holoscan SDK has an application that can be run with the EVT camera, but there are other applications that can be run without EVT camera.

### 4.2.1 Installing EVT Hardware

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

**Note:** The recommended length of SFP28 cable is no more than 2 meters due to signal integrity issues.

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

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

---

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

---

## 4.2.2 Installing EVT Software

The Emergent SDK needs to be installed in order to compile and run the Clara Holoscan applications with EVT camera. The latest tested version of Emergent SDK with Clara Holoscan SDK is eSDK 2.36.02 Linux Ubuntu 20.04.04 Kernel 5.10.65 JP 5.0 HP and can be downloaded from [here](#). The Emergent SDK comes with headers, libraries and examples. To install the SDK refer to the Software Installation section of EVT [Camera User's Manual](#). Users need to log in to find be able to download Camera User's Manual.

---

**Note:**

- The Emergent SDK depends on Rivermax SDK v1.11.11 and Mellanox OFED Network Drivers v5.7 which needs to be installed by SDK Manager.
- To avoid installing Rivermax SDK and Mellanox OFED Network Drivers as part of Emergent SDK, use following command.

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

---

## 4.2.3 Post EVT Software Installation Steps

After installation of the software, execute below steps to bring up the camera node on Clara devkit in dGPU mode.

1. Restart openibd to configure Mellanox device, if not already.

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

2. Find out the logical name of the ethernet interface being used to connect EVT camera to Mellanox CX NIC using below command.

```
sudo ibdev2netdev -v
```

An example of what output would look like is:

```
0007:03:00.0 mlx5_0 (MT4125 - MCX623106AN-CDAT) ConnectX-6 Dx EN adapter card, 100GbE,
↪Dual-port QSFP56, PCIe 4.0 x16, No Crypto fw 22.33.1048 port 1 (ACTIVE) ==> eth1 (Up)
0007:03:00.1 mlx5_1 (MT4125 - MCX623106AN-CDAT) ConnectX-6 Dx EN adapter card, 100GbE,
↪Dual-port QSFP56, PCIe 4.0 x16, No Crypto fw 22.33.1048 port 1 (DOWN ) ==> eth2 (Down)
```

In above example, the camera is connected to ACTIVE port eth1.

---

**Note:** The logical name of the ethernet interface can be anything and does not need to be eth1 as in above example.

---

3. Configure the NIC with IP address, if not already during the [Installing EVT hardware](#) step. The following command uses the logical name of the ethernet interface found in step 2.

```
sudo ifconfig eth1 down
sudo ifconfig eth1 192.168.1.100 mtu 9000
sudo ifconfig eth1 up
```

## 4.2.4 Testing the EVT Camera

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

## 4.2.5 Troubleshooting

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

**Solution:**

- Make sure that the ConnectX is configured with the correct IP address. Follow section [Post EVT Software Installation Steps](#)

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

**Solutions:**

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

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

**Solution:**

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

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

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

## CLARA HOLOSCAN DEVELOPMENT GUIDE

Welcome to the Holoscan SDK development guide!

Here you will learn *Holoscan core concepts* and *get started with the simple endoscopy application*.

Then, we will *walk you through the Graph eXecution Framework (GXF) extensions*, and how to use Holoscan C++ API to *wrap them as Holoscan operators*, *compose/build* and *run MyReco* application as an example.

### 5.1 Holoscan Core Concepts

Since Holoscan Embedded SDK version 0.3.0, we are introducing a new framework with C++ API for the creation of applications.

The Holoscan API provides an easier and more flexible way to create applications using *GXF's features*.

It is designed to be used as a drop-in replacement for the GXF's API and provides a common interface for GXF components.

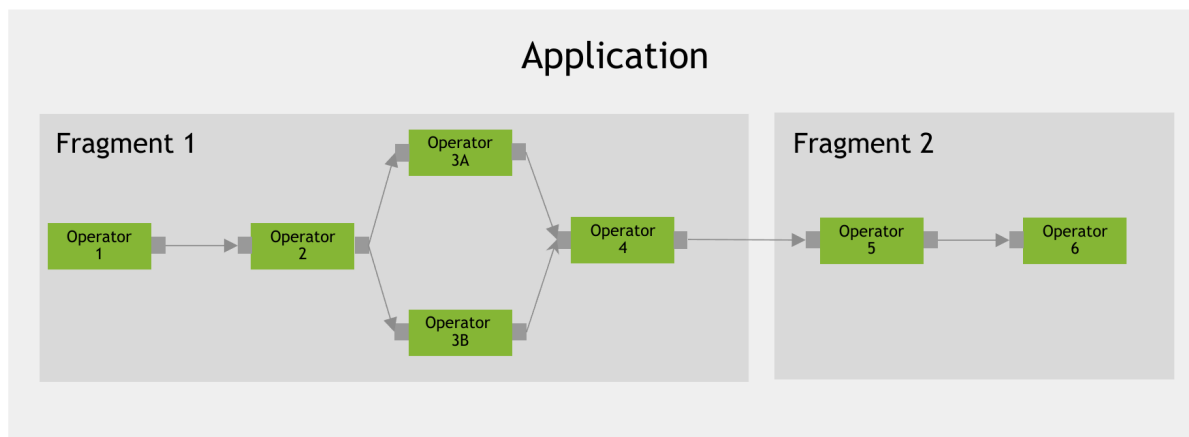


Fig. 5.1: Core concepts: Application

The core concepts of the Holoscan API are:

- **Application:** An application acquires and processes streaming data. An application is a collection of fragments where each fragment can be allocated to execute on a physical node of a Holoscan cluster.
- **Fragment:** A fragment is a building block of the Application. It is a *Directed Acyclic Graph (DAG)* of operators. A fragment can be assigned to a physical node of a Holoscan cluster during execution. The run-time execution

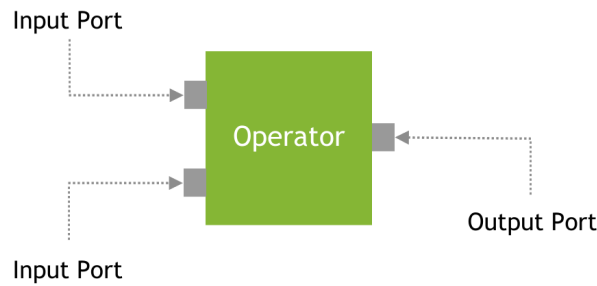


Fig. 5.2: Core concepts: Port

manages communication across fragments. In a Fragment, Operators (Graph Nodes) are connected to each other by flows (Graph Edges).

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

As of version 0.3.0, the Holoscan API provides a new convenient way to compose GXF Codelets as GXF Operators into Application workflows, without the need to write YAML files. The Holoscan API enables a more flexible/scalable approach to create applications.

## 5.2 Getting Started

Let's get started with the Holoscan SDK.

The following figure shows a workflow graph of the simple endoscopy tool tracking application that consists of a single fragment as an application.

The fragment consists of six operators that we provide as part of the Holoscan SDK. The operators are:

- **Video Stream Replayer:** This operator replays a video stream from a file. It is a GXF Operator (*VideoStreamReplayerOp*) that wraps a GXF Codelet (*VideoStreamReplayer*).
- **Visualizer Format Converter:** This operator converts the image format from RGB888 (24-bit pixel) to RGBA8888(32-bit pixel) for visualization for the Tool Tracking Visualizer. It is a GXF Operator (*FormatConverterOp*) that wraps a GXF Codelet (*FormatConverter*).



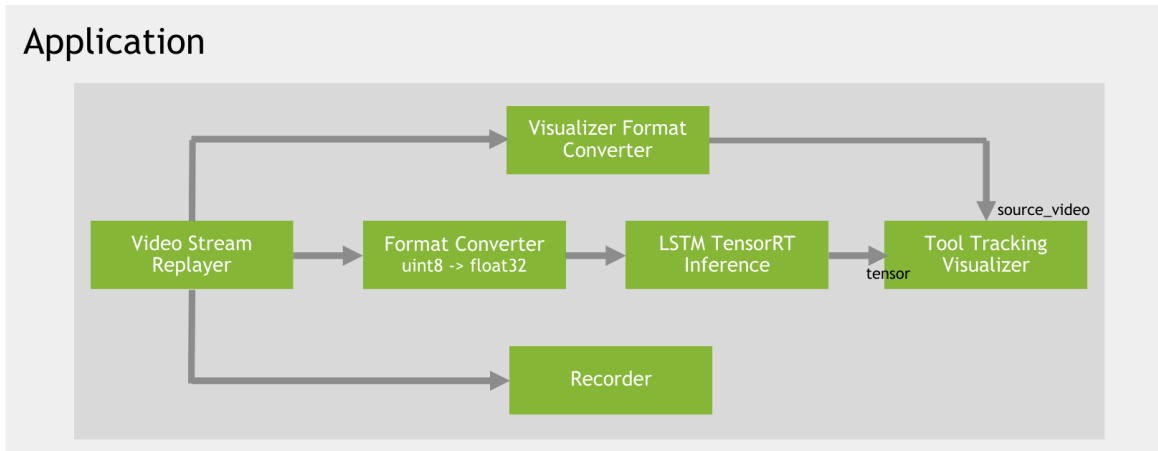


Fig. 5.3: Simple Endoscopy Workflow

- **Tool Tracking Visualizer:** This operator visualizes the tool tracking results. It is a GXF Operator (`ToolTrackingVizOp`) that wraps a GXF Codelet (`visualizer_tool_tracking::Sink`).
- **Format Converter:** This operator converts the data type of the image from `uint8` to `float32` for feeding into the tool tracking model. It is a GXF Operator (`FormatConverterOp`) that wraps a GXF Codelet (`FormatConverter`).
- **LSTM TensorRT Inference:** This operator performs the inference of the tool tracking model. It is a GXF Operator (`LSTMTensorRTInferenceOp`) that wraps a GXF Codelet (`custom_lstm_inference::TensorRtInference`).
- **Recorder:** This operator records the video stream to a file. It is a GXF Operator (`VideoStreamRecorderOp`) that wraps a GXF Codelet (`EntityRecorder`).

### 5.2.1 Code Example

Let's see how we can create the application by composing the operators.

The following code snippet shows how to create the application with Holoscan SDK's C++ API.

**Code Snippet:** `apps/experiments/simple/simple.cpp`

Listing 5.1: `apps/experiments/simple/simple.cpp`

```

18 #include <holoscan/holoscan.hpp>
19 #include <holoscan/std_ops.hpp>
20
21 class App : public holoscan::Application {
22 public:
23     void compose() override {
24         using namespace holoscan;
25
26         auto replayer = make_operator<ops::VideoStreamReplayerOp>("replayer", from_config(
27             ↪ "replayer"));
  
```

(continues on next page)

(continued from previous page)

```

28     auto recorder = make_operator<ops::VideoStreamRecorderOp>("recorder", from_config(
    ↪ "recorder"));
29
30     auto format_converter = make_operator<ops::FormatConverterOp>(
31         "format_converter",
32         from_config("format_converter_replayer"),
33         Arg("pool") = make_resource<BlockMemoryPool>("pool", 1, 4919041, 2));
34
35     auto lstm_inferer = make_operator<ops::LSTMTensorRTInferenceOp>(
36         "lstm_inferer",
37         from_config("lstm_inference"),
38         Arg("pool") = make_resource<UnboundedAllocator>("pool"),
39         Arg("cuda_stream_pool") = make_resource<CudaStreamPool>("cuda_stream", 0, 0, 0, ↪
    ↪ 1, 5));
40
41     auto visualizer_format_converter = make_operator<ops::FormatConverterOp>(
42         "visualizer_format_converter",
43         from_config("visualizer_format_converter_replayer"),
44         Arg("pool") = make_resource<BlockMemoryPool>("pool", 1, 6558720, 2));
45
46     auto visualizer = make_operator<ops::ToolTrackingVizOp>(
47         "visualizer",
48         from_config("visualizer"),
49         Arg("pool") = make_resource<UnboundedAllocator>("pool"));
50
51     // Flow definition
52     add_flow(replayer, visualizer_format_converter);
53     add_flow(visualizer_format_converter, visualizer, {"tensor", "source_video"});
54
55     add_flow(replayer, format_converter);
56     add_flow(format_converter, lstm_inferer);
57     add_flow(lstm_inferer, visualizer, {"tensor", "tensor"});
58
59     add_flow(replayer, recorder);
60 }
61 };
62
63 int main() {
64     App app;
65     app.config("apps/endoscopy_tool_tracking/app_config.yaml");
66     app.run();
67
68     return 0;
69 }

```

In `main()` method, we create an instance of the `App` class that inherits from `holoscan::Application`. The `App` class overrides the `compose()` function to define the application's flow graph. The `compose()` function is called by the `run()` function of the `holoscan::Application` class.

Before we call `run()`, we need to set the application configuration by calling the `config()` function.

The configuration file is a YAML file that contains the configuration of the operators and the application. The path to the configuration file is passed to the `config()` function as a string.

The configuration file for the simple application is located at `apps/endoscopy_tool_tracking/app_config.yaml`. Let's take a look at the configuration file.

**Code Snippet:** `apps/endoscopy_tool_tracking/app_config.yaml`

Listing 5.2: `apps/endoscopy_tool_tracking/app_config.yaml`

```

34 ...
35
36 replayer:
37   directory: "/workspace/test_data/endoscopy/video"
38   basename: "surgical_video"
39   frame_rate: 0 # as specified in timestamps
40   repeat: true # default: false
41   realtime: true # default: true
42   count: 0 # default: 0 (no frame count restriction)
43
44 ...

```

In `compose()`, we create the operators and add them to the application flow graph. The operators are created using the `make_operator()` function. The `make_operator()` function takes the operator name and the operator configuration as arguments. The operator name is used to identify the operator in the flow graph. The operator configuration is `holoscan::ArgList` object(s) that contains the operator's parameter values, or `holoscan::Arg` object(s).

The operator configuration (`holoscan::ArgList` object) is created using the `from_config()` function with a string argument that contains the name of the key in the configuration file. For example, `from_config("replayer")` creates an `holoscan::ArgList` object that contains the arguments of the `replayer` operator (such as values for 'directory', 'basename', 'frame\_rate', 'repeat', 'realtime', and 'count' parameters).

For the Operator parameters that are not defined in the configuration file, we can pass them as `holoscan::Arg` objects to the `make_operator()` function. For example, the `format_converter` operator has a `pool` parameter that is not defined in the configuration file. We pass the `pool` parameter as an `holoscan::Arg` object to the `make_operator()` function, using `make_resource()` function to create the `holoscan::Arg` object. This section shows the available resources that can be used to create an operator resource.

After creating the operators, we add the operators to the application flow graph using the `add_flow()` function.

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

The following code snippet creates edges between the operators in the flow graph as shown in [Fig. 5.3](#).

```

add_flow(replayer, visualizer_format_converter);
add_flow(visualizer_format_converter, visualizer, {"tensor", "source_video"});

add_flow(replayer, format_converter);
add_flow(format_converter, lstm_inferer);
add_flow(lstm_inferer, visualizer, {"tensor", "tensor"});

add_flow(replayer, recorder);

```

## 5.2.2 Build and Run the Application

Let's build and run the application.

The code shown in the previous section is available in Clara Holoscan Embedded SDK repository(<https://github.com/NVIDIA/clara-holoscan-embedded-sdk>).

Please make sure that you have [NVIDIA Container Toolkit](#) installed on your system and that NVIDIA Container Toolkit is configured to use the NVIDIA driver installed on your system.

First, we need to clone the Clara Holoscan Embedded SDK repository.

```
git clone https://github.com/NVIDIA/clara-holoscan-embedded-sdk.git
cd clara-holoscan-embedded-sdk
```

Next, we need to install GXF package that contains GXF libraries and headers required to build the Holoscan application.

```
./run install_gxf
```

Now, we can build the application.

```
./run build
```

It will take some time to create the Docker image and build the application.

After the build is complete, the sample applications (including the simple endoscopy application whose code is located at [apps/experiment/simple](#)) are available under the build directory.

The simple endoscopy application binary is located at `build/apps/experiment/simple/endoscopy_tool_tracking_simple`.

Let's run the application.

```
# Launch the docker container, mounting the current directory as /workspace/holoscan-sdk
./run launch
```

Inside the docker container (the current directory would be `/workspace/holoscan-sdk/build`), we can run the application.

```
export LD_LIBRARY_PATH=$(pwd):$(pwd)/lib:$LD_LIBRARY_PATH
./apps/experiments/simple/endoscopy_tool_tracking_simple
```

When the application is first launched, it will create a TensorRT engine file (`.engine`) under `test_data/endoscopy/model/tool_loc_convlstm_engines/` directory and it will take some time to create the engine file.

After the engine file is created, the application will start running.

Congratulations! You have successfully built and run the simple endoscopy application.

In the next sections, we will see how to create your application (`MyRecorder`) that records the video frames from the video file and save them to the disk.

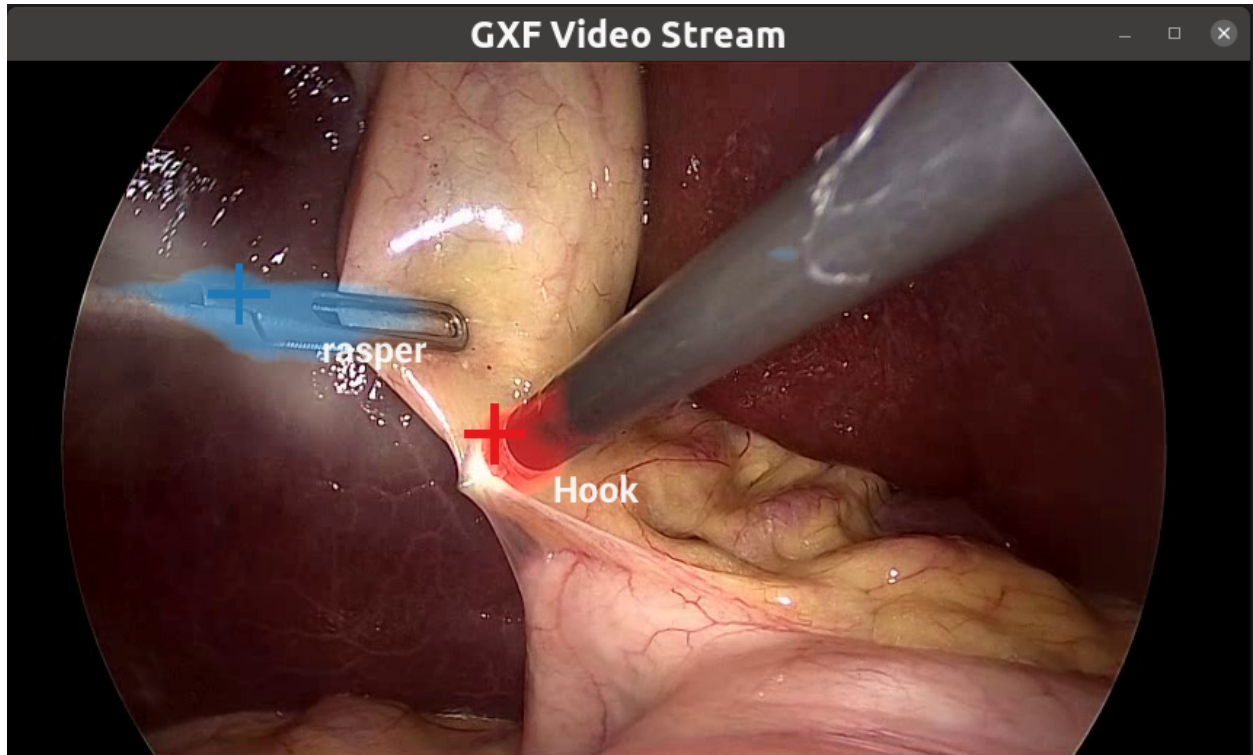


Fig. 5.4: Endoscopy application with tool tracking

## 5.3 Developing Holoscan GXF Extensions

GXF components in Holoscan can perform a multitude of sub-tasks ranging from data transformations, to memory management, to entity scheduling. In this section, we will explore an `nvidia::gxf::Codelet` component which in Holoscan is known as a “GXF extension”. *Holoscan (GXF) extensions* are typically concerned with application-specific sub-tasks such as data transformations, AI model inference, and the like.

### 5.3.1 Extension Lifecycle

The lifecycle of a `Codelet` is composed of the following five stages.

1. **initialize** - called only once when the codelet is created for the first time, and use of light-weight initialization.
2. **deinitialize** - called only once before the codelet is destroyed, and used for light-weight deinitialization.
3. **start** - called multiple times over the lifecycle of the codelet according to the order defined in the lifecycle, and used for heavy initialization tasks such as allocating memory resources.
4. **stop** - called multiple times over the lifecycle of the codelet according to the order defined in the lifecycle, and used for heavy deinitialization tasks such as deallocation of all resources previously assigned in **start**.
5. **tick** - called when the codelet is triggered, and is called multiple times over the codelet lifecycle; even multiple times between **start** and **stop**.

The flow between these stages is detailed in [Fig. 5.5](#).

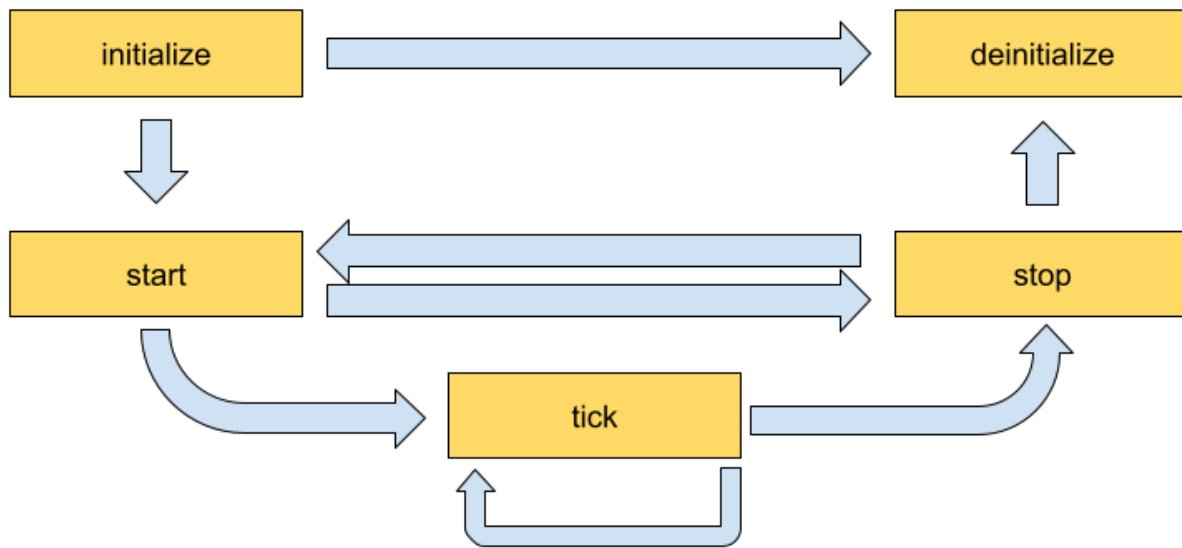


Fig. 5.5: Sequence of method calls in the lifecycle of a Holoscan extension

### 5.3.2 Implementing an Extension

In this section, we will implement a simple recorder that will highlight the actions we would perform in the lifecycle methods. The recorder receives data in the input queue and records the data to a configured location on the disk. The output format of the recorder files is the GXF-formatted index/binary replay files (the format is also used for the data in the sample applications), where the `gxf_index` file contains timing and sequence metadata that refer to the binary/tensor data held in the `gxf_entities` file.

#### Declare the Class That Will Implement the Extension Functionality

The developer can create their Holoscan extension by extending the `Codelet` class, implementing the extension functionality by overriding the lifecycle methods, and defining the parameters the extension exposes at the application level via the `registerInterface` method. To define our recorder component we would need to implement some of the methods in the `Codelet`.

First, clone the Holoscan project from [here](https://github.com/NVIDIA/clara-holoscan-embedded-sdk) and create a folder to develop our extension such as under `gxf_extensions/my_recorder`.

**Tip:** Using Bash we create a Holoscan extension folder as follows.

```
git clone https://github.com/NVIDIA/clara-holoscan-embedded-sdk.git
cd clara-holoscan-embedded-sdk
mkdir -p gxf_extensions/my_recorder
```

In our extension folder, we create a header file `my_recorder.hpp` with a declaration of our Holoscan component.

Listing 5.3: gxf\_extensions/my\_recorder/my\_recorder.hpp

```

1  #include <string>
2
3  #include "gxf/core/handle.hpp"
4  #include "gxf/std/codelet.hpp"
5  #include "gxf/std/receiver.hpp"
6  #include "gxf/std/transmitter.hpp"
7  #include "gxf/serialization/file_stream.hpp"
8  #include "gxf/serialization/entity_serializer.hpp"
9
10
11 class MyRecorder : public nvidia::gxf::Codelet {
12 public:
13     gxf_result_t registerInterface(nvidia::gxf::Registrar* registrar) override;
14     gxf_result_t initialize() override;
15     gxf_result_t deinitialize() override;
16
17     gxf_result_t start() override;
18     gxf_result_t tick() override;
19     gxf_result_t stop() override;
20
21 private:
22     nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::Receiver>> receiver_;
23     nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::EntitySerializer>> my_
24     ↪serializer_;
25     nvidia::gxf::Parameter<std::string> directory_;
26     nvidia::gxf::Parameter<std::string> basename_;
27     nvidia::gxf::Parameter<bool> flush_on_tick_;
28
29     // File stream for data index
30     nvidia::gxf::FileStream index_file_stream_;
31     // File stream for binary data
32     nvidia::gxf::FileStream binary_file_stream_;
33     // Offset into binary file
34     size_t binary_file_offset_;
35 };

```

### Declare the Parameters to Expose at the Application Level

Next, we can start implementing our lifecycle methods in the `my_recorder.cpp` file, which we also create in `gxf_extensions/my_recorder` path.

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

Listing 5.4: registerInterface in gxf\_extensions/my\_recorder/my\_recorder.cpp

```

1  #include "my_recorder.hpp"
2
3  gxf_result_t MyRecorder::registerInterface(nvidia::gxf::Registrar* registrar) {
4      nvidia::gxf::Expected<void> result;
5      result &= registrar->parameter(

```

(continues on next page)

(continued from previous page)

```

6     receiver_, "receiver", "Entity receiver",
7     "Receiver channel to log");
8 result &= registrar->parameter(
9     my_serializer_, "serializer", "Entity serializer",
10    "Serializer for serializing input data");
11 result &= registrar->parameter(
12    directory_, "out_directory", "Output directory path",
13    "Directory path to store received output");
14 result &= registrar->parameter(
15    basename_, "basename", "File base name",
16    "User specified file name without extension",
17    nvidia::gxf::Registrar::NoDefaultParameter(), GXF_PARAMETER_FLAGS_OPTIONAL);
18 result &= registrar->parameter(
19    flush_on_tick_, "flush_on_tick", "Boolean to flush on tick",
20    "Flushes output buffer on every `tick` when true", false); // default value `false`
21 return nvidia::gxf::ToResultCode(result);
22 }

```

If we are creating a pure GXF application (see *Creating the GXF Application Definition* section in the GXF documentation), in the application YAML, our component's parameters can be specified in the following format. Don't worry about what it means for now. With Holoscan API, the application is defined in C++ code instead of the YAML file, and the parameter values are set in the application code or via the configuration file (YAML).

Listing 5.5: Example parameters for MyRecorder component

```

1 name: my_recorder_entity
2 components:
3   - name: my_recorder_component
4     type: MyRecorder
5     parameters:
6       receiver: receiver
7       serializer: my_serializer
8       out_directory: /home/user/out_path
9       basename: my_output_file # optional
10      # flush_on_tick: false # optional

```

Note that all the parameters exposed at the application level are mandatory except for `flush_on_tick`, which defaults to `false`, and `basename`, whose default is handled at `initialize()` below.

## Implement the Lifecycle Methods

This extension does not need to perform any heavy-weight initialization tasks, so we will concentrate on `initialize()`, `tick()`, and `deinitialize()` methods which define the core functionality of our component. At initialization, we will create a file stream and keep track of the bytes we write on `tick()` via `binary_file_offset`.

Listing 5.6: `initialize` in `gxf_extensions/my_recorder/my_recorder.cpp`

```

24 gxf_result_t MyRecorder::initialize() {
25     // Create path by appending receiver name to directory path if basename is not provided
26     std::string path = directory_.get() + '/';
27     if (const auto& basename = basename_.try_get()) {
28         path += basename.value();

```

(continues on next page)



(continued from previous page)

```

29 } else {
30     path += receiver_>name();
31 }
32
33 // Initialize index file stream as write-only
34 index_file_stream_ = nvidia::gxf::FileStream("", path +
↳nvidia::gxf::FileStream::kIndexFileExtension);
35
36 // Initialize binary file stream as write-only
37 binary_file_stream_ = nvidia::gxf::FileStream("", path +
↳nvidia::gxf::FileStream::kBinaryFileExtension);
38
39 // Open index file stream
40 nvidia::gxf::Expected<void> result = index_file_stream_.open();
41 if (!result) {
42     return nvidia::gxf::ToResultCode(result);
43 }
44
45 // Open binary file stream
46 result = binary_file_stream_.open();
47 if (!result) {
48     return nvidia::gxf::ToResultCode(result);
49 }
50 binary_file_offset_ = 0;
51
52 return GXF_SUCCESS;
53 }

```

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

Listing 5.7: deinitialize in gxf\_extensions/my\_recorder/my\_recorder.cpp

```

55 gxf_result_t MyRecorder::deinitialize() {
56     // Close binary file stream
57     nvidia::gxf::Expected<void> result = binary_file_stream_.close();
58     if (!result) {
59         return nvidia::gxf::ToResultCode(result);
60     }
61
62     // Close index file stream
63     result = index_file_stream_.close();
64     if (!result) {
65         return nvidia::gxf::ToResultCode(result);
66     }
67
68     return GXF_SUCCESS;
69 }

```

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

Listing 5.8: start/stop in gxf\_extensions/my\_recorder/my\_recorder.cpp

```

71 gxf_result_t MyRecorder::start() {
72     return GXF_SUCCESS;
73 }
74
75 gxf_result_t MyRecorder::stop() {
76     return GXF_SUCCESS;
77 }

```

**Tip:** For a detailed implementation of `start()` and `stop()`, and how memory management can be handled therein, please refer to the implementation of the [AJA Video source extension](#).

Finally, we write the component-specific functionality of our extension by implementing `tick()`.

Listing 5.9: tick in gxf\_extensions/my\_recorder/my\_recorder.cpp

```

79 gxf_result_t MyRecorder::tick() {
80     // Receive entity
81     nvidia::gxf::Expected<nvidia::gxf::Entity> entity = receiver_>receive();
82     if (!entity) {
83         return nvidia::gxf::ToResultCode(entity);
84     }
85
86     // Write entity to binary file
87     nvidia::gxf::Expected<size_t> size = my_serializer_>serializeEntity(entity.value(), &
88     ↪ binary_file_stream_);
89     if (!size) {
90         return nvidia::gxf::ToResultCode(size);
91     }
92
93     // Create entity index
94     nvidia::gxf::EntityIndex index;
95     index.log_time = std::chrono::system_clock::now().time_since_epoch().count();
96     index.data_size = size.value();
97     index.data_offset = binary_file_offset_;
98
99     // Write entity index to index file
100    nvidia::gxf::Expected<size_t> result = index_file_stream_.writeTrivialType(&index);
101    if (!result) {
102        return nvidia::gxf::ToResultCode(result);
103    }
104    binary_file_offset_ += size.value();
105
106    if (flush_on_tick_) {
107        // Flush binary file output stream
108        nvidia::gxf::Expected<void> result = binary_file_stream_.flush();
109        if (!result) {
110            return nvidia::gxf::ToResultCode(result);
111        }

```

(continues on next page)

(continued from previous page)

```

112 // Flush index file output stream
113 result = index_file_stream_.flush();
114 if (!result) {
115     return nvidia::gxf::ToResultCode(result);
116 }
117 }
118
119 return GXF_SUCCESS;
120 }

```

## Register the Extension as a Holoscan Component

As a final step, we must register our extension so it is recognized as a component and loaded by the application executor. For this we create a simple declaration in `my_recorder_ext.cpp` as follows.

Listing 5.10: `gxf_extensions/my_recorder/my_recorder_ext.cpp`

```

1 #include "gxf/std/extension_factory_helper.hpp"
2
3 #include "my_recorder.hpp"
4
5 GXF_EXT_FACTORY_BEGIN()
6 GXF_EXT_FACTORY_SET_INFO(0xb891cef3ce754825, 0x9dd3dcac9bbd8483, "MyRecorderExtension",
7     "My example recorder extension", "NVIDIA", "0.1.0", "LICENSE");
8 GXF_EXT_FACTORY_ADD(0x2464fabf91b34ccf, 0xb554977fa22096bd, MyRecorder,
9     nvidia::gxf::Codelet, "My example recorder codelet.");
10 GXF_EXT_FACTORY_END()

```

`GXF_EXT_FACTORY_SET_INFO` configures the extension with the following information in order:

- UUID which can be generated using `scripts/generate_extension_uuids.py` which defines the **extension id**
- extension name
- extension description
- author
- extension version
- license text

`GXF_EXT_FACTORY_ADD` registers the newly built extension as a valid `Codelet` component with the following information in order:

- UUID which can be generated using `scripts/generate_extension_uuids.py` which defines the **component id** (this must be different from the extension id),
- fully qualified extension class,
- fully qualifies base class,
- component description

To build a shared library for our new extension which can be loaded by a Holoscan application at runtime we use a CMake file under `gxf_extensions/my_recorder/CMakeLists.txt` with the following content.

Listing 5.11: gxf\_extensions/my\_recorder/CMakeLists.txt

```

1  # Create library
2  add_library(my_recorder_lib SHARED
3      my_recorder.cpp
4      my_recorder.hpp
5  )
6  target_link_libraries(my_recorder_lib
7      PUBLIC
8      GXF::std
9      GXF::serialization
10     yaml-cpp
11 )
12
13 # Create extension
14 add_library(my_recorder SHARED
15     my_recorder_ext.cpp
16 )
17 target_link_libraries(my_recorder
18     PUBLIC my_recorder_lib
19 )
20
21 # Install GXF extension as a component 'holoscan-embedded-gxf_extensions'
22 install_gxf_extension(my_recorder) # this will also install my_recorder_lib
23 # install_gxf_extension(my_recorder_lib) # this statement is not necessary because this_
    ↪ library follows `<extension library name>_lib` convention.

```

Here, we create a library `my_recorder_lib` with the implementation of the lifecycle methods, and the extension `my_recorder` which exposes the C API necessary for the application runtime to interact with our component.

To make our extension discoverable from the project root we add the line

```
add_subdirectory(my_recorder)
```

to the CMake file `gxf_extensions/CMakeLists.txt`.

---

**Tip:** To build our extension, we can follow the steps in the [README](#).

---

At this point, we have a complete extension that records data coming into its receiver queue to the specified location on the disk using the GXF-formatted binary/index files.

## 5.4 Wrapping a GXF Codelet as a Holoscan Operator (C++ API)

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

With Holoscan C++ API, we can wrap GXF Components (including Codelets) in an *Entity* as a Holoscan Operator. Then, we can compose Operators programmatically, to create a Holoscan application.

For our C++ API-based application, we create the directory `apps/my_recorder_app` with our `MyRecorderOp` Operator implementation.

Listing 5.12: apps/my\_recorder\_app/my\_recorder\_op.hpp

```

1  #ifndef APPS_MY_RECORDER_APP_MY_RECORDER_OP_HPP
2  #define APPS_MY_RECORDER_APP_MY_RECORDER_OP_HPP
3
4  #include "holoscan/core/gxf/gxf_operator.hpp"
5
6  namespace holoscan::ops {
7
8  class MyRecorderOp : public holoscan::ops::GXFOperator {
9  public:
10     HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER(MyRecorderOp, holoscan::ops::GXFOperator)
11
12     MyRecorderOp() = default;
13
14     const char* gxf_typename() const override { return "MyRecorder"; }
15
16     void setup(OperatorSpec& spec) override;
17
18     void initialize() override;
19
20 private:
21     Parameter<holoscan::IOSpec*> receiver_;
22     Parameter<std::shared_ptr<holoscan::Resource>> my_serializer_;
23     Parameter<std::string> directory_;
24     Parameter<std::string> basename_;
25     Parameter<bool> flush_on_tick_;
26 };
27
28 } // namespace holoscan::ops
29
30 #endif /* APPS_MY_RECORDER_APP_MY_RECORDER_OP_HPP */

```

holoscan::ops::MyRecorderOp class wraps a MyRecorder GXF Codelet by inheriting holoscan::ops::GXFOperator.

To wrap a GXF Codelet as a Holoscan Operator, we need to implement the following functions:

- `const char* gxf_typename() const override`: return the GXF type name of the Codelet. The fully-qualified class name (MyRecorder) for the GXF Codelet is specified.
- `void setup(OperatorSpec& spec) override`: setup the OperatorSpec with the inputs/outputs and parameters of the Operator.
- `void initialize() override`: initialize the Operator.

HOLOSCAN\_OPERATOR\_FORWARD\_ARGS\_SUPER(MyRecorderOp, holoscan::ops::GXFOperator)) macro is used to forward the arguments of the constructor to the base class.

Then, we need to define the fields of the MyRecorderOp class.

Let's see the fields of the MyRecorderOp class. You can see that the same fields with the same names (but different types) are defined in both the MyRecorderOp class and the *MyRecorder* GXF Codelet.

Listing 5.13: Parameter declarations in apps/my\_recorder\_app/my\_recorder\_op.hpp

```

22  nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::Receiver>> receiver_;
23  nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::EntitySerializer>> my_
    ↪ serializer_;
24  nvidia::gxf::Parameter<std::string> directory_;
25  nvidia::gxf::Parameter<std::string> basename_;
26  nvidia::gxf::Parameter<bool> flush_on_tick_;

```

In the MyRecorderOp class, the followings are changed:

- holoscan::Parameter type is used instead of nvidia::gxf::Parameter type
- holoscan::IOSpec\* type is used instead of nvidia::gxf::Handle<nvidia::gxf::Receiver>> or nvidia::gxf::Handle<nvidia::gxf::Transmitter>> type
- std::shared\_ptr<holoscan::Resource>> type is used instead of nvidia::gxf::Handle<T>> type (such as nvidia::gxf::Handle<nvidia::gxf::EntitySerializer>>)

The implementation of the setup(OperatorSpec& spec) function and the initialize() function are as follows:

Listing 5.14: apps/my\_recorder\_app/my\_recorder\_op.cpp

```

1  #include "../my_recorder_op.hpp"
2
3  #include "holoscan/core/fragment.hpp"
4  #include "holoscan/core/gxf/entity.hpp"
5  #include "holoscan/core/operator_spec.hpp"
6
7  #include "holoscan/core/resources/gxf/video_stream_serializer.hpp"
8
9  namespace holoscan::ops {
10
11  void MyRecorderOp::setup(OperatorSpec& spec) {
12      auto& input = spec.input<:gxf::Entity>("input");
13      // Above is same with the following two lines (a default condition is assigned to the
    ↪ input port if not specified):
14      //
15      //      auto& input = spec.input<:gxf::Entity>("input")
16      //                      .condition(ConditionType::kMessageAvailable, Arg("min_size") =
    ↪ 1);
17
18      spec.param(receiver_, "receiver", "Entity receiver", "Receiver channel to log", &
    ↪ input);
19      spec.param(my_serializer_,
20                  "serializer",
21                  "Entity serializer",
22                  "Serializer for serializing input data");
23      spec.param(directory_, "out_directory", "Output directory path", "Directory path to
    ↪ store received output");
24      spec.param(basename_, "basename", "File base name", "User specified file name without
    ↪ extension");
25      spec.param(flush_on_tick_,
26                  "flush_on_tick",

```

(continues on next page)

(continued from previous page)

```

27         "Boolean to flush on tick",
28         "Flushes output buffer on every `tick` when true",
29         false);
30     }
31
32     void MyRecorderOp::initialize() {
33         // Set up prerequisite parameters before calling GXFOperator::initialize()
34         auto frag = fragment();
35         auto serializer =
36             frag->make_resource<holoscan::VideoStreamSerializer>("serializer");
37         add_arg(Arg("serializer") = serializer);
38
39         GXFOperator::initialize();
40     }
41
42 } // namespace holoscan::ops

```

In the `setup(OperatorSpec& spec)` function, we set up the inputs/outputs and parameters of the Operator.

Please compare the content of the function with `MyRecorder` class's `registerInterface` function. You can see that `setup(OperatorSpec& spec)` function is very similar to the `registerInterface(OperatorSpec& spec)` function in the `MyRecorder` class.

In C++ API, GXF Receiver and Transmitter components (such as `DoubleBufferReceiver` and `DoubleBufferTransmitter`) are considered as input and output ports of the Operator so we register the inputs/outputs of the Operator with `input<T>` and `output<T>` functions (where T is the data type of the port).

Compared to *the pure GXF application that does the same job*, the *SchedulingTerm* definitions of an Entity in *GXF Application YAML* are specified as Conditions (e.g., `holoscan::MessageAvailableCondition` and `holoscan::DownstreamMessageAffordableCondition`) on the input/output ports.

The following statements

```

auto& input = spec.input<::gxf::Entity>("input");
// Above is same with the following two lines (a default condition is assigned to the
// input port if not specified):
//
//     auto& input = spec.input<::gxf::Entity>("input")
//                               .condition(ConditionType::kMessageAvailable, Arg("min_size") =
//                               1);
spec.param(receiver_, "receiver", "Entity receiver", "Receiver channel to log", &
input);

```

represent the following highlighted statements of *GXF Application YAML*:

Listing 5.15: A part of `apps/my_recorder_app_gxf/my_recorder_gxf.yaml`

```

35 name: recorder
36 components:
37   - name: input
38     type: nvidia::gxf::DoubleBufferReceiver
39   - name: allocator
40     type: nvidia::gxf::UnboundedAllocator
41   - name: component_serializer

```

(continues on next page)

(continued from previous page)

```

42  type: nvidia::gxf::StdComponentSerializer
43  parameters:
44    allocator: allocator
45  - name: entity_serializer
46    type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
↳nvidia::gxf::EntitySerializer
47    parameters:
48      component_serializers: [component_serializer]
49  - type: MyRecorder
50    parameters:
51      receiver: input
52      serializer: entity_serializer
53      out_directory: "/tmp"
54      basename: "tensor_out"
55  - type: nvidia::gxf::MessageAvailableSchedulingTerm
56    parameters:
57      receiver: input
58      min_size: 1

```

In the same way, if we had a Transmitter GXF component, we would have the following statements (Please see available constants for `holoscan::ConditionType`):

```

auto& output = spec.output<:gxf::Entity>("output");
// Above is same with the following two lines (a default condition is assigned to the
↳output port if not specified):
//
//   auto& output = spec.output<:gxf::Entity>("output")
//                               .condition(ConditionType::kDownstreamMessageAffordable, Arg(
↳"min_size") = 1);

```

In the `initialize()` function, we set up the pre-defined parameters such as `serializer`.

```

auto frag = fragment();
auto serializer =
    frag->make_resource<holoscan::VideoStreamSerializer>("serializer");
add_arg(Arg("serializer") = serializer); // set 'serializer' parameter with 'serializer'
↳resource.

```

Holoscan C++ API provides `holoscan::VideoStreamSerializer` class (include `holoscan/core/resources/gxf/video_stream_serializer.hpp` and `src/core/resources/gxf/video_stream_serializer.cpp`) for `nvidia::holoscan::stream_playback::VideoStreamSerializer` GXF component and above statements covers the highlighted statements of *GXF Application YAML*:

Listing 5.16: Another part of `apps/my_recorder_app_gxf/my_recorder_gxf.yaml`

```

35 name: recorder
36 components:
37   - name: input
38     type: nvidia::gxf::DoubleBufferReceiver
39   - name: allocator
40     type: nvidia::gxf::UnboundedAllocator
41   - name: component_serializer
42     type: nvidia::gxf::StdComponentSerializer

```

(continues on next page)



(continued from previous page)

```

43     parameters:
44         allocator: allocator
45     - name: entity_serializer
46       type: nvidia::holoscan::stream_playback::VideoStreamSerializer  # inheriting from
↳nvidia::gxf::EntitySerializer
47     parameters:
48         component_serializers: [component_serializer]
49     - type: MyRecorder
50     parameters:
51         receiver: input
52         serializer: entity_serializer
53         out_directory: "/tmp"
54         basename: "tensor_out"
55     - type: nvidia::gxf::MessageAvailableSchedulingTerm
56     parameters:
57         receiver: input
58         min_size: 1

```

## 5.5 Creating the Holoscan Application (C++ API)

The following code snippet shows how to create the Holoscan Application using the C++ API.

Listing 5.17: apps/my\_recorder\_app/main.cpp

```

1  #include <holoscan/holoscan.hpp>
2  #include <holoscan/std_ops.hpp>
3
4  #include "../my_recorder_op.hpp"
5
6  class App : public holoscan::Application {
7  public:
8
9      void compose() override {
10         using namespace holoscan;
11
12         HOLOSCAN_LOG_DEBUG("In App::compose() method");
13
14         auto replayer = make_operator<ops::VideoStreamReplayerOp>("replayer", from_config(
↳"replayer"));
15         // auto replayer = make_operator<ops::VideoStreamReplayerOp>("replayer", from_config(
↳"replayer"),
16         //                               Arg("frame_rate") = 30.f, // same with Arg("frame_rate", 30.f)
17         //                               Arg("repeat") = true);
18         auto recorder = make_operator<ops::MyRecorderOp>("recorder", from_config("recorder
↳"));
19
20         HOLOSCAN_LOG_INFO("replayer.directory: {}", from_config("replayer.directory").as
↳<std::string>());
21
22         // Flow definition

```

(continues on next page)

(continued from previous page)

```

23     add_flow(replayer, recorder);
24     // Above is same with:
25     //
26     // // replayer's output port named 'output' is connected to recorder's input port named
    ↪ 'input'
27     // add_flow(replayer, recorder, {"output", "input"});
28     // // or,
29     // // replayer has only one output port and recorder has only one input port so
    ↪ names can be omitted
30     // add_flow(replayer, recorder, {"", ""});
31 }
32 };
33
34 int main(int argc, char** argv) {
35     holoscan::load_env_log_level();
36
37     auto app = holoscan::make_application<App>();
38     app->config("apps/my_recorder_app/app_config.yaml");
39     app->run();
40
41     return 0;
42 }

```

The App class is the main class of the Holoscan Application. It inherits from `holoscan::Application` class.

In the `compose()` function, we create the operators and flows of the Holoscan Application.

We can call `make_operator()` to create an operator of type `<OperatorT>` and pass parameter values to the operator.

`from_config()` function is used to get parameter values from the configuration file (`apps/my_recorder_app/app_config.yaml`) that is passed to the `config()` function in the main function.

Let's create a configuration file for the Holoscan Application (C++ API) in the `apps/my_recorder_app/app_config.yaml` file.

Listing 5.18: `apps/my_recorder_app/app_config.yaml`

```

1  # 'extensions' has the same content with 'build/apps/my_recorder_app_gxf/my_recorder_gxf_
    ↪ manifest.yaml' file.
2  extensions:
3      - libgxf_std.so
4      - libgxf_cuda.so
5      - libgxf_multimedia.so
6      - libgxf_serialization.so
7      - ./gxf_extensions/my_recorder/libmy_recorder.so
8      - ./gxf_extensions/stream_playback/libstream_playback.so
9
10 # Configururaton for 'holoscan::ops::VideoStreamReplayerOp' Operator
11 replayer:
12     directory: "/workspace/test_data/endoscopy/video"
13     basename: "surgical_video"
14     frame_rate: 0    # as specified in timestamps
15     repeat: false   # default: false
16     realtime: true   # default: true

```

(continues on next page)

(continued from previous page)

```

17  count: 0          # default: 0 (no frame count restriction)
18
19  # Configururaton for 'holoscan::ops::MyRecorderOp' Operator
20  recorder:
21    out_directory: "/tmp"
22    basename: "tensor_out"

```

In `app_config.yaml`, it has the information of GXF extension paths (that is same with the content of `build/apps/my_recorder_app_gxf/my_recorder_gxf_manifest.yaml` file if you have followed the [Creating the GXF Application Definition](#) section) and some parameter values for `VideoStreamReplayerOp` and `MyRecorderOp` operators.

The `extensions` field in this YAML configuration file is a list of GXF extension paths. The first four paths are the paths of GXF core extensions that are required for the `VideoStreamReplayerOp` and `MyRecorderOp` operators (you can find the paths under the `build/lib` directory once you have built the Holoscan SDK). The last two paths are the paths of GXF extensions that are required for the `MyRecorderOp` and `VideoStreamReplayerOp` operator (you can find the paths under the `build` directory once you have built the Holoscan SDK).

Compared with `my_recorder_gxf.yaml` file, `app_config.yaml` file is concise and easier to read. It is also easier to modify the parameter values of operators (codelets). In GXF Application YAML, we had to specify graph edges between graph nodes through `nvidia::gxf::Connection` component. With C++ API, we can connect Operator nodes programmatically with `add_flow()` function.

On top of the parameter values from the configuration file, if you want to override the parameter values, you can pass them to the operator as an argument of the `make_operator()` function. For example, if you want to override the parameter values of `replayer`, you can do it as follows:

```

auto replayer = make_operator<ops::VideoStreamReplayerOp>("replayer", from_config(
    ↪ "replayer"),
    Arg("frame_rate") = 30.f, // same with Arg("frame_rate", 30.f)
    Arg("repeat") = true);

```

In `main()` function, we create the Holoscan Application and pass the configuration file to the `config()` function.

`holoscan::load_env_log_level()` function is used to set the log level of the Holoscan Application from the environment variable `HOLOSCAN_LOG_LEVEL`:

`HOLOSCAN_LOG_LEVEL` can be set to one of the following values:

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- CRITICAL
- OFF

```
export HOLOSCAN_LOG_LEVEL=TRACE
```

We can call `HOLOSCAN_LOG_XXX()` macros to log messages. The format string follows the `fmtlib` format string syntax.

An Application object is created in the `main()` function through `make_application()` function and it is launched by calling `run()` function.

Once `main.cpp` file is available, we need to declare a CMake file `apps/my_recorder_app/CMakeLists.txt` as follows.

Listing 5.19: `apps/my_recorder_app/CMakeLists.txt`

```

1 add_executable(my_recorder_app
2     my_recorder_op.hpp
3     my_recorder_op.cpp
4     main.cpp
5 )
6
7 target_link_libraries(my_recorder_app
8     PRIVATE
9     holoscan-embedded
10 )
11
12 # Download the associated dataset if needed
13 if(HOLOSCAN_DOWNLOAD_DATASETS)
14     add_dependencies(my_recorder_app endoscopy_data)
15 endif()
16
17 # Copy config file
18 file(COPY "${CMAKE_CURRENT_SOURCE_DIR}/app_config.yaml" DESTINATION "${CMAKE_CURRENT_
19     ↪BINARY_DIR}")
20
21 # Get relative folder path for the app
22 file(RELATIVE_PATH app_relative_dest_path ${CMAKE_SOURCE_DIR} ${CMAKE_CURRENT_SOURCE_DIR}
23     ↪)
24
25 # Install the app
26 install(TARGETS "my_recorder_app"
27     DESTINATION "${app_relative_dest_path}"
28     COMPONENT "holoscan-embedded-apps"
29 )
30 install(FILES "${CMAKE_CURRENT_SOURCE_DIR}/app_config.yaml"
31     DESTINATION ${app_relative_dest_path}
32     COMPONENT "holoscan-embedded-apps"
33 )

```

In the `apps/my_recorder_app/CMakeLists.txt` file, we declare the executable `my_recorder_app` and link the `holoscan-embedded` library. We also copy the configuration file `apps/my_recorder_app/app_config.yaml` to the same folder under `${CMAKE_CURRENT_BINARY_DIR}` to make it available to the application.

Finally, we install the application to the `holoscan-embedded-apps` component by calling `install()` function.

To make this Holoscan application discoverable by the build, in the root of the repository, we add the following line

```
add_subdirectory(my_recorder_app)
```

to `apps/CMakeLists.txt`.

## 5.6 Running the Holoscan MyRecorder Application (C++ API)

To run our application in a local development container:

1. Follow the instructions under the [Using a Development Container](#) section steps 1-5 (try clearing the CMake cache by removing the build folder before compiling).

You can execute the following commands to build

```
./run install_gxf
# ./run clear_cache # if you want to clear build/install/cache folders
./run build
```

2. Our application can now be run in the development container using the command, where \$(pwd) is the path to the build folder:

You can execute `./run launch` to run the development container.

```
./run launch
```

Then, you can execute the following commands to run the application

```
LD_LIBRARY_PATH=$(pwd):$(pwd)/lib:$LD_LIBRARY_PATH ./apps/my_recorder_app/my_
↪ recorder_app
```

inside the development container.

```
@LINUX:/workspace/holoscan-sdk/build$ LD_LIBRARY_PATH=$(pwd):$(pwd)/lib:$LD_LIBRARY_
↪ PATH ./apps/my_recorder_app/my_recorder_app
2022-08-24 12:36:48.685 INFO /workspace/holoscan-sdk/src/core/executors/gxf/gxf_
↪ executor.cpp@39: Creating context
[2022-08-24 12:36:48.685] [holoscan] [info] [gxf_executor.cpp:64] Loading_
↪ extensions...
[2022-08-24 12:36:48.692] [holoscan] [info] [main.cpp:20] replayer.directory: /
↪ workspace/test_data/endoscopy/video
[2022-08-24 12:36:48.692] [holoscan] [info] [gxf_executor.cpp:222] Activating Graph.
↪ ...
[2022-08-24 12:36:48.693] [holoscan] [info] [gxf_executor.cpp:224] Running Graph...
[2022-08-24 12:36:48.693] [holoscan] [info] [gxf_executor.cpp:226] Waiting for_
↪ completion...
2022-08-24 12:36:48.693 INFO gxf/std/greedy_scheduler.cpp@170: Scheduling 2_
↪ entities
2022-08-24 12:37:16.084 INFO /workspace/holoscan-sdk/gxf_extensions/stream_
↪ playback/video_stream_replayer.cpp@144: Reach end of file or playback count_
↪ reaches to the limit. Stop ticking.
2022-08-24 12:37:16.084 INFO gxf/std/greedy_scheduler.cpp@329: Scheduler stopped:_
↪ Some entities are waiting for execution, but there are no periodic or async_
↪ entities to get out of the deadlock.
2022-08-24 12:37:16.084 INFO gxf/std/greedy_scheduler.cpp@353: Scheduler finished.
[2022-08-24 12:37:16.084] [holoscan] [info] [gxf_executor.cpp:228] Deactivating_
↪ Graph...
2022-08-24 12:37:16.085 INFO /workspace/holoscan-sdk/src/core/executors/gxf/gxf_
↪ executor.cpp@49: Destroying context
```

You can set `HOLOSCAN_LOG_LEVEL` environment variable to `DEBUG` (or other levels such as ‘TRACE’) to see more logs.

```

@LINUX:/workspace/holoscan-sdk/build$ export HOLOSCHAN_LOG_LEVEL=DEBUG
@LINUX:/workspace/holoscan-sdk/build$ LD_LIBRARY_PATH=$(pwd):$(pwd)/lib:$LD_LIBRARY_
↳PATH ./apps/my_recorder_app/my_recorder_app
2022-08-24 12:41:01.616 INFO /workspace/holoscan-sdk/src/core/executors/gxf/gxf_
↳executor.cpp@39: Creating context
[2022-08-24 12:41:01.616] [holoscan] [info] [gxf_executor.cpp:64] Loading_
↳extensions...
[2022-08-24 12:41:01.622] [holoscan] [debug] [main.cpp:12] In App::compose() method
[2022-08-24 12:41:01.622] [holoscan] [debug] [fragment.hpp:62] Creating operator
↳'replayer'
[2022-08-24 12:41:01.622] [holoscan] [debug] [fragment.hpp:84] Creating resource
↳'entity_serializer'
[2022-08-24 12:41:01.622] [holoscan] [debug] [fragment.hpp:84] Creating resource
↳'component_serializer'
[2022-08-24 12:41:01.622] [holoscan] [debug] [fragment.hpp:84] Creating resource
↳'allocator'
[2022-08-24 12:41:01.622] [holoscan] [debug] [fragment.hpp:106] Creating condition
↳'boolean_scheduling_term'
[2022-08-24 12:41:01.623] [holoscan] [debug] [fragment.hpp:62] Creating operator
↳'recorder'
[2022-08-24 12:41:01.623] [holoscan] [debug] [fragment.hpp:84] Creating resource
↳'serializer'
[2022-08-24 12:41:01.623] [holoscan] [debug] [fragment.hpp:84] Creating resource
↳'component_serializer'
[2022-08-24 12:41:01.623] [holoscan] [debug] [fragment.hpp:84] Creating resource
↳'allocator'
[2022-08-24 12:41:01.623] [holoscan] [info] [main.cpp:20] replayer.directory: /
↳workspace/test_data/endoscopy/video
[2022-08-24 12:41:01.623] [holoscan] [debug] [gxf_executor.cpp:107] Operator:_
↳replayer
[2022-08-24 12:41:01.623] [holoscan] [debug] [gxf_executor.cpp:115] Next_
↳operator: recorder
[2022-08-24 12:41:01.623] [holoscan] [debug] [gxf_executor.cpp:125] Port:_
↳output -> input
[2022-08-24 12:41:01.623] [holoscan] [debug] [gxf_executor.cpp:107] Operator:_
↳recorder
[2022-08-24 12:41:01.623] [holoscan] [info] [gxf_executor.cpp:222] Activating Graph.
↳..
[2022-08-24 12:41:01.713] [holoscan] [info] [gxf_executor.cpp:224] Running Graph...
[2022-08-24 12:41:01.713] [holoscan] [info] [gxf_executor.cpp:226] Waiting for_
↳completion...
2022-08-24 12:41:01.713 INFO gxf/std/greedy_scheduler.cpp@170: Scheduling 2_
↳entities
2022-08-24 12:41:29.093 INFO /workspace/holoscan-sdk/gxf_extensions/stream_
↳playback/video_stream_replayer.cpp@144: Reach end of file or playback count_
↳reaches to the limit. Stop ticking.
2022-08-24 12:41:29.093 INFO gxf/std/greedy_scheduler.cpp@329: Scheduler stopped:_
↳Some entities are waiting for execution, but there are no periodic or async_
↳entities to get out of the deadlock.
2022-08-24 12:41:29.093 INFO gxf/std/greedy_scheduler.cpp@353: Scheduler finished.
[2022-08-24 12:41:29.093] [holoscan] [info] [gxf_executor.cpp:228] Deactivating_
↳Graph...
2022-08-24 12:41:29.209 INFO /workspace/holoscan-sdk/src/core/executors/gxf/gxf_
↳executor.cpp@49: Destroying context

```

(continues on next page)

(continued from previous page)

A successful run (it takes about 30 secs) will result in output files (tensor\_out.gxf\_index and tensor\_out.gxf\_entities in /tmp) that match the original input files (surgical\_video.gxf\_index and surgical\_video.gxf\_entities under test\_data/endoscopy/video) exactly.

```
@LINUX:/workspace/holoscan-sdk/build$ ls -al /tmp
total 821392
drwxrwxrwt 1 root root      4096 Aug 24 12:36 .
drwxr-xr-x 1 root root      4096 Aug 24 12:36 ..
drwxrwxrwt 2 root root      4096 Aug 11 21:42 .X11-unix
-rw-r--r-- 1 1000 1000    738674 Aug 24 12:41 gxf_log
-rw-r--r-- 1 1000 1000 840054484 Aug 24 12:41 tensor_out.gxf_entities
-rw-r--r-- 1 1000 1000    16392 Aug 24 12:41 tensor_out.gxf_index

@LINUX:/workspace/holoscan-sdk/build$ ls -al ../test_data/endoscopy/video/
total 839116
drwxr-xr-x 2 1000 1000      4096 Aug 24 02:08 .
drwxr-xr-x 4 1000 1000      4096 Aug 24 02:07 ..
-rw-r--r-- 1 1000 1000 19164125 Jun 17 16:31 raw.mp4
-rw-r--r-- 1 1000 1000 840054484 Jun 17 16:31 surgical_video.gxf_entities
-rw-r--r-- 1 1000 1000    16392 Jun 17 16:31 surgical_video.gxf_index
```





## CLARA HOLOSCAN SAMPLE APPLICATIONS

This section explains how to run the Clara Holoscan sample applications. Three sample applications are provided with the SDK:

1. Tool tracking in endoscopy video using an LSTM model
2. Hi-speed endoscopy using high resolution and high frame rate cameras
3. Semantic segmentation bone contours with hyperechoic lines

Each application comes with support for an AJA capture card or replay from a video file included in the sample application container. More information regarding the AI models used for these applications can be found under the Overview section of this document.

---

**Tip:** To run the sample applications, please follow the instructions in [NGC website](#) or [Github repository](#).

Please also ensure that X11 is configured to allow commands from docker:

```
xhost +local:docker
```

---

### 6.1 Endoscopy Tool Tracking Application

Digital endoscopy is a key technology for medical screenings and minimally invasive surgeries. Using real-time AI workflows to process and analyze the video signal produced by the endoscopic camera, this technology helps medical professionals with anomaly detection and measurements, image enhancements, alerts, and analytics.

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

#### 6.1.1 Input source: Video Stream Replayer

The GXF pipeline in a graph form is defined at `apps/endoscopy_tool_tracking_gxf/tracking_replayer.yaml` in [Holoscan Embedded SDK Github Repository](#).

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

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

- *Format Converter*: Convert image format from RGB888 (24-bit pixel) to RGBA8888(32-bit pixel) for visualization (*Tool Tracking Visualizer*)

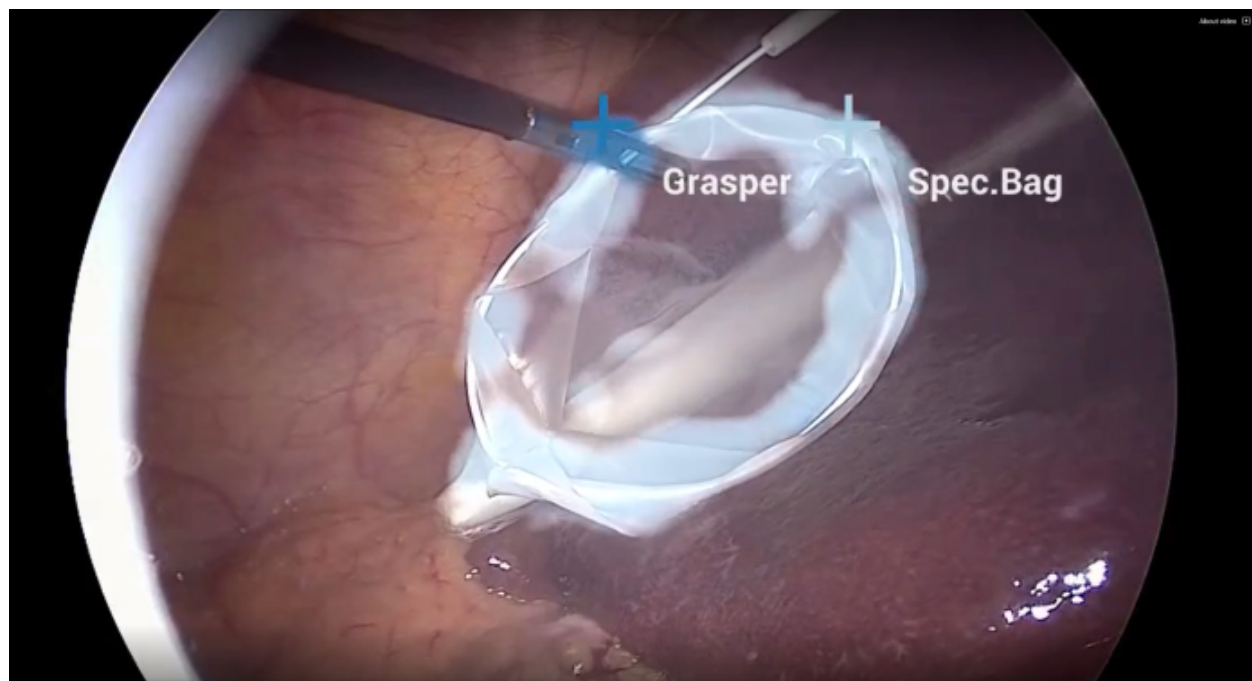


Fig. 6.1: Endoscopy image from a gallbladder surgery showing AI-powered frame-by-frame tool identification and tracking. Image courtesy of Research Group Camma, IHU Strasbourg and the University of Strasbourg

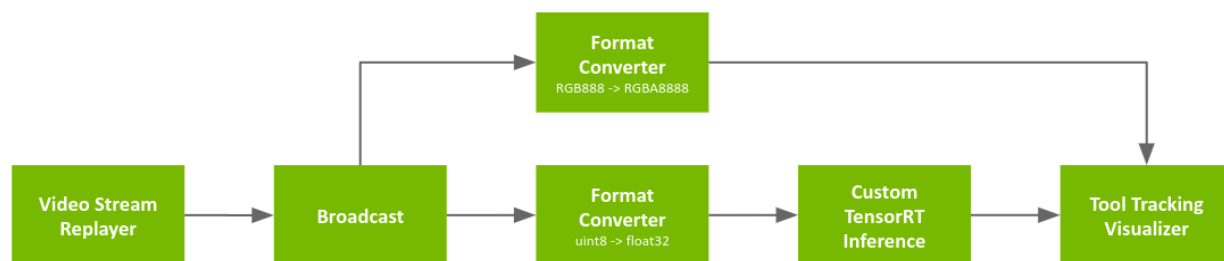


Fig. 6.2: Tool tracking application workflow with replay from file

- *Format Converter*: Convert the data type of the image from uint8 to float32 for feeding into the tool tracking model (by *Custom TensorRT Inference*)

Then, *Tool Tracking Visualizer* uses outputs from the first Format Converter and Custom TensorRT Inference to render overlay frames (mask/point/text) on top of the original video frames.

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

In the runtime container (from NGC):

```
cd /opt/holoscan_sdk

# Endoscopy tool tracking (GXF) from recorded video
./apps/endoscopy_tool_tracking_gxf/tracking_replayer

# Endoscopy tool tracking (C++ API) from recorded video
# 1. Make sure that 'source' is set to 'replayer' in app_config.yaml
sed -i -e 's#^source:.*#source: replayer#' ./apps/endoscopy_tool_tracking/app_config.yaml
# 2. Run the application
./apps/endoscopy_tool_tracking/endoscopy_tool_tracking
```

In the development container (from source):

```
cd /workspace/holoscan-sdk/build

# Endoscopy tool tracking (GXF) from recorded video
./apps/endoscopy_tool_tracking_gxf/tracking_replayer

# Endoscopy tool tracking (C++ API) from recorded video
# 1. Make sure that 'source' is set to 'replayer' in app_config.yaml
sed -i -e 's#^source:.*#source: replayer#' ./apps/endoscopy_tool_tracking/app_config.yaml
# 2. Run the application
LD_LIBRARY_PATH=$(pwd):$(pwd)/lib:$LD_LIBRARY_PATH ./apps/endoscopy_tool_tracking/
↪endoscopy_tool_tracking
```

### 6.1.2 Input source: AJA

The GXF pipeline in a graph form is defined at `apps/endoscopy_tool_tracking_gxf/tracking_aja.yaml` in [Holoscan Embedded SDK Github Repository](#).

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

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

- *AJA Source*: Get video frame from AJA HDMI capture card (pixel format is RGBA8888 with the resolution of 1920x1080)
- *Format Converter*: Convert image format from RGB8888 (32-bit pixel) to RGBA888 (24-bit pixel) for recording (*Video Stream Recorder*)
- *Video Stream Recorder*: Record input frames into a file

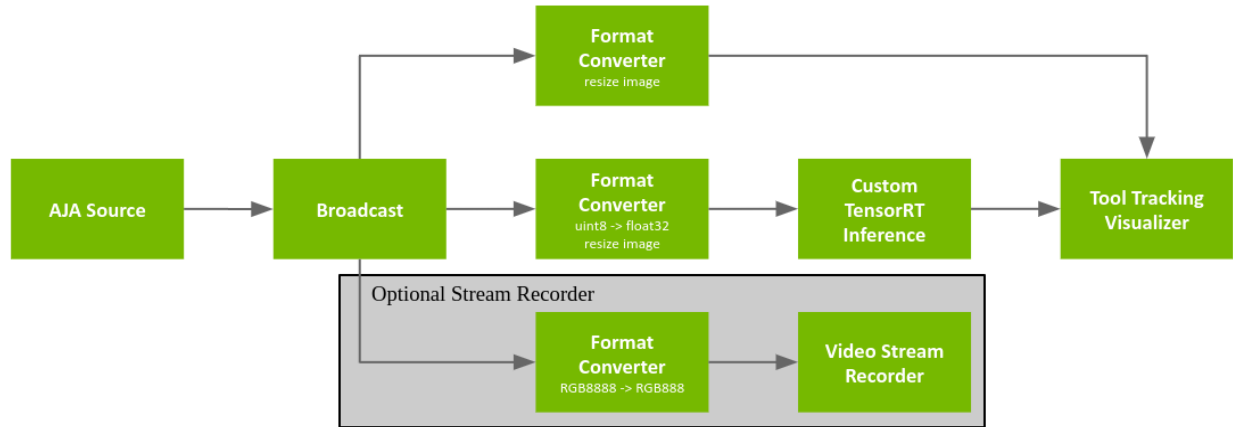


Fig. 6.3: AJA tool tracking app

Please follow these steps to run the Endoscopy Tool Tracking Application:

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

In the runtime container (from NGC):

```

cd /opt/holoscan_sdk/

# Endoscopy tool tracking (GXF) with AJA
./apps/endoscopy_tool_tracking_gxf/tracking_aja

# Endoscopy tool tracking (C++ API) with AJA
# 1. Make sure that 'source' is set to 'aja' in app_config.yaml
# (To enable recording, you can also update the value for 'do_record' to 'true'.)
sed -i -e 's#^source:.*#source: aja#' ./apps/endoscopy_tool_tracking/app_config.yaml
# 2. Run the application
./apps/endoscopy_tool_tracking/endoscopy_tool_tracking

```

In the development container (from source):

```

cd /workspace/holoscan-sdk/build

# Endoscopy tool tracking (GXF) with AJA
./apps/endoscopy_tool_tracking_gxf/tracking_aja

# Endoscopy tool tracking (C++ API) with AJA
# 1. Make sure that 'source' is set to 'aja' in app_config.yaml
# (To enable recording, you can also update the value for 'do_record' to 'true'.)
sed -i -e 's#^source:.*#source: aja#' ./apps/endoscopy_tool_tracking/app_config.yaml
# 2. Run the application
LD_LIBRARY_PATH=$(pwd):$(pwd)/lib:$LD_LIBRARY_PATH ./apps/endoscopy_tool_tracking/
↪ endoscopy_tool_tracking

```

## 6.2 Hi-Speed Endoscopy Application

The hi-speed endoscopy application showcases how high resolution cameras can be used to capture the scene, processed on GPU and displayed at high frame rate using the GXF framework. This application requires Emergent Vision Technologies camera and a display with high refresh rate to keep up with camera's framerate. This sections also explains how to *enable GSYNC for the display*, if GSYNC enabled monitor is available, how to *install and enable GPUDirect RDMA*, and how to *enable exclusive display mode* for better performance.

Note that this application is meant to be run directly on the device without using docker.

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



Fig. 6.4: Hi-Speed Endoscopy App

The data acquisition happens using `emergent-source`, by default it is set to 4200x2160 at 240Hz. The acquired data is then demosaiced in GPU using CUDA via `bayer-demosaic` and displayed through `holoviz-viewer`.

Follow below steps to run the Hi-Speed Endoscopy Application:

**Tip:** To run the Hi-Speed Endoscopy Application, follow below commands after setting up the *Holoscan SDK* to run from source and your *EVT camera*.

On the local environment (from source):

1. Configure and build the project with `HOLOSCAN_BUILD_HI_SPEED_ENDO_APP` option as ON.

```

cd ${PATH_TO_SDK_REPOSITORY}
cmake -S . -B build \
  -D CMAKE_BUILD_TYPE=Release \
  -D CUDAToolkit_ROOT:PATH=/usr/local/cuda \
  -D CMAKE_CUDA_COMPILER:PATH=/usr/local/cuda/bin/nvcc \
  -D HOLOSCAN_BUILD_HI_SPEED_ENDO_APP=ON
cmake --build build -j

```

2. Run the application

```

cd ${PATH_TO_SDK_REPOSITORY}/build
sudo ./apps/hi_speed_endoscopy_gxf/hi_speed_endoscopy

```

Currently this application has hardcoded the camera controls within the `emergent-source`. Once the user updated the `gxf-extension`, the project would need to be rebuild as mentioned in step 1. above. For more information on the controls, refer to [EVT Camera Attributes Manual](#).

### 6.2.1 Enable G-SYNC for Display

To get better performance, the application can be run with a G-SYNC enabled display. To enable G-SYNC for the display, a G-SYNC enabled display is required. This app has been tested with two G-SYNC enabled displays: [Asus ROG Swift PG279QM](#) and [Asus ROG Swift 360 Hz PG259QNR](#).

Follow below steps to enable G-SYNC for the display using `nvidia-settings`.

1. Open `nvidia-settings` using terminal. This step requires a graphical user interface.

```
nvidia-settings
```

This will open the NVIDIA Settings window.

2. Click on **X Server Display Configuration** and then **Advanced** button. This will show option **Allow G-SYNC on monitor not validated as G-SYNC compatible**, select the option and click **Apply**. The window would look like below.
3. To show the refresh rate and G-SYNC label on the display window, click on **OpenGL Settings** for the selected display. Now click **Allow G-SYNC/G-SYNC Compatible** and **Enable G-SYNC/G-SYNC Compatible Visual Indicator** options and click **Quit**. This step is shown in below image. The Gsync indicator will be at the top right screen once the application is running.

### 6.2.2 Installing and Enabling GPUDirect RDMA

The GPUDirect drivers must be installed to enable the use of GPUDirect when using an RTX6000 or RTX A6000 add-in dGPU.

---

**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
```

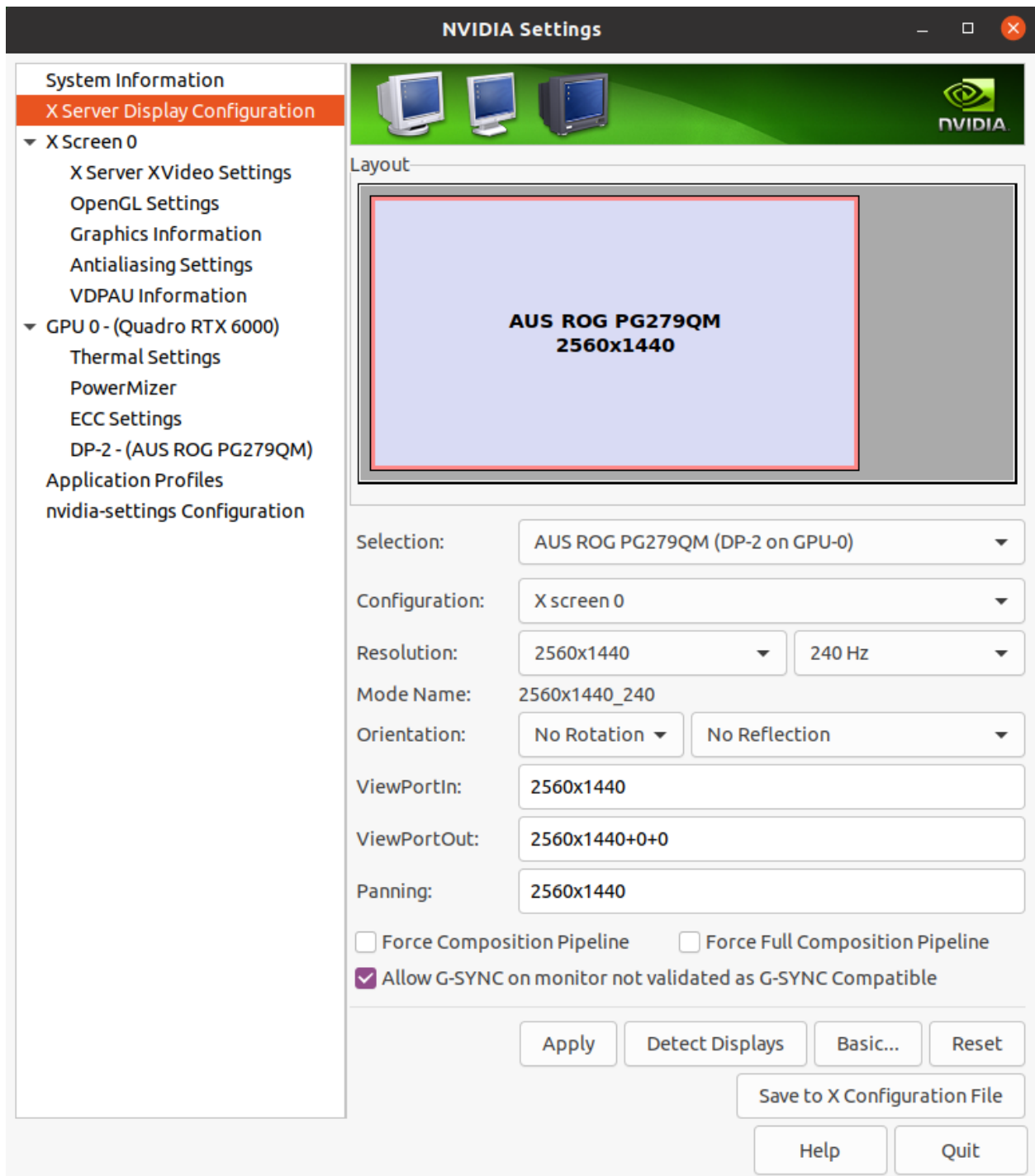


Fig. 6.5: Enable G-SYNC for the current display

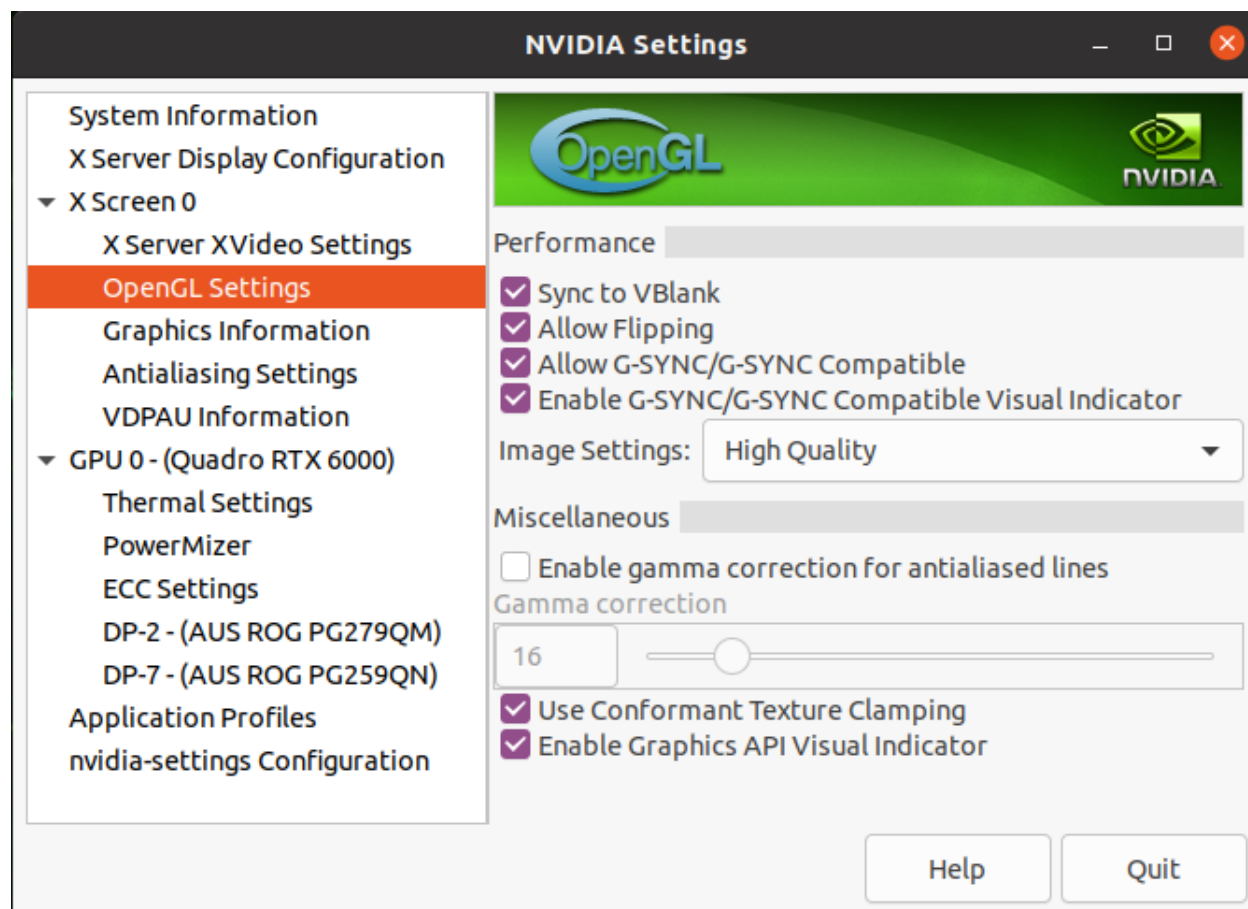


Fig. 6.6: Enable Visual Indicator for the current display



**Note:** To enable the GPUDirect RDMA on NVIDIA IGX Orin Developer Kit, update your firmware with instructions from the [NVIDIA IGX Orin Developer Kit User Guide](#) or the below command needs to be executed at every single bootup.

```
sudo setpci -s 0007:02:00.0 ecap_acs+6.w=0
```

3. Update the hi-speed-endoscopy application to set `use_rdma` as `true`.

```
vi ${PATH_TO_SDK_REPOSITORY}/apps/hi_speed_endoscopy_gxf/hi_speed_endoscopy.yaml
# Set `use_rdma` as `true`
```

Save the file and build the application again. To build the source locally please refer to README.md.

4. To run application, use below run command:

```
cd ${PATH_TO_SDK_REPOSITORY}/build
sudo MELLANOX_RINGBUFF_FACTOR=14 ./apps/hi_speed_endoscopy_gxf/hi_speed_endoscopy
```

**Note:** The MELLANOX\_RINGBUFF\_FACTOR is used by EVT driver to decide how much BAR1 size memory would be used on the dGPU. It can be changed to different number based for different use cases.

## 6.2.3 Enabling Exclusive Display Mode

By default, the application uses a borderless fullscreen window which is managed by the window manager. Because the window manager also manages other applications, the hi-speed-endoscopy application may suffer a performance hit. To improve performance, exclusive display mode can be used which allows the application to bypass the window manager and render directly to the display.

To enable exclusive display follow below steps.

1. Find the name of the display connected using `xrandr`. As an example the output of the `xrandr` could look like below.

```
$ xrandr
Screen 0: minimum 8 x 8, current 4480 x 1440, maximum 32767 x 32767
DP-0 disconnected (normal left inverted right x axis y axis)
DP-1 disconnected (normal left inverted right x axis y axis)
DP-2 connected primary 2560x1440+1920+0 (normal left inverted right x axis y axis) 600mm
↳x 340mm
   2560x1440    59.98 + 239.97* 199.99  144.00  120.00  99.95
   1024x768     60.00
   800x600      60.32
   640x480      59.94
DP-3 disconnected (normal left inverted right x axis y axis)
DP-4 disconnected (normal left inverted right x axis y axis)
DP-5 disconnected (normal left inverted right x axis y axis)
DP-6 disconnected (normal left inverted right x axis y axis)
DP-7 connected 1920x1080+0+0 (normal left inverted right x axis y axis) 543mm x 302mm
   1920x1080    60.00*+ 119.88  59.94  50.00  23.98
   1280x720     59.94  50.00
   1024x768     60.00
```

(continues on next page)

(continued from previous page)

800x600	60.32	
720x576	50.00	
720x480	59.94	
640x480	59.94	59.93

USB-C-0 disconnected (normal left inverted right x axis y axis)

In this example DP-2 is the name of the display connected to the Clara devkit that will be used for exclusive display.

The name of the display can also be found in tab X Server Display Configuration in nvidia-settings. See figure *Enable G-SYNC for the current display*.

2. Update the hi-speed-endoscopy application to use the display name, resolution and framerate used by the connected display. In addition to that, add `use_exclusive_display` as `true` otherwise the exclusive display will not be enabled in the app.

```
vi ${PATH_TO_SDK_REPOSITORY}/apps/hi_speed_endoscopy_gxf/hi_speed_endoscopy.yaml
# Add below lines as parameters for entity holoviz and component_
↪nvidia::holoscan::HolovizViewer
# display_name: DP-2
# width: 2560
# height: 1440
# framerate: 240
# use_exclusive_display: true
```

Save the file and build the application again. To build the source locally please refer to README.md.

3. If a **single display** is connected, ssh to Clara devkit and stop the X server.

```
ssh ${DEVKIT_USER}@${IP_ADDRESS}
export DISPLAY=:1
xhost +
sudo systemctl stop display-manager
```

#### Note:

- Set `${DEVKIT_USER}` and `${IP_ADDRESS}` to your Clara devkit credentials.
- Display `:1` is just an example, it could be `:0` or different.
- To start the display manager, after done running the application, use command:

```
sudo systemctl start display-manager
```

If **multiple displays** are connected, the display to be used in exclusive mode needs to be disabled in the nvidia-settings. Open the X Server Display Configuration tab, select the display and under Configuration select Disabled. Press Apply.

**Note:** To enable the display after done running the application, start nvidia-settings. Open the X Server Display Configuration tab, select the display and under Configuration select X screen 0. Press Apply.

Now, run the application.

## 6.3 Ultrasound Segmentation Application & Customization

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

This application performs an automatic segmentation of the spine from a trained AI model for the purpose of scoliosis visualization and measurement.

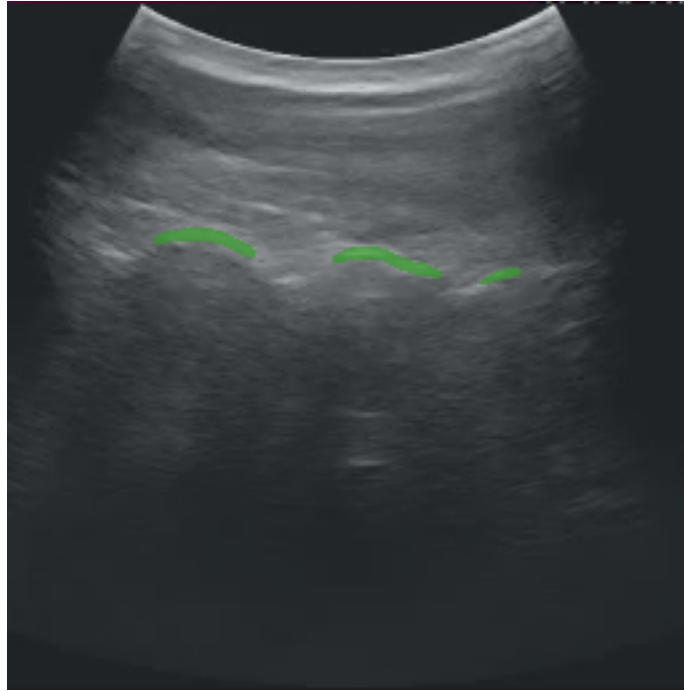


Fig. 6.7: Spine segmentation of ultrasound data

### 6.3.1 Input source: Video Stream Replayer

The replayer pipeline is defined in `apps/ultrasound_segmentation/segmentation_replayer.yaml` in [Holoscan Embedded SDK Github Repository](#).

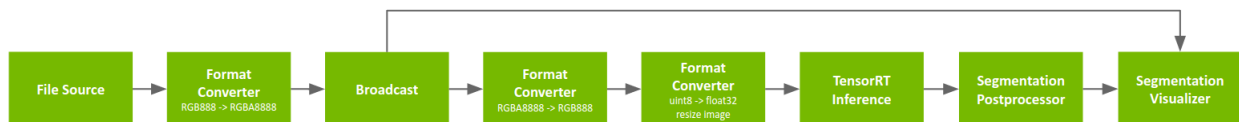


Fig. 6.8: Segmentation application with replay from file

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

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

- In the *inference* branch the video frames are converted to floating-point precision using the *format converter*, pixel-wise segmentation is performed, and the segmentation result is post-processed for the visualizer.

- The *visualizer* receives the original frame as well as the result of the inference branch to show an overlay.

---

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

In the runtime container (from NGC):

```
cd /opt/holoscan_sdk
./apps/ultrasound_segmentation_gxf/segmentation_replayer
```

In the development container (from source):

```
cd /workspace/holoscan-sdk/build
./apps/ultrasound_segmentation_gxf/segmentation_replayer
```

---

### 6.3.2 Input source: AJA

The AJA pipeline is defined in `apps/ultrasound_segmentation/segmentation_aja.yaml` in [Holoscan Embedded SDK Github Repository](#).

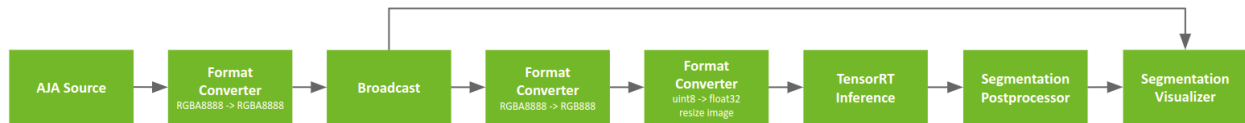


Fig. 6.9: AJA segmentation app

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

---

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

In the runtime container (from NGC):

```
cd /opt/holoscan_sdk
./apps/ultrasound_segmentation_gxf/segmentation_aja
```

In the development container (from source):

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

---

### 6.3.3 Bring Your Own Model (BYOM) - Customizing the Ultrasound Segmentation Application For Your Model

This section shows how the user can easily modify the [ultrasound segmentation app](#) to run a different segmentation model, even of an entirely different modality. In this use case we will use the ultrasound application to implement a polyp segmentation model to run on a Colonoscopy sample video.

At this time the runtime containers contain only binaries of the sample applications, meaning users may not modify the extensions. However, the users can substitute the ultrasound model with their own and add, remove, or replace the extensions used in the application.

As a first step, please go to the [Colonoscopy Sample Application Data NGC Resource](#) to download the model and video data.

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

The sample ultrasound segmentation model expects a gray-scale image of 256 x 256 and outputs a semantic segmentation of the same size with two channels representing bone contours with hyperechoic lines (foreground) and hyperechoic acoustic shadow (background).

**Warning:** Currently, the sample apps are able to load ONNX models, or [TensorRT](#) engine files built for the architecture on which you will be running the model only. **TRT engines are automatically generated from ONNX by the application when it is run.**

If you are converting your model from PyTorch to ONNX, chances are your input is NCHW, and will need to be converted to NHWC. An example transformation script is included with the colonoscopy sample downloaded above, and is found inside the resource as `model/graph_surgeon.py`. You may need to modify the dimensions as needed before modifying your model as:

```
python graph_surgeon.py {YOUR_MODEL_NAME}.onnx {DESIRED_OUTPUT_NAME}.onnx
```

Note that this step is optional if you are directly using ONNX models.

To get a better understanding of the model, applications such as [netron.app](#) can be used.

We will now substitute the model and sample video to inference upon as follows.

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

```
docker run -it --rm --runtime=nvidia \
  -e NVIDIA_DRIVER_CAPABILITIES=graphics,video,compute,utility \
  -v ${my_model_path_dir}:/workspace/my_model \
  -v ${my_data_path_dir}:/workspace/my_data \
  -v /tmp/.X11-unix:/tmp/.X11-unix \
  -e DISPLAY=${DISPLAY} \
  nvcr.io/nvidia/clara-holoscan/clara_holoscan_sample_runtime:v0.3.0-arm64
```

2. Check that the model and data correctly appear under `/workspace/my_model` and `/workspace/my_data`.
3. Now we are ready to make the required modifications to the ultrasound sample application to have the colonoscopy model load.

```
cd /opt/holoscan_sdk
vi ./apps/ultrasound_segmentation_gxf/segmentation_replayer.yaml
```

4. In the editor navigate to the first entity source, and under type `nvidia::holoscan::stream_playback::VideoStreamReplay` we will modify the following for our input video:

- a. `directory:` `"/workspace/my_data"`
- b. `basename:` `"colonoscopy"`

**Tip:** In general, to be able to play a desired video through a custom model we first need to convert the video file into a GXF replayable tensor format. This step has already been done for the colonoscopy example, but for a custom video perform the following actions inside the container.

```
apt update && DEBIAN_FRONTEND=noninteractive apt install -y ffmpeg
cd /workspace
git clone https://github.com/NVIDIA/clara-holoscan-embedded-sdk.git
cd clara-holoscan-embedded-sdk/scripts
ffmpeg -i /workspace/my_data/${my_video} -pix_fmt rgb24 -f rawvideo pipe:1 |
python3 convert_video_to_gxf_entities.py --width ${my_width} --height ${my_height}
--directory /workspace/my_data --basename my_video
```

The above commands should yield two Holoscan tensor replayer files in `/workspace/my_data`, namely `my_video.gxf_index` and `my_video.gxf_entities`.

5. In the editor navigate to the `segmentation_preprocessor` entity. Under type `nvidia::holoscan::formatconverter::FormatConverter` we will modify the following parameters to fit the input dimensions of our colonoscopy model:

- a. `resize_width:` `512`
- b. `resize_height:` `512`

6. In the editor navigate to the `segmentation_inference` entity. We will modify the `nvidia::gxf::TensorRtInference` type where we want to specify the input and output names.

- a. Specify the location of your ONNX files as:

```
model_file_path: /workspace/my_model/colon.onnx
```

- b. Specify the location of TensorRT engines as:

```
engine_cache_dir: /workspace/my_model/cache
```

- c. Specify the names of the inputs specified in your model under `input_binding_names`. In the case of ONNX models converted from PyTorch inputs names take the form `INPUT__0`.

- d. Specify the names of the inputs specified in your model under `output_binding_names`. In the case of ONNX models converted from PyTorch and then the `graph_surgeon.py` conversion, names take the form `output_old`.

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

```
- type: nvidia::gxf::TensorRtInference
  parameters:
    input_binding_names:
      - MY_MODEL_INPUT_NAME
```

(continues on next page)

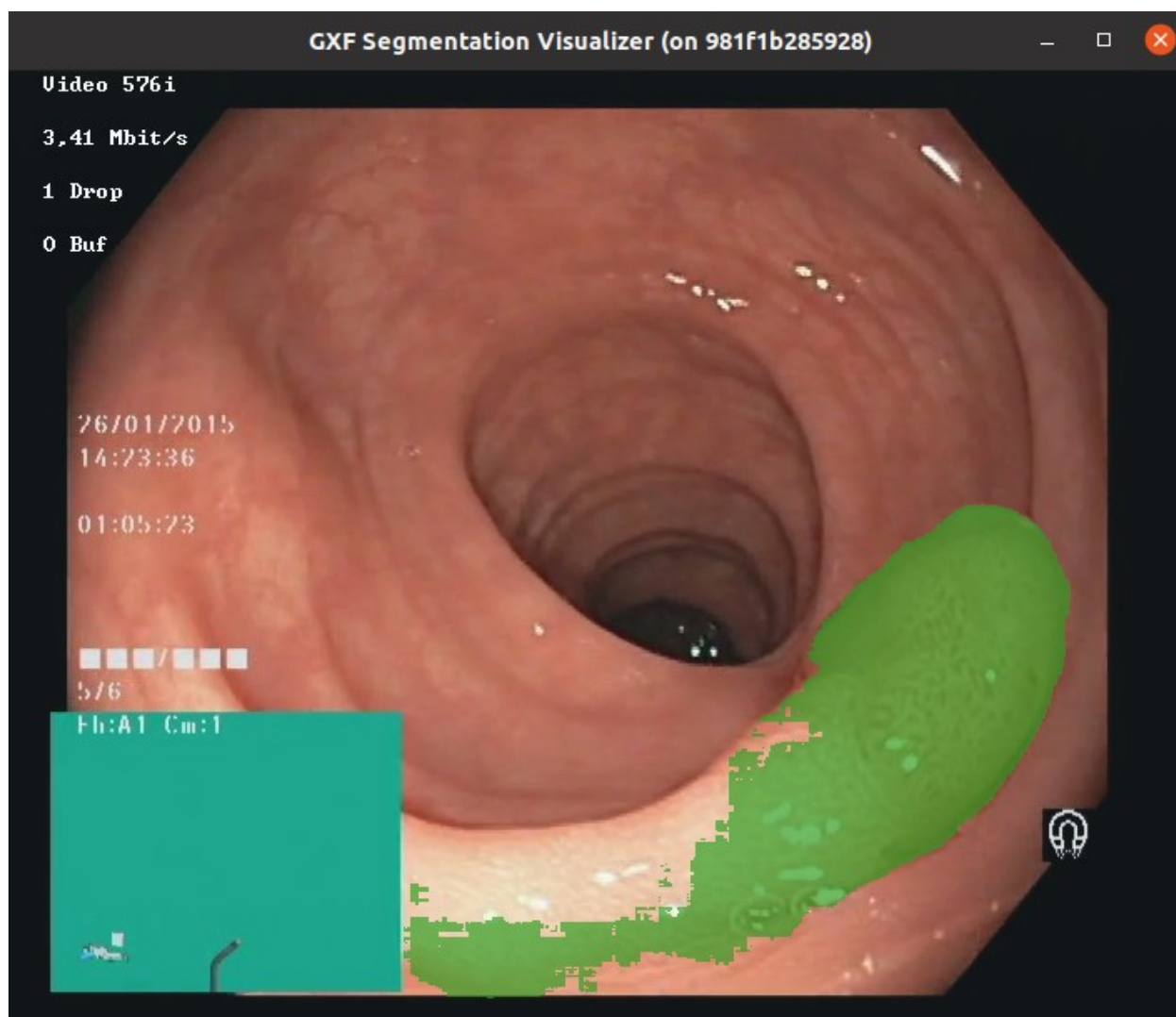
(continued from previous page)

```
output_binding_names:  
- MY_MODEL_OUTPUT_NAME
```

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

7. In the entity `segmentation_postprocessor`, make the following change: `network_output_type: sigmoid`.
8. In the entity `segmentation_visualizer`, we will make the following changes under `nvidia::holoscan::segmentation_visualizer::Visualizer` to correctly input the dimensions of our video and the output dimensions of our model:
  - a. `image_width: 720`
  - b. `image_height: 576`
  - c. `class_index_width: 512`
  - d. `class_index_height: 512`
9. Run the application with the new model and data.

```
cd /opt/holoscan_sdk  
./apps/ultrasound_segmentation_gxf/segmentation_replayer
```





## CLARA HOLOSCAN GXF EXTENSIONS

### 7.1 GXF Built-in Extensions

#### 7.1.1 Std

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

Please see *GXF Standard Extension* for more details.

##### `nvidia::gxf::Broadcast`

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

##### Parameters

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

#### 7.1.2 Serialization

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

Please see *GXF Serialization Extension* for more details.

### `nvidia::gxf::EntityRecorder`

Serializes incoming messages and writes them to a file.

#### Parameters

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

## 7.2 Holoscan SDK GXF Extensions

### 7.2.1 V4L2

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

### `nvidia::holoscan::V4L2Source`

V4L2 Source Codelet.

#### Parameters

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

- **numBuffers**: Number of V4L2 buffers to use (default: 2)
  - type: `uint32_t`

### 7.2.2 AJA

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

#### `nvidia::holoscan::AJASource`

AJA Source Codelet.

#### Parameters

- **signal**: Output signal
  - type: `gxf::Handle<gxf::Transmitter>`
- **device**: Device specifier (default: `0`)
  - type: `std::string`
- **channel**: NTV2Channel to use (default: `0` (NTV2\_CHANNEL1))
  - type: `NTV2Channel`
- **width**: Width of the stream (default: `1920`)
  - type: `uint32_t`
- **height**: Height of the stream (default: `1080`)
  - type: `uint32_t`
- **framerate**: Framerate of the stream (default: `60`)
  - type: `uint32_t`
- **rdma**: Enable RDMA (default: `false`)
  - type: `bool`

### 7.2.3 Stream Playback

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

## `nvidia::holoscan::stream_playback::VideoStreamReplayer`

VideoStreamReplayer codelet.

### Parameters

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

**nvidia::holoscan::stream\_playback::VideoStreamSerializer**

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

**7.2.4 Format Converter**

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

**nvidia::holoscan::formatconverter::FormatConverter**

This codelet executes the following processes:

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

## Parameters

- **in**: Input channel
  - type: `gxf::Handle<gxf::Receiver>`
- **in\_tensor\_name**: Name of the input tensor (default: "")
  - type: `std::string`
- **in\_dtype**: Source data type (default: "")
  - type: `std::string`
  - If not specified, input data type is guessed from the input tensor.
- **out**: Output channel
  - type: `gxf::Handle<gxf::Transmitter>`
- **out\_tensor\_name**: Name of the output tensor (default: "")
  - type: `std::string`
- **out\_dtype**: Destination data type
  - type: `std::string`
- **scale\_min**: Minimum value of the scale (default: 0.f)
  - type: `float`
- **scale\_max**: Maximum value of the scale (default: 1.f)
  - type: `float`
- **alpha\_value**: Alpha value that can be used to fill the alpha channel when converting RGB888 to RGBA8888 (default: 255)
  - type: `uint8_t`
- **resize\_width**: Width for resize. No actions if this value is zero (default: 0)
  - type: `int32_t`
- **resize\_height**: Height for resize. No actions if this value is zero (default: 0)
  - type: `int32_t`
- **resize\_mode**: Mode for resize. 4 (NPPI\_INTER\_CUBIC) if this value is zero (default: 0)
  - type: `int32_t`

```

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

```

(continues on next page)

(continued from previous page)

```

        16 = NPPI_INTER_LANCZOS           /**< Lanczos filtering. */
        17 = NPPI_INTER_LANCZOS3_ADVANCED /**< Generic Lanczos filtering.
↪with order 3. */
(int)0x80000000 = NPPI_SMOOTH_EDGE       /**< Smooth edge filtering.
↪(0x80000000 = 134217728)*/

```

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

## 7.2.5 TensorRT

The `tensor_rt` extension provides the TensorRT inference codelet.

### `nvidia::holoscan::TensorRtInference`

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

### Parameters

- **model\_file\_path**: Path to ONNX model to be loaded
  - type: `std::string`
- **engine\_cache\_dir**: Path to a directory containing cached generated engines to be serialized and loaded from
  - type: `std::string`
- **plugins\_lib\_namespace**: Namespace used to register all the plugins in this library (default: "")
  - type: `std::string`
- **force\_engine\_update**: Always update engine regard less of existing engine file. Such conversion may take minutes (default: false)
  - type: `bool`
- **input\_tensor\_names**: Names of input tensors in the order to be fed into the model
  - type: `std::vector<std::string>`
- **input\_binding\_names**: Names of input bindings as in the model in the same order of what is provided in `input_tensor_names`
  - type: `std::vector<std::string>`
- **output\_tensor\_names**: Names of output tensors in the order to be retrieved from the model
  - type: `std::vector<std::string>`
- **output\_binding\_names**: Names of output bindings in the model in the same order of of what is provided in `output_tensor_names`

- type: `std::vector<std::string>`
- **pool**: Allocator instance for output tensors
  - type: `gxf::Handle<gxf::Allocator>`
- **cuda\_stream\_pool**: Instance of `gxf::CudaStreamPool` to allocate CUDA stream
  - type: `gxf::Handle<gxf::CudaStreamPool>`
- **max\_workspace\_size**: Size of working space in bytes (default: 67108864 (64MB))
  - type: `int64_t`
- **dla\_core**: DLA Core to use. Fallback to GPU is always enabled. Default to use GPU only (optional)
  - type: `int64_t`
- **max\_batch\_size**: Maximum possible batch size in case the first dimension is dynamic and used as batch size (default: 1)
  - type: `int32_t`
- **enable\_fp16\_**: Enable inference with FP16 and FP32 fallback (default: `false`)
  - type: `bool`
- **verbose**: Enable verbose logging on console (default: `false`)
  - type: `bool`
- **relaxed\_dimension\_check**: Ignore dimensions of 1 for input tensor dimension check (default: `true`)
  - type: `bool`
- **clock**: Instance of clock for publish time (optional)
  - type: `gxf::Handle<gxf::Clock>`
- **rx**: List of receivers to take input tensors
  - type: `std::vector<gxf::Handle<gxf::Receiver>>`
- **tx**: Transmitter to publish output tensors
  - type: `gxf::Handle<gxf::Transmitter>`

## 7.2.6 OpenGL

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

### `nvidia::holoscan::OpenGLRenderer`

OpenGL Renderer Codelet.



## Parameters

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

## 7.2.7 Segmentation Post Processor

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

**`nvidia::holoscan::segmentation_postprocessor::Postprocessor`**

Segmentation Postprocessor codelet.

## Parameters

- **in**: Input channel
  - type: `gxf::Handle<gxf::Receiver>`
- **in\_tensor\_name**: Name of the input tensor (default: "")
  - type: `std::string`
- **network\_output\_type**: Network output type (default: `softmax`)
  - type: `std::string`
- **out**: Output channel
  - type: `gxf::Handle<gxf::Transmitter>`
- **data\_format**: Data format of network output (default: `hwc`)
  - type: `std::string`
- **allocator**: Output Allocator
  - type: `gxf::Handle<gxf::Allocator>`

## 7.2.8 Segmentation Visualizer

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

**`nvidia::holoscan::segmentation_visualizer::Visualizer`**

OpenGL Segmentation Visualizer codelet.

### Parameters

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

## 7.2.9 Custom LSTM Inference

The `custom_lstm_inference` extension provides LSTM (Long-Short Term Memory) stateful inference module using TensorRT.

**nvidia::holoscan::custom\_lstm\_inference::TensorRtInference**

Codelet, taking input tensors and feeding them into TensorRT for LSTM inference.

This implementation is based on `nvidia::gxf::TensorRtInference`. `input_state_tensor_names` and `output_state_tensor_names` parameters are added to specify tensor names for states in LSTM model.

**Parameters**

- **model\_file\_path**: Path to ONNX model to be loaded
  - type: `std::string`
- **engine\_cache\_dir**: Path to a directory containing cached generated engines to be serialized and loaded from
  - type: `std::string`
- **plugins\_lib\_namespace**: Namespace used to register all the plugins in this library (default: "")
  - type: `std::string`
- **force\_engine\_update**: Always update engine regard less of existing engine file. Such conversion may take minutes (default: false)
  - type: `bool`
- **input\_tensor\_names**: Names of input tensors in the order to be fed into the model
  - type: `std::vector<std::string>`
- **input\_state\_tensor\_names**: Names of input state tensors that are used internally by TensorRT
  - type: `std::vector<std::string>`
- **input\_binding\_names**: Names of input bindings as in the model in the same order of what is provided in `input_tensor_names`
  - type: `std::vector<std::string>`
- **output\_tensor\_names**: Names of output tensors in the order to be retrieved from the model
  - type: `std::vector<std::string>`
- **input\_state\_tensor\_names**: Names of output state tensors that are used internally by TensorRT
  - type: `std::vector<std::string>`
- **output\_binding\_names**: Names of output bindings in the model in the same order of of what is provided in `output_tensor_names`
  - type: `std::vector<std::string>`
- **pool**: Allocator instance for output tensors
  - type: `gxf::Handle<gxf::Allocator>`
- **cuda\_stream\_pool**: Instance of `gxf::CudaStreamPool` to allocate CUDA stream
  - type: `gxf::Handle<gxf::CudaStreamPool>`
- **max\_workspace\_size**: Size of working space in bytes (default: 671088641 (64MB))
  - type: `int64_t`
- **dla\_core**: DLA Core to use. Fallback to GPU is always enabled. Default to use GPU only (optional)
  - type: `int64_t`

- **max\_batch\_size**: Maximum possible batch size in case the first dimension is dynamic and used as batch size (default: 1)
  - type: `int32_t`
- **enable\_fp16\_**: Enable inference with FP16 and FP32 fallback (default: `false`)
  - type: `bool`
- **verbose**: Enable verbose logging on console (default: `false`)
  - type: `bool`
- **relaxed\_dimension\_check**: Ignore dimensions of 1 for input tensor dimension check (default: `true`)
  - type: `bool`
- **clock**: Instance of clock for publish time (optional)
  - type: `gxf::Handle<gxf::Clock>`
- **rx**: List of receivers to take input tensors
  - type: `std::vector<gxf::Handle<gxf::Receiver>>`
- **tx**: Transmitter to publish output tensors
  - type: `gxf::Handle<gxf::Transmitter>`

## 7.2.10 Visualizer Tool Tracking

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

### `nvidia::holoscan::visualizer_tool_tracking::Sink`

Surgical Tool Tracking Viz codelet.

#### Parameters

- **videoframe\_vertex\_shader\_path**: Path to vertex shader to be loaded
  - type: `std::string`
- **videoframe\_fragment\_shader\_path**: Path to fragment shader to be loaded
  - type: `std::string`
- **tooltip\_vertex\_shader\_path**: Path to vertex shader to be loaded
  - type: `std::string`
- **tooltip\_fragment\_shader\_path**: Path to fragment shader to be loaded
  - type: `std::string`
- **num\_tool\_classes**: Number of different tool classes
  - type: `int32_t`
- **num\_tool\_pos\_components**: Number of components of the tool position vector (default: 2)
  - type: `int32_t`

- **tool\_tip\_colors:** Color of the tool tips, a list of RGB values with components between 0 and 1 (default: 12 qualitative classes color scheme from colorbrewer2)
  - type: `std::vector<std::vector<float>>`
- **overlay\_img\_vertex\_shader\_path:** Path to vertex shader to be loaded
  - type: `std::string`
- **overlay\_img\_fragment\_shader\_path:** Path to fragment shader to be loaded
  - type: `std::string`
- **overlay\_img\_width:** Width of overlay image
  - type: `int32_t`
- **overlay\_img\_height:** Height of overlay image
  - type: `int32_t`
- **overlay\_img\_channels:** Number of Overlay Image Channels
  - type: `int32_t`
- **overlay\_img\_layers:** Number of Overlay Image Layers
  - type: `int32_t`
- **overlay\_img\_colors:** Color of the image overlays, a list of RGB values with components between 0 and 1 (default: 12 qualitative classes color scheme from colorbrewer2)
  - type: `std::vector<std::vector<float>>`
- **tool\_labels:** List of tool names (default: [])
  - type: `std::vector<std::string>`
- **label\_sans\_font\_path:** Path for sans font to be loaded
  - type: `std::string`
- **label\_sans\_bold\_font\_path:** Path for sans bold font to be loaded
  - type: `std::string`
- **in:** List of input channels
  - type: `std::vector<gxf::Handle<gxf::Receiver>>`
- **in\_tensor\_names:** Names of input tensors (default: "")
  - type: `std::vector<std::string>`
- **in\_width:** Width of the image (default: 640)
  - type: `int32_t`
- **in\_height:** Height of the image (default: 480)
  - type: `int32_t`
- **in\_channels:** Number of channels (default: 3)
  - type: `int16_t`
- **in\_bytes\_per\_pixel:** Number of bytes per pixel of the image (default: 1)
  - type: `uint8_t`
- **alpha\_value:** Alpha value that can be used when converting RGB888 to RGBA8888 (default: 255)

- type: `uint8_t`
- **pool**: Pool to allocate the output message.
  - type: `gxf::Handle<gxf::Allocator>`
- **window\_close\_scheduling\_term**: `BooleanSchedulingTerm` to stop the codelet from ticking after all messages are published.
  - type: `gxf::Handle<gxf::BooleanSchedulingTerm>`

## 7.2.11 Holoscan Test Mock

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

### `nvidia::holoscan::mocks::VideoBufferMock`

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

#### Parameters

- **in\_width**: Width of the image (default: 640)
  - type: `int32_t`
- **in\_height**: Height of the image (default: 480)
  - type: `int32_t`
- **in\_channels**: Number of input channels (default: 3)
  - type: `int16_t`
- **in\_bytes\_per\_pixel**: Number of bytes per pixel of the image (default: 1)
  - type: `int8_t`
- **out\_tensor\_name**: Name of the output tensor (default: "")
  - type: `std::string`
- **out**: Output channel
  - type: `gxf::Handle<gxf::Transmitter>`
- **pool**: Pool to allocate the output message
  - type: `gxf::Handle<gxf::Allocator>`

### 7.2.12 Emergent

The `emergent_source` extension supports an Emergent Vision Technologies camera as the video source. The datastream from this camera is transferred through Mellanox ConnectX SmartNIC using Rivermax SDK.

**`nvidia::holoscan::EmergentSource`**

Emergent Source codelet

#### Parameters

- **signal**: Output signal
  - type: `gxf::Handle<gxf::Transmitter>`
- **width**: Width of the stream (default: 4200)
  - type: `uint32_t`
- **height**: Height of the stream (default: 2160)
  - type: `uint32_t`
- **framerate**: Framerate of the stream (default: 240)
  - type: `uint32_t`
- **rdma**: Enable RDMA (default: false)
  - type: `bool`

### 7.2.13 Bayer Demosaic

The `bayer_demosaic` extension performs color filter array (CFA) interpolation for 1-channel inputs of 8 or 16-bit unsigned integer and outputs an RGB or RGBA image.

**`nvidia::holoscan::BayerDemosaic`**

Bayer Demosaic codelet

#### Parameters

- **receiver**: Input queue to component accepting `gxf::Tensor` or `gxf::VideoBuffer`
  - type: `gxf::Handle<gxf::Receiver>`
- **transmitter**: Output queue of component for `gxf::Tensor` types
  - type: `gxf::Handle<gxf::Transmitter>`
- **in\_tensor\_name**: Name of the expected input tensor (default: "")
  - type: `std::string`
- **out\_tensor\_name**: Name of the output tensor generated by component (default: "")
  - type: `std::string`

- **pool**: Pool to allocate output message contents
  - type: `gxf::Handle<gxf::Allocator>`
- **cuda\_stream\_pool**: Instance of `gxf::CudaStreamPool` to allocate CUDA streams
  - type: `gxf::Handle<gxf::CudaStreamPool>`
- **interpolation\_mode**: Interpolation model to be used for demosaicing (default: `0` (UNDEFINED)). For details on possible interpolation modes consult [NPP documentation](#). Currently NPP *only* supports UNDEFINED as interpolation mode
  - type: `int`
- **bayer\_grid\_pos**: Bayer grid position (default: `2` (GBRG)). For details on possible Bayer grid position values consult [NPP documentation](#)
  - type: `int`
- **generate\_alpha**: Boolean value indicating whether output image should be RGB (`false`) or RGBA (`true`) (default: `false`)
  - type: `bool`
- **alpha\_value**: Alpha value desired at the output of the component when `generate_alpha` is set to `true`
  - type: `int`

## 7.2.14 Holoviz Viewer

The `holoviz_viewer` extension provides a high-speed viewer (built on Vulkan SDK) to visualize RGB or RGBA images.

### `nvidia::holoscan::HolovizViewer`

Holoviz Viewer codelet

#### Parameters

- **receiver**: Input queue to component accepting `gxf::Tensor`
  - type: `gxf::Handle<gxf::Receiver>`
- **input\_image\_name**: Name of the expected input tensor (default: `""`)
  - type: `std::string`
- **window\_title**: Title on window canvas (default: `Holoviz Viewer`)
  - type: `std::string`
- **display\_name**: Name of Display as shown with `xrandr` (default: `DP-0`)
  - type: `std::string`
- **width**: Width of the stream (default: `2560`)
  - type: `uint32_t`
- **height**: Height of the stream (default: `1440`)
  - type: `uint32_t`



- **framerate**: Framerate of the stream (default: 240)
  - type: uint32\_t
- **use\_exclusive\_display**: Enable exclusive display (default: false)
  - type: bool



## CLARA HOLOVIZ

### 8.1 Overview

Clara Holoviz is a SDK used by Clara Holoscan for visualizing data. Clara Holoviz composites real time streams of frames with multiple different other layers like segmentation mask layers, geometry layers and GUI layers.

For maximum performance Clara Holoviz makes use of [Vulkan](#), which is already installed as part of the Nvidia driver.

### 8.2 Concepts

Clara Holoviz uses the concept of the immediate mode design pattern for its API, inspired by the [Dear ImGui](#) library. The difference to the retained mode, for which most APIs are designed for, is, that there are no objects created and stored by the application. This makes it easy to quickly build and change an Clara Holoviz app.

### 8.3 Usage

The code below creates a window and displays an image.

```
namespace viz = clara::holoviz;

viz::Init("Holoviz Example");

viz::Begin();
viz::BeginImageLayer();
viz::ImageHost(width, height, viz::ImageFormat::R8G8B8A8_UNORM, image_data);
viz::EndLayer();
viz::End();
```

Result:

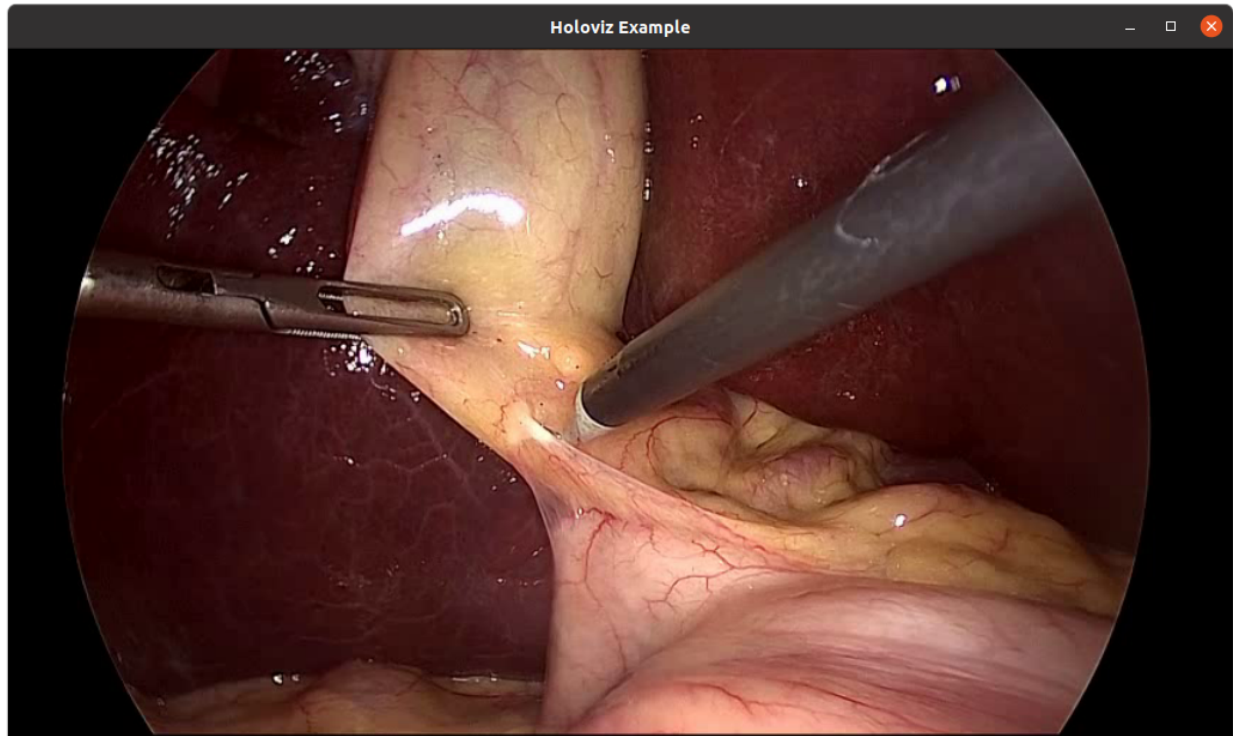


Fig. 8.1: Holoviz example app

## 8.4 Layers

The core entity of Clara Holoviz are layers. A layer is a two-dimensional image object, multiple layers are composited to form the final output.

These layer types are supported by Clara HoloViz:

- image layer
- geometry layer
- GUI layer

Layers have opacity and priority. The priority determines the rendering order of the layers. Opacity is used to blend transparent layers over other layers.

## VIDEO PIPELINE LATENCY TOOL

The Clara Developer Kits excel as a high-performance computing platform by combining high-bandwidth video I/O components and the compute capabilities of an NVIDIA GPU to meet the needs of the most demanding video processing and inference applications.

For many video processing applications located at the edge—especially those designed to augment medical instruments and aid live medical procedures—minimizing the latency added between image capture and display, often referred to as the end-to-end latency, is of the utmost importance.

While it is generally easy to measure the individual processing time of an isolated compute or inference algorithm by simply measuring the time that it takes for a single frame (or a sequence of frames) to be processed, it is not always so easy to measure the complete end-to-end latency when the video capture and display is incorporated as this usually involves external capture hardware (e.g. cameras and other sensors) and displays.

In order to establish a baseline measurement of the minimal end-to-end latency that can be achieved with the Clara Developer Kits and various video I/O hardware and software components, the Clara Holoscan SDK includes a sample latency measurement tool.

### 9.1 Requirements

#### 9.1.1 Hardware

The latency measurement tool requires the use of a Clara AGX Developer Kit in dGPU mode, and operates by having an output component generate a sequence of known video frames that are then transferred back to an input component using a physical loopback cable.

Testing the latency of any of the HDMI modes that output from the GPU requires a DisplayPort to HDMI adapter or cable (see *Example Configurations*, below). Note that this cable must support the mode that is being tested — for example, the UHD mode will only be available if the cable is advertised to support “4K Ultra HD (3840 x 2160) at 60 Hz”.

Testing the latency of an optional AJA Video Systems device requires a supported AJA SDI or HDMI capture device (see *AJA Video Systems* for the list of supported devices), along with the HDMI or SDI cable that is required for the configuration that is being tested (see *Example Configurations*, below).

### 9.1.2 Software

The following additional software components are required and are installed either by the Clara Holoscan SDK installation or in the *Installation* steps below:

- CUDA 11.1 or newer (<https://developer.nvidia.com/cuda-toolkit>)
- CMake 3.10 or newer (<https://cmake.org/>)
- GLFW 3.2 or newer (<https://www.glfw.org/>)
- GStreamer 1.14 or newer (<https://gstreamer.freedesktop.org/>)
- GTK 3.22 or newer (<https://www.gtk.org/>)
- pkg-config 0.29 or newer (<https://www.freedesktop.org/wiki/Software/pkg-config/>)

The following is optional to enable DeepStream support (for RDMA support from the *GStreamer Producer*):

- DeepStream 5.1 or newer (<https://developer.nvidia.com/deepstream-sdk>)

The following is optional to enable AJA Video Systems support:

- AJA NTV2 SDK 16.1 or newer (See *AJA Video Systems* for details on installing the AJA NTV2 SDK and drivers).

## 9.2 Installation

### 9.2.1 Downloading the Source

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

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

### 9.2.2 Installing Software Requirements

CUDA is installed automatically during the dGPU setup. The rest of the software requirements are installed with the following:

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

### 9.2.3 Building

Start by creating a build folder within the loopback-latency directory:

```
$ cd clara-holoscan-perf-tools/loopback-latency
$ mkdir build
$ cd build
```

CMake is then used to build the tool and output the loopback-latency binary to the current directory:

```
$ cmake ..
$ make -j
```

**Note:** If the error `No CMAKE_CUDA_COMPILER could be found` is encountered, make sure that the `nvcc` executable can be found by adding the CUDA runtime location to your `PATH` variable:

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

#### Enabling DeepStream Support

DeepStream support enables RDMA when using the *GStreamer Producer*. To enable DeepStream support, the `DEEPPSTREAM_SDK` path must be appended to the `cmake` command with the location of the DeepStream SDK. For example, when building against DeepStream 5.1, replace the `cmake` command above with the following:

```
$ cmake -DDEEPPSTREAM_SDK=/opt/nvidia/deepstream/deepstream-5.1 ..
```

#### Enabling AJA Support

To enable AJA support, the `NTV2_SDK` path must be appended to the `cmake` command with the location of the NTV2 SDK in which both the headers and compiled libraries (i.e. `libajantv2`) exist. For example, if the NTV2 SDK is in `/home/nvidia/ntv2`, replace the `cmake` command above with the following:

```
$ cmake -DNTV2_SDK=/home/nvidia/ntv2 ..
```

## 9.3 Example Configurations

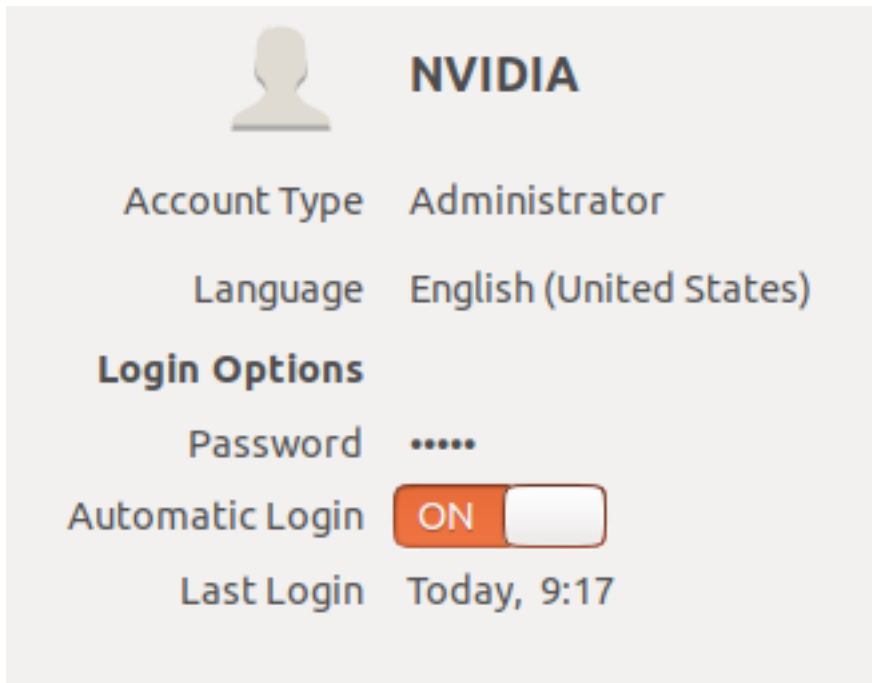
**Note:** When testing a configuration that outputs from the GPU, the tool currently only supports a display-less environment in which the loopback cable is the only cable attached to the GPU. Because of this, any tests that output from the GPU must be performed using a remote connection such as SSH from another machine. When this is the case, make sure that the `DISPLAY` environment variable is set to the ID of the X11 display you are using (e.g. in `~/.bashrc`):

```
export DISPLAY=:0
```

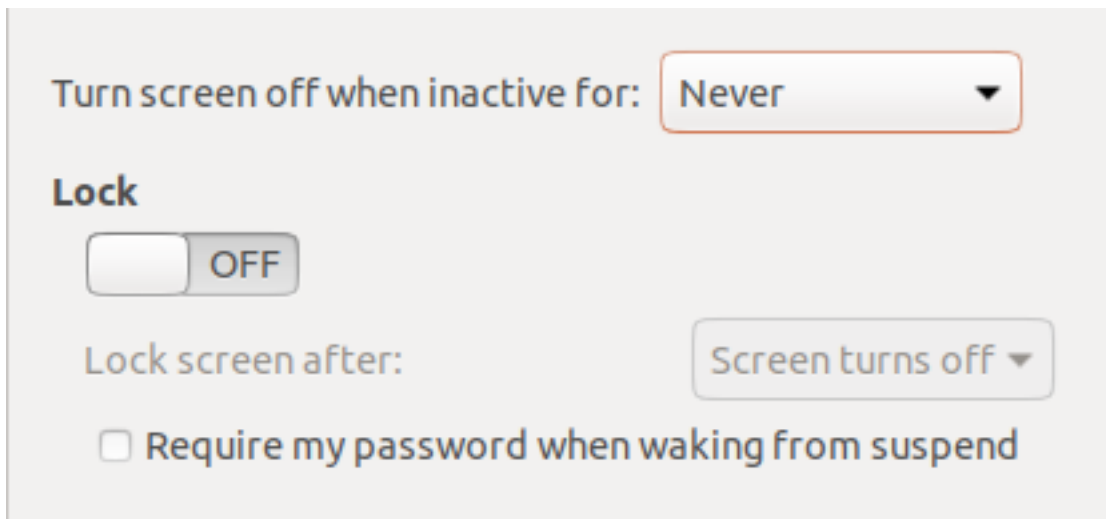
It is also required that the system is logged into the desktop and that the system does not sleep or lock when the latency tool is being used. This can be done by temporarily attaching a display to the system to do the following:

1. Open the **Ubuntu System Settings**

2. Open **User Accounts**, click **Unlock** at the top right, and enable **Automatic Login**:



3. Return to **All Settings** (top left), open **Brightness & Lock**, and disable sleep and lock as pictured:



Make sure that the display is detached again after making these changes.

See the [Producers](#) section for more details about GPU-based producers (i.e. [OpenGL](#) and [GStreamer](#)).

---



### 9.3.1 GPU To Onboard HDMI Capture Card

In this configuration, a DisplayPort to HDMI cable is connected from the GPU to the onboard HDMI capture card. This configuration supports the *OpenGL* and *GStreamer* producers, and the *V4L2* and *GStreamer* consumers.



Fig. 9.1: DP-to-HDMI Cable Between GPU and Onboard HDMI Capture Card

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

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

### 9.3.2 GPU to AJA HDMI Capture Card

In this configuration, a DisplayPort to HDMI cable is connected from the GPU to an HDMI input channel on an AJA capture card. This configuration supports the *OpenGL* and *GStreamer* producers, and the *AJA consumer* using an AJA HDMI capture card.



Fig. 9.2: DP-to-HDMI Cable Between GPU and AJA KONA HDMI Capture Card (Channel 1)

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

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

### 9.3.3 AJA SDI to AJA SDI

In this configuration, an SDI cable is attached between either two channels on the same device or between two separate devices (pictured is a loopback between two channels of a single device). This configuration must use the *AJA producer* and *AJA consumer*.

For example, the following can be used to measure the pictured configuration using a single device with a loopback between channels 1 and 2. Note that the tool defaults to use channel 1 for the producer and channel 2 for the consumer, so the `channel` parameters can be omitted.

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

If instead there are two AJA devices being connected, the following can be used to measure a configuration in which they are both connected to channel 1:



Fig. 9.3: SDI Cable Between Channel 1 and 2 of a Single AJA Corvid 44 Capture Card

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

## 9.4 Operation Overview

The latency measurement tool operates by having a **producer** component generate a sequence of known video frames that are output and then transferred back to an input **consumer** component using a physical loopback cable. Timestamps are compared throughout the life of the frame to measure the overall latency that the frame sees during this process, and these results are summarized when all of the frames have been received and the measurement completes. See [Producers](#), [Consumers](#), and [Example Configurations](#) for more details.

### 9.4.1 Frame Measurements

Each frame that is generated by the tool goes through the following steps in order, each of which has its time measured and then reported when all frames complete.

#### 1. CUDA Processing

In order to simulate a real-world GPU workload, the tool first runs a CUDA kernel for a user-specified amount of loops (defaults to zero). This step is described below in [Simulating GPU Workload](#).

#### 2. Render on GPU

After optionally simulating a GPU workload, every producer then generates its frames using the GPU, either by a common CUDA kernel or by another method that is available to the producer's API (such as the OpenGL

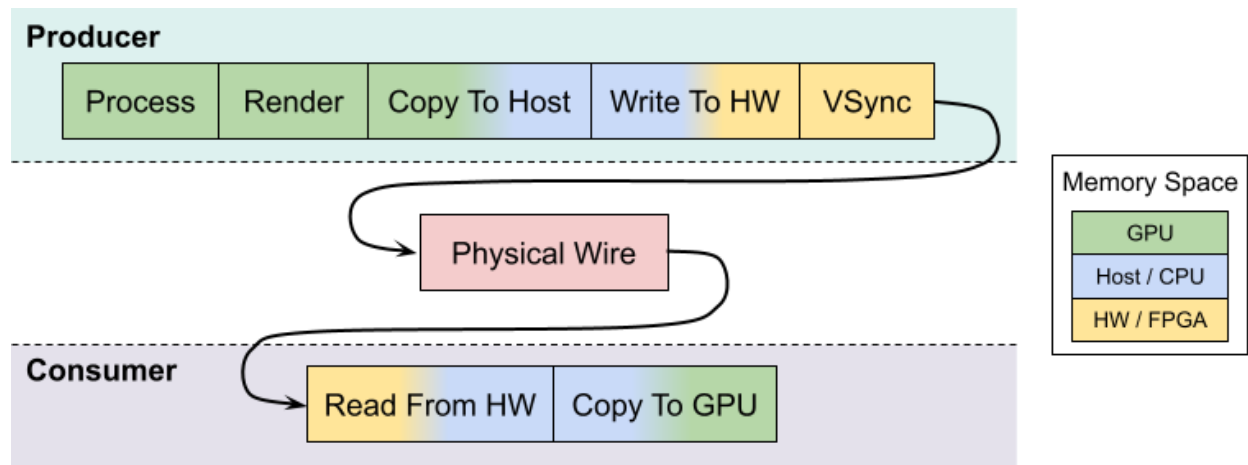


Fig. 9.4: Latency Tool Frame Lifespan (RDMA Disabled)

producer).

This step is expected to be very fast (<100us), but higher times may be seen if overall system load is high.

### 3. Copy To Host

Once the frame has been generated on the GPU, it may be necessary to copy the frame to host memory in order for the frame to be output by the producer component (for example, an AJA producer with RDMA disabled).

If a host copy is not required (i.e. RDMA is enabled for the producer), this time should be zero.

### 4. Write to HW

Some producer components require frames to be copied to peripheral memory before they can be output (for example, an AJA producer requires frames to be copied to the external frame stores on the AJA device). This copy may originate from host memory if RDMA is disabled for the producer, or from GPU memory if RDMA is enabled.

If this copy is not required, e.g. the producer outputs directly from the GPU, this time should be zero.

### 5. VSync Wait

Once the frame is ready to be output, the producer hardware must wait for the next VSync interval before the frame can be output.

The sum of this VSync wait and all of the preceding steps is expected to be near a multiple of the frame interval. For example, if the frame rate is 60Hz then the sum of the times for steps 1 through 5 should be near a multiple of 16666us.

### 6. Wire Time

The wire time is the amount of time that it takes for the frame to transfer across the physical loopback cable. This should be near the time for a single frame interval.

### 7. Read From HW

Once the frame has been transferred across the wire and is available to the consumer, some consumer components require frames to be copied from peripheral memory into host (RDMA disabled) or GPU (RDMA enable) memory. For example, an AJA consumer requires frames to be copied from the external frame store of the AJA device.

If this copy is not required, e.g. the consumer component writes received frames directly to host/GPU memory, this time should be zero.

## 8. Copy to GPU

If the consumer received the frame into host memory, the final step required for processing the frame with the GPU is to copy the frame into GPU memory.

If RDMA is enabled for the consumer and the frame was previously written directly to GPU memory, this time should be zero.

Note that if RDMA is enabled on the producer and consumer sides then the GPU/host copy steps above, 3 and 8 respectively, are effectively removed since RDMA will copy directly between the video HW and the GPU. The following shows the same diagram as above but with RDMA enabled for both the producer and consumer.

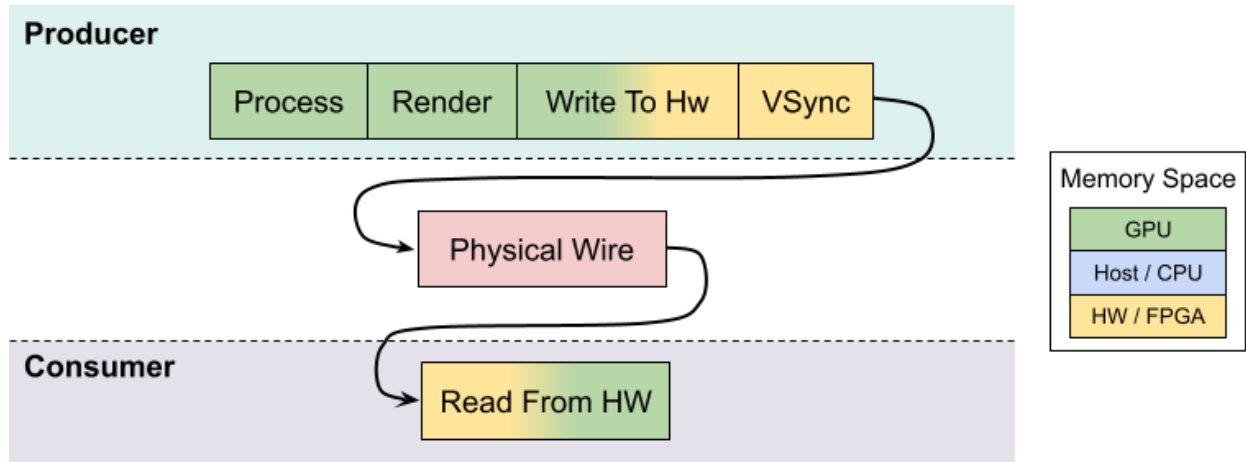


Fig. 9.5: Latency Tool Frame Lifespan (RDMA Enabled)

## 9.4.2 Interpreting The Results

The following shows example output of the above measurements from the tool when testing a 4K stream at 60Hz from an AJA producer to an AJA consumer, both with RDMA disabled, and no GPU/CUDA workload simulation. Note that all time values are given in microseconds.

```
$ ./loopback-latency -p aja -p.rdma 0 -c aja -c.rdma 0 -f 4k
```



```

Format: 4096x2160 RGBA @ 60Hz

Producer: AJA
  Device: 0
  Channel: NTV2_CHANNEL1
  RDMA: 0

Consumer: AJA
  Device: 0
  Channel: NTV2_CHANNEL2
  RDMA: 0

Measuring 600 frames...Done!

CUDA Processing: avg =      0, min =      0, max =      64
Render on GPU:   avg =    144, min =     94, max =    386
Copy To Host:    avg =   5788, min =   4145, max =   7024
Write To HW:     avg =   9468, min =   8219, max =   9916
Vsync Wait:     avg =   1245, min =    126, max =   2608
Wire Time:      avg =  16745, min =  16547, max =  17379
Read From HW:   avg =   7086, min =   6983, max =   7357
Copy To GPU:    avg =   4282, min =   3805, max =   6304
=====
Total:          avg =  44764, min =  44122, max =  46680

```

While this tool measures the producer times followed by the consumer times, the expectation for real-world video processing applications is that this order would be reversed. That is to say, the expectation for a real-world application is that it would capture, process, and output frames in the following order (with the component responsible for measuring that time within this tool given in parentheses):

1. **Read from 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)

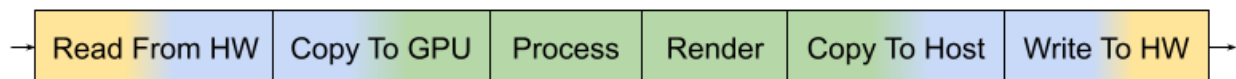


Fig. 9.6: Real Application Frame Lifespan

To illustrate this, the tool sums and displays the total producer and consumer times, then provides the **Estimated Application Times** as the total sum of all of these steps (i.e. steps 1 through 6, above).

(continued from above)

```

Producer (Process and Write to HW)
=====
Microseconds: avg = 15403, min = 14074, max = 16495
Frames: avg = 0.924, min = 0.844, max = 0.99

Consumer (Read from HW and Copy to GPU)
=====
Microseconds: avg = 11369, min = 10856, max = 13381
Frames: avg = 0.682, min = 0.651, max = 0.803

Estimated Application Times (Read + Process + Write)
=====
Microseconds: avg = 26772, min = 25101, max = 29204
Frames: avg = 1.61, min = 1.51, max = 1.75

```

Once a real-world application captures, processes, and outputs a frame, it would still be required that this final output waits for the next VSync interval before it is actually sent across the physical wire to the display hardware. Using this assumption, the tool then estimates one final value for the **Final Estimated Latencies** by doing the following:

1. Take the **Estimated Application Time** (from above)
2. Round it up to the next VSync interval
3. Add the physical wire time (i.e. a frame interval)

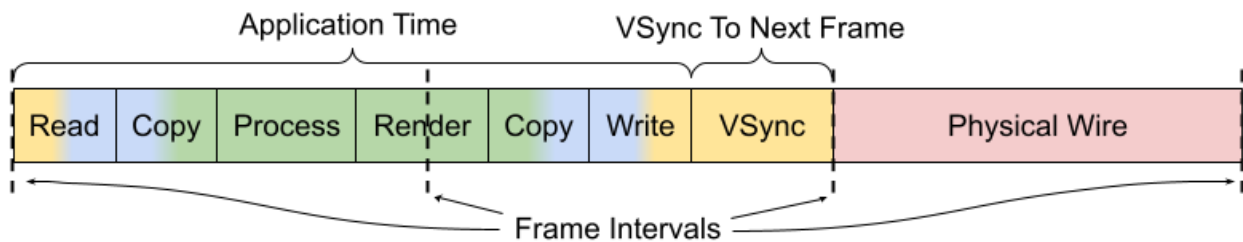


Fig. 9.7: Final Estimated Latency with VSync and Physical Wire Time

Continuing this example using a frame interval of 16666us (60Hz), this means that the average **Final Estimated Latency** is determined by:

1. Average application time = **26772**
2. Round up to next VSync interval = **33332**
3. Add physical wire time (+16666) = **49998**

These times are also reported as a multiple of frame intervals.

(continued from above)

```

Final Estimated Latencies (Processing + Vsync + Wire)
=====
Microseconds: avg = 49998, min = 49998, max = 49998
Frames: avg = 3, min = 3, max = 3

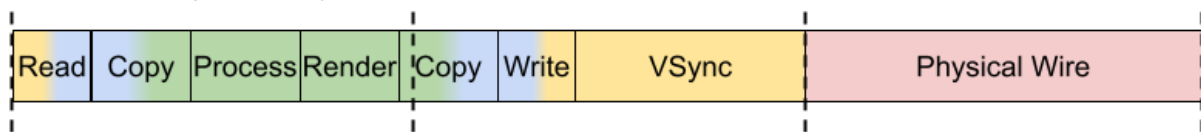
```

Using this example, we should then expect that the total end-to-end latency that is seen by running this pipeline using these components and configuration is 3 frame intervals (49998us).

### 9.4.3 Reducing Latency With RDMA

The previous example uses an AJA producer and consumer for a 4K @ 60Hz stream, however RDMA was disabled for both components. Because of this, the additional copies between the GPU and host memory added more than 10000us of latency to the pipeline, causing the application to exceed one frame interval of processing time per frame and therefore a total frame latency of 3 frames. If RDMA is enabled, these GPU and host copies can be avoided so the processing latency is reduced by more than 10000us. More importantly, however, this also allows the total processing time to fit within a single frame interval so that the total end-to-end latency can be reduced to just 2 frames.

RDMA Disabled (3 Frames)



RDMA Enabled (2 Frames)

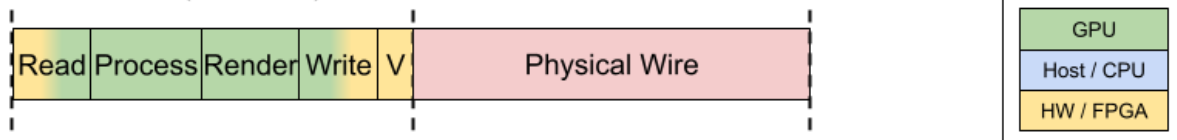


Fig. 9.8: Reducing Latency With RDMA

The following shows the above example repeated with RDMA enabled.

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



```

Format: 4096x2160 RGBA @ 60Hz

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

Consumer: AJA
  Device: 0
  Channel: NTV2_CHANNEL2
  RDMA: 1

Measuring 600 frames...Done!

CUDA Processing: avg =      0, min =      0, max =      74
Render on GPU:   avg =    122, min =     94, max =    356
Copy To Host:    avg =      0, min =      0, max =     35
Write To HW:     avg =   8209, min =   7453, max =   8856
Vsync Wait:     avg =   8314, min =   6338, max =  10036
Wire Time:       avg =  16650, min =  14814, max =  18391
Read From HW:    avg =   6041, min =   5962, max =   6931
Copy To GPU:     avg =      0, min =      0, max =     30
=====
Total:           avg =  39343, min =  37668, max =  41081

Producer (Process and Write to HW)
=====
  Microseconds: avg =   8334, min =   7580, max =   8988
    Frames: avg =    0.5, min =   0.455, max =   0.539

Consumer (Read from HW and Copy to GPU)
=====
  Microseconds: avg =   6042, min =   5962, max =   6932
    Frames: avg =   0.363, min =   0.358, max =   0.416

Estimated Application Times (Read + Process + Write)
=====
  Microseconds: avg =  14377, min =  13627, max =  15233
    Frames: avg =   0.863, min =   0.818, max =   0.914

Final Estimated Latencies (Processing + Vsync + Wire)
=====
  Microseconds: avg =  33332, min =  33332, max =  33332
    Frames: avg =      2, min =      2, max =      2

```

### 9.4.4 Simulating GPU Workload

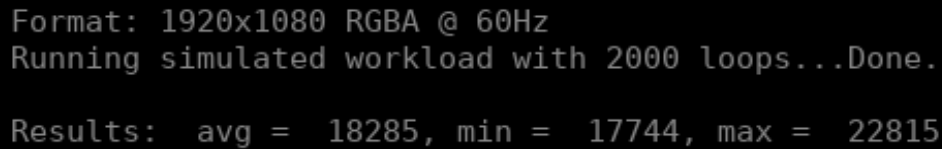
By default the tool measures what is essentially a pass-through video pipeline; that is, no processing of the video frames is performed by the system. While this is useful for measuring the minimum latency that can be achieved by the video input and output components, it's not very indicative of a real-world use case in which the GPU is used for compute-intensive processing operations on the video frames between the input and output — for example, an object detection algorithm that applies an overlay to the output frames.

While it may be relatively simple to measure the runtime latency of the processing algorithms that are to be applied to the video frames — by simply measuring the runtime of running the algorithm on a single or stream of frames — this may not be indicative of the effects that such processing might have on the overall system load, which may further increase the latency of the video input and output components.

In order to estimate the total latency when an additional GPU workload is added to the system, the latency tool has an `-s {count}` option that can be used to run an arbitrary CUDA loop the specified number of times before the producer actually generates a frame. The expected usage for this option is as follows:

1. The per-frame runtime of the actual GPU processing algorithm is measured outside of the latency measurement tool.
2. The latency tool is repeatedly run with just the `-s {count}` option, adjusting the `{count}` parameter until the time that it takes to run the simulated loop approximately matches the actual processing time that was measured in the previous step.

```
$ ./loopback-latency -s 2000
```



```
Format: 1920x1080 RGBA @ 60Hz
Running simulated workload with 2000 loops...Done.

Results:  avg = 18285, min = 17744, max = 22815
```

3. The latency tool is run with the full producer (`-p`) and consumer (`-c`) options used for the video I/O, along with the `-s {count}` option using the loop count that was determined in the previous step.

---

**Note:** The following example shows that approximately half of the frames received by the consumer were duplicate/repeated frames. This is due to the fact that the additional processing latency of the producer causes it to exceed a single frame interval, and so the producer is only able to output a new frame every second frame interval.

---

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

```

Format: 1920x1080 RGBA @ 60Hz

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

Consumer: AJA
  Device: 0
  Channel: NTV2_CHANNEL2
  RDMA: 1

Simulating processing with 2000 CUDA loops per frame.

Measuring 600 frames...Done!

WARNING: Frames were skipped or repeated!
Frames received: 301
Frames skipped: 0
Frames repeated: 299

CUDA Processing: avg = 17153, min = 16877, max = 17569
Render on GPU:   avg = 50, min = 34, max = 116
Copy To Host:    avg = 0, min = 0, max = 19
Write To HW:     avg = 1785, min = 1721, max = 1849
Vsync Wait:      avg = 14321, min = 13782, max = 14718
Wire Time:       avg = 16723, min = 16360, max = 33470
Read From HW:    avg = 1502, min = 1442, max = 1726
Copy To GPU:     avg = 0, min = 0, max = 0
=====
Total:           avg = 51541, min = 51164, max = 68238

Producer (Process and Write to HW)
=====
  Microseconds: avg = 18991, min = 18689, max = 19405
  Frames: avg = 1.14, min = 1.12, max = 1.16

Consumer (Read from HW and Copy to GPU)
=====
  Microseconds: avg = 1502, min = 1443, max = 1726
  Frames: avg = 0.0901, min = 0.0866, max = 0.104

Estimated Application Times (Read + Process + Write)
=====
  Microseconds: avg = 20493, min = 20191, max = 20967
  Frames: avg = 1.23, min = 1.21, max = 1.26

Final Estimated Latencies (Processing + Vsync + Wire)
=====
  Microseconds: avg = 49998, min = 49998, max = 49998
  Frames: avg = 3, min = 3, max = 3

WARNING: Frames were skipped or repeated. These times only
include frames that were actually received, and the times
include only the first instance each frame was received.

```

---

**Tip:** To get the most accurate estimation of the latency that would be seen by a real world application, the best thing to do would be to run the actual frame processing algorithm used by the application during the latency measurement. This could be done by modifying the `SimulateProcessing` function in the latency tool source code.

---

## 9.5 Graphing Results

The latency tool includes a `-o {file}` option that can be used to output a CSV file with all of the measured times for every frame. This file can then be used with the `graph_results.py` script that is included with the tool in order to generate a graph of the measurements.

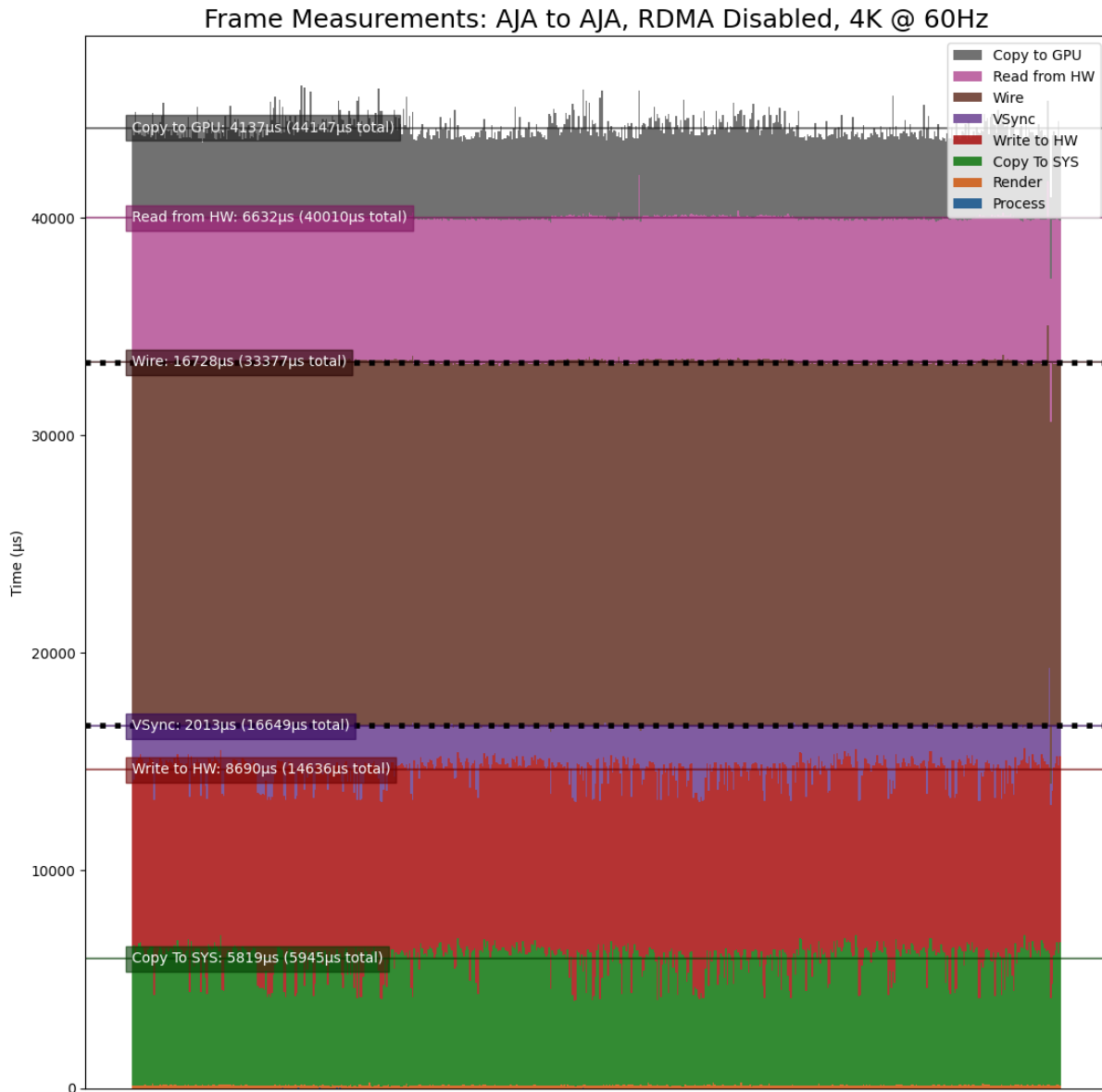
For example, if the latencies are measured using:

```
$ ./loopback-latency -p aja -c aja -o latencies.csv
```

The graph can then be generated using the following, which will open a window on the desktop to display the graph:

```
$ ./graph_results.py --file latencies.csv
```

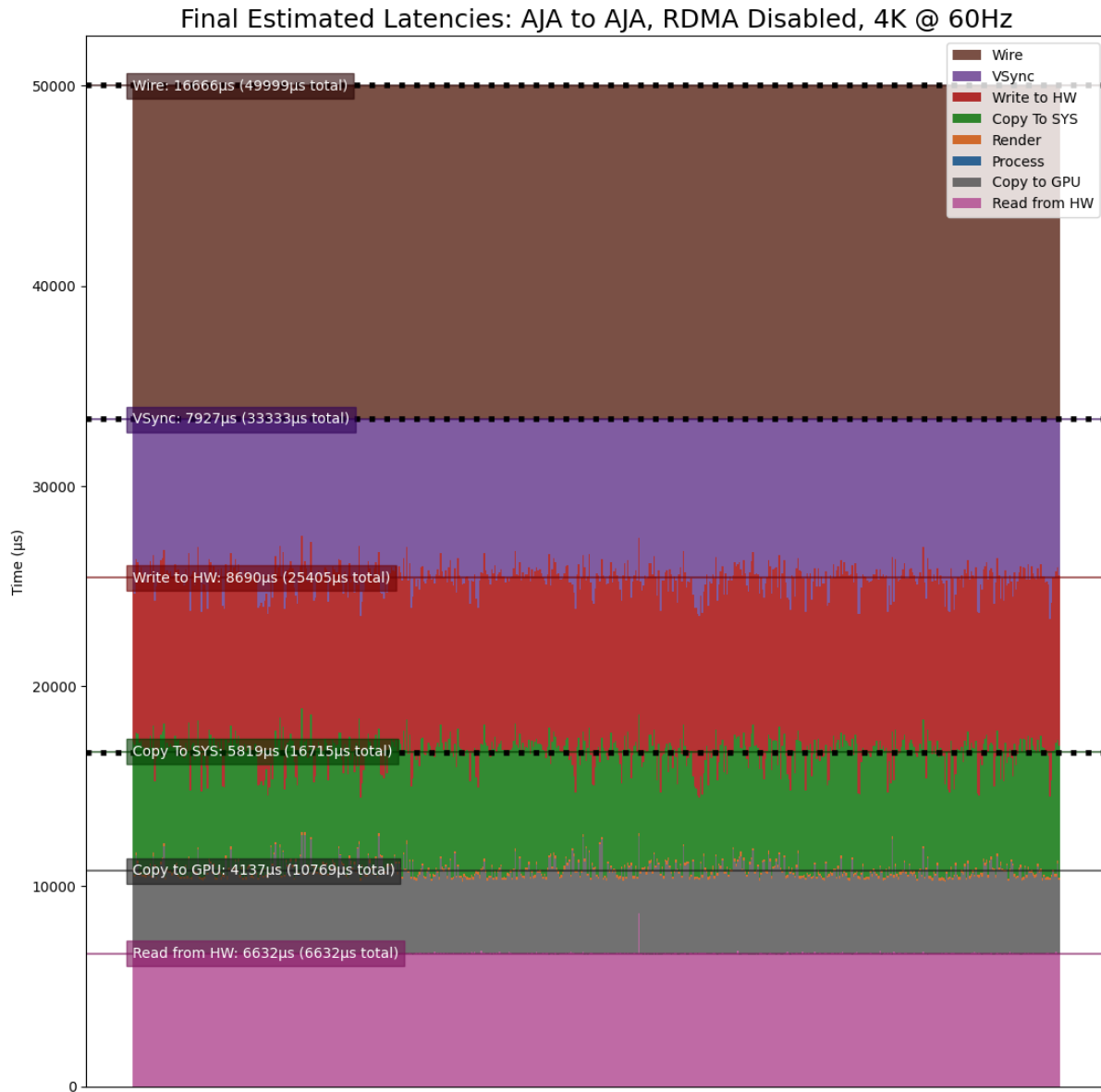
The graph can also be output to a PNG image file instead of opening a window on the desktop by providing the `--png {file}` option to the script. The following shows an example graph for an AJA to AJA measurement of a 4K @ 60Hz stream with RDMA disabled (as shown as an example in [Interpreting The Results](#), above).



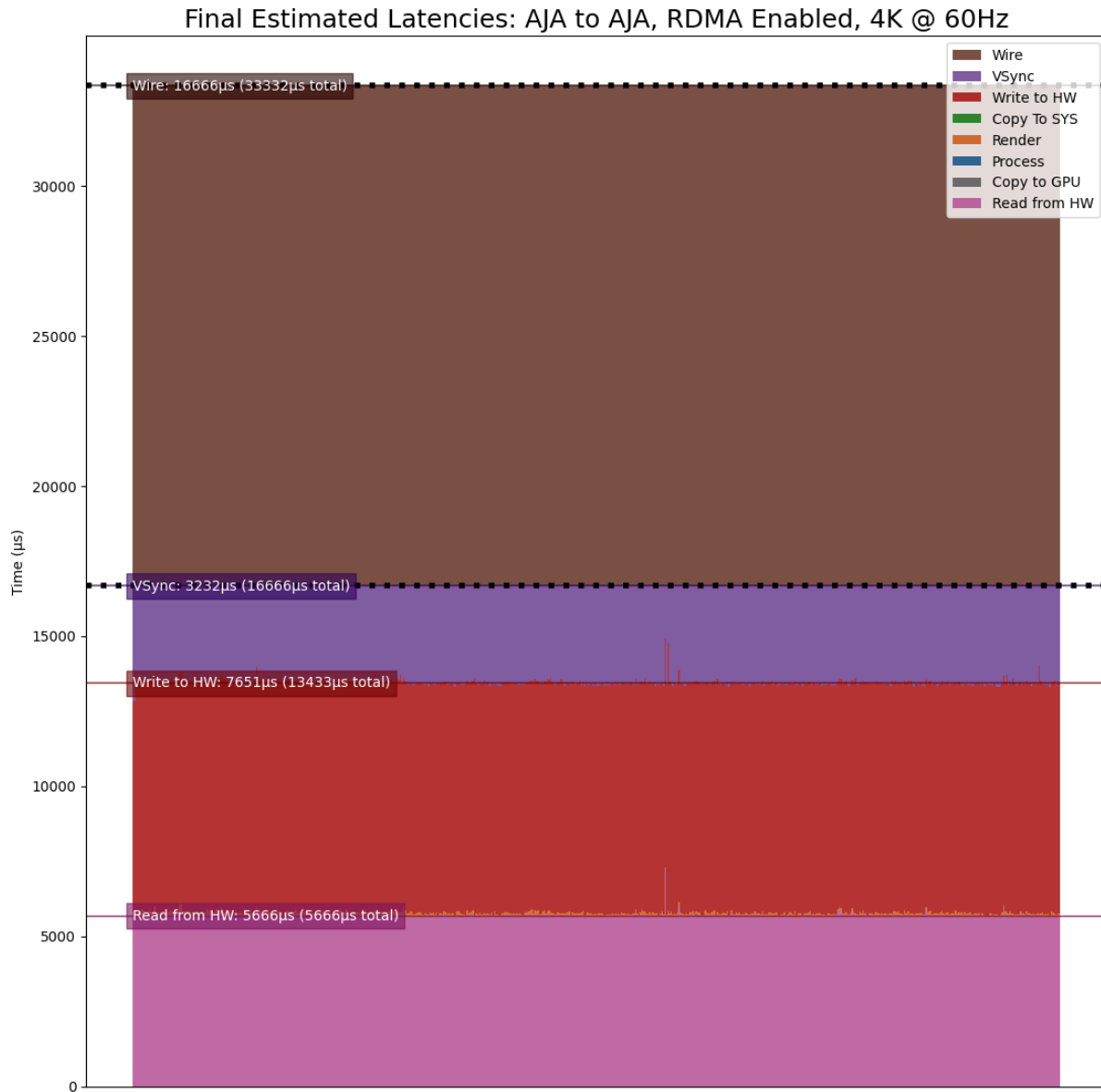
Note that this is showing the times for 600 frames, from left to right, with the life of each frame beginning at the bottom and ending at the top. The dotted black lines represent frame VSync intervals (every 16666 $\mu$ s).

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.

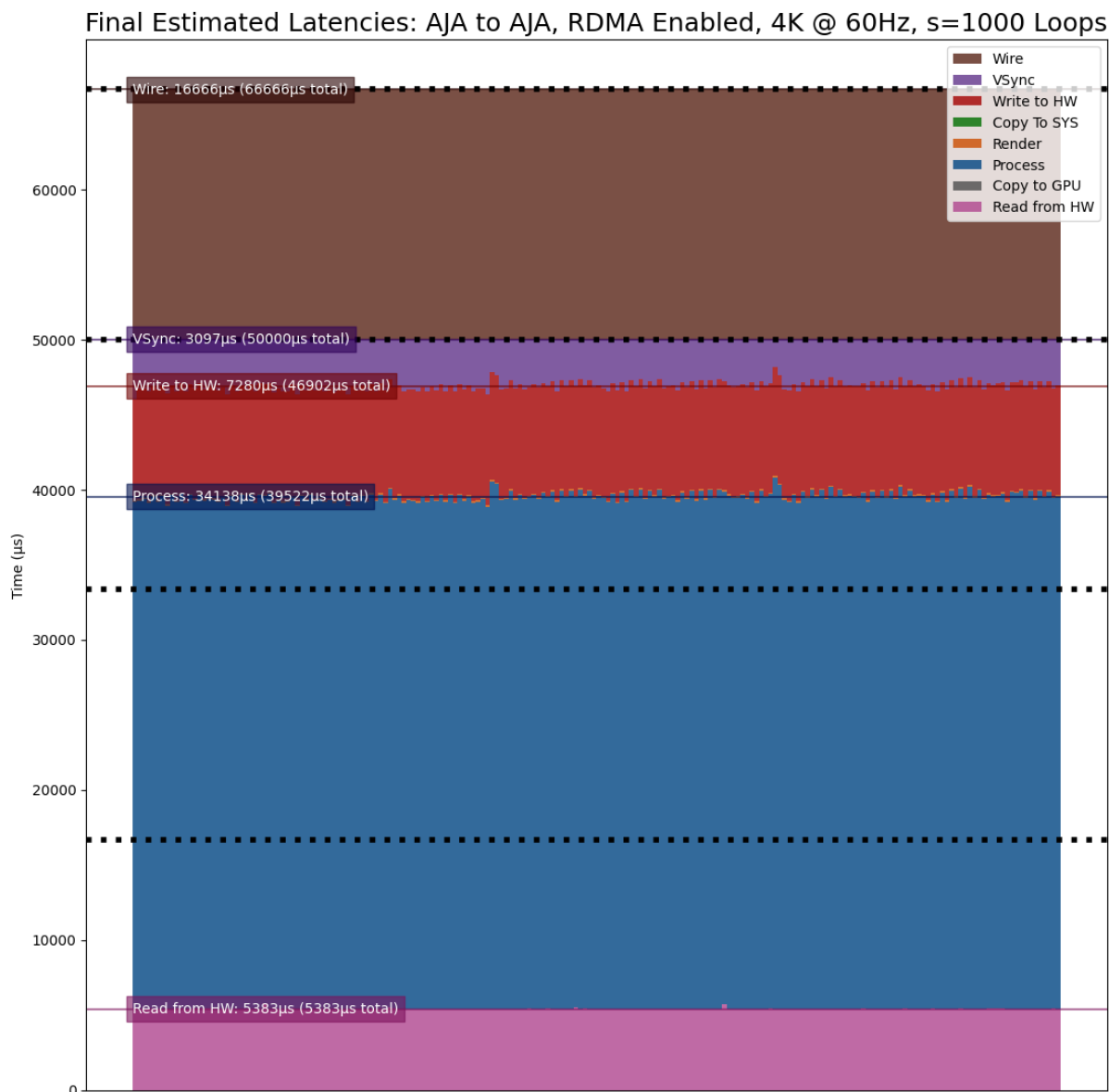
The following graphs the **Final Estimated Latencies** using the same data file as the graph above. Note that this shows a total of 3 frames of expected latency.



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.



## 9.6 Producers

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



### 9.6.1 OpenGL GPU Direct Rendering (HDMI)

This producer (`gl`) uses OpenGL to render frames directly on the GPU for output via the HDMI connectors on the GPU. This is currently expected to be the lowest latency path for GPU video output.

OpenGL Producer Notes:

- The video generated by this producer is rendered full-screen to the primary display. As of this version, this component has only been tested in a display-less environment in which the loop-back HDMI cable is the only cable attached to the GPU (and thus is the primary display). It may also be required to use the `xrandr` tool to configure the HDMI output — the tool will provide the `xrandr` commands needed if this is the case.
- Since OpenGL renders directly to the GPU, the `p.rdma` flag is not supported and RDMA is always considered to be enabled for this producer.

### 9.6.2 GStreamer GPU Rendering (HDMI)

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

GStreamer Producer Notes:

- The tool must be built with DeepStream support in order for this producer to support RDMA (see [Enabling DeepStream Support](#) for details).
- The video generated by this producer is rendered full-screen to the primary display. As of this version, this component has only been tested in a display-less environment in which the loop-back HDMI cable is the only cable attached to the GPU (and thus is the primary display). It may also be required to use the `xrandr` tool to configure the HDMI output — the tool will provide the `xrandr` commands needed if this is the case.
- Since the output of the generated frames is handled internally by the `nveglglessink` plugin, the timing of when the frames are output from the GPU are not known. Because of this, the *Wire Time* that is reported by this producer includes all of the time that the frame spends between being passed to the `nveglglessink` and when it is finally received by the consumer.

### 9.6.3 AJA Video Systems (SDI)

This producer (`aja`) outputs video frames from an AJA Video Systems device that supports video playback.

AJA Producer Notes:

- The latency tool must be built with AJA Video Systems support in order for this producer to be available (see [Building](#) for details).
- The following parameters can be used to configure the AJA device and channel that are used to output the frames:
  - `-p.device {index}`  
Integer specifying the device index (i.e. 0 or 1). Defaults to 0.
  - `-p.channel {channel}`  
Integer specifying the channel number, starting at 1 (i.e. 1 specifies `NTV2_CHANNEL_1`). Defaults to 1.
- The `p.rdma` flag can be used to enable (1) or disable (0) the use of RDMA with the producer. If RDMA is to be used, the AJA drivers loaded on the system must also support RDMA.
- The only AJA device that have currently been verified to work with this producer is the [Corvid 44 12G BNC \(SDI\)](#).

## 9.7 Consumers

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

### 9.7.1 V4L2 (Onboard HDMI Capture Card)

This consumer (`v4l2`) uses the V4L2 API directly in order to capture frames using the HDMI capture card that is onboard the Clara 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.

### 9.7.2 GStreamer (Onboard HDMI Capture Card)

This consumer (`gst`) also captures frames from the onboard HDMI capture card, but uses the `v4l2src` GStreamer plugin that wraps the V4L2 API to support capturing frames for using within a GStreamer pipeline.

GStreamer Consumer Notes:

- The onboard HDMI capture card is locked to a specific frame resolution and frame rate (1080p @ 60Hz), and so **1080** is the only supported format when using this consumer.
- The `-c.device {device}` parameter can be used to specify the path to the device that is being used to capture the frames (defaults to `/dev/video0`).
- The `v4l2src` GStreamer plugin does not support RDMA, and so the `c.rdma` option is ignored.

### 9.7.3 AJA Video Systems (SDI and HDMI)

This consumer (`aja`) captures video frames from an AJA Video Systems device that supports video capture. This can be either an SDI or an HDMI video capture card.

AJA Consumer Notes:

- The latency tool must be built with AJA Video Systems support in order for this producer to be available (see [Building](#) for details).
- The following parameters can be used to configure the AJA device and channel that are used to capture the frames:
  - `-c.device {index}`  
Integer specifying the device index (i.e. 0 or 1). Defaults to 0.
  - `-c.channel {channel}`  
Integer specifying the channel number, starting at 1 (i.e. 1 specifies `NTV2_CHANNEL_1`). Defaults to 2.
- The `c.rdma` flag can be used to enable (1) or disable (0) the use of RDMA with the consumer. If RDMA is to be used, the AJA drivers loaded on the system must also support RDMA.

- The only AJA devices that have currently been verified to work with this consumer are the [KONA HDMI](#) (for HDMI) and [Corvid 44 12G BNC](#) (for SDI).

## 9.8 Troubleshooting

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

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

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

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

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

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

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

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

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

```
ERROR: The requested format (1920x1080 @ 60Hz) does not match
       the current display mode (1024x768 @ 60Hz)
       Please set the display mode with the xrandr tool using
       the following comand:

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

But using the xrandr command provided produces an error:

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

**Solution:** Try the following:

1. Ensure that no other displays are connected to the GPU.
2. Check the output of an xrandr command to see that the requested format is supported. The following shows an example of what the onboard HDMI capture card should support. Note that each row of the supported modes shows the resolution on the left followed by all of the supported frame rates for that resolution to the right.

```
$ xrandr
Screen 0: minimum 8 x 8, current 1920 x 1080, maximum 32767 x 32767
DP-0 disconnected (normal left inverted right x axis y axis)
DP-1 disconnected (normal left inverted right x axis y axis)
DP-2 disconnected (normal left inverted right x axis y axis)
DP-3 disconnected (normal left inverted right x axis y axis)
DP-4 disconnected (normal left inverted right x axis y axis)
```

(continues on next page)

(continued from previous page)

```

DP-5 connected primary 1920x1080+0+0 (normal left inverted right x axis y axis)
↪1872mm x 1053mm
  1920x1080    60.00*+  59.94   50.00   29.97   25.00   23.98
  1680x1050    59.95
  1600x900     60.00
  1440x900     59.89
  1366x768     59.79
  1280x1024    75.02   60.02
  1280x800     59.81
  1280x720     60.00   59.94   50.00
  1152x864     75.00
  1024x768     75.03   70.07   60.00
   800x600     75.00   72.19   60.32
   720x576     50.00
   720x480     59.94
   640x480     75.00   72.81   59.94
DP-6 disconnected (normal left inverted right x axis y axis)
DP-7 disconnected (normal left inverted right x axis y axis)
USB-C-0 disconnected (normal left inverted right x axis y axis)

```

3. If a UHD or 4K mode is being requested, ensure that the DisplayPort to HDMI cable that is being used supports that mode.
  4. If the `xrandr` output still does not show the mode that is being requested but it should be supported by the cable and capture device, try rebooting the device.
3. **Problem:** One of the following errors is output:

```
ERROR: Select timeout on /dev/video0
```

```
ERROR: Failed to get the monitor mode (is the display cable attached?)
```

```
ERROR: Could not find frame color (0,0,0) in producer records.
```

These errors mean that either the capture device is not receiving frames, or the frames are empty (the producer will never output black frames, (0,0,0)).

**Solution:** Check the output of `xrandr` to ensure that the loopback cable is connected and the capture device is recognized as a display. If the following is output, showing no displays attached, this could mean that the loopback cable is either not connected properly or is faulty. Try connecting the cable again and/or replacing the cable.

```

$ xrandr
Screen 0: minimum 8 x 8, current 1920 x 1080, maximum 32767 x 32767
DP-0 disconnected (normal left inverted right x axis y axis)
DP-1 disconnected (normal left inverted right x axis y axis)
DP-2 disconnected (normal left inverted right x axis y axis)
DP-3 disconnected (normal left inverted right x axis y axis)
DP-4 disconnected (normal left inverted right x axis y axis)
DP-5 disconnected primary 1920x1080+0+0 (normal left inverted right x axis y axis)
↪0mm x 0mm
DP-6 disconnected (normal left inverted right x axis y axis)
DP-7 disconnected (normal left inverted right x axis y axis)

```

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

**ERROR:** Could not find frame color (27,28,26) in producer records.

Colors near this particular value (27,28,26) are displayed on the Ubuntu lock screen, which prevents the latency tool from rendering frames properly. Note that the color value may differ slightly from (27,28,26).

**Solution:**

Follow the steps provided in the note at the top of the Example Configurations section to *enable automatic login and disable the Ubuntu lock screen*.



## 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 you can find containers by looking through the Clara Holoscan collections link:

[Clara Holoscan Collection 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.

### 10.1 ARM Container

[Clara Holoscan Sample Applications container](#) is available for Clara developer kits.

### 10.2 x86 Container

Beginning with SDK v0.3, an x86\_64 version of the [Clara Holoscan Sample Applications container](#) is available as well.

The main requirement for this version of the container is the Ubuntu 20.04 operating system, and the container requires a Turing or Ampere GPU.

We recommend specifically either of these GPUs:

- NVIDIA RTX 6000
- NVIDIA RTX A6000

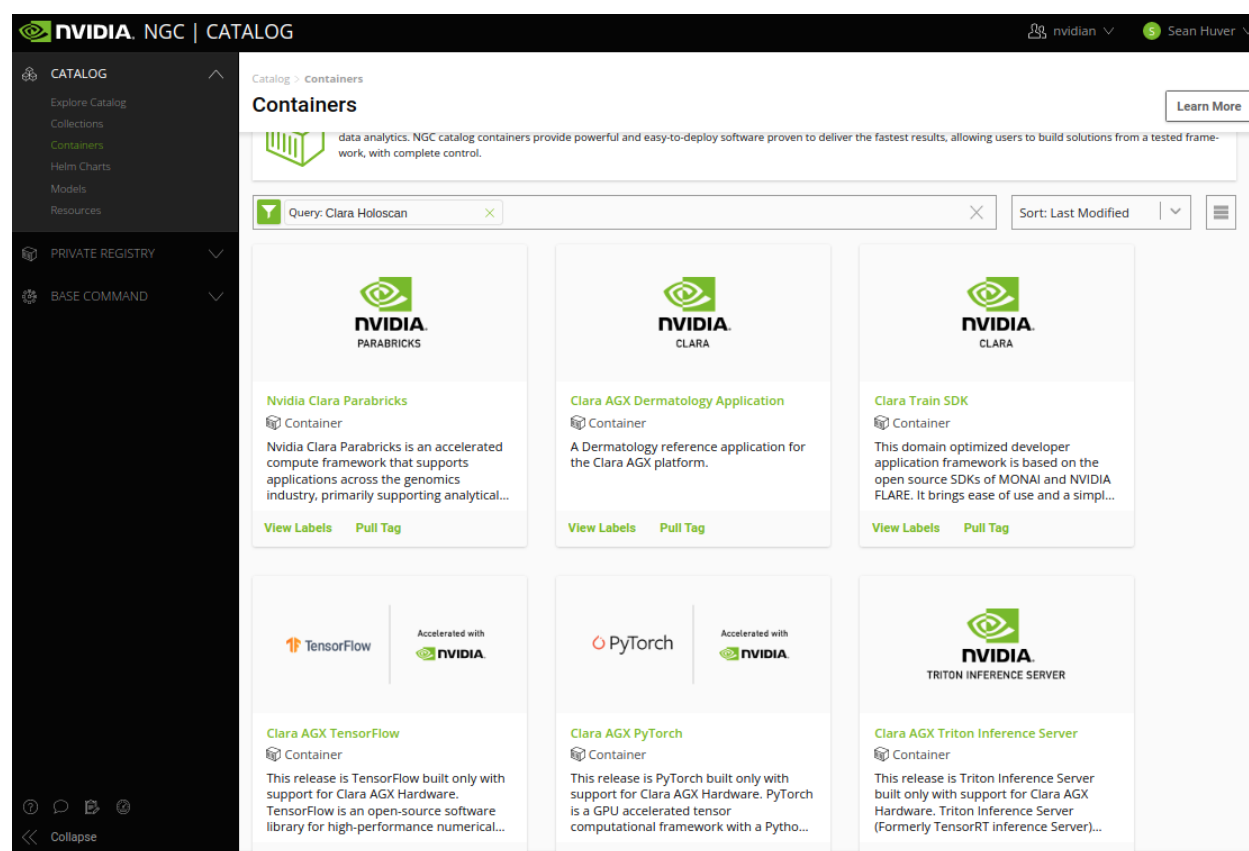


Fig. 10.1: NGC Catalog



## RELEVANT TECHNOLOGIES

Clara Holoscan accelerates streaming AI applications by leveraging both hardware and software. On the software side, the Clara Holoscan SDK 0.3.0 relies on multiple core technologies to achieve low latency and high throughput, including:

### 11.1 Graph Execution Framework (GXF)

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

Those are the key terms used throughout this guide:

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

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

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

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

- **Scheduler and Scheduling Terms:** components that determine how and when the `tick()` of a Codelet executes. This can be single or multithreaded, support conditional execution, asynchronous scheduling, and other custom behavior.
- **Memory Allocator:** provides a system for up-front allocating a large contiguous memory pool and then re-using regions as needed. Memory can be pinned to the device (enabling zero-copy between Codelets when messages are not modified) or host or customized for other potential behavior.
- **Receivers, Transmitters, and Message Router:** a message passing system between Codelets that supports zero-copy.

- **Tensor:** the common message type is a tensor. It provides a simple abstraction for numeric data that can be allocated, serialized, sent between Codelets, etc. Tensors can be rank 1 to 7 supporting a variety of common data types like arrays, vectors, matrices, multi-channel images, video, regularly sampled time-series data, and higher dimensional constructs popular with deep learning flows.
- **Parameters:** configuration variables that specify constants used by the Codelet loaded from the application yaml file modifiable without recompiling.

### 11.1.1 GXF Entities by Example

Let us look at an example of a Holoscan entity to try to understand its general anatomy. As an example let's start with the entity definition for an image format converter entity named `format_converter_entity` as shown below.

Listing 11.1: An example GXF Application YAML snippet

```

1 %YAML 1.2
2 ---
3 # other entities declared
4 ---
5 name: format_converter_entity
6 components:
7   - name: in_tensor
8     type: nvidia::gxf::DoubleBufferReceiver
9   - type: nvidia::gxf::MessageAvailableSchedulingTerm
10    parameters:
11      receiver: in_tensor
12      min_size: 1
13   - name: out_tensor
14     type: nvidia::gxf::DoubleBufferTransmitter
15   - type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
16    parameters:
17      transmitter: out_tensor
18      min_size: 1
19   - name: pool
20     type: nvidia::gxf::BlockMemoryPool
21    parameters:
22      storage_type: 1
23      block_size: 4919040 # 854 * 480 * 3 (channel) * 4 (bytes per pixel)
24      num_blocks: 2
25   - name: format_converter_component
26     type: nvidia::holoscan::formatconverter::FormatConverter
27    parameters:
28      in: in_tensor
29      out: out_tensor
30      out_tensor_name: source_video
31      out_dtype: "float32"
32      scale_min: 0.0
33      scale_max: 255.0
34      pool: pool
35 ---
36 # other entities declared
37 ---
38 components:

```

(continues on next page)

(continued from previous page)

```

39 - name: input_connection
40   type: nvidia::gxf::Connection
41   parameters:
42     source: upstream_entity/output
43     target: format_converter/in_tensor
44 ---
45 components:
46 - name: output_connection
47   type: nvidia::gxf::Connection
48   parameters:
49     source: format_converter/out_tensor
50     target: downstream_entity/input
51 ---
52 name: scheduler
53 components:
54 - type: nvidia::gxf::GreedyScheduler

```

Above:

1. The entity `format_converter_entity` receives a message in its `in_tensor` message from an upstream entity `upstream_entity` as declared in the `input_connection`.
2. The received message is passed to the `format_converter_component` component to convert the tensor element precision from `uint8` to `float32` and scale any input in the `[0, 255]` intensity range.
3. The `format_converter_component` component finally places the result in the `out_tensor` message so that its result is made available to a downstream entity (`downstream_entity` as declared in `output_connection`).
4. The `Connection` components tie the inputs and outputs of various components together, in the above case `upstream_entity/output` -> `format_converter_entity/in_tensor` and `format_converter_entity/out_tensor` -> `downstream_entity/input`.
5. The `scheduler` entity declares a `GreedyScheduler` “system component” which orchestrates the execution of the entities declared in the graph. In the specific case of `GreedyScheduler` entities are scheduled to run exclusively, where no more than one entity can run at any given time.

The YAML snippet above can be visually represented as follows.

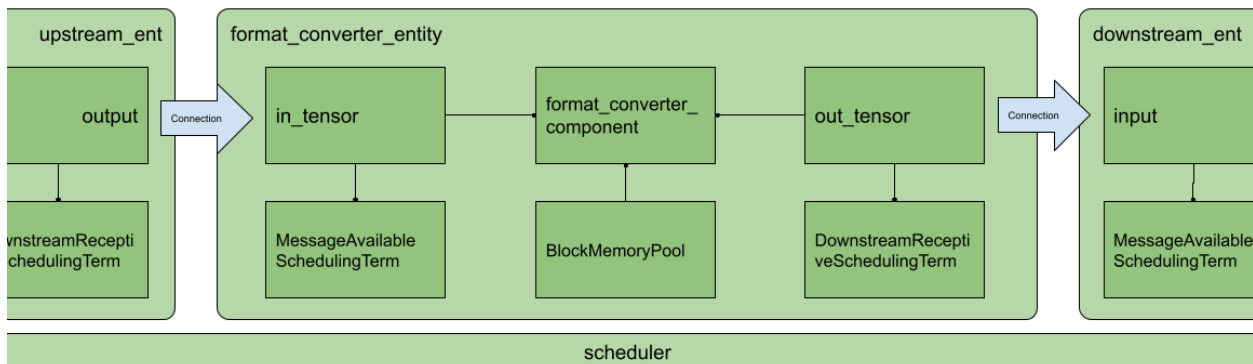


Fig. 11.1: Arrangement of components and entities in a Holoscan application

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

- `MessageAvailableSchedulingTerm` is a component that takes a `Receiver` (in this case DoubleBufferReceiver named in_tensor) and alerts the graph Executor that a message is available. This alert triggers format_converter_component`.`
- `DownstreamReceptiveSchedulingTerm` is a component that takes a `Transmitter` (in this case `DoubleBufferTransmitter` named `out_tensor`) and alerts the graph `Executor` that a message has been placed on the output.
- `BlockMemoryPool` provides two blocks of almost 5MB allocated on the GPU device and is used by `format_converted_ent` to allocate the output tensor where the converted data will be placed within the `format converted` component.

Together these components allow the entity to perform a specific function and coordinate communication with other entities in the graph via the declared scheduler.

More generally, an entity can be thought of as a collection of components where components can be passed to one another to perform specific subtasks (e.g. event triggering or message notification, format conversion, memory allocation), and an application as a graph of entities.

The scheduler is a component of type `nvidia::gxf::System` which orchestrates the execution components in each entity at application runtime based on triggering rules.

### 11.1.2 Data Flow and Triggering Rules

Entities communicate with one another via messages which may contain one or more payloads. Messages are passed and received via a component of type `nvidia::gxf::Queue` from which both `nvidia::gxf::Receiver` and `nvidia::gxf::Transmitter` are derived. Every entity that receives and transmits messages has at least one receiver and one transmitter queue.

Holoscan uses the `nvidia::gxf::SchedulingTerm` component to coordinate data access and component orchestration for a `Scheduler` which invokes execution through the `tick()` function in each `Codelet`.

**Tip:** A `SchedulingTerm` defines a specific condition that is used by an entity to let the scheduler know when it's ready for execution.

In the above example, we used a `MessageAvailableSchedulingTerm` to trigger the execution of the components waiting for data from `in_tensor` receiver queue, namely `format_converter_component`.

Listing 11.2: `MessageAvailableSchedulingTerm`

```

1 - type: nvidia::gxf::MessageAvailableSchedulingTerm
2   parameters:
3     receiver: in_tensor
4     min_size: 1

```

Similarly, `DownStreamReceptiveSchedulingTerm` checks whether the `out_tensor` transmitter queue has at least one outgoing message in it. If there are one or more outgoing messages, `DownStreamReceptiveSchedulingTerm` will notify the scheduler which in turn attempts to place the message in the receiver queue of a downstream entity. If, however, the downstream entity has a full receiver queue, the message is held in the `out_tensor` queue as a means to handle back-pressure.

Listing 11.3: `DownstreamReceptiveSchedulingTerm`

```

1 - type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
2   parameters:

```

(continues on next page)

(continued from previous page)

```

3 transmitter: out_tensor
4 min_size: 1

```

If we were to draw the entity in *Fig. 11.1* in greater detail it would look something like the following.

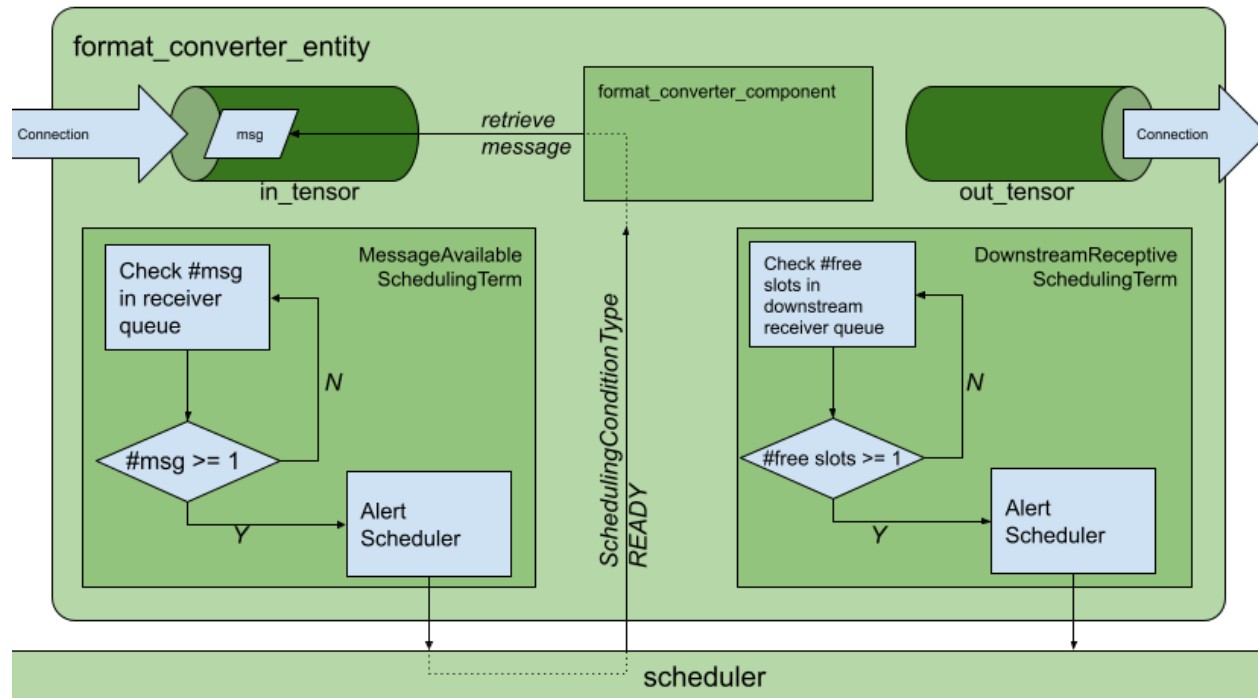


Fig. 11.2: Receive and transmit Queues and SchedulingTerms in entities.

Up to this point, we have covered the “entity component system” at a high level and showed the functional parts of an entity, namely, the messaging queues and the scheduling terms that support the execution of components in the entity. To complete the picture, the next section covers the anatomy and lifecycle of a component, and how to handle events within it.

### 11.1.3 Creating the GXF Application Definition

Please follow *Developing Holoscan GXF Extensions* section first for a detailed explanation of the GXF extension development process.

For our application, we create the directory `apps/my_recorder_app_gxf` with the application definition file `my_recorder_gxf.yaml`. The `my_recorder_gxf.yaml` application is as follows:

Listing 11.4: `apps/my_recorder_app_gxf/my_recorder_gxf.yaml`

```

1 %YAML 1.2
2 ---
3 name: replay
4 components:
5   - name: output

```

(continues on next page)

(continued from previous page)

```

6     type: nvidia::gxf::DoubleBufferTransmitter
7   - name: allocator
8     type: nvidia::gxf::UnboundedAllocator
9   - name: component_serializer
10    type: nvidia::gxf::StdComponentSerializer
11    parameters:
12      allocator: allocator
13   - name: entity_serializer
14     type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
↳nvidia::gxf::EntitySerializer
15    parameters:
16      component_serializers: [component_serializer]
17   - type: nvidia::holoscan::stream_playback::VideoStreamReplayer
18     parameters:
19       transmitter: output
20       entity_serializer: entity_serializer
21       boolean_scheduling_term: boolean_scheduling
22       directory: "/workspace/test_data/endoscopy/video"
23       basename: "surgical_video"
24       frame_rate: 0 # as specified in timestamps
25       repeat: false # default: false
26       realtime: true # default: true
27       count: 0 # default: 0 (no frame count restriction)
28   - name: boolean_scheduling
29     type: nvidia::gxf::BooleanSchedulingTerm
30   - type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
31     parameters:
32       transmitter: output
33       min_size: 1
34   ---
35   name: recorder
36   components:
37   - name: input
38     type: nvidia::gxf::DoubleBufferReceiver
39   - name: allocator
40     type: nvidia::gxf::UnboundedAllocator
41   - name: component_serializer
42     type: nvidia::gxf::StdComponentSerializer
43     parameters:
44       allocator: allocator
45   - name: entity_serializer
46     type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
↳nvidia::gxf::EntitySerializer
47     parameters:
48       component_serializers: [component_serializer]
49   - type: MyRecorder
50     parameters:
51       receiver: input
52       serializer: entity_serializer
53       out_directory: "/tmp"
54       basename: "tensor_out"
55   - type: nvidia::gxf::MessageAvailableSchedulingTerm

```

(continues on next page)

(continued from previous page)

```

56     parameters:
57         receiver: input
58         min_size: 1
59     ---
60 components:
61     - name: input_connection
62       type: nvidia::gxf::Connection
63       parameters:
64         source: replayer/output
65         target: recorder/input
66     ---
67 name: scheduler
68 components:
69     - name: clock
70       type: nvidia::gxf::RealtimeClock
71     - name: greedy_scheduler
72       type: nvidia::gxf::GreedyScheduler
73       parameters:
74         clock: clock

```

Above:

- The replayer reads data from `/workspace/test_data/endoscopy/video/surgical_video.gxf_[index|entities]` files, deserializes the binary data to a `nvidia::gxf::Tensor` using `VideoStreamSerializer`, and puts the data on an output message in the `replayer/output` transmitter queue.
- The `input_connection` component connects the `replayer/output` transmitter queue to the `recorder/input` receiver queue.
- The recorder reads the data in the `input` receiver queue, uses `StdEntitySerializer` to convert the received `nvidia::gxf::Tensor` to a binary stream, and outputs to the `/tmp/tensor_out.gxf_[index|entities]` location specified in the parameters.
- The scheduler component, while not explicitly connected to the application-specific entities, performs the orchestration of the components discussed in the *Data Flow and Triggering Rules*.

Note the use of the `component_serializer` in our newly built recorder. This component is declared separately in the entity

```

- name: entity_serializer
  type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
↳ nvidia::gxf::EntitySerializer
  parameters:
    component_serializers: [component_serializer]

```

and passed into `MyRecorder` via the `serializer` parameter which we exposed in the *extension development section* (*Declare the Parameters to Expose at the Application Level*).

```

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

```

For our app to be able to load (and also compile where necessary) the extensions required at runtime, we need to declare a CMake file `apps/my_recorder_app_gxf/CMakeLists.txt` as follows.

Listing 11.5: `apps/my_recorder_app_gxf/CMakeLists.txt`

```
1 list(APPEND APP_COMMON_EXTENSIONS
2   GFX::std
3   GFX::cuda
4   GFX::multimedia
5   GFX::serialization
6 )
7
8 create_gxe_application(
9   NAME my_recorder_gxf
10  YAML my_recorder_gxf.yaml
11  EXTENSIONS
12    ${APP_COMMON_EXTENSIONS}
13    my_recorder
14    stream_playback
15 )
16
17 # Support automatic datasets download at build time
18
19 # Create a CMake target for the my recorder test
20 add_custom_target(my_recorder_gxf ALL)
21
22 # Download the associated dataset if needed
23 if(HOLOSCAN_DOWNLOAD_DATASETS)
24   add_dependencies(my_recorder_gxf endoscopy_data)
25 endif()
```

In the declaration of `create_gxe_application` we list:

- `my_recorder` component declared in the CMake file of the *extension development section* under the `EXTENSIONS` argument
- the existing `stream_playback` Holoscan extension which reads data from disk

We also create a dependency between `my_recorder_gxf` and `endoscopy_data` targets so that it downloads `endoscopy test data` when building the application.

To make our newly built application discoverable by the build, in the root of the repository, we add the following line

```
add_subdirectory(my_recorder_app_gxf)
```

to `apps/CMakeLists.txt`.

We now have a minimal working application to test the integration of our newly built `MyRecorder` extension.



### 11.1.4 Running the GXF Recorder Application

To run our application in a local development container:

1. Follow the instructions under the [Using a Development Container](#) section steps 1-5 (try clearing the CMake cache by removing the build folder before compiling).

You can execute the following commands to build

```
./run install_gxf
./run build
# ./run clear_cache # if you want to clear build/install/cache folders
```

2. Our test application can now be run in the development container using the command

```
./apps/my_recorder_app_gxf/my_recorder_gxf
```

from inside the development container.

(You can execute `./run launch` to run the development container.)

```
@LINUX:/workspace/holoscan-sdk/build$ ./apps/my_recorder_app_gxf/my_recorder_gxf
2022-08-24 04:46:47.333 INFO gxf/gxe/gxe.cpp@230: Creating context
2022-08-24 04:46:47.339 INFO gxf/gxe/gxe.cpp@107: Loading app: 'apps/my_recorder_
↪app_gxf/my_recorder_gxf.yaml'
2022-08-24 04:46:47.339 INFO gxf/std/yaml_file_loader.cpp@117: Loading GXF_
↪entities from YAML file 'apps/my_recorder_app_gxf/my_recorder_gxf.yaml'...
2022-08-24 04:46:47.340 INFO gxf/gxe/gxe.cpp@291: Initializing...
2022-08-24 04:46:47.437 INFO gxf/gxe/gxe.cpp@298: Running...
2022-08-24 04:46:47.437 INFO gxf/std/greedy_scheduler.cpp@170: Scheduling 2_
↪entities
2022-08-24 04:47:14.829 INFO /workspace/holoscan-sdk/gxf_extensions/stream_
↪playback/video_stream_replayer.cpp@144: Reach end of file or playback count_
↪reaches to the limit. Stop ticking.
2022-08-24 04:47:14.829 INFO gxf/std/greedy_scheduler.cpp@329: Scheduler stopped:_
↪Some entities are waiting for execution, but there are no periodic or async_
↪entities to get out of the deadlock.
2022-08-24 04:47:14.829 INFO gxf/std/greedy_scheduler.cpp@353: Scheduler finished.
2022-08-24 04:47:14.829 INFO gxf/gxe/gxe.cpp@320: Deinitializing...
2022-08-24 04:47:14.863 INFO gxf/gxe/gxe.cpp@327: Destroying context
2022-08-24 04:47:14.863 INFO gxf/gxe/gxe.cpp@333: Context destroyed.
```

A successful run (it takes about 30 secs) will result in output files (`tensor_out.gxf_index` and `tensor_out.gxf_entities` in `/tmp`) that match the original input files (`surgical_video.gxf_index` and `surgical_video.gxf_entities` under `test_data/endoscopy/video`) exactly.

```
@LINUX:/workspace/holoscan-sdk/build$ ls -al /tmp/
total 821384
drwxrwxrwt 1 root root 4096 Aug 24 04:37 .
drwxr-xr-x 1 root root 4096 Aug 24 04:36 ..
drwxrwxrwt 2 root root 4096 Aug 11 21:42 .X11-unix
-rw-r--r-- 1 1000 1000 729309 Aug 24 04:47 gxf_log
-rw-r--r-- 1 1000 1000 840054484 Aug 24 04:47 tensor_out.gxf_entities
-rw-r--r-- 1 1000 1000 16392 Aug 24 04:47 tensor_out.gxf_index
```

(continues on next page)

(continued from previous page)

```
@LINUX:/workspace/holoscan-sdk/build$ ls -al ../test_data/endoscopy/video/
total 839116
drwxr-xr-x 2 1000 1000      4096 Aug 24 02:08 .
drwxr-xr-x 4 1000 1000      4096 Aug 24 02:07 ..
-rw-r--r-- 1 1000 1000 19164125 Jun 17 16:31 raw.mp4
-rw-r--r-- 1 1000 1000 840054484 Jun 17 16:31 surgical_video.gxf_entities
-rw-r--r-- 1 1000 1000    16392 Jun 17 16:31 surgical_video.gxf_index
```

## 11.1.5 GXF User Guide

### Overview

GXF is a modular and extensible framework to build high-performance AI applications.

- Enable developers to reuse components and app graphs between different products to build their own applications.
- Enable developers to use common data formats.
- Enable developers with tools to build and analyze their applications.

### GXF Core

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

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

### GXF Extensions

A GXF extension is a shared library and/or header file containing one or more components.

## Graph Specification

Graph Specification is a format to describe high-performance AI applications in a modular and extensible way. It allows writing applications in a standard format and sharing components across multiple applications without code modification. Graph Specification is based on entity-composition pattern. Every object in graph is represented with entity (aka Node) and components. Developers implement custom components which can be added to entity to achieve the required functionality.

## Concepts

The graph contains nodes which follow an entity-component design pattern implementing the “composition over inheritance” paradigm. A node itself is just a light-weight object which owns components. Components define how a node interacts with the rest of the applications. For example, nodes be connected to pass data between each other. A special component, called compute component, is used to execute the code based on certain rules. Typically a compute component would receive data, execute some computation and publish data.

## Graph

A graph is a data-driven representation of an AI application. Implementing an application by using programming code to create and link objects results in a monolithic and hard to maintain program. Instead a graph object is used to structure an application. The graph can be created using specialized tools and it can be analyzed to identify potential problems or performance bottlenecks. The graph is loaded by the graph runtime to be executed.

The functional blocks of a graph are defined by the set of nodes which the graph owns. Nodes can be queried via the graph using certain query functions. For example, it is possible to search for a node by its name.

## SubGraph

A subgraph is a graph with additional node for interfaces. It points to the components which are accessible outside this graph. In order to use a subgraph in an existing graph or subgraph, the developer needs to create an entity where a component of the type `nvidia::gxf::Subgraph` is contained. Inside the Subgraph component a corresponding subgraph can be loaded from the yaml file indicated by *location* property and instantiated in the parent graph.

System makes the components from interface available to the parent graph when a sub-graph is loaded in the parent graph. It allows users to link sub-graphs in parent with defined interface.

A subgraph interface can be defined as follows:

```
---
interfaces:
  - name: iname # the name of the interface for the access from the parent graph
    target: n_entity/n_component # the true component in the subgraph that is represented
    ↪by the interface
```

### Node

Graph Specification uses an entity-component design principle for nodes. This means that a node is a light-weight object whose main purpose is to own components. A node is a composition of components. Every component is in exactly one node. In order to customize a node a developer does not derive from node as a base class, but instead composes objects out of components. Components can be used to provide a rich set of functionality to a node and thus to an application.

### Components

Components are the main functional blocks of an application. Graph runtime provides a couple of components which implement features like properties, code execution, rules and message passing. It also allows a developer to extend the runtime by injecting her own custom components with custom features to fit a specific use case.

The most common component is a codelet or compute component which is used for data processing and code execution. To implement a custom codelet you'll need to implement a certain set of functions like *start* and *stop*. A special system - the *scheduler* - will call these functions at the specified time. Typical examples of triggering code execution are: receiving a new message from another node, or performing work on a regular schedule based on a time trigger.

### Edges

Nodes can receive data from other nodes by connecting them with an edge. This essential feature allows a graph to represent a compute pipeline or a complicated AI application. An input to a node is called sink while an output is called source. There can be zero, one or multiple inputs and outputs. A source can be connected to multiple sinks and a sink can be connected to multiple sources.

### Extension

An extension is a compiled shared library of a logical group of component type definitions and their implementations along with any other asset files that are required for execution of the components. Some examples of asset files are model files, shared libraries that the extension library links to and hence required to run, header and development files that enable development of additional components and extensions that use components from the extension.

An extension library is a runtime loadable module compiled with component information in a standard format that allows the graph runtime to load the extension and retrieve further information from it to:

- Allow the runtime to create components using the component types in the extension.
- Query information regarding the component types in the extension:
  - The component type name
  - The base type of the component
  - A string description of the component
  - Information of parameters of the component – parameter name, type, description etc.,
- Query information regarding the extension itself - Name of the extension, version, license, author and a string description of the extension.

## Graph File Format

Graph file stores list of dependencies and list of entities. Each entity has a unique name and list of components. Each component has a name, a type and properties. Properties are stored as key-value pairs. *Dependencies* describe extensions used in graph with version required. *Dependencies* information is used by registry to download the correct versions of all the required extensions.

```
%YAML 1.2
---
dependencies:
- extension: StandardExtension
  uuid: 8ec2d5d6-b5df-48bf-8dee-0252606fdd7e
  version: 1.0.0
- extension: test
  uuid: 346eecbc-9039-37da-8456-44fe9ac6492c
  version: 1.0.0
---
name: source
components:
- name: signal
  type: sample::test::ping
- type: nvidia::gxf::CountSchedulingTerm
  parameters:
    count: 10
---
components:
- type: nvidia::gxf::GreedyScheduler
  parameters:
    realtime: false
    max_duration_ms: 1000000
```

## Graph Execution Engine

Graph Execution Engine is used to execute AI application graphs. It accepts multiple graph files as input, and all graphs are executed in same process context. It also needs manifest files as input which includes list of extensions to load. It must list all extensions required for the graph.

```
gxe --help
Flags from gxf/gxe/gxe.cpp:
-app (GXF app file to execute. Multiple files can be comma-separated)
  type: string default: ""
-graph_directory (Path to a directory for searching graph files.)
  type: string default: ""
-log_file_path (Path to a file for logging.) type: string default: ""
-manifest (GXF manifest file with extensions. Multiple files can be
  comma-separated) type: string default: ""
-severity (Set log severity levels: 0=None, 1=Error, 2=Warning, 3=Info,
  4=Debug. Default: Info) type: int32 default: 3
```

## GXF Core C APIs

### Context

#### Create context

```
gxf_result_t GxfContextCreate(gxf_context_t* context);
```

Creates a new GXF context

A GXF context is required for all almost all GXF operations. The context must be destroyed with ‘GxfContextDestroy’. Multiple contexts can be created in the same process, however they can not communicate with each other.

parameter: `context` The new GXF context is written to the given pointer.

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

#### Create a context from a shared context

```
gxf_result_t GxfContextCreate1(gxf_context_t shared, gxf_context_t* context);
```

Creates a new runtime context from shared context.

A shared runtime context is used for sharing entities between graphs running within the same process.

parameter: `shared` A valid GXF shared context.

parameter: `context` The new GXF context is written to the given pointer

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

#### Destroy context

```
gxf_result_t GxfContextDestroy(gxf_context_t context);
```

Destroys a GXF context

Every GXF context must be destroyed by calling this function. The context must have been previously created with ‘GxfContextCreate’. This will also destroy all entities and components which were created as part of the context.

parameter: `context` A valid GXF context.

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

#### Extensions

Maximum number of extensions in a context can be 1024.

## Load Extensions from a file

```
gxf_result_t GxfLoadExtension(gxf_context_t context, const char* filename);
```

Loads extension in the given context from file.

parameter: context A valid GXF context

parameter: filename A valid filename.

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

*This function will be deprecated.*

## Load Extension libraries

```
gxf_result_t GxfLoadExtensions(gxf_context_t context, const GxfLoadExtensionsInfo* info);
```

Loads GXF extension libraries

Loads one or more extensions either directly by their filename or indirectly by loading manifest files. Before a component can be added to a GXF entity the GXF extension shared library providing the component must be loaded. An extensions must only be loaded once.

To simplify loading multiple extensions at once the developer can create a manifest file which lists all extensions he needs. This function will then load all extensions listed in the manifest file. Multiple manifest may be loaded, however each extensions may still be loaded only a single time.

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

A manifest file is a YAML file with a single top-level entry 'extensions' followed by a list of filenames of GXF extension shared libraries.

parameter: context A valid GXF context

parameter: filename A valid filename.

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

```
gxf_result_t GxfLoadExtensionManifest(gxf_context_t context, const char*
manifest_filename);
```

Loads extensions from manifest file.

parameter: context A valid GXF context.

parameter: filename A valid filename.

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

*This function will be deprecated.*

## Load Metadata files

```
gxf_result_t GxfLoadExtensionMetadataFiles(gxf_context_t context, const char* const*
filenames, uint32_t count);
```

Loads an extension registration metadata file

Reads a metadata file of the contents of an extension used for registration. These metadata files can be used to resolve typename and TID's of components for other extensions which depend on them. Metadata files do not contain the actual implementation of the extension and must be loaded only to run the extension query API's on extension libraries which have the actual implementation and only depend on the metadata for type resolution.

If some components of extension B depend on some components in extension A: - Load metadata file for extension A  
- Load extension library for extension B using 'GxfLoadExtensions' - Run extension query api's on extension B and it's components.

parameter: `context` A valid GXF context.

parameter: `filenames` absolute paths of metadata files generated by the registry during extension registration

parameter: `count` The number of metadata files to be loaded

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

## Register component

```
gxf_result_t GxfRegisterComponent(gxf_context_t context, gxf_tid_t tid, const char* name,
const char* base_name);
```

Registers a component with a GXF extension

A GXF extension need to register all of its components in the extension factory function. For convenience the helper macros in `gxf/std/extension_factory_helper.hpp` can be used.

The developer must choose a unique GXF tid with two random 64-bit integers. The developer must ensure that every GXF component has a unique tid. The name of the component must be the fully qualified C++ type name of the component. A component may only have a single base class and that base class must be specified with its fully qualified C++ type name as the parameter 'base\_name'.

ref: `gxf/std/extension_factory_helper.hpp` ref: `core/type_name.hpp`

parameter: `context` A valid GXF context

parameter: `tid` The chosen GXF tid

parameter: `name` The type name of the component

parameter: `base_name` The type name of the base class of the component

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



## Graph Execution

### Loads a list of entities from YAML file

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

parameter: context A valid GXF context

parameter: filename A valid YAML filename.

parameter: params\_override An optional array of strings used for override parameters in yaml file.

parameter: num\_overrides Number of optional override parameter strings.

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

### Set the root folder for searching YAML files during loading

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

parameter: context A valid GXF context

parameter: path Path to root folder for searching YAML files during loading

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

### Loads a list of entities from YAML text

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

parameter: context A valid GXF context

parameter: text A valid YAML text.

parameter: params\_override An optional array of strings used for override parameters in yaml file.

parameter: num\_overrides Number of optional override parameter strings.

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

### Activate all system components

```
gxf_result_t GxfGraphActivate(gxf_context_t context);
```

parameter: context A valid GXF context

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

### Deactivate all System components

```
gxf_result_t GxfGraphDeactivate(gxf_context_t context);
```

parameter: context A valid GXF context

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

### Starts the execution of the graph asynchronously

```
gxf_result_t GxfGraphRunAsync(gxf_context_t context);
```

parameter: context A valid GXF context

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

### Interrupt the execution of the graph

```
gxf_result_t GxfGraphInterrupt(gxf_context_t context);
```

parameter: context A valid GXF context

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

### Waits for the graph to complete execution

```
gxf_result_t GxfGraphWait(gxf_context_t context);
```

parameter: context A valid GXF context

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

### Runs all System components and waits for their completion

```
gxf_result_t GxfGraphRun(gxf_context_t context);
```

parameter: context A valid GXF context

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

## Entities

### Create an entity

```
gxf_result_t GxfEntityCreate(gxf_context_t context, gxf_uid_t* eid);
```

Creates a new entity and updates the eid to the unique identifier of the newly created entity.

*This method will be deprecated.*

```
gxf_result_t GxfCreateEntity((gxf_context_t context, const GxfEntityCreateInfo* info,  
gxf_uid_t* eid);
```

Create a new GXF entity.

Entities are light-weight containers to hold components and form the basic building blocks of a GXF application. Entities are created when a GXF file is loaded, or they can be created manually using this function. Entities created with this function must be destroyed using ‘GxfEntityDestroy’. After the entity was created components can be added to it with ‘GxfComponentAdd’. To start execution of codelets on an entity the entity needs to be activated first. This can happen automatically using ‘GXF\_ENTITY\_CREATE\_PROGRAM\_BIT’ or manually using ‘GxfEntityActivate’.

parameter **context**: GXF context that creates the entity. parameter **info**: pointer to a GxfEntityCreateInfo structure containing parameters affecting the creation of the entity. parameter **eid**: pointer to a gxf\_uid\_t handle in which the resulting entity is returned. returns: GXF\_SUCCESS if the operation was successful, or otherwise one of the GXF error codes.

### Activate an entity

```
gxf_result_t GxfEntityActivate(gxf_context_t context, gxf_uid_t eid);
```

Activates a previously created and inactive entity

Activating an entity generally marks the official start of its lifetime and has multiple implications: - If mandatory parameters, i.e. parameter which do not have the flag “optional”, are not set the operation will fail.

- All components on the entity are initialized.
- All codelets on the entity are scheduled for execution. The scheduler will start calling start, tick and stop functions as specified by scheduling terms.
- After activation trying to change a dynamic parameters will result in a failure.
- Adding or removing components of an entity after activation will result in a failure.

parameter: **context** A valid GXF context

parameter: **eid** UID of a valid entity

returns: GXF error code

### Deactivate an entity

```
gxf_result_t GxfEntityDeactivate(gxf_context_t context, gxf_uid_t eid);
```

Deactivates a previously activated entity

---

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

---

Deactivating an entity generally marks the official end of its lifetime and has multiple implications:

- All codelets are removed from the schedule. Already running entities are run to completion.
- All components on the entity are deinitialized.
- Components can be added or removed again once the entity was deactivated.
- Mandatory and non-dynamic parameters can be changed again.

parameter: **context** A valid GXF context

parameter: **eid** UID of a valid entity

returns: GXF error code

## Destroy an entity

```
gxf_result_t GxfEntityDestroy(gxf_context_t context, gxf_uid_t eid);
```

Destroys a previously created entity

Destroys an entity immediately. The entity is destroyed even if the reference count has not yet reached 0. If the entity is active it is deactivated first.

Note: This function can block for the same reasons as 'GxfEntityDeactivate'.

parameter: `context` A valid GXF context

parameter: `eid` The returned UID of the created entity

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

## Find an entity

```
gxf_result_t GxfEntityFind(gxf_context_t context, const char* name, gxf_uid_t* eid);
```

Finds an entity by its name

parameter: `context` A valid GXF context

parameter: `name` A C string with the name of the entity. Ownership is not transferred.

parameter: `eid` The returned UID of the entity

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

## Find all entities

```
gxf_result_t GxfEntityFindAll(gxf_context_t context, uint64_t* num_entities, gxf_uid_t* entities);
```

Finds all entities in the current application

Finds and returns all entity ids for the current application. If more than *max\_entities* exist only *max\_entities* will be returned. The order and selection of entities returned is arbitrary.

parameter: `context` A valid GXF context

parameter: `num_entities` In/Out: the max number of entities that can fit in the buffer/the number of entities that exist in the application

parameter: `entities` A buffer allocated by the caller for returned UIDs of all entities, with capacity for *num\_entities*.

returns: `GXF_SUCCESS` if the operation was successful, `GXF_QUERY_NOT_ENOUGH_CAPACITY` if more entites exist in the application than *max\_entities*, or otherwise one of the GXF error codes.

### Increase reference count of an entity

```
gxf_result_t GxfEntityRefCountInc(gxf_context_t context, gxf_uid_t eid);
```

Increases the reference count for an entity by 1.

By default reference counting is disabled for an entity. This means that entities created with 'GxfEntityCreate' are not automatically destroyed. If this function is called for an entity with disabled reference count, reference counting is enabled and the reference count is set to 1. Once reference counting is enabled an entity will be automatically destroyed if the reference count reaches zero, or if 'GxfEntityCreate' is called explicitly.

parameter: `context` A valid GXF context

parameter: `eid` The UID of a valid entity

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

### Decrease reference count of an entity

```
gxf_result_t GxfEntityRefCountDec(gxf_context_t context, gxf_uid_t eid);
```

Decreases the reference count for an entity by 1.

See 'GxfEntityRefCountInc' for more details on reference counting.

parameter: `context` A valid GXF context

parameter: `eid` The UID of a valid entity

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

### Get status of an entity

```
gxf_result_t GxfEntityGetStatus(gxf_context_t context, gxf_uid_t eid,  
gxf_entity_status_t* entity_status);
```

Gets the status of the entity.

See 'gxf\_entity\_status\_t' for the various status.

parameter: `context` A valid GXF context

parameter: `eid` The UID of a valid entity

parameter: `entity_status` output; status of an entity eid

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

### Get state of an entity

```
gxf_result_t GxfEntityGetState(gxf_context_t context, gxf_uid_t eid, entity_state_t*  
entity_state);
```

Gets the state of the entity.

See 'gxf\_entity\_status\_t' for the various status.

parameter: `context` A valid GXF context

parameter: `eid` The UID of a valid entity

parameter: `entity_state` output; behavior status of an entity `eid` used by the behavior tree parent codelet

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

### Notify entity of an event

```
gxf_result_t GxfEntityEventNotify(gxf_context_t context, gxf_uid_t eid);
```

Notifies the occurrence of an event and inform the scheduler to check the status of the entity

The entity must have an ‘AsynchronousSchedulingTerm’ scheduling term component and it must be in “EVENT\_WAITING” state for the notification to be acknowledged.

See ‘AsynchronousEventState’ for various states

parameter: `context` A valid GXF context

parameter: `eid` The UID of a valid entity

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

### Components

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

### Get component type identifier

```
gxf_result_t GxfComponentTypeId(gxf_context_t context, const char* name, gxf_tid_t* tid);
```

Gets the GXF unique type ID (TID) of a component

Get the unique type ID which was used to register the component with GXF. The function expects the fully qualified C++ type name of the component including namespaces.

Example of a valid component type name: “nvidia::gxf::test::PingTx”

parameter: `context` A valid GXF context

parameter: `name` The fully qualified C++ type name of the component

parameter: `tid` The returned TID of the component

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

### Get component type name

```
gxf_result_t GxfComponentTypeName(gxf_context_t context, gxf_tid_t tid, const char** name);
```

Gets the fully qualified C++ type name GXF component typename

Get the unique typename of the component with which it was registered using one of the `GXF_EXT_FACTORY_ADD*()` macros

parameter: `context` A valid GXF context

parameter: `tid` The unique type ID (TID) of the component with which the component was registered

parameter: `name` The returned name of the component

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

### Get component name

```
gxf_result_t GxfComponentName(gxf_context_t context, gxf_uid_t cid, const char** name);
```

Gets the name of a component

Each component has a user-defined name which was used in the call to ‘GxfComponentAdd’. Usually the name is specified in the GXF application file.

parameter: `context` A valid GXF context

parameter: `cid` The unique object ID (UID) of the component

parameter: `name` The returned name of the component

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

### Get unique identifier of the entity of given component

```
gxf_result_t GxfComponentEntity(gxf_context_t context, gxf_uid_t cid, gxf_uid_t* eid);
```

Gets the unique object ID of the entity of a component

Each component has a unique ID with respect to the context and is stored in one entity. This function can be used to retrieve the ID of the entity to which a given component belongs.

parameter: `context` A valid GXF context

parameter: `cid` The unique object ID (UID) of the component

parameter: `eid` The returned UID of the entity

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

### Add a new component

```
gxf_result_t GxfComponentAdd(gxf_context_t context, gxf_uid_t eid, gxf_tid_t tid, const char* name, gxf_uid_t* cid);
```

Adds a new component to an entity

An entity can contain multiple components and this function can be used to add a new component to an entity. A component must be added before an entity is activated, or after it was deactivated. Components must not be added to active entities. The order of components is stable and identical to the order in which components are added (see ‘GxfComponentFind’).

parameter: `context` A valid GXF context

parameter: `eid` The unique object ID (UID) of the entity to which the component is added.

parameter: `tid` The unique type ID (TID) of the component to be added to the entity.

parameter: `name` The name of the new component. Ownership is not transferred.

parameter: `cid` The returned UID of the created component

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

### Add component to entity interface

```
gxf_result_t GxfComponentAddToInterface(gxf_context_t context, gxf_uid_t eid, gxf_uid_t cid, const char* name);
```

Adds an existing component to the interface of an entity

An entity can hold references to other components in its interface, so that when finding a component in an entity, both the component this entity holds and those it refers to will be returned. This supports the case when an entity contains a subgraph, then those components that have been declared in the subgraph interface will be put to the interface of the parent entity.

parameter: `context` A valid GXF context

parameter: `eid` The unique object ID (UID) of the entity to which the component is added.

parameter: `cid` The unique object ID of the component.

parameter: `name` The name of the new component. Ownership is not transferred.

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

### Find a component in an entity

```
gxf_result_t GxfComponentFind(gxf_context_t context, gxf_uid_t eid, gxf_tid_t tid, const char* name, int32_t* offset, gxf_uid_t* cid);
```

Finds a component in an entity

Searches components in an entity which satisfy certain criteria: component type, component name, and component min index. All three criteria are optional; in case no criteria is given the first component is returned. The main use case for “component min index” is a repeated search which continues at the index which was returned by a previous search.

In case no entity with the given criteria was found `GXF_ENTITY_NOT_FOUND` is returned.

parameter: `context` A valid GXF context

parameter: `eid` The unique object ID (UID) of the entity which is searched.

parameter: `tid` The component type ID (TID) of the component to find (optional)

parameter: `name` The component name of the component to find (optional). Ownership not transferred.

parameter: `offset` The index of the first component in the entity to search. Also contains the index of the component which was found.

parameter: `cid` The returned UID of the searched component

returns: `GXF_SUCCESS` if a component matching the criteria was found, `GXF_ENTITY_NOT_FOUND` if no component matching the criteria was found, or otherwise one of the GXF error codes.



### Get type identifier for a component

```
gxf_result_t GxfComponentType(gxf_context_t context, gxf_uid_t cid, gxf_tid_t* tid);
```

Gets the component type ID (TID) of a component

parameter: `context` A valid GXF context

parameter: `cid` The component object ID (UID) for which the component type is requested.

parameter: `tid` The returned TID of the component

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

### Gets pointer to component

```
gxf_result_t GxfComponentPointer(gxf_context_t context, gxf_uid_t uid, gxf_tid_t tid, void** pointer);
```

Verifies that a component exists, has the given type, gets a pointer to it.

parameter: `context` A valid GXF context

parameter: `uid` The component object ID (UID).

parameter: `tid` The expected component type ID (TID) of the component

parameter: `pointer` The returned pointer to the component object.

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

## Primitive Parameters

### 64-bit floating point

#### Set

```
gxf_result_t GxfParameterSetFloat64(gxf_context_t context, gxf_uid_t uid, const char* key, double value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` a double value

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

## Get

```
gxf_result_t GxfParameterGetFloat64(gxf_context_t context, gxf_uid_t uid, const char* key, double* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the double value.

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

## 64-bit signed integer

### Set

```
gxf_result_t GxfParameterSetInt64(gxf_context_t context, gxf_uid_t uid, const char* key, int64_t value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value 64-bit integer value to set.

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

### Get

```
gxf_result_t GxfParameterGetInt64(gxf_context_t context, gxf_uid_t uid, const char* key, int64_t* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the 64-bit integer value.

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

## 64-bit unsigned integer

### Set

```
gxf_result_t GxfParameterSetUInt64(gxf_context_t context, gxf_uid_t uid, const char* key, uint64_t value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value unsigned 64-bit integer value to set.

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

### Get

```
gxf_result_t GxfParameterGetUInt64(gxf_context_t context, gxf_uid_t uid, const char* key,
uint64_t* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the unsigned 64-bit integer value.

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

## 32-bit signed integer

### Set

```
gxf_result_t GxfParameterSetInt32(gxf_context_t context, gxf_uid_t uid, const char* key,
int32_t value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value 32-bit integer value to set.

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

### Get

```
gxf_result_t GxfParameterGetInt32(gxf_context_t context, gxf_uid_t uid, const char* key,
int32_t* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the 32-bit integer value.

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

## String parameter

### Set

```
gxf_result_t GxfParameterSetStr(gxf_context_t context, gxf_uid_t uid, const char* key,
const char* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value A char array containing value to set.

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

### Get

```
gxf_result_t GxfParameterGetStr(gxf_context_t context, gxf_uid_t uid, const char* key,
const char** value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to a char\* array to get the value.

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

## Boolean

### Set

```
gxf_result_t GxfParameterSetBool(gxf_context_t context, gxf_uid_t uid, const char* key,
bool value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value A boolean value to set.

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

## Get

```
gxf_result_t GxfParameterGetBool(gxf_context_t context, gxf_uid_t uid, const char* key,
bool* value);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value pointer to get the boolean value.

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

## Handle

### Set

```
gxf_result_t GxfParameterSetHandle(gxf_context_t context, gxf_uid_t uid, const char* key,
gxf_uid_t cid);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: cid Unique identifier to set.

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

### Get

```
gxf_result_t GxfParameterGetHandle(gxf_context_t context, gxf_uid_t uid, const char* key,
gxf_uid_t* cid);
```

parameter: context A valid GXF context.

parameter: uid A valid component identifier.

parameter: key A valid name of a component to set.

parameter: value Pointer to a unique identifier to get the value.

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

## Vector Parameters

To set or get the vector parameters of a component, users can use the following C-APIs for various data types:

## Set 1-D Vector Parameters

Users can call `gxf_result_t GxfParameterSet1D(DataType"Vector"(gxf_context_t context, gxf_uid_t uid, const char* key, data_type* value, uint64_t length)`

value should point to an array of the data to be set of the corresponding type. The size of the stored array should match the length argument passed.

See the table below for all the supported data types and their corresponding function signatures.

parameter: key The name of the parameter

parameter: value The value to set of the parameter

parameter: length The length of the vector parameter

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

Table 11.1: Supported Data Types to Set 1D Vector Parameters

Function Name	data_type
<code>GxfParameterSet1DFloat64Vector(...)</code>	<code>double</code>
<code>GxfParameterSet1DInt64Vector(...)</code>	<code>int64_t</code>
<code>GxfParameterSet1DUInt64Vector(...)</code>	<code>uint64_t</code>
<code>GxfParameterSet1DInt32Vector(...)</code>	<code>int32_t</code>

## Set 2-D Vector Parameters

Users can call `gxf_result_t GxfParameterSet2D(DataType"Vector"(gxf_context_t context, gxf_uid_t uid, const char* key, data_type** value, uint64_t height, uint64_t width)`

value should point to an array of array (and not to the address of a contiguous array of data) of the data to be set of the corresponding type. The length of the first dimension of the array should match the height argument passed and similarly the length of the second dimension of the array should match the width passed.

See the table below for all the supported data types and their corresponding function signatures.

parameter: key The name of the parameter

parameter: value The value to set of the parameter

parameter: height The height of the 2-D vector parameter

parameter: width The width of the 2-D vector parameter

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

Table 11.2: Supported Data Types to Set 2D Vector Parameters

Function Name	data_type
<code>GxfParameterSet2DFloat64Vector(...)</code>	<code>double</code>
<code>GxfParameterSet2DInt64Vector(...)</code>	<code>int64_t</code>
<code>GxfParameterSet2DUInt64Vector(...)</code>	<code>uint64_t</code>
<code>GxfParameterSet2DInt32Vector(...)</code>	<code>int32_t</code>

## Get 1-D Vector Parameters

Users can call `gxf_result_t GxfParameterGet1D"DataType"Vector(gxf_context_t context, gxf_uid_t uid, const char* key, data_type** value, uint64_t* length)` to get the value of a 1-D vector.

Before calling this method, users should call `GxfParameterGet1D"DataType"VectorInfo(gxf_context_t context, gxf_uid_t uid, const char* key, uint64_t* length)` to obtain the length of the vector parameter and then allocate at least that much memory to retrieve the value.

value should point to an array of size greater than or equal to `length` allocated by user of the corresponding type to retrieve the data. If the `length` doesn't match the size of stored vector then it will be updated with the expected size.

See the table below for all the supported data types and their corresponding function signatures.

parameter: key The name of the parameter

parameter: value The value to set of the parameter

parameter: length The length of the 1-D vector parameter obtained by calling `GxfParameterGet1D"DataType"VectorInfo(...)`

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

Function Name	data_type
<code>GxfParameterGet1DFloat64Vector(...)</code>	<code>double</code>
<code>GxfParameterGet1DInt64Vector(...)</code>	<code>int64_t</code>
<code>GxfParameterGet1DUInt64Vector(...)</code>	<code>uint64_t</code>
<code>GxfParameterGet1DInt32Vector(...)</code>	<code>int32_t</code>

## Get 2-D Vector Parameters

Users can call `gxf_result_t GxfParameterGet2D"DataType"Vector(gxf_context_t context, gxf_uid_t uid, const char* key, data_type** value, uint64_t* height, uint64_t* width)` to get the value of a 2D vector.

Before calling this method, users should call `GxfParameterGet1D"DataType"VectorInfo(gxf_context_t context, gxf_uid_t uid, const char* key, uint64_t* height, uint64_t* width)` to obtain the height and width of the 2D-vector parameter and then allocate at least that much memory to retrieve the value.

value should point to an array of array of height (size of first dimension) greater than or equal to `height` and width (size of the second dimension) greater than or equal to `width` allocated by user of the corresponding type to get the data. If the `height` or `width` don't match the height and width of the stored vector then they will be updated with the expected values.

See the table below for all the supported data types and their corresponding function signatures.

parameter": key The name of the parameter

parameter": value Allocated array to get the value of the parameter

parameter": height The height of the 2-D vector parameter obtained by calling `GxfParameterGet2D"DataType"VectorInfo(...)`

parameter": width The width of the 2-D vector parameter obtained by calling `GxfParameterGet2D"DataType"VectorInfo(...)`

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

Function Name	data_type
GxfParameterGet2DFloat64Vector(...)	double
GxfParameterGet2DInt64Vector(...)	int64_t
GxfParameterGet2DUInt64Vector(...)	uint64_t
GxfParameterGet2DInt32Vector(...)	int32_t

## Information Queries

### Get Meta Data about the GXF Runtime

```
gxf_result_t GxfRuntimeInfo(gxf_context_t context, gxf_runtime_info* info);
```

parameter: context A valid GXF context.

parameter: info pointer to gxf\_runtime\_info object to get the meta data.

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

### Get description and list of components in loaded Extension

```
gxf_result_t GxfExtensionInfo(gxf_context_t context, gxf_tid_t tid, gxf_extension_info_t* info);
```

parameter: context A valid GXF context.

parameter: tid The unique identifier of the extension.

parameter: info pointer to gxf\_extension\_info\_t object to get the meta data.

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

### Get description and list of parameters of Component

```
gxf_result_t GxfComponentInfo(gxf_context_t context, gxf_tid_t tid, gxf_component_info_t* info);
```

Note: Parameters are only available after at least one instance is created for the Component.

parameter: context A valid GXF context.

parameter: tid The unique identifier of the component.

parameter: info pointer to gxf\_component\_info\_t object to get the meta data.

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



### Get parameter type description

Gets a string describing the parameter type

```
const char* GxfParameterTypeStr(gxf_parameter_type_t param_type);
```

parameter: `param_type` Type of parameter to get info about.

returns: C-style string description of the parameter type.

### Get flag type description

Gets a string describing the flag type

```
const char* GxfParameterFlagTypeStr(gxf_parameter_flags_t flag_type);
```

parameter: `flag_type` Type of flag to get info about.

returns: C-style string description of the flag type.

### Get parameter description

Gets description of specific parameter. Fails if the component is not instantiated yet.

```
gxf_result_t GxfGetParameterInfo(gxf_context_t context, gxf_tid_t cid, const char* key,
gxf_parameter_info_t* info);
```

parameter: `context` A valid GXF context.

parameter: `cid` The unique identifier of the component.

parameter: `key` The name of the parameter.

parameter: `info` Pointer to a `gxf_parameter_info_t` object to get the value.

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

### Redirect logs to a file

Redirect console logs to the provided file.

```
gxf_result_t GxfGetParameterInfo(gxf_context_t context, FILE* fp);
```

parameter: `context` A valid GXF context.

parameter: `fp` File path for the redirected logs.

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

## Miscellaneous

### Get string description of error

```
const char* GxfResultStr(gxf_result_t result);
```

Gets a string describing an GXF error code.

The caller does not get ownership of the return C string and must not delete it.

parameter: `result` A GXF error code

returns: A pointer to a C string with the error code description.

## The GXF Scheduler

The execution of entities in a graph is governed by the scheduler and the scheduling terms associated with every entity. A scheduler is a component responsible for orchestrating the execution of all the entities defined in a graph. A scheduler typically keeps track of the graph entities and their current execution states and passes them on to a `nvidia::gxf::EntityExecutor` component when ready for execution. The following diagram depicts the flow for an entity execution.

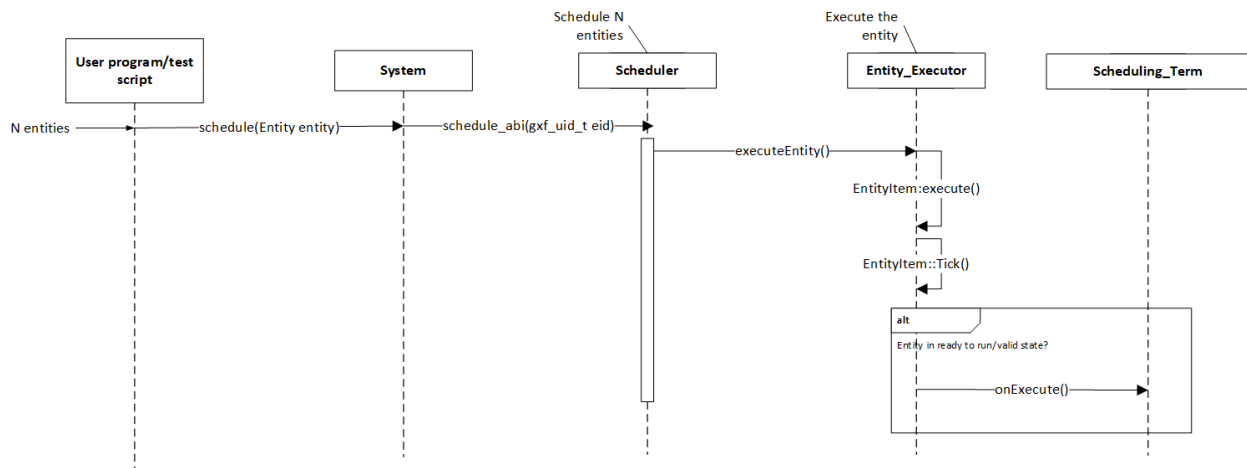


Fig. 11.3: Entity execution sequence

As shown in the sequence diagram, the schedulers begin executing the graph entities via the `nvidia::gxf::System::runAsync_abi()` interface and continue this process until it meets the certain ending criteria. A single entity can have multiple codelets. These codelets are executed in the same order in which they were defined in the entity. A failure in execution of any single codelet stops the execution of all the entities. Entities are naturally unscheduled from execution when any one of their scheduling term reaches NEVER state.

Scheduling terms are components used to define the execution readiness of an entity. An entity can have multiple scheduling terms associated with it and each scheduling term represents the state of an entity using `SchedulingCondition`.

The table below shows various states of `nvidia::gxf::SchedulingConditionType` described using `nvidia::gxf::SchedulingCondition`.

SchedulingConditionType	Description
NEVER	Entity will never execute again
READY	Entity is ready for execution
WAIT	Entity may execute in the future
WAIT_TIME	Entity will be ready for execution after specified duration
WAIT_EVENT	Entity is waiting on an asynchronous event with unknown time interval

Schedulers define deadlock as a condition when there are no entities which are in READY, WAIT\_TIME or WAIT\_EVENT state which guarantee execution at a future point in time. This implies all the entities are in WAIT state for which the scheduler does not know if they ever will reach the READY state in the future. The scheduler can be configured to stop when it reaches such a state using the stop\_on\_deadlock parameter, else the entities are polled to check if any of them have reached READY state. max\_duration configuration parameter can be used to stop execution of all entities regardless of their state after a specified amount of time has elapsed.

There are two types of schedulers currently supported by GXF:

1. Greedy Scheduler
2. Multithread Scheduler

### Greedy Scheduler

This is a basic single threaded scheduler which tests scheduling term greedily. It is great for simple use cases and predictable execution but may incur a large overhead of scheduling term execution, making it unsuitable for large applications. The scheduler requires a clock to keep track of time. Based on the choice of clock the scheduler will execute differently. If a Realtime clock is used the scheduler will execute in real-time. This means pausing execution - sleeping the thread, until periodic scheduling terms are due again. If a ManualClock is used scheduling will happen “time-compressed”. This means flow of time is altered to execute codelets in immediate succession.

The GreedyScheduler maintains a running count of entities which are in READY, WAIT\_TIME and WAIT\_EVENT states. The following activity diagram depicts the gist of the decision making for scheduling an entity by the greedy scheduler:

### Greedy Scheduler Configuration

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

Parameter name	Description
clock	The clock used by the scheduler to define the flow of time. Typical choices are RealtimeClock or ManualClock
max_duration_ms	The maximum duration for which the scheduler will execute (in ms). If not specified, the scheduler will run until all work is done. If periodic terms are present this means the application will run indefinitely
stop_on_deadlock	If stop_on_deadlock is disabled, the GreedyScheduler constantly polls for the status of all the waiting entities to check if any of them are ready for execution.

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

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

(continues on next page)

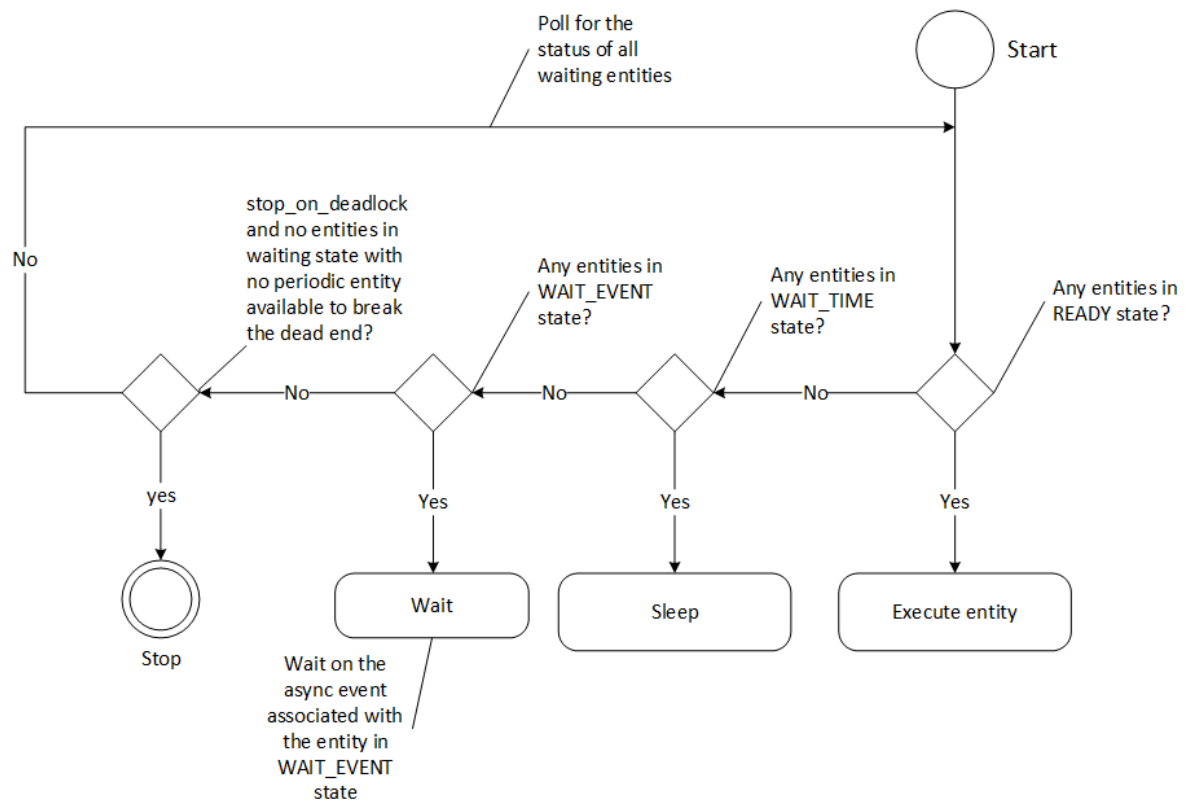


Fig. 11.4: Greedy Scheduler Activity Diagram

(continued from previous page)

```

max_duration_ms: 3000
clock: misc/clock
stop_on_deadlock: true
---
name: misc
components:
- name: clock
  type: nvidia::gxf::ManualClock

```

## Multithread Scheduler

The MultiThread scheduler is more suitable for large applications with complex execution patterns. The scheduler consists of a dispatcher thread which checks the status of an entity and dispatches it to a thread pool of worker threads responsible for executing them. Worker threads enqueue the entity back on to the dispatch queue upon completion of execution. The number of worker threads can be configured using worker\_thread\_number parameter. The MultiThread scheduler also manages a dedicated queue and thread to handle asynchronous events. The following activity diagram demonstrates the gist of the multithread scheduler implementation.

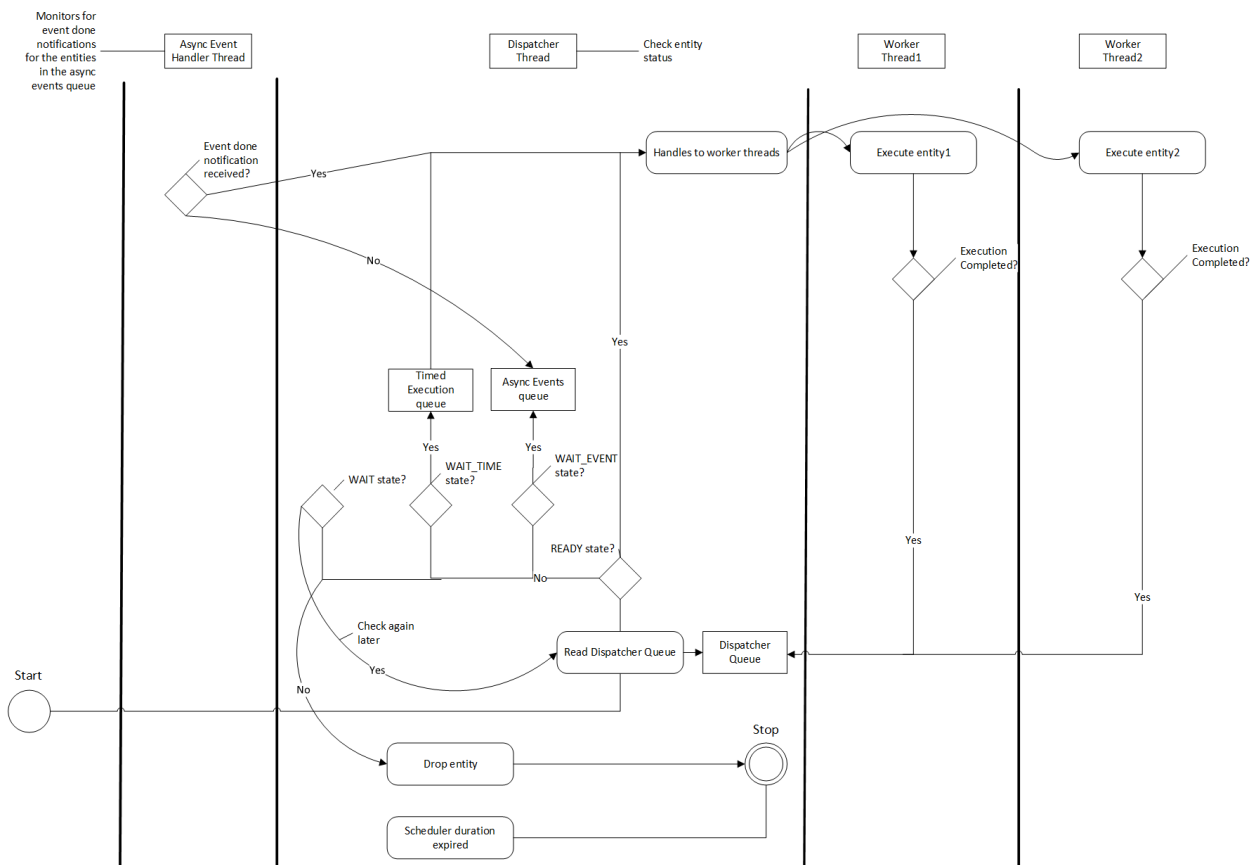


Fig. 11.5: MultiThread Scheduler Activity Diagram

As depicted in the diagram, when an entity reaches WAIT\_EVENT state, it's moved to a queue where they wait to receive event done notification. The asynchronous event handler thread is responsible for moving entities to the dispatcher upon receiving event done notification. The dispatcher thread also maintains a running count of the number

of entities in READY, WAIT\_EVENT and WAIT\_TIME states and uses these statistics to check if the scheduler has reached a deadlock. The scheduler also needs a clock component to keep track of time and it is configured using the clock parameter.

MultiThread scheduler is more resource efficient compared to the Greedy Scheduler and does not incur any additional overhead for constantly polling the states of scheduling terms. The `check_recession_period_ms` parameter can be used to configure the time interval the scheduler must wait to poll the state of entities which are in WAIT state.

## Multithread Scheduler Configuration

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

Parameter name	Description
clock	The clock used by the scheduler to define the flow of time. Typical choices are RealtimeClock or ManualClock.
max_duration_ms	The maximum duration for which the scheduler will execute (in ms). If not specified, the scheduler will run until all work is done. If periodic terms are present this means the application will run indefinitely.
check_recess_period_ms	Duration to sleep before checking the condition of an entity again [ms]. This is the maximum duration for which the scheduler would wait when an entity is not yet ready to run.
stop_on_deadlock	If enabled the scheduler will stop when all entities are in a waiting state, but no periodic entity exists to break the dead end. Should be disabled when scheduling conditions can be changed by external actors, for example by clearing queues manually.
worker_thread_number	Number of threads.

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

```
name: scheduler
components:
- type: nvidia::gxf::MultiThreadScheduler
  parameters:
    max_duration_ms: 5000
    clock: misc/clock
    worker_thread_number: 5
    check_recession_period_ms: 3
    stop_on_deadlock: false
---
name: misc
components:
- name: clock
  type: nvidia::gxf::RealtimeClock
```

## Epoch Scheduler

The Epoch scheduler is used for running loads in externally managed threads. Each run is called an Epoch. The scheduler goes over all entities that are known to be active and executes them one by one. If the epoch budget is provided (in ms), it would keep running all codelets until the budget is consumed or no codelet is ready. It might run over budget since it guarantees to cover all codelets in epoch. In case the budget is not provided, it would go over all the codelets once and execute them only once.

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

Parameter name	Description
clock	The clock used by the scheduler to define the flow of time. Typical choice is a RealtimeClock.

The following code snippet configures an Epoch scheduler:

```
name: scheduler
components:
- name: clock
  type: nvidia::gxf::RealtimeClock
- name: epoch
  type: nvidia::gxf::EpochScheduler
parameters:
  clock: clock
```

Note that the epoch scheduler is intended to run from an external thread. The `runEpoch(float budget_ms);` can be used to set the `budget_ms` and run the scheduler from the external thread. If the specified budget is not positive, all the nodes are executed once.

## SchedulingTerms

A SchedulingTerm defines a specific condition that is used by an entity to let the scheduler know when it's ready for execution. There are various scheduling terms currently supported by GXF.

### PeriodicSchedulingTerm

An entity associated with `nvidia::gxf::PeriodicSchedulingTerm` is ready for execution after periodic time intervals specified using its `recess_period` parameter. The `PeriodicSchedulingTerm` can either be in `READY` or `WAIT_TIME` state.

Example usage:

```
- name: scheduling_term
  type: nvidia::gxf::PeriodicSchedulingTerm
parameters:
  recess_period: 500000000
```

### CountSchedulingTerm

An entity associated with `nvidia::gxf::CountSchedulingTerm` is executed for a specific number of times specified using its count parameter. The `CountSchedulingTerm` can either be in `READY` or `NEVER` state. The scheduling term reaches the `NEVER` state when the entity has been executed count number of times.

Example usage:

```
- name: scheduling_term
  type: nvidia::gxf::CountSchedulingTerm
  parameters:
    count: 42
```

### MessageAvailableSchedulingTerm

An entity associated with `nvidia::gxf::MessageAvailableSchedulingTerm` is executed when the associated receiver queue has at least a certain number of elements. The receiver is specified using the `receiver` parameter of the scheduling term. The minimum number of messages that permits the execution of the entity is specified by `min_size`. An optional parameter for this scheduling term is `front_stage_max_size`, the maximum front stage message count. If this parameter is set, the scheduling term will only allow execution if the number of messages in the queue does not exceed this count. It can be used for codelets which do not consume all messages from the queue.

In the example shown below, the minimum size of the queue is configured to be 4. This means the entity will not be executed until there are at least 4 messages in the queue.

```
- type: nvidia::gxf::MessageAvailableSchedulingTerm
  parameters:
    receiver: tensors
    min_size: 4
```

### MultiMessageAvailableSchedulingTerm

An entity associated with `nvidia::gxf::MultiMessageAvailableSchedulingTerm` is executed when a list of provided input receivers combined have at least a given number of messages. The `receivers` parameter is used to specify a list of the input channels/receivers. The minimum number of messages needed to permit the entity execution is set by `min_size` parameter.

Consider the example shown below. The associated entity will be executed when the number of messages combined for all the three receivers is at least the `min_size`, i.e. 5.

```
- name: input_1
  type: nvidia::gxf::test::MockReceiver
  parameters:
    max_capacity: 10
- name: input_2
  type: nvidia::gxf::test::MockReceiver
  parameters:
    max_capacity: 10
- name: input_3
  type: nvidia::gxf::test::MockReceiver
  parameters:
    max_capacity: 10
```

(continues on next page)



(continued from previous page)

```
- type: nvidia::gxf::MultiMessageAvailableSchedulingTerm
parameters:
  receivers: [input_1, input_2, input_3]
  min_size: 5
```

## BooleanSchedulingTerm

An entity associated with `nvidia::gxf::BooleanSchedulingTerm` is executed when its internal state is set to tick. The parameter `enable_tick` is used to control the entity execution. The scheduling term also has two APIs `enable_tick()` and `disable_tick()` to toggle its internal state. The entity execution can be controlled by calling these APIs. If `enable_tick` is set to false, the entity is not executed (Scheduling condition is set to NEVER). If `enable_tick` is set to true, the entity will be executed (Scheduling condition is set to READY). Entities can toggle the state of the scheduling term by maintaining a handle to it.

Example usage:

```
- type: nvidia::gxf::BooleanSchedulingTerm
parameters:
  enable_tick: true
```

## AsynchronousSchedulingTerm

`AsynchronousSchedulingTerm` is primarily associated with entities which are working with asynchronous events happening outside of their regular execution performed by the scheduler. Since these events are non-periodic in nature, `AsynchronousSchedulingTerm` prevents the scheduler from polling the entity for its status regularly and reduces CPU utilization. `AsynchronousSchedulingTerm` can either be in READY, WAIT, WAIT\_EVENT or NEVER states based on asynchronous event it's waiting on.

The state of an asynchronous event is described using `nvidia::gxf::AsynchronousEventState` and is updated using the `setEventState` API.

AsynchronousEventState	Description
READY	Init state, first tick is pending
WAIT	Request to an async service yet to be sent, nothing to do but wait
EVENT_WAITING	Request sent to an async service, pending event done notification
EVENT_DONE	Event done notification received, entity ready to be ticked
EVENT_NEVER	Entity does not want to be ticked again, end of execution

Entities associated with this scheduling term most likely have an asynchronous thread which can update the state of the scheduling term outside of its regular execution cycle performed by the gxf scheduler. When the scheduling term is in WAIT state, the scheduler regularly polls for the state of the entity. When the scheduling term is in EVENT\_WAITING state, schedulers will not check the status of the entity again until they receive an event notification which can be triggered using the `GxfEntityEventNotify` api. Setting the state of the scheduling term to EVENT\_DONE automatically sends this notification to the scheduler. Entities can use the EVENT\_NEVER state to indicate the end of its execution cycle.

Example usage:

```
- name: async_scheduling_term
type: nvidia::gxf::AsynchronousSchedulingTerm
```

### DownstreamReceptiveSchedulingTerm

This scheduling term specifies that an entity shall be executed if the receiver for a given transmitter can accept new messages.

Example usage:

```
- name: downstream_st
  type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
  parameters:
    transmitter: output
    min_size: 1
```

### TargetTimeSchedulingTerm

This scheduling term permits execution at a user-specified timestamp. The timestamp is specified on the clock provided.

Example usage:

```
- name: target_st
  type: nvidia::gxf::TargetTimeSchedulingTerm
  parameters:
    clock: clock/manual_clock
```

### ExpiringMessageAvailableSchedulingTerm

This scheduling waits for a specified number of messages in the receiver. The entity is executed when the first message received in the queue is expiring or when there are enough messages in the queue. The `receiver` parameter is used to set the receiver to watch on. The parameters `max_batch_size` and `max_delay_ns` dictate the maximum number of messages to be batched together and the maximum delay from first message to wait before executing the entity respectively.

In the example shown below, the associated entity will be executed when the number of messages in the queue is greater than `max_batch_size`, i.e 5, or when the delay from the first message to current time is greater than `max_delay_ns`, i.e 10000000.

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

## AND Combined

An entity can be associated with multiple scheduling terms which define its execution behavior. Scheduling terms are AND combined to describe the current state of an entity. For an entity to be executed by the scheduler, all the scheduling terms must be in READY state and conversely, the entity is unscheduled from execution whenever any one of the scheduling term reaches NEVER state. The priority of various states during AND combine follows the order NEVER, WAIT\_EVENT, WAIT, WAIT\_TIME, and READY.

Example usage:

```
components:
- name: integers
  type: nvidia::gxf::DoubleBufferTransmitter
- name: fibonacci
  type: nvidia::gxf::DoubleBufferTransmitter
- type: nvidia::gxf::CountSchedulingTerm
  parameters:
    count: 100
- type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
  parameters:
    transmitter: integers
    min_size: 1
```

## StandardExtension

Most commonly used interfaces and components in Gxf Core.

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

## Interfaces

### nvidia::gxf::Codelet

Interface for a component which can be executed to run custom code.

- Component ID: 5c6166fa-6eed-41e7-bbf0-bd48cd6e1014
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/codelet.hpp

### **nvidia::gxf::Clock**

Interface for clock components which provide time.

- Component ID: 779e61c2-ae70-441d-a26c-8ca64b39f8e7
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/clock.hpp

### **nvidia::gxf::System**

Component interface for systems which are run as part of the application run cycle.

- Component ID: d1febca1-80df-454e-a3f2-715f2b3c6e69
- Base Type: nvidia::gxf::Component

### **nvidia::gxf::Queue**

Interface for storing entities in a queue.

- Component ID: 792151bf-3138-4603-a912-5ca91828dea8
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/queue.hpp

### **nvidia::gxf::Router**

Interface for classes which are routing messages in and out of entities.

- Component ID: 8b317aad-f55c-4c07-8520-8f66db92a19e
- Defined in: gxf/std/router.hpp

### **nvidia::gxf::Transmitter**

Interface for publishing entities.

- Component ID: c30cc60f-0db2-409d-92b6-b2db92e02cce
- Base Type: nvidia::gxf::Queue
- Defined in: gxf/std/transmitter.hpp

### **nvidia::gxf::Receiver**

Interface for receiving entities.

- Component ID: a47d2f62-245f-40fc-90b7-5dc78ff2437e
- Base Type: nvidia::gxf::Queue
- Defined in: gxf/std/receiver.hpp

### **nvidia::gxf::Scheduler**

A simple poll-based single-threaded scheduler which executes codelets.

- Component ID: f0103b75-d2e1-4d70-9b13-3fe5b40209be
- Base Type: nvidia::gxf::System
- Defined in: nvidia/gxf/system.hpp

### **nvidia::gxf::SchedulingTerm**

Interface for terms used by a scheduler to determine if codelets in an entity are ready to step.

- Component ID: 184d8e4e-086c-475a-903a-69d723f95d19
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/scheduling\_term.hpp

### **nvidia::gxf::Allocator**

Provides allocation and deallocation of memory.

- Component ID: 3cdd82d0-2326-4867-8de2-d565dbe28e03
- Base Type: nvidia::gxf::Component
- Defined in: nvidia/gxf/allocator.hpp

### **nvidia::gxf::Monitor**

Monitors entities during execution.

- Component ID: 9ccf9421-b35b-8c79-e1f0-97dc23bd38ea
- Base Type: nvidia::gxf::Component
- Defined in: nvidia/gxf/monitor.hpp

## Components

### **nvidia::gxf::RealtimeClock**

A real-time clock which runs based off a system steady clock.

- Component ID: 7b170b7b-cf1a-4f3f-997c-bfea25342381
- Base Type: nvidia::gxf::Clock

## Parameters

### **initial\_time\_offset**

The initial time offset used until time scale is changed manually.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_FLOAT64

### **initial\_time\_scale**

The initial time scale used until time scale is changed manually.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_FLOAT64

### **use\_time\_since\_epoch**

If true, clock time is time since epoch + initial\_time\_offset at initialize(). Otherwise clock time is initial\_time\_offset at initialize().

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL

### **nvidia::gxf::ManualClock**

A manual clock which is instrumented manually.

- Component ID: 52fa1f97-eba8-472a-a8ca-4cff1a2c440f
- Base Type: nvidia::gxf::Clock

## Parameters

### **initial\_timestamp**

The initial timestamp on the clock (in nanoseconds).

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT64

### **nvidia::gxf::SystemGroup**

A group of systems.

- Component ID: 3d23d470-0aed-41c6-ac92-685c1b5469a0
- Base Type: nvidia::gxf::System

### **nvidia::gxf::MessageRouter**

A router which sends transmitted messages to receivers.

- Component ID: 84fd5d56-fda6-4937-0b3c-c283252553d8
- Base Type: nvidia::gxf::Router

### **nvidia::gxf::RouterGroup**

A group of routers.

- Component ID: ca64ee14-2280-4099-9f10-d4b501e09117
- Base Type: nvidia::gxf::Router

### **nvidia::gxf::DoubleBufferTransmitter**

A transmitter which uses a double-buffered queue where messages are pushed to a backstage after they are published.

- Component ID: 0c3c0ec7-77f1-4389-aef1-6bae85bddc13
- Base Type: nvidia::gxf::Transmitter

## Parameters

### **capacity**

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64
- Default: 1

### **policy**

0: pop, 1: reject, 2: fault.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64
- Default: 2

### **nvidia::gxf::DoubleBufferReceiver**

A receiver which uses a double-buffered queue where new messages are first pushed to a backstage.

- Component ID: ee45883d-bf84-4f99-8419-7c5e9deac6a5
- Base Type: nvidia::gxf::Receiver

### **Parameters**

#### **capacity**

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64
- Default: 1

### **policy**

0: pop, 1: reject, 2: fault

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64
- Default: 2

### **nvidia::gxf::Connection**

A component which establishes a connection between two other components.

- Component ID: cc71afae-5ede-47e9-b267-60a5c750a89a
- Base Type: nvidia::gxf::Component



## Parameters

### source

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Transmitter

### target

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Receiver

## nvidia::gxf::PeriodicSchedulingTerm

A component which specifies that an entity shall be executed periodically.

- Component ID: d392c98a-9b08-49b4-a422-d5fe6cd72e3e
- Base Type: nvidia::gxf::SchedulingTerm

## Parameters

### recess\_period

The recess period indicates the minimum amount of time which has to pass before the entity is permitted to execute again. The period is specified as a string containing of a number and an (optional) unit. If no unit is given the value is assumed to be in nanoseconds. Supported units are: Hz, s, ms. Example: 10ms, 10000000, 0.2s, 50Hz.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_STRING

## nvidia::gxf::CountSchedulingTerm

A component which specifies that an entity shall be executed exactly a given number of times.

- Component ID: f89da2e4-fddf-4aa2-9a80-1119ba3fde05
- Base Type: nvidia::gxf::SchedulingTerm

## Parameters

### count

The total number of time this term will permit execution.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT64

## nvidia::gxf::TargetTimeSchedulingTerm

A component where the next execution time of the entity needs to be specified after every tick.

- Component ID: e4aaf5c3-2b10-4c9a-c463-ebf6084149bf
- Base Type: nvidia::gxf::SchedulingTerm

## Parameters

### clock

The clock used to define target time.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Clock

## nvidia::gxf::DownstreamReceptiveSchedulingTerm

A component which specifies that an entity shall be executed if receivers for a certain transmitter can accept new messages.

- Component ID: 9de75119-8d0f-4819-9a71-2aeaefd23f71
- Base Type: nvidia::gxf::SchedulingTerm

## Parameters

### min\_size

The term permits execution if the receiver connected to the transmitter has at least the specified number of free slots in its back buffer.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64

### transmitter

The term permits execution if this transmitter can publish a message, i.e. if the receiver which is connected to this transmitter can receive messages.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gxf::Transmitter`

### **`nvidia::gxf::MessageAvailableSchedulingTerm`**

A scheduling term which specifies that an entity can be executed when the total number of messages over a set of input channels is at least a given number of messages.

- Component ID: `fe799e65-f78b-48eb-beb6-e73083a12d5b`
- Base Type: `nvidia::gxf::SchedulingTerm`

### **Parameters**

#### **`front_stage_max_size`**

If set the scheduling term will only allow execution if the number of messages in the front stage does not exceed this count. It can for example be used in combination with codelets which do not clear the front stage in every tick.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_UINT64

#### **`min_size`**

The scheduling term permits execution if the given receiver has at least the given number of messages available.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64

#### **`receiver`**

The scheduling term permits execution if this channel has at least a given number of messages available.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gxf::Receiver`

### **nvidia::gxf::MultiMessageAvailableSchedulingTerm**

A component which specifies that an entity shall be executed when a queue has at least a certain number of elements.

- Component ID: f15dbeaa-afd6-47a6-9ffc-7afd7e1b4c52
- Base Type: nvidia::gxf::SchedulingTerm

#### **Parameters**

##### **min\_size**

The scheduling term permits execution if all given receivers together have at least the given number of messages available.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64

##### **receivers**

The scheduling term permits execution if the given channels have at least a given number of messages available.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Receiver

### **nvidia::gxf::ExpiringMessageAvailableSchedulingTerm**

A component which tries to wait for specified number of messages in queue for at most specified time.

- Component ID: eb22280c-76ff-11eb-b341-cf6b417c95c9
- Base Type: nvidia::gxf::SchedulingTerm

#### **Parameters**

##### **clock**

Clock to get time from.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Clock

**max\_batch\_size**

The maximum number of messages to be batched together.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT64

**max\_delay\_ns**

The maximum delay from first message to wait before submitting workload anyway.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT64

**receiver**

Receiver to watch on.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Receiver

**nvidia::gxf::BooleanSchedulingTerm**

A component which acts as a boolean AND term that can be used to control the execution of the entity.

- Component ID: e07a0dc4-3908-4df8-8134-7ce38e60fbef
- Base Type: nvidia::gxf::SchedulingTerm

**nvidia::gxf::AsynchronousSchedulingTerm**

A component which is used to inform of that an entity is dependent upon an async event for its execution.

- Component ID: 56be1662-ff63-4179-9200-3fcd8dc38673
- Base Type: nvidia::gxf::SchedulingTerm

## **nvidia::gxf::GreedyScheduler**

A simple poll-based single-threaded scheduler which executes codelets.

- Component ID: 869d30ca-a443-4619-b988-7a52e657f39b
- Base Type: nvidia::gxf::Scheduler

### **Parameters**

#### **clock**

The clock used by the scheduler to define flow of time. Typical choices are a `RealtimeClock` or a `ManualClock`.

- Flags: `GXF_PARAMETER_FLAGS_OPTIONAL`
- Type: `GXF_PARAMETER_TYPE_HANDLE`
- Handle Type: `nvidia::gxf::Clock`

#### **max\_duration\_ms**

The maximum duration for which the scheduler will execute (in ms). If not specified the scheduler will run until all work is done. If periodic terms are present this means the application will run indefinitely.

- Flags: `GXF_PARAMETER_FLAGS_OPTIONAL`
- Type: `GXF_PARAMETER_TYPE_INT64`

#### **realtime**

This parameter is deprecated. Assign a clock directly.

- Flags: `GXF_PARAMETER_FLAGS_OPTIONAL`
- Type: `GXF_PARAMETER_TYPE_BOOL`

#### **stop\_on\_deadlock**

If enabled the scheduler will stop when all entities are in a waiting state, but no periodic entity exists to break the dead end. Should be disabled when scheduling conditions can be changed by external actors, for example by clearing queues manually.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_BOOL`

## **nvidia::gxf::MultiThreadScheduler**

A multi thread scheduler that executes codelets for maximum throughput.

- Component ID: de5e0646-7fa5-11eb-a5c4-330ebfa81bbf
- Base Type: nvidia::gxf::Scheduler

### **Parameters**

#### **check\_recession\_perios\_ms**

The maximum duration for which the scheduler would wait (in ms) when an entity is not ready to run yet.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT64

#### **clock**

The clock used by the scheduler to define flow of time. Typical choices are a `RealtimeClock` or a `ManualClock`.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Clock

#### **max\_duration\_ms**

The maximum duration for which the scheduler will execute (in ms). If not specified the scheduler will run until all work is done. If periodic terms are present this means the application will run indefinitely.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_INT64

#### **stop\_on\_deadlock**

If enabled the scheduler will stop when all entities are in a waiting state, but no periodic entity exists to break the dead end. Should be disabled when scheduling conditions can be changed by external actors, for example by clearing queues manually.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL

### **worker\_thread\_number**

Number of threads.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT64
- Default: 1

### **nvidia::gxf::BlockMemoryPool**

A memory pools which provides a maximum number of equally sized blocks of memory.

- Component ID: 92b627a3-5dd3-4c3c-976c-4700e8a3b96a
- Base Type: nvidia::gxf::Allocator

### **Parameters**

#### **block\_size**

The size of one block of memory in byte. Allocation requests can only be fulfilled if they fit into one block. If less memory is requested still a full block is issued.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64

#### **do\_not\_use\_cuda\_malloc\_host**

If enabled operator new will be used to allocate host memory instead of cudaMallocHost.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL
- Default: True

#### **num\_blocks**

The total number of blocks which are allocated by the pool. If more blocks are requested allocation requests will fail.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64



**storage\_type**

The memory storage type used by this allocator. Can be kHost (0) or kDevice (1) or kSystem (2).

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT32
- Default: 0

**nvidia::gxf::UnboundedAllocator**

Allocator that uses dynamic memory allocation without an upper bound.

- Component ID: c3951b16-a01c-539f-d87e-1dc18d911ea0
- Base Type: nvidia::gxf::Allocator

**Parameters****do\_not\_use\_cuda\_malloc\_host**

If enabled operator new will be used to allocate host memory instead of cudaMallocHost.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL
- Default: True

**nvidia::gxf::Tensor**

A component which holds a single tensor.

- Component ID: 377501d6-9abf-447c-a617-0114d4f33ab8
- Defined in: gxf/std/tensor.hpp

**nvidia::gxf::Timestamp**

Holds message publishing and acquisition related timing information.

- Component ID: d1095b10-5c90-4bbc-bc89-601134cb4e03
- Defined in: gxf/std/timestamp.hpp

**nvidia::gxf::Metric**

Collects, aggregates, and evaluates metric data.

- Component ID: f7cef803-5beb-46f1-186a-05d3919842ac
- Base Type: nvidia::gxf::Component

## Parameters

### **aggregation\_policy**

Aggregation policy used to aggregate individual metric samples. Choices: {mean, min, max}.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_STRING

### **lower\_threshold**

Lower threshold of the metric's expected range.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_FLOAT64

### **upper\_threshold**

Upper threshold of the metric's expected range.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_FLOAT64

## **nvidia::gxf::JobStatistics**

Collects runtime statistics.

- Component ID: 2093b91a-7c82-11eb-a92b-3f1304ecc959
- Base Type: nvidia::gxf::Component

## Parameters

### **clock**

The clock component instance to retrieve time from.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Clock

**codelet\_statistics**

If set to true, JobStatistics component will collect performance statistics related to codelets.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_BOOL

**json\_file\_path**

If provided, all the collected performance statistics data will be dumped into a json file.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_STRING

**nvidia::gxf::Broadcast**

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

- Component ID: 3daadb31-0bca-47e5-9924-342b9984a014
- Base Type: nvidia::gxf::Codelet

**Parameters****mode**

The broadcast mode. Can be Broadcast or RoundRobin.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_CUSTOM

**source**

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Receiver

## **nvidia::gxf::Gather**

All messages arriving on any input channel are published on the single output channel.

- Component ID: 85f64c84-8236-4035-9b9a-3843a6a2026f
- Base Type: nvidia::gxf::Codelet

### **Parameters**

#### **sink**

The output channel for gathered messages.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Transmitter

#### **tick\_source\_limit**

Maximum number of messages to take from each source in one tick. 0 means no limit.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT64

## **nvidia::gxf::TensorCopier**

Copies tensor either from host to device or from device to host.

- Component ID: c07680f4-75b3-189b-8886-4b5e448e7bb6
- Base Type: nvidia::gxf::Codelet

### **Parameters**

#### **allocator**

Memory allocator for tensor data

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Allocator

#### **mode**

Configuration to select what tensors to copy:

1. kCopyToDevice (0) - copies to device memory, ignores device allocation
2. kCopyToHost (1) - copies to pinned host memory, ignores host allocation
3. kCopyToSystem (2) - copies to system memory, ignores system allocation.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT32

**receiver**

Receiver for incoming entities.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Receiver

**transmitter**

Transmitter for outgoing entities.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Transmitter

**nvidia::gxf::TimedThrottler**

Publishes the received entity respecting the timestamp within the entity.

- Component ID: ccf7729c-f62c-4250-5cf7-f4f3ec80454b
- Base Type: nvidia::gxf::Codelet

**Parameters****execution\_clock**

Clock on which the codelet is executed by the scheduler.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Clock

**receiver**

Channel to receive messages that need to be synchronized.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gxf::Receiver`

### **scheduling\_term**

Scheduling term for executing the codelet.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gxf::TargetTimeSchedulingTerm`

### **throttling\_clock**

Clock which the received entity timestamps are based on.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gxf::Clock`

### **transmitter**

Transmitter channel publishing messages at appropriate timesteps.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gxf::Transmitter`

### **`nvidia::gxf::Vault`**

Safely stores received entities for further processing.

- Component ID: `1108cb8d-85e4-4303-ba02-d27406ee9e65`
- Base Type: `nvidia::gxf::Codelet`

## Parameters

### drop\_waiting

If too many messages are waiting the oldest ones are dropped.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL

### max\_waiting\_count

The maximum number of waiting messages. If exceeded the codelet will stop pulling messages out of the input queue.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64

### source

Receiver from which messages are taken and transferred to the vault.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Receiver

## nvidia::gxf::Subgraph

Helper component to import a subgraph.

- Component ID: 576eedd7-7c3f-4d2f-8c38-8baa79a3d231
- Base Type: nvidia::gxf::Component

## Parameters

### location

Yaml source of the subgraph.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_STRING

### **nvidia::gxf::EndOfStream**

A component which represents end-of-stream notification.

- Component ID: 8c42f7bf-7041-4626-9792-9eb20ce33cce
- Defined in: gxf/std/eos.hpp

### **nvidia::gxf::Synchronization**

Component to synchronize messages from multiple receivers based on the `acq_time`.

- Component ID: f1cb80d6-e5ec-4dba-9f9e-b06b0def4443
- Base Type: nvidia::gxf::Codelet

### **Parameters**

#### **inputs**

All the inputs for synchronization. Number of inputs must match that of the outputs.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Receiver

#### **outputs**

All the outputs for synchronization. Number of outputs must match that of the inputs.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Transmitter

### **signed char**

- Component ID: 83905c6a-ca34-4f40-b474-cf2cde8274de

### **unsigned char**

- Component ID: d4299e15-0006-d0bf-8cbd-9b743575e155



**short int**

- Component ID: 9e1dde79-3550-307d-e81a-b864890b3685

**short unsigned int**

- Component ID: 958cbdef-b505-bcc7-8a43-dc4b23f8cead

**int**

- Component ID: b557ec7f-49a5-08f7-a35e-086e9d1ea767

**unsigned int**

- Component ID: d5506b68-5c86-fedb-a2a2-a7bae38ff3ef

**long int**

- Component ID: c611627b-6393-365f-d234-1f26bfa8d28f

**long unsigned int**

- Component ID: c4385f5b-6e25-01d9-d7b5-6e7cad704e8

**float**

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

**double**

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

**bool**

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

## CudaExtension

Extension for CUDA operations.

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

## Components

### **nvidia::gxf::CudaStream**

Holds and provides access to native `cudaStream_t`.

`nvidia::gxf::CudaStream` handle must be allocated by `nvidia::gxf::CudaStreamPool`. Its lifecycle is valid until explicitly recycled through `nvidia::gxf::CudaStreamPool.releaseStream()` or implicitly until `nvidia::gxf::CudaStreamPool` is deactivated.

You may call `stream()` to get the native `cudaStream_t` handle, and to submit GPU operations. After the submission, GPU takes over the input tensors/buffers and keeps them in use. To prevent host carelessly releasing these in-use buffers, CUDA Codelet needs to call `record(event, input_entity, sync_cb)` to extend `input_entity`'s lifecycle until GPU completely consumes it. Alternatively, you may call `record(event, event_destroy_cb)` for native `cudaEvent_t` operations and free in-use resource via `event_destroy_cb`.

It is required to have a `nvidia::gxf::CudaStreamSync` in the graph pipeline after all the CUDA operations. See more details in `nvidia::gxf::CudaStreamSync`

- Component ID: 5683d692-7884-11eb-9338-c3be62d576be
- Defined in: `gxf/cuda/cuda_stream.hpp`

### **nvidia::gxf::CudaStreamId**

Holds CUDA stream Id to deduce `nvidia::gxf::CudaStream` handle.

`stream_cid` should be `nvidia::gxf::CudaStream` component id.

- Component ID: 7982aeac-37f1-41be-ade8-6f00b4b5d47c
- Defined in: `gxf/cuda/cuda_stream_id.hpp`

### **nvidia::gxf::CudaEvent**

Holds and provides access to native `cudaEvent_t` handle.

When a `nvidia::gxf::CudaEvent` is created, you'll need to initialize a native `cudaEvent_t` through `init(flags, dev_id)`, or set third party event through `initWithEvent(event, dev_id, free_fnc)`. The event keeps valid until `deinit` is called explicitly otherwise gets recycled in destructor.

- Component ID: f5388d5c-a709-47e7-86c4-171779bc64f3
- Defined in: `gxf/cuda/cuda_event.hpp`

## **nvidia::gxf::CudaStreamPool**

CudaStream allocation.

You must explicitly call `allocateStream()` to get a valid `nvidia::gxf::CudaStream` handle. This component would hold all the its allocated `nvidia::gxf::CudaStream` entities until `releaseStream(stream)` is called explicitly or the `CudaStreamPool` component is deactivated.

- Component ID: 6733bf8b-ba5e-4fae-b596-af2d1269d0e7
- Base Type: `nvidia::gxf::Allocator`

### **Parameters**

#### **dev\_id**

GPU device id.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_INT32`
- Default Value: 0

#### **stream\_flags**

Flag values to create CUDA streams.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_INT32`
- Default Value: 0

#### **stream\_priority**

Priority values to create CUDA streams.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_INT32`
- Default Value: 0

#### **reserved\_size**

User-specified file name without extension.

- Flags: `GXF_PARAMETER_FLAGS_NONE`

- Type: GXF\_PARAMETER\_TYPE\_INT32
- Default Value: 1

#### **max\_size**

Maximum Stream Size.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_INT32
- Default Value: 0, no limitation.

### **nvidia::gxf::CudaStreamSync**

Synchronize all CUDA streams which are carried by message entities.

This codelet is required to get connected in the graph pipeline after all CUDA ops codelets. When a message entity is received, it would find all of the `nvidia::gxf::CudaStreamId` in that message, and extract out each `nvidia::gxf::CudaStream`. With each `CudaStream` handle, it synchronizes all previous `nvidia::gxf::CudaStream.record()` events, along with all submitted GPU operations before this point.

---

**Note:** `CudaStreamSync` must be set in the graph when `nvidia::gxf::CudaStream.record()` is used, otherwise it may cause memory leak.

---

- Component ID: 0d1d8142-6648-485d-97d5-277eed00129c
- Base Type: `nvidia::gxf::Codelet`

### **Parameters**

#### **rx**

Receiver to receive all messages carrying `nvidia::gxf::CudaStreamId`.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gxf::Receiver`

#### **tx**

Transmitter to send messages to downstream.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gxf::Transmitter`

## MultimediaExtension

Extension for multimedia related data types, interfaces and components in GXF Core.

- UUID: 6f2d1afc-1057-481a-9da6-a5f61fed178e
- Version: 2.0.0
- Author: NVIDIA

## Components

### nvidia::gxf::AudioBuffer

AudioBuffer is similar to Tensor component in the standard extension and holds memory and metadata corresponding to an audio buffer.

- Component ID: a914cac6-5f19-449d-9ade-8c5cdcebe7c3

AudioBufferInfo structure captures the following metadata:

Field	Description
channels	Number of channels in an audio frame
samples	Number of samples in an audio frame
sampling_rate	sampling rate in Hz
bytes_per_sample	Number of bytes required per sample
audio_format	AudioFormat of an audio frame
audio_layout	AudioLayout of an audio frame

Supported AudioFormat types:

AudioFormat	Description
GXF_AUDIO_FORMAT_S16LE	16-bit signed PCM audio
GXF_AUDIO_FORMAT_F32LE	32-bit floating-point audio

Supported AudioLayout types:

AudioLayout	Description
GXF_AUDIO_LAYOUT_INTERLEAVED	Data from all the channels to be interleaved - LRLRLR
GXF_AUDIO_LAYOUT_NON_INTERLEAVED	Data from all the channels not to be interleaved - LLLRRR

### nvidia::gxf::VideoBuffer

VideoBuffer is similar to Tensor component in the standard extension and holds memory and metadata corresponding to a video buffer.

- Component ID: 16ad58c8-b463-422c-b097-61a9acc5050e

VideoBufferInfo structure captures the following metadata:

Field	Description
width	width of a video frame
height	height of a video frame
color_format	VideoFormat of a video frame
color_planes	ColorPlane(s) associated with the VideoFormat
surface_layout	SurfaceLayout of the video frame

Supported VideoFormat types:

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

Supported SurfaceLayout types:

SurfaceLayout	Description
GXF_SURFACE_LAYOUT_PITCH_LINEAR	pitch linear surface memory
GXF_SURFACE_LAYOUT_BLOCK_LINEAR	block linear surface memory

## SerializationExtension

Extension for serializing messages.

- UUID: bc573c2f-89b3-d4b0-8061-2da8b11fe79a
- Version: 2.0.0
- Author: NVIDIA

## Interfaces

### **nvidia::gxf::ComponentSerializer**

Interface for serializing components.

- Component ID: 8c76a828-2177-1484-f841-d39c3fa47613
- Base Type: `nvidia::gxf::Component`
- Defined in: `gxf/serialization/component_serializer.hpp`

## Components

### **nvidia::gxf::EntityRecorder**

Serializes incoming messages and writes them to a file.

- Component ID: 9d5955c7-8fda-22c7-f18f-ea5e2d195be9
- Base Type: `nvidia::gxf::Codelet`

## Parameters

### **receiver**

Receiver channel to log.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_HANDLE`
- Handle Type: `nvidia::gxf::Receiver`

### **serializers**

List of component serializers to serialize entities.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_CUSTOM`
- Custom Type: `std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>`

### **directory**

Directory path for storing files.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_STRING`

### **basename**

User specified file name without extension.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_STRING

### **flush\_on\_tick**

Flushes output buffer on every tick when true.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL

### **nvidia::gxf::EntityReplayer**

De-serializes and publishes messages from a file.

- Component ID: fe827c12-d360-c63c-8094-32b9244d83b6
- Base Type: nvidia::gxf::Codelet

## **Parameters**

### **transmitter**

Transmitter channel for replaying entities.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Transmitter

### **serializers**

List of component serializers to serialize entities.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>



**directory**

Directory path for storing files.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_STRING

**batch\_size**

Number of entities to read and publish for one tick.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_UINT64

**ignore\_corrupted\_entities**

If an entity could not be de-serialized, it is ignored by default; otherwise a failure is generated.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL

**nvidia::gxf::StdComponentSerializer**

Serializer for Timestamp and Tensor components.

- Component ID: c0e6b36c-39ac-50ac-ce8d-702e18d8bfff7
- Base Type: nvidia::gxf::ComponentSerializer

**Parameters****allocator**

Memory allocator for tensor components.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Allocator

## TensorRTEExtension

Components with TensorRT inference capability.

- UUID: d43f23e4-b9bf-11eb-9d18-2b7be630552b
- Version: 2.0.0
- Author: NVIDIA

## Components

### nvidia::gxf::TensorRtInference

Codelet taking input tensors and feed them into TensorRT for inference.

- Component ID: 06a7f0e0-b9c0-11eb-8cd6-23c9c2070107
- Base Type: nvidia::gxf::Codelet

## Parameters

### model\_file\_path

Path to ONNX model to be loaded.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_STRING

### engine\_file\_path

Path to the generated engine to be serialized and loaded from.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_STRING

### force\_engine\_update

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

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL
- Default: False

**input\_tensor\_names**

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

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_STRING

**input\_binding\_names**

Names of input bindings as in the model in the same order of what is provided in input\_tensor\_names.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_STRING

**output\_tensor\_names**

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

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_STRING

**output\_binding\_names**

Names of output bindings in the model in the same order of of what is provided in output\_tensor\_names.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_STRING

**pool**

Allocator instance for output tensors.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gfx::Allocator

### **cuda\_stream\_pool**

Instance of `gxf::CudaStreamPool` to allocate CUDA stream.

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

### **max\_workspace\_size**

Size of working space in bytes. Default to 64MB

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

### **dla\_core**

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

- Flags: `GXF_PARAMETER_FLAGS_OPTIONAL`
- Type: `GXF_PARAMETER_TYPE_INT64`

### **max\_batch\_size**

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

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_INT32`
- Default: 1

### **enable\_fp16\_**

Enable inference with FP16 and FP32 fallback.

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

**verbose**

Enable verbose logging on console. Default to false.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL
- Default: False

**relaxed\_dimension\_check**

Ignore dimensions of 1 for input tensor dimension check.

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_BOOL
- Default: True

**clock**

Instance of clock for publish time.

- Flags: GXF\_PARAMETER\_FLAGS\_OPTIONAL
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gfx::Clock`

**rx**

List of receivers to take input tensors

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: `nvidia::gfx::Receiver`

**tx**

Transmitter to publish output tensors

- Flags: GXF\_PARAMETER\_FLAGS\_NONE
- Type: GXF\_PARAMETER\_TYPE\_HANDLE
- Handle Type: nvidia::gxf::Transmitter

## 11.2 Rivermax SDK

Clara Developer Kits can be used along with the [NVIDIA Rivermax SDK](#) to provide an extremely efficient network connection using the onboard ConnectX 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 discrete GPU.

The instructions below describe the steps to test the Rivermax SDK with the Developer Kits. 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 can be installed onto the Developer Kit via SDK Manager by selecting it as an additional SDK during the HoloPack installation.

---

---

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

---

### 11.2.1 Testing Rivermax and GPUDirect

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

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

```
$ sudo lshw -class network
*-network:0
    description: Ethernet interface
    product: MT28908 Family [ConnectX-6]
    vendor: Mellanox Technologies
    physical id: 0
    bus info: pci@0000:09:00.0
    <b>logical name: enp9s0f0</b>
    version: 00
    serial: 48:b0:2d:13:9b:6b
    capacity: 10Gbit/s
    width: 64 bits
```

(continues on next page)

(continued from previous page)

```

    clock: 33MHz
    capabilities: pciexpress vpd msix pm bus_master cap_list ethernet physical_
→10000bt-fd 100000bt-fd autonegotiation
    configuration: autonegotiation=on broadcast=yes driver=mlx5_core_
→driverversion=5.4-1.0.3 duplex=full firmware=20.27.4006 (NVD0000000001) ip=10.0.0.
→2 latency=0 link=yes multicast=yes
    resources: iomemory:180-17f irq:33 memory:1818000000-1819ffffff

```

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

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

a. Bring up the network:

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

b. Build the sample apps:

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

**Note:** The 1.8.21 path above corresponds to the path where the Rivermax SDK package was installed. 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
| Source IP: 10.0.0.1
| Destination IP: 10.0.0.2
| Destination port: 5001
| Number of flows: 1
| Rate limit bps: 0
| Rate limit max burst in packets: 0
| Memory address: 0x7f986e3010
| Memory length: 59883520[B]
| Memory key: 40308
| Number of user requested chunks: 1
| Number of application chunks: 5
| Number of packets in chunk: 8192

```

(continues on next page)

(continued from previous page)

```
| Packet's payload size: 1462
+*****
```

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

- a. Bring up the network:

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

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

```
$ 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 installed. 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:

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

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

## 11.3 GPUDirect RDMA

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

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

NVIDIA takes advantage of RDMA in many of its SDKs, including [Rivermax](#) for GPUDirect support with [ConnectX](#) network adapters, and [GPUDirect Storage](#) for transfers between a GPU and storage device. NVIDIA is also committed to supporting hardware vendors enable RDMA within their own drivers, an example of which is provided by [AJA Video Systems](#) as part of a partnership with NVIDIA for the Clara Holoscan SDK. The [AJASource extension](#) is an example of how the SDK can leverage RDMA.

For more information about GPUDirect RDMA, see the following:

- [GPUDirect RDMA Documentation](#)



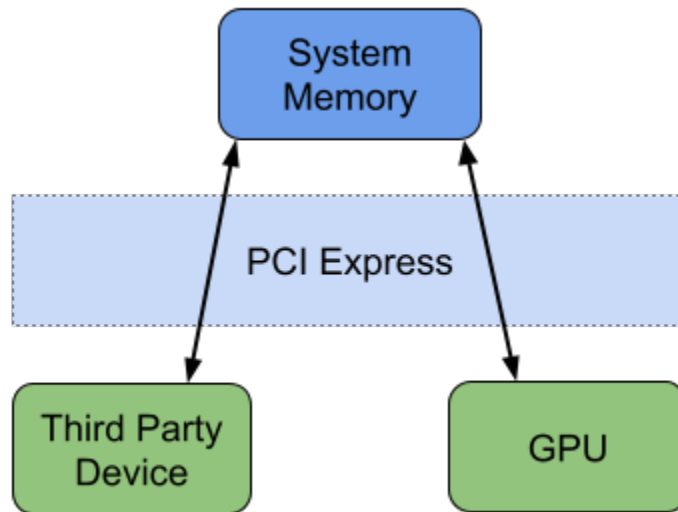


Fig. 11.6: Data Transfer Between PCIe Device and GPU Without GPUDirect RDMA

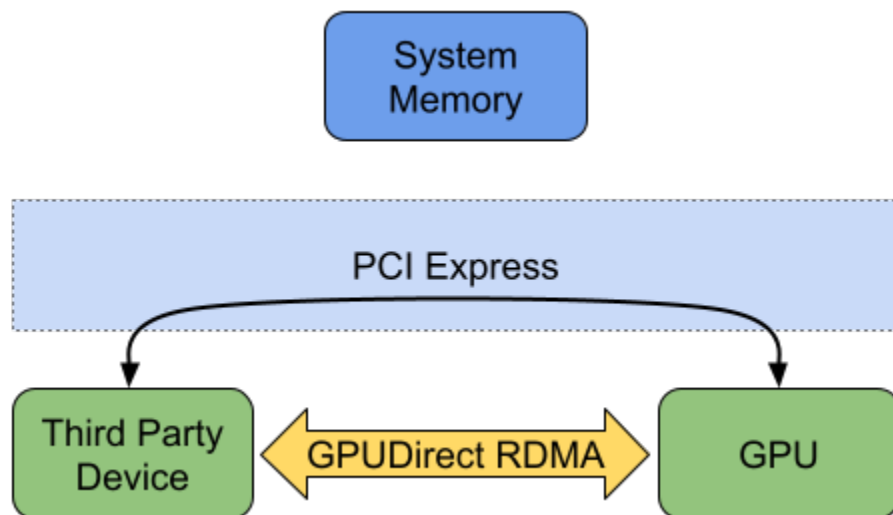


Fig. 11.7: Data Transfer Between PCIe Device and GPU With GPUDirect RDMA

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

## 11.4 TensorRT Optimized Inference

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

GXF comes with a TensorRT base extension which is extended in the Holoscan SDK: the updated [TensorRT extension](#) is able to selectively load a cached TensorRT model based on the system GPU specifications, making it ideal to interface with the Clara Developer Kits.

## 11.5 CUDA and OpenGL Interoperability

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

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

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

## 11.6 Accelerated Image Transformations

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