

nVIDIA®

Holoscan SDK User Guide

Release 0.5.1

NVIDIA Corporation

Jun 08, 2023

INTRODUCTION

1 Overview	1
2 Relevant Technologies	3
3 SDK Installation	5
4 Additional Setup	7
5 Third Party Hardware Setup	15
6 Holoscan Core Concepts	23
7 Holoscan by Example	25
8 Creating an Application	61
9 Creating Operators	77
10 Built-in Operators and Extensions	109
11 Logging	113
12 Visualization Module	115
13 Inference Module	123
14 GXF Core concepts	129
15 Holoscan and GXF	131
16 GXF by Example	133
17 Using Holoscan Operators in GXF Applications	149
18 GXF User Guide	151
19 Video Pipeline Latency Tool	225

CHAPTER ONE

OVERVIEW

NVIDIA Holoscan is the AI sensor processing platform that combines hardware systems for low-latency sensor and network connectivity, optimized libraries for data processing and AI, and core microservices to run streaming, imaging, and other applications, from embedded to edge to cloud. It can be used to build streaming AI pipelines for a variety of domains, including Medical Devices, High Performance Computing at the Edge, Industrial Inspection and more.

Note: In previous releases, the prefix [Clara](#) was used to define Holoscan as a platform designed initially for [medical devices](#). As Holoscan has grown, its potential to serve other areas has become apparent. With version 0.4.0, we're proud to announce that the Holoscan SDK is now officially built to be domain-agnostic and can be used to build sensor AI applications in multiple domains. Note that some of the content of the SDK (sample applications) or the documentation might still appear to be healthcare-specific pending additional updates. Going forward, domain specific content will be hosted on the [HoloHub](#) repository.

The Holoscan SDK assists developers by providing:

1. Various installation strategies

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

2. C++ and Python APIs

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

3. Built-in Operators

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

4. Minimal Examples

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

5. 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 [ES] 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 the Holoscan Developer Kits. The measurements taken by this tool can then be displayed with a comprehensive and easy-to-read visualization of the data.

6. Documentation

The Holoscan SDK documentation is composed of:

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

RELEVANT TECHNOLOGIES

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

- *Rivermax and GPUDirect RDMA*
- *Graph Execution Framework (GXF)*
- *TensorRT TensorRT Optimized Inference*
- *Interoperability between CUDA and rendering frameworks*
- *Accelerated Image Transformations*

2.1 Rivermax and GPUDirect RDMA

The Holoscan 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](#) for RDMA. This technology avoids unnecessary memory copies and CPU overhead by copying data directly to or from pinned GPU memory, and supports both the integrated GPU or the discrete GPU.

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

For more information about GPUDirect RDMA, see the following:

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

2.2 Graph Execution Framework

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

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

2.3 TensorRT Optimized Inference

NVIDIA [TensorRT](#) is a deep learning inference framework based on CUDA that provided the highest optimizations to run on NVIDIA GPUs, including the Holoscan 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.

Similarly, the new *inference module* leverages TensorRT and provides the ability to execute multiple inferences in parallel.

Warning: The *TensorRT extension* will be deprecated in favor of operators leveraging the new *inference module* in a future release.

2.4 Interoperability between CUDA and rendering frameworks

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

- The [OpenGL](#) and Segmentation Visualizer extensions use the [OpenGL interoperability](#) functions provided by the CUDA runtime API.
- The [Holoviz](#) module uses the [external resource interoperability](#) functions of the low-level CUDA driver application programming interface, the Vulkan external memory and [external semaphore](#) extensions.

Warning: The *OpenGL extension* will be deprecated in favor of *Vulkan/Holoviz* in a future release.

2.5 Accelerated Image Transformations

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

SDK INSTALLATION

The Holoscan SDK requires a specific software stack to build and run. We provide two:

- The *Development Stack* for Holoscan Developer Kits based on Holopack (Jetson L4T based), and for x86_64 Linux compute platforms, ideal for development and testing of the SDK.
- The *Deployment Stack* for Holoscan Developer Kits based on OpenEmbedded (Yocto build system), recommended to limit your stack to only the software components required to run your Holoscan application. The runtime Board Support Package (BSP) can be optimized with respect to memory usage, speed, security and power requirements.

3.1 Development Software Stack

3.1.1 Prerequisites

Holoscan Developer Kits (aarch64)

Set up your developer kit:

Developer Kit	User Guide	HoloPack	GPU
NVIDIA IGX Orin	Guide	2.0	dGPU or iGPU
NVIDIA IGX Orin [ES]	Guide	1.2	dGPU only
NVIDIA Clara AGX	Guide	1.2	dGPU only

x86_64

You'll need the following to use the Holoscan SDK on x86_64:

- OS: Ubuntu 20.04
- NVIDIA discrete GPU (dGPU)
 - Ampere or above recommended for best performance
 - Quadro/NVIDIA RTX necessary for [GPUDirect RDMA](#) support
 - Tested with [NVIDIA RTX 6000](#) and [NVIDIA RTX A6000](#)
- NVIDIA dGPU drivers: 510.73.08 or above
- For Rivermax support (optional):
 - [NVIDIA ConnectX SmartNIC](#)

- OFED Network Drivers: 5.8
- GPUDirect Drivers: 1.1
- Rivermax SDK: 1.20 (for local development only)

Additional software dependencies are needed, which vary based on how you choose to install the SDK (see section below).

Refer to the following sections in this user guide for *additional setup* or to support *third-party technologies*, such as AJA cards or Emergent cameras.

3.1.2 Install the SDK

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

- The **Holoscan container image on NGC** includes the Holoscan libraries, GXF extensions, headers, example source code, and sample datasets, as well as all the dependencies that were tested with Holoscan. It is the recommended way to run the Holoscan examples, while still allowing you to create your own C++ and Python Holoscan application.
- The **Holoscan python wheels on PyPI** are the simplest way for Python developers to get started with the SDK using `pip install holoscan`. The wheels include the SDK libraries, not the example applications or the sample datasets.
- The **Holoscan Debian package on NGC** includes the libraries, headers, example applications and CMake configurations needed for both C++ and Python developers. It does not include sample datasets.
- The **Holoscan SDK source repository** provides reference implementations as well as infrastructure for building the libraries and example applications yourself.

Refer to the documentation in each of those for specific install and run instructions.

3.2 Deployment Software Stack

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

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

ADDITIONAL SETUP

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

4.1 Setting-up GPUDirect RDMA

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

Note: This is not required for *AJA cards* support as they use their own driver (NTV2) which implements GPUDirect RDMA. However, this is required for *Emergent cameras* support, as their SDK (eSDK) uses the NVIDIA GPUDirect drivers.

HoloPack 2.0

The GPUDirect drivers (nvidia peermem) are installed with HoloPack 2.0 in dGPU mode. However - at this time - they must be reconfigured after installing MOFED drivers (either as part of the Rivermax SDK option in SDK Manager, or manually), then loaded manually to enable the use of GPUDirect RDMA with NVIDIA's Quadro/workstation GPUs.

```
# Ensure you've installed MOFED drivers first through SDKM (Rivermax SDK) or manually
sudo dpkg-reconfigure nvidia-dkms-520 && \
insmod /var/lib/dkms/nvidia/520.61.05/5.10.104-tegra/arm64/module/nvidia-peermem.ko
```

HoloPack 1.2

The GPUDirect drivers (nvidia peermem) are installed after switching to dGPU mode on HoloPack 1.2. However - at this time - they must be reconfigured after installing MOFED drivers (either as part of the Rivermax SDK option in SDK Manager, or manually), then loaded manually to enable the use of GPUDirect RDMA with NVIDIA's Quadro/workstation GPUs.

```
# Ensure you've installed MOFED drivers first through SDKM (Rivermax SDK) or manually
sudo dpkg-reconfigure nvidia-dkms-510 && \
insmod /var/lib/dkms/nvidia/510.73.08/5.10.65-tegra/aarch64/module/nvidia-peermem.ko
```

HoloPack 1.1

The GPUDirect drivers (nvidia peermem) must be manually installed to enable the use of GPUDirect RDMA with NVIDIA's Quadro/workstation GPUs. They are not installed as part of Holopack 1.1 when selecting Rivermax SDK in the SDK Manager at this time.

1. Download the [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
```

3. Verify the `nv_peer_mem` service is running:

```
sudo service nv_peer_mem status
```

4. 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
```

Warning: There is a known issue that prevents GPU RDMA from being enabled on the NVIDIA IGX Orin [ES] Developer Kit without a firmware update or running a manual command. Refer to the instructions in the [NVIDIA IGX Orin \[ES\] Developer Kit User Guide](#) for instructions.

4.1.1 Testing with Rivermax

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

Note: The Rivermax SDK can be installed onto the Developer Kit via SDK Manager by selecting it as an additional SDK during the HoloPack installation. Access to the Rivermax SDK Developer Program as well as a valid Rivermax software license is required to use the Rivermax SDK.

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.

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

```
rivermax_sdk=$HOME/Documents/Rivermax/1.8.21
```

Install path might differ in future versions of Rivermax.

1. Determine the logical name for the ConnectX devices that are used by each system. This can be done by using the `lshw -class network` command, finding the `product:` entry for the ConnectX device, and making note of the `logical name:` that corresponds to that device. For example, this output on a Developer Kit shows the onboard ConnectX device using the `enp9s0f0` 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:</b> enp9s0f0</b>
    version: 00
    serial: 48:b0:2d:13:9b:6b
    capacity: 10Gbit/s
    width: 64 bits
    clock: 33MHz
    capabilities: pciexpress vpd msix pm bus_master cap_list ethernet physical
    ↳ 1000bt-fd 10000bt-fd autonegotiation
    configuration: autonegotiation=on broadcast=yes driver=mlx5_core
    ↳ driverversion=5.4-1.0.3 duplex=full firmware=20.27.4006 (NVD0000000001) ip=10.0.0.
    ↳ 2 latency=0 link=yes multicast=yes
    resources: iomemory:180-17f irq:33 memory:1818000000-1819fffff
```

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.

- Bring up the network:

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

- Build the sample apps:

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

- 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
```

(continues on next page)

(continued from previous page)

```

| 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
| 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 ${rivermax_sdk}/apps
$ make CUDA=y
```

- 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>`.

4.2 Enabling G-SYNC

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

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

- Asus ROG Swift PG279QM
 - Asus ROG Swift 360 Hz PG259QNR
-

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

1. Open the “NVIDIA Settings” Graphical application (`nvidia-settings` in Terminal).
2. Click on `X Server Display Configuration` then the `Advanced` button. This will show the `Allow G-SYNC on monitor not validated as G-SYNC compatible` option. Enable the option and click `Apply`:
3. To show the refresh rate and G-SYNC label on the display window, click on `OpenGL Settings` for the selected display. Now click `Allow G-SYNC/G-SYNC Compatible` and `Enable G-SYNC/G-SYNC Compatible Visual Indicator` options 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.

4.3 Disabling Variable Backlight

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

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

4.4 Enabling Exclusive Display Mode

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

4.5 Use both Integrated and Discrete GPUs on Holoscan developer kits

Due to symbols redundancy between the nvgpu drivers for integrated GPU (iGPU), and the openrm drivers for discrete GPU (dGPU), additional steps are required to be able to leverage them both in parallel on the Holoscan developer kits at this time.

Starting with the Holoscan SDK 0.5, we provide a utility container on NGC named [L4T Compute Assist](#) designed to enable iGPU compute access on the development stack configured for dGPU.

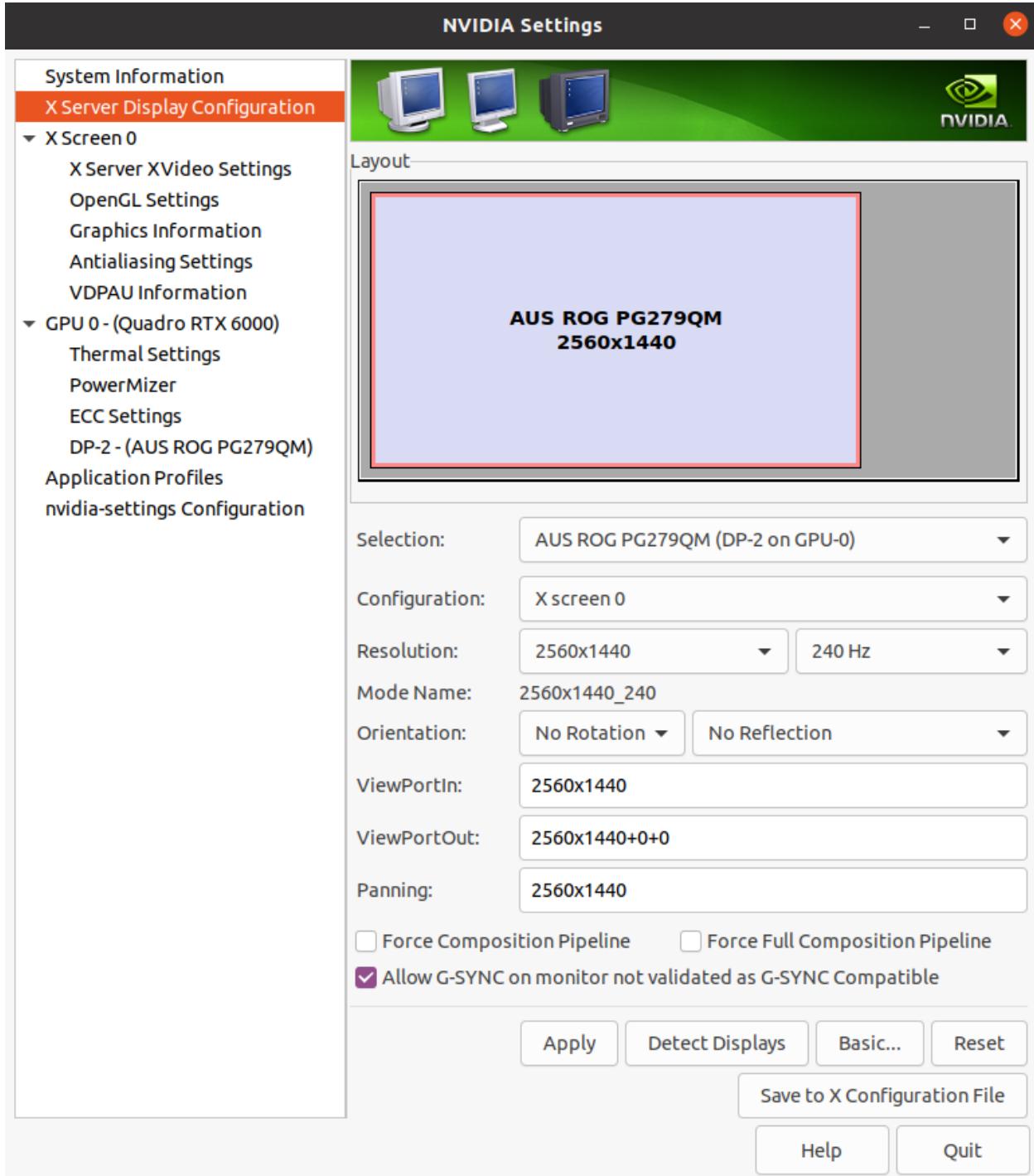


Fig. 4.1: Enable G-SYNC for the current display

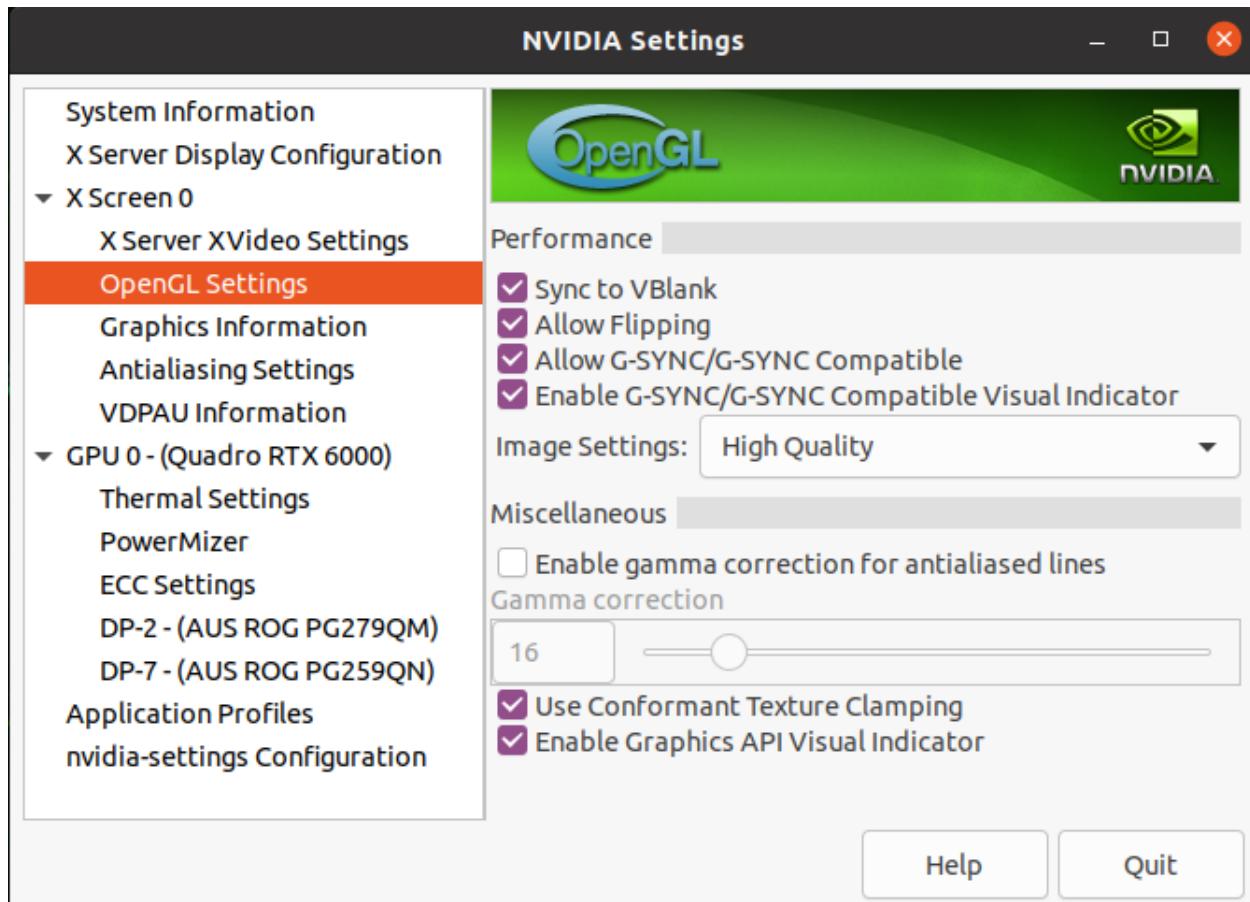


Fig. 4.2: Enable Visual Indicator for the current display

Note: This container enables using the iGPU for compute capabilities only (not rendering).

THIRD PARTY HARDWARE SETUP

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

5.1 AJA Video Systems

AJA provides a wide range of proven, professional video I/O devices, and thanks to a partnership between NVIDIA and AJA, Holoscan 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 sysmem to GPU copies are eliminated from the processing pipeline.

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

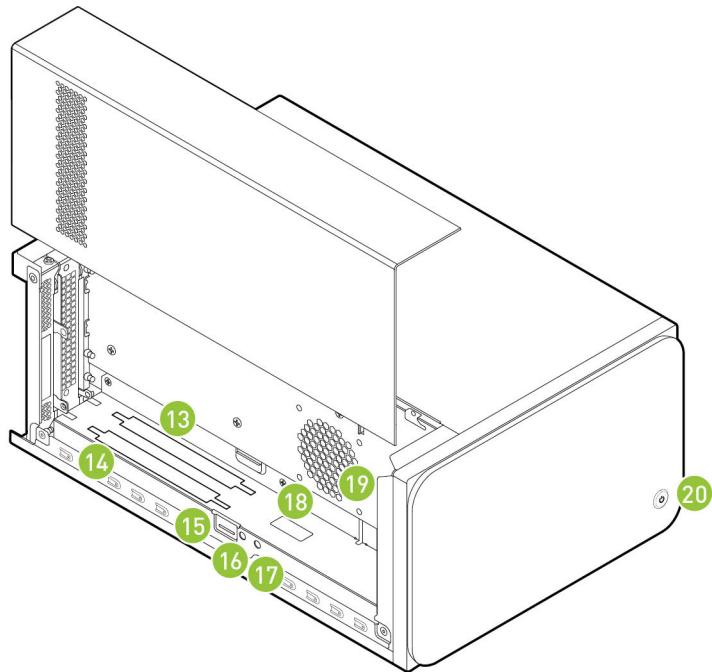
Note: The addition of an AJA device to a Holoscan Developer Kit is optional. The Holoscan SDK has elements that can be run with an AJA device with the additional features mentioned above, but those elements can also run without AJA. For example, there are Holoscan sample applications that have an AJA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

5.1.1 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.



5.1.2 Installing the AJA Software

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

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

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

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

- [Downloading the AJA NTV2 SDK Source](#)
- [Building the AJA NTV2 Drivers](#)

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

- [Loading the AJA NTV2 Drivers](#)

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

- [Building and Installing the AJA NTV2 SDK](#)
 - [Testing the AJA Device](#)
-

Downloading the AJA NTV2 SDK Source

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

```
$ git clone https://github.com/nvidia-holoscan/ntv2.git
$ export NTV2=$(pwd)/ntv2
```

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.

```
$ export AJA_RDMA=1
$ export AJA_IGPU=0 # Or 1 to run on the integrated GPU of the IGX Orin Devkit
  ↵(L4T >= 35.4)
$ make -j --directory ${NTV2}/ajadriver/linux
```

Loading the AJA NTV2 Drivers

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

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

Building and Installing the AJA NTV2 SDK

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

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

Testing the AJA Device

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

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

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

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

(continues on next page)

(continued from previous page)

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
```

- 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
```

5.1.3 Using AJA Devices in Containers

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

5.1.4 Troubleshooting

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

Solutions:

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

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

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

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

- Enter a temporary password and reboot the system.
- Upon reboot press any key when you see the blue screen MOK Management
- Select Change Secure Boot state
- Enter the password your selected
- Select Yes to disable Secure Book in shim-signed
- After reboot you can verify again that SecureBoot validation is disabled in shim.

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

Solutions:

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

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

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

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

3. **Problem:** The `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.

5.2 Emergent Vision Technologies (EVT)

Thanks to a collaboration with [Emergent Vision Technologies](#), the Holoscan SDK now supports EVT high-speed cameras.

Note: The addition of an EVT camera to the Holoscan Developer Kits 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.

5.2.1 Installing EVT Hardware

The EVT cameras can be connected to Holoscan Developer Kits through [Mellanox ConnectX SmartNIC](#), with the most simple connection method being a single cable between a camera and the 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 Holoscan SDK application has been tested using a SFP28 copper cable of 2M or less. Longer copper cables or optical cables and optical modules can be used but these have not been tested as a part of this development.

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

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

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

5.2.2 Installing EVT Software

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

Note: The Emergent SDK depends on Rivermax SDK v1.20 and Mellanox OFED Network Drivers v5.8 which are pre-installed by the SDK Manager on the Holoscan Developer Kits. To avoid duplicate installation of the Rivermax SDK and the Mellanox OFED Network Drivers use the following command when installing the Emergent SDK:

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

5.2.3 Post EVT Software Installation Steps

After installation of the software, execute the steps below to bring up the camera node on the Holoscan devkits 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.
- if above command does not yield anything, do following and try again:

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

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
```

5.2.4 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](#).

5.2.5 Troubleshooting

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

Solution:

- Make sure that the MLNX ConnectX SmartNIC is configured with the correct IP address. Follow section [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 application in the home window.

Solution:

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

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

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

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

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

HOLOSCAN CORE CONCEPTS

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

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

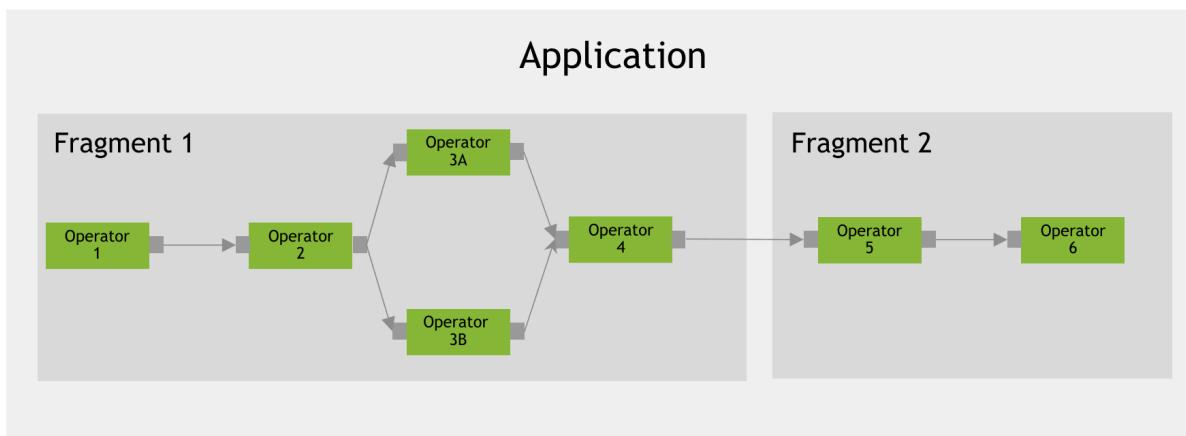


Fig. 6.1: Core concepts: Application

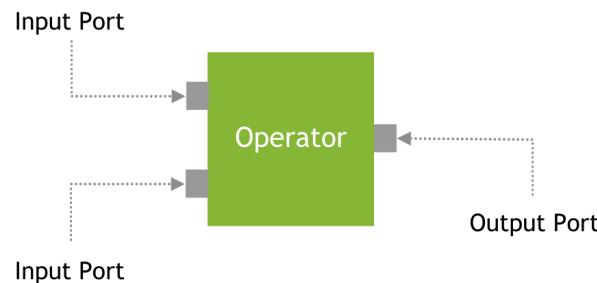


Fig. 6.2: Core concepts: Port

The core concepts of the Holoscan API are:

- **Application:** An application acquires and processes streaming data. An application is a collection of fragments where each fragment can be allocated to execute on a physical node of a Holoscan cluster.
- **Fragment:** A fragment is a building block of the Application. It is a *Directed Acyclic Graph* (DAG) of operators. A fragment can be assigned to a physical node of a Holoscan cluster during execution. The run-time execution manages communication across fragments. In a Fragment, Operators (Graph Nodes) are connected to each other by flows (Graph Edges).
- **Operator:** An operator is the most basic unit of work in this framework. An Operator receives streaming data at an input port, processes it, and publishes it to one of its output ports. A *Codelet* in GXF would be replaced by an Operator in the Holoscan SDK. An Operator encapsulates Receivers and Transmitters of a GXF *Entity* as Input/Output Ports of the Operator.
- **(Operator) Resource:** Resources such as system memory or a GPU memory pool that an Operator needs to perform its job. Resources are allocated during the initialization phase of the application. This matches the semantics of GXF's Memory Allocator or any other components derived from the Component class in GXF.
- **Condition:** A condition is a predicate that can be evaluated at runtime to determine if an operator should execute. This matches the semantics of GXF's *Scheduling Term*.
- **Port:** 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.

HOLOSCAN BY EXAMPLE

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

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

7.1 Hello World

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

In this example we'll cover:

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

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

7.1.1 Defining the `HelloWorldApp` class

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

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

C++

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

Python

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

7.1.2 Defining the HelloWorldApp workflow

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

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

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

C++

```

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

```

Python

```

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

```

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

C++

```

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

```

(continues on next page)

(continued from previous page)

```

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

```

Python

```

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

```

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

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

7.1.3 Running the Application

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

```
Hello World!
```

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

7.2 Ping Simple

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

In this example we'll cover:

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

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

7.2.1 Operators and Workflow

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

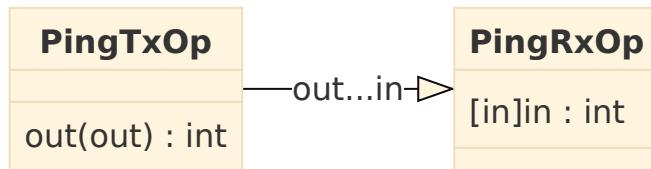


Fig. 7.1: A linear workflow

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

7.2.2 Connecting Operators

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

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

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

C++

```

1 #include <holoscan/holoscan.hpp>
2 #include <holoscan/operators/ping_tx/ping_tx.hpp>
3 #include <holoscan/operators/ping_rx/ping_rx.hpp>
4
5 class MyPingApp : public holoscan::Application {
6 public:
7     void compose() override {
8         using namespace holoscan;
9         // Create the tx and rx operators
10        auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
11        auto rx = make_operator<ops::PingRxOp>("rx");
12
13        // Connect the operators into the workflow: tx -> rx
14        add_flow(tx, rx);
15    }
16};
17
18 int main(int argc, char** argv) {
19     auto app = holoscan::make_application<MyPingApp>();
20     app->run();

```

(continues on next page)

(continued from previous page)

```

21     return 0;
22 }
23

```

- The header files that define **PingTxOp** and **PingRxOp** are included on lines 2 and 3 respectively.
- We create an instance of the **PingTxOp** using the `make_operator()` function (line 9) with the name “tx” and constrain it’s `compute()` method to execute 10 times.
- We create an instance of the **PingRxOp** using the `make_operator()` function (line 10) with the name “rx”.
- The tx and rx operators are connected using `add_flow()` (line 12)

Python

```

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

```

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

7.2.3 Running the Application

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

```

Rx message value: 1
Rx message value: 2
Rx message value: 3
Rx message value: 4
Rx message value: 5
Rx message value: 6
Rx message value: 7

```

(continues on next page)

(continued from previous page)

```
Rx message value: 8
Rx message value: 9
Rx message value: 10
```

7.3 Ping Custom Op

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

In this example we will cover:

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

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

7.3.1 Operators and Workflow

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

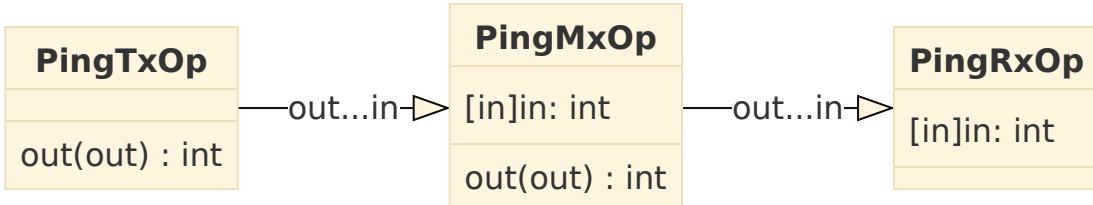


Fig. 7.2: A linear workflow with new custom operator

Compared to the previous example, we are adding a new **PingMxOp** operator between the **PingTxOp** and **PingRxOp** operators. This new operator takes as input an integer, multiplies it by a constant factor, and then sends the new value to **PingRxOp**. You can think of this custom operator as doing some data processing on an input stream before sending the result to downstream operators.

7.3.2 Configuring Operator Input and Output Ports

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

7.3.3 Configuring Operator Parameters

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

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

C++

```

1 #include <holoscan/holoscan.hpp>
2 #include <holoscan/operators/ping_tx/ping_tx.hpp>
3 #include <holoscan/operators/ping_rx/ping_rx.hpp>
4
5 namespace holoscan::ops {
6
7     class PingMxOp : public Operator {
8         public:
9             HOLOSCAN_OPERATOR_FORWARD_ARGS(PingMxOp)
10
11             PingMxOp() = default;
12
13             void setup(OperatorSpec& spec) override {
14                 spec.input<int>("in");
15                 spec.output<int>("out");
16                 spec.param(multiplier_, "multiplier", "Multiplier", "Multiply the input by this value",
17                             "2");
18             }
19
20             void compute(InputContext& op_input, OutputContext& op_output, ExecutionContext& op_context)
21             override {
22                 auto value = op_input.receive<int>("in");
23
24                 std::cout << "Middle message value: " << *(value.get()) << std::endl;
25
26                 // Multiply the value by the multiplier parameter
27                 *(value.get()) *= multiplier_;
28
29                 op_output.emit(value);
30             }
31
32     private:
33         Parameter<int> multiplier_;
34     };
35 }
```

(continues on next page)

(continued from previous page)

```

32 };
33
34 } // namespace holoscan::ops

```

- The `PingMxOp` class inherits from the `Operator` base class (line 7).
- The `HOLOSCAN_OPERATOR_FORWARD_ARGS` macro (line 9) is syntactic sugar to help forward an operator's constructor arguments to the `Operator` base class, and is a convenient shorthand to avoid having to manually define constructors for your operator with the necessary parameters.
- Input/output ports with the names “in”/“out” are added to the operator spec on lines 14 and 15 respectively. The port type of both ports are `int` as indicated by the template argument `<int>`.
- We add a “multiplier” parameter to the operator spec (line 16) with a default value of 2. This parameter is tied to the private “`multiplier_`” data member.
- In the `compute()` method, we receive the integer data from the operator’s “in” port (line 20), print it’s value, multiply it’s value by the multiplicative factor, and send the new value downstream (line 27).
- On line 25, note that the data being passed between the operators has the type `std::shared_ptr<int>`, a shared pointer to a `int` object. To get the raw pointer to the integer, `value.get()` is first called within the parenthesis.
- The call to `op_output.emit(value)` on line 27 is equivalent to `op_output.emit(value, "out")` since this operator has only 1 output port. If the operator has more than 1 output port, then the port name is required.

Python

```

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

```

(continues on next page)

(continued from previous page)

```
27
28     op_output.emit(value, "out")
```

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

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

C++

```
35 class MyPingApp : public holoscan::Application {
36 public:
37     void compose() override {
38         using namespace holoscan;
39         // Define the tx, mx, rx operators, allowing tx operator to execute 10 times
40         auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
41         auto mx = make_operator<ops::PingMxOp>("mx", Arg("multiplier", 3));
42         auto rx = make_operator<ops::PingRxOp>("rx");
43
44         // Define the workflow: tx -> mx -> rx
45         add_flow(tx, mx);
46         add_flow(mx, rx);
47     }
48 };
49
50 int main(int argc, char** argv) {
51     auto app = holoscan::make_application<MyPingApp>();
52     app->run();
53
54     return 0;
55 }
```

- The tx, mx, and rx operators are created in the compose() method on lines 40-42.
- The custom mx operator is created in exactly the same way with make_operator() (line 41) as the built-in operators, and configured with a “multiplier” parameter initialized to 3 which overrides the parameter’s default value of 2 (line 16).
- The workflow is defined by connecting tx to mx, and mx to rx using add_flow() on lines 45-46.

Python

```

29 class MyPingApp(Application):
30     def compose(self):
31         # Define the tx, mx, rx operators, allowing the tx operator to execute 10 times
32         tx = PingTxOp(self, CountCondition(self, 10), name="tx")
33         mx = PingMxOp(self, name="mx", multiplier=3)
34         rx = PingRxOp(self, name="rx")
35
36         # Define the workflow: tx -> mx -> rx
37         self.add_flow(tx, mx)
38         self.add_flow(mx, rx)
39
40
41 if __name__ == "__main__":
42     app = MyPingApp()
43     app.run()

```

- The tx, mx, and rx operators are created in the `compose()` method on lines 32-34.
- The custom mx operator is created in exactly the same way as the built-in operators (line 33), and configured with a “multiplier” parameter initialized to 3 which overrides the parameter’s default value of 2 (line 19).
- The workflow is defined by connecting tx to mx, and mx to rx using `add_flow()` on lines 37-38.

7.3.4 Message Data Types

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

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

C++

```

1 #include "holoscan/operators/ping_tx/ping_tx.hpp"
2
3 namespace holoscan::ops {
4
5     void PingTxOp::setup(OperatorSpec& spec) {
6         spec.output<int>("out");
7     }
8
9     void PingTxOp::compute(InputContext&, OutputContext& op_output, ExecutionContext&) {
10        auto value = std::make_shared<int>(index_++);
11        op_output.emit(value, "out");
12    }
13
14 } // namespace holoscan::ops

```

- The “out” port of the `PingTxOp` has the type `int` (line 6).

- The type returned by the call to `std::make_shared<int>()` on line 10 is `std::shared_ptr<int>`, and this shared pointer to an integer is what is published to the “out” port when calling `emit()` (line 11).
- The message received by the downstream **PingMxOp** operator when it calls `op_input.receive<int>()` has the type `std::shared_ptr<int>`.

Python

```

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

```

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

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

7.3.5 Running the Application

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

```

Middle message value: 1
Rx message value: 3
Middle message value: 2
Rx message value: 6
Middle message value: 3
Rx message value: 9
Middle message value: 4
Rx message value: 12
Middle message value: 5
Rx message value: 15
Middle message value: 6
Rx message value: 18
Middle message value: 7
Rx message value: 21

```

(continues on next page)

(continued from previous page)

```
Middle message value: 8
Rx message value: 24
Middle message value: 9
Rx message value: 27
Middle message value: 10
Rx message value: 30
```

7.4 Ping Multi Port

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

In this example we will cover:

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

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

7.4.1 Operators and Workflow

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

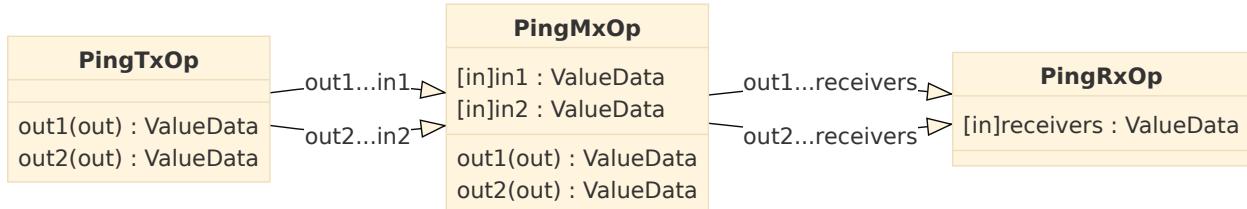


Fig. 7.3: A workflow with multiple inputs and outputs

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

7.4.2 User Defined Data Types

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

C++

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

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

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

Python

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

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

7.4.3 Defining an Explicit Number of Inputs and Outputs

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

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

C++

```

18 namespace holoscan::ops {
19
20 class PingTxOp : public Operator {
21 public:
22     HOLOSCAN_OPERATOR_FORWARD_ARGS(PingTxOp)
23
24     PingTxOp() = default;
25
26     void setup(OperatorSpec& spec) override {
27         spec.output<ValueData>("out1");
28         spec.output<ValueData>("out2");
29     }
30
31     void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override {
32         auto value1 = std::make_shared<ValueData>(index_++);
33         op_output.emit(value1, "out1");
34
35         auto value2 = std::make_shared<ValueData>(index_++);
36         op_output.emit(value2, "out2");
37     };
38     int index_ = 1;
39 };

```

- We configure the output ports with the `ValueData` type on lines 27 and 28 using `spec.output<ValueData>()`.
- The values are then sent out using `op_output.emit()` on lines 33 and 36. The port name is required since there is more than one port on this operator.

Note: Data passed between operators is wrapped in shared pointers (`std::shared_ptr`), hence the call to `std::make_shared<ValueData>(...)` on lines 32 and 35.

Python

```

19 class PingTxOp(Operator):
20     """Simple transmitter operator.
21
22     This operator has:
23     outputs: "out1", "out2"
24
25     On each tick, it transmits a `ValueData` object at each port. The
26     transmitted values are even on port1 and odd on port2 and increment with
27     each call to compute.

```

(continues on next page)

(continued from previous page)

```

28     """
29
30     def __init__(self, fragment, *args, **kwargs):
31         self.index = 1
32         super().__init__(fragment, *args, **kwargs)
33
34     def setup(self, spec: OperatorSpec):
35         spec.output("out1")
36         spec.output("out2")
37
38     def compute(self, op_input, op_output, context):
39         value1 = ValueData(self.index)
40         self.index += 1
41         op_output.emit(value1, "out1")
42
43         value2 = ValueData(self.index)
44         self.index += 1
45         op_output.emit(value2, "out2")

```

- We configure the output ports on lines 35 and 36 using `spec.output()`. There is no need to reference the type (`ValueData`) in Python.
- The values are then sent out using `op_output.emit()` on lines 41 and 45.

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

C++

```

40 class PingMxOp : public Operator {
41 public:
42     HOLOSCAN_OPERATOR_FORWARD_ARGS(PingMxOp)
43
44     PingMxOp() = default;
45
46     void setup(OperatorSpec& spec) override {
47         spec.input<ValueData>("in1");
48         spec.input<ValueData>("in2");
49         spec.output<ValueData>("out1");
50         spec.output<ValueData>("out2");
51         spec.param(multiplier_, "multiplier", "Multiplier", "Multiply the input by this value
52             ", 2);
53     }
54
55     void compute(InputContext& op_input, OutputContext& op_output, ExecutionContext&)
56     override {
57         auto value1 = op_input.receive<ValueData>("in1");
58         auto value2 = op_input.receive<ValueData>("in2");
59
60         HOLOSCAN_LOG_INFO("Middle message received (count: {})", count_++);
61
62         HOLOSCAN_LOG_INFO("Middle message value1: {}", value1->data());
63         HOLOSCAN_LOG_INFO("Middle message value2: {}", value2->data());

```

(continues on next page)

(continued from previous page)

```

62
63     // Multiply the values by the multiplier parameter
64     value1->data(value1->data() * multiplier_);
65     value2->data(value2->data() * multiplier_);

66
67     op_output.emit(value1, "out1");
68     op_output.emit(value2, "out2");
69 }

70
71 private:
72     int count_ = 1;
73     Parameter<int> multiplier_;
74 }
```

- We configure the input ports with the `ValueData` type on lines 47 and 48 using `spec.input<ValueData>()`.
- The values are received using `op_input.receive()` on lines 55 and 56 using the port names. The received values are of type `std::shared_ptr<ValueData>`.

Python

```

46 class PingMxOp(Operator):
47     """Example of an operator modifying data.
48
49     This operator has:
50         inputs: "in1", "in2"
51         outputs: "out1", "out2"
52
53     The data from each input is multiplied by a user-defined value.
54     """
55
56     def __init__(self, fragment, *args, **kwargs):
57         self.count = 1
58         super().__init__(fragment, *args, **kwargs)
59
60     def setup(self, spec: OperatorSpec):
61         spec.input("in1")
62         spec.input("in2")
63         spec.output("out1")
64         spec.output("out2")
65         spec.param("multiplier", 2)
66
67     def compute(self, op_input, op_output, context):
68         value1 = op_input.receive("in1")
69         value2 = op_input.receive("in2")
70         print(f"Middle message received (count: {self.count})")
71         self.count += 1
72
73         print(f"Middle message value1: {value1.data}")
74         print(f"Middle message value2: {value2.data}")
```

(continues on next page)

(continued from previous page)

```

76     # Multiply the values by the multiplier parameter
77     value1.data *= self.multiplier
78     value2.data *= self.multiplier
79
80     op_output.emit(value1, "out1")
81     op_output.emit(value2, "out2")

```

Sending messages of arbitrary data types is pretty straightforward in Python. The code to define the operator input ports (lines 61–62), and to receive them (lines 68, 69) did not change when we went from passing `int` to `ValueData` objects.

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

7.4.4 Receiving Any Number of Inputs

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

C++

```

75 class PingRxOp : public Operator {
76 public:
77     HOLOSCAN_OPERATOR_FORWARD_ARGS(PingRxOp)
78
79     PingRxOp() = default;
80
81     void setup(OperatorSpec& spec) override {
82         spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.", {});
83     }
84
85     void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {
86         auto value_vector = op_input.receive<std::vector<ValueData>>("receivers");
87
88         HOLOSCAN_LOG_INFO("Rx message received (count: {}, size: {})", count_++, value_
89         <vector.size());
90
91         HOLOSCAN_LOG_INFO("Rx message value1: {}", value_vector[0]->data());
92         HOLOSCAN_LOG_INFO("Rx message value2: {}", value_vector[1]->data());
93     }
94
95     private:
96         Parameter<std::vector<IOSpec*>> receivers_;
97         int count_ = 1;
98     };
99 } // namespace holoscan::ops

```

- In the operator's `setup()` method, we define a parameter `receivers` (line 82) that is tied to the private data member `receivers_` (line 95) of type `Parameter<std::vector<IOSpec*>>`.

- The values are retrieved using `op_input.receive<std::vector<ValueData>>(...)`.
- `value_vector`'s type is `std::vector<std::shared_ptr<ValueData>>` (line 86).

Python

```

82 class PingRxOp(Operator):
83     """Simple receiver operator.
84
85     This operator has:
86         input: "receivers"
87
88     This is an example of a native operator that can dynamically have any
89     number of inputs connected to its "receivers" port.
90     """
91
92     def __init__(self, fragment, *args, **kwargs):
93         self.count = 1
94         super().__init__(fragment, *args, **kwargs)
95
96     def setup(self, spec: OperatorSpec):
97         spec.param("receivers", kind="receivers")
98
99     def compute(self, op_input, op_output, context):
100        values = op_input.receive("receivers")
101        print(f"Rx message received (count: {self.count}, size: {len(values)}")
102        self.count += 1
103        print(f"Rx message value1: {values[0].data}")
104        print(f"Rx message value2: {values[1].data}")

```

- In Python, a port that can be connected to multiple upstream ports is created by defining a parameter and setting the argument `kind="receivers"` (line 97).
- The call to `receive()` returns a tuple of `ValueData` objects (line 100).

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

C++

```

100 class MyPingApp : public holoscan::Application {
101 public:
102     void compose() override {
103         using namespace holoscan;
104
105         // Define the tx, mx, rx operators, allowing the tx operator to execute 10 times
106         auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
107         auto mx = make_operator<ops::PingMxOp>("mx", Arg("multiplier", 3));
108         auto rx = make_operator<ops::PingRxOp>("rx");
109
110         // Define the workflow
111         add_flow(tx, mx, {"out1", "in1"}, {"out2", "in2"});
112         add_flow(mx, rx, {"out1", "receivers"}, {"out2", "receivers"});
113     }

```

(continues on next page)

(continued from previous page)

```

114 };
115
116 int main(int argc, char** argv) {
117     holoscan::load_env_log_level();
118     auto app = holoscan::make_application<MyPingApp>();
119     app->run();
120
121     return 0;
122 }
```

Python

```

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

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

Note: The `load_env_log_level()` function loads the logging level from the `HOLOSCAN_LOG_LEVEL` environment variable .

7.4.5 Running the Application

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

```

2023-03-17 01:00:28.448 INFO  /workspace/holoscan-sdk/src/core/executors/gxf/gxf_
↳ executor.cpp@71: Creating context
[2023-03-17 01:00:28.459] [holoscan] [info] [gxf_executor.cpp:100] Loading extensions_
↳ from configs...
[2023-03-17 01:00:28.459] [holoscan] [info] [gxf_executor.cpp:285] Activating Graph...
[2023-03-17 01:00:28.460] [holoscan] [info] [gxf_executor.cpp:287] Running Graph...
```

(continues on next page)

(continued from previous page)

```
[2023-03-17 01:00:28.460] [holoscan] [info] [gxf_executor.cpp:289] Waiting for ↵
↳ completion...
[2023-03-17 01:00:28.461 INFO gxf/std/greedy_scheduler.cpp@184: Scheduling 3 entities
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:77] Middle message ↵
↳ received (count: 1)
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:79] Middle message ↵
↳ value1: 1
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:80] Middle message ↵
↳ value2: 2
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:108] Rx message ↵
↳ received (count: 1, size: 2)
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:110] Rx message value1: ↵
↳ 3
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:111] Rx message value2: ↵
↳ 6
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:77] Middle message ↵
↳ received (count: 2)
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:79] Middle message ↵
↳ value1: 3
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:80] Middle message ↵
↳ value2: 4
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:108] Rx message ↵
↳ received (count: 2, size: 2)
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:110] Rx message value1: ↵
↳ 9
[2023-03-17 01:00:28.461] [holoscan] [info] [ping_multi_port.cpp:111] Rx message value2: ↵
↳ 12
...
[2023-03-17 01:00:28.463] [holoscan] [info] [ping_multi_port.cpp:110] Rx message value1: ↵
↳ 51
[2023-03-17 01:00:28.463] [holoscan] [info] [ping_multi_port.cpp:111] Rx message value2: ↵
↳ 54
[2023-03-17 01:00:28.464] [holoscan] [info] [ping_multi_port.cpp:77] Middle message ↵
↳ received (count: 10)
[2023-03-17 01:00:28.464] [holoscan] [info] [ping_multi_port.cpp:79] Middle message ↵
↳ value1: 19
[2023-03-17 01:00:28.464] [holoscan] [info] [ping_multi_port.cpp:80] Middle message ↵
↳ value2: 20
[2023-03-17 01:00:28.464] [holoscan] [info] [ping_multi_port.cpp:108] Rx message ↵
↳ received (count: 10, size: 2)
[2023-03-17 01:00:28.464] [holoscan] [info] [ping_multi_port.cpp:110] Rx message value1: ↵
↳ 57
[2023-03-17 01:00:28.464] [holoscan] [info] [ping_multi_port.cpp:111] Rx message value2: ↵
↳ 60
...
2023-03-17 01:00:28.464 INFO gxf/std/greedy_scheduler.cpp@367: Scheduler finished.
[2023-03-17 01:00:28.464] [holoscan] [info] [gxf_executor.cpp:291] Deactivating Graph...
2023-03-17 01:00:28.464 INFO /workspace/holoscan-sdk/src/core/executors/gxf/gxf_ ↵
↳ executor.cpp@88: Destroying context
```

Note: Depending on your log level you may see more or fewer messages. The output above was generated using the

default value of INFO.

7.5 Video Replayer

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

In this example we'll cover:

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

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

7.5.1 Operators and Workflow

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

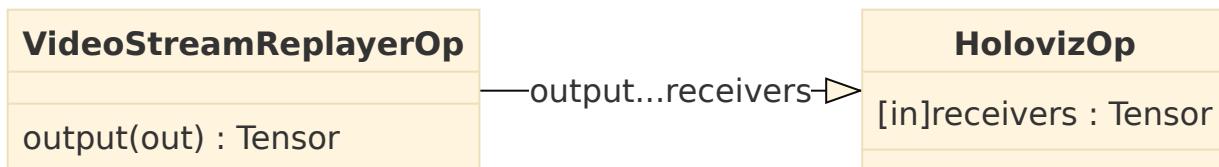


Fig. 7.4: Workflow to load and display video from a file

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

7.5.2 Video Stream Replayer Operator

The built-in video stream replayer operator can be used to replay a video stream that has been encoded as gxf entities. You can use the `convert_video_to_gxf_entities.py` script to encode a video file as gxf entities for use by this operator.

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

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

7.5.3 Holoviz Operator

The built-in Holoviz operator provides the functionality of the [Visualization Module](#) for Holoscan SDK applications to visualize data. Holoviz composites real time streams of frames with multiple different other layers like segmentation mask layers, geometry layers and GUI layers.

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

7.5.4 Application Configuration File (YAML)

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

Tip: For C++ applications, the configuration file can be a nice way to set the behavior of the application at runtime without having to recompile the code.

This example uses the following configuration file to configure the parameters for the replayer and Holoviz operators. The full list of parameters can be found at [operators-video-stream-replayer](#) and [operators-holoviz](#).

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

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

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

C++

```

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

```

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

Python

```

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

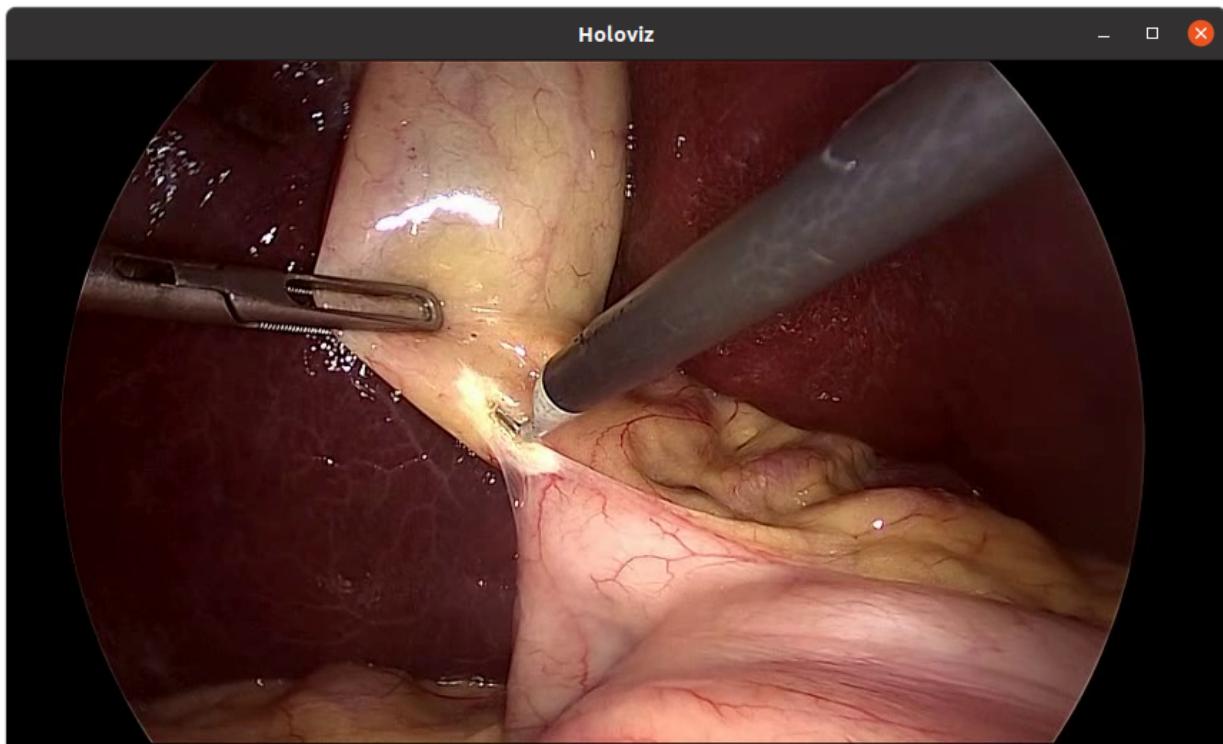
- The built-in **VideoStreamReplayerOp** and **HolovizOp** operators are imported on line 5.
- We create an instance of **VideoStreamReplayerOp** named “replayer” with parameters initialized from the YAML configuration file using `**self.kwargs()` (lines 28-30).
- For the python script, the path to the gxf entity video data is not set in the application configuration file but

determined by the code on lines 7 and 23 and is passed directly as the “directory” argument (line 29). This allows more flexibility for the user to run the script from any directory by setting the HOLOSCAN_SAMPLE_DATA_PATH directory (line 7).

- We create an instance of **HolovizOp** named “holoviz” with parameters initialized from the YAML configuration file using `**self.kwargs()` (line 31).
- The “output” port of “replayer” operator is connected to the “receivers” port of the “holoviz” operator and defines the application workflow (line 34).
- The application’s YAML configuration file contains the parameters for our operators, and is loaded on line 45. If no argument is passed to the python script, the application looks for a file with the name “video_replayer.yaml” in the same directory as the script (line 39), otherwise it treats the argument as the path to the app’s YAML configuration file (lines 41-42).

7.5.5 Running the Application

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



7.6 Bring Your Own Model (BYOM)

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

In this example we’ll cover:

- the usage of `FormatConverterOp`, `MultiAIInferenceOp`, `SegmentationPostprocessorOp` operators to add AI inference into the workflow

- how to modify the existing code in this example to create an ultrasound segmentation application to visualize the results from a spinal scoliosis segmentation model

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

7.6.1 Operators and Workflow

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



Fig. 7.5: The BYOM inference workflow

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

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

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

7.6.2 Prerequisites

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

```

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

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

Input video

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

Note: To use your own video data, you can use the `convert_video_to_gxf_entities.py` script from [here](#) to encode your video.

Input model

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

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

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

7.6.3 Understanding the Application Code

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

Python

```
 1 import os
 2 from argparse import ArgumentParser
 3
 4 from holoscan.core import Application
 5 from holoscan.logger import load_env_log_level
 6 from holoscan.operators import (
 7     FormatConverterOp,
 8     HolovizOp,
 9     MultiAIIInferenceOp,
10     SegmentationPostprocessorOp,
11     VideoStreamReplayerOp,
12 )
13 from holoscan.resources import UnboundedAllocator
14
15
16 class BYOMApp(Application):
17     def __init__(self, data):
18         """Initialize the application
19
20             Parameters
21             -----
```

(continues on next page)

(continued from previous page)

```

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

```

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

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

Python

```

43     def compose(self):
44         host_allocator = UnboundedAllocator(self, name="host_allocator")
45
46         source = VideoStreamReplayerOp(
47             self, name="replayer", directory=self.video_dir, **self.kwargs("replayer")
48         )
49
50         preprocessor = FormatConverterOp(
51             self, name="preprocessor", pool=host_allocator, **self.kwargs("preprocessor")
52         )
53
54         inference = MultiAIInferenceOp(
55             self,
56             name="inference",
57             allocator=host_allocator,

```

(continues on next page)

(continued from previous page)

```

58     model_path_map=self.model_path_map,
59     **self.kwargs("inference"),
60 )
61
62     postprocessor = SegmentationPostprocessorOp(
63         self, name="postprocessor", allocator=host_allocator, **self.kwargs(
64             "postprocessor")
65     )
66
67     viz = HolovizOp(self, name="viz", **self.kwargs("viz"))

```

- An instance of the `UnboundedAllocator` resource class is created (line 44) and used by subsequent operators for memory allocation. This allocator allocates memory dynamically on the host as needed. For applications where latency becomes an issue, there is the `BlockMemoryPool` allocator.
- The preprocessor operator (line 50) takes care of converting the input video from the source video to a format that can be used by the AI model.
- The inference operator (line 54) feeds the output from the preprocessor to the AI model to perform inference.
- The postprocessor operator (line 62) postprocesses the output from the inference operator before passing it downstream to the visualizer. Here, the segmentation postprocessor checks the probabilities output from the model to determine which class is most likely and emits this class index. This is then used by the `Holoviz` operator to create a segmentation mask overlay.

YAML

```

1 %YAML 1.2
2 replayer: # VideoStreamReplayer
3   basename: "surgical_video"
4   frame_rate: 0 # as specified in timestamps
5   repeat: true # default: false
6   realtime: true # default: true
7   count: 0 # default: 0 (no frame count restriction)
8
9 preprocessor: # FormatConverter
10   out_tensor_name: source_video
11   out_dtype: "float32"
12   resize_width: 512
13   resize_height: 512
14
15 inference: # MultiAIInference
16   backend: "trt"
17   pre_processor_map:
18     "byom_model": ["source_video"]
19   inference_map:
20     "byom_model": "output"
21   in_tensor_names: ["source_video"]
22   out_tensor_names: ["output"]
23
24 postprocessor: # SegmentationPostprocessor
25   in_tensor_name: output

```

(continues on next page)

(continued from previous page)

```

26 # network_output_type: None
27 data_format: nchw
28
29 viz: # Holoviz
30   width: 854
31   height: 480
32   color_lut: [
33     [0.65, 0.81, 0.89, 0.1],
34   ]

```

- The preprocessor converts the tensors to float32 values (line 11) and ensures that the image is resized to 512x512 (line 12–13).
- The pre_processor_map parameter (lines 17–18) maps the model name(s) to input tensor name(s). Here, “source_video” matches the output tensor name of the preprocessor (line 10). The inference_map parameter maps the model name(s) to the output tensor name(s). Here, “output”, matches the input tensor name of the postprocessor (line 25). in_tensor_names is a list of all the input tensor names (line 21). out_tensor_names is a list of all the output tensor names (line 22). For more details on MultiAIInferenceOp parameters, see [Customizing the MultiAI Inference Operator](#) or refer to [Inference Module](#).
- The network_output_type parameter is commented out on line 26 to remind ourselves that this second branch is currently not generating anything interesting. If not specified, this parameter defaults to “softmax” for SegmentationPostprocessorOp.
- The color lookup table defined on lines 32–34 is used here to create a segmentation mask overlay. The values of each entry in the table are RGBA values between 0.0 and 1.0. For the alpha value, 0.0 is fully transparent and 1.0 is fully opaque.

Finally, we define the application and workflow.

Python

```

67 # Define the workflow
68 self.add_flow(source, viz, {"output": "receivers"})
69 self.add_flow(source, preprocessor, {"output": "source_video"})
70 self.add_flow(preprocessor, inference, {"tensor": "receivers"})
71 self.add_flow(inference, postprocessor, {"transmitter": "in_tensor"})
72 self.add_flow(postprocessor, viz, {"out_tensor": "receivers"})
73
74
75 if __name__ == "__main__":
76     # Parse args
77     parser = ArgumentParser(description="BYOM demo application.")
78     parser.add_argument(
79         "-d",
80         "--data",
81         default="none",
82         help="Set the data path",
83     )
84
85     args = parser.parse_args()
86
87     load_env_log_level()

```

(continues on next page)

(continued from previous page)

```

88 config_file = os.path.join(os.path.dirname(__file__), "byom.yaml")
89
90 app = BYOMApp(data=args.data)
91 app.config(config_file)
92 app.run()
93

```

- The `add_flow()` on line 68 defines the first branch to display the original video.
- The `add_flow()` commands from line 69–72 defines the second branch to display the segmentation mask overlay.

7.6.4 Modifying the Application for Ultrasound Segmentation

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

We will need to modify the python and yaml files to change our application to the ultrasound segmentation application.

Python

```

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

```

- Update `self.model_path_map` to the ultrasound segmentation model (lines 20–23).

- Update `self.video_dir` to point to the directory of the ultrasound video files (line 25).

YAML

```

1  replayer: # VideoStreamReplayer
2    basename: "ultrasound_256x256"
3    frame_rate: 0 # as specified in timestamps
4    repeat: true # default: false
5    realtime: true # default: true
6    count: 0 # default: 0 (no frame count restriction)
7
8  preprocessor: # FormatConverter
9    out_tensor_name: source_video
10   out_dtype: "float32"
11   resize_width: 256
12   resize_height: 256
13
14 inference: # MultiaAIInference
15   backend: "trt"
16   pre_processor_map:
17     "byom_model": ["source_video"]
18   inference_map:
19     "byom_model": "output"
20   in_tensor_names: ["source_video"]
21   out_tensor_names: ["output"]
22
23 postprocessor: # SegmentationPostprocessor
24   in_tensor_name: output
25   network_output_type: softmax
26   data_format: nchw
27
28 viz: # Holoviz
29   width: 854
30   height: 480
31   color_lut: [
32     [0.65, 0.81, 0.89, 0.1],
33     [0.2, 0.63, 0.17, 0.7]
34   ]

```

- Update `basename` to the basename of the ultrasound video files (line 2).
- The AI model expects the width and height of the images to be 256x256, update the preprocessor's parameters to resize the input to 256x256 (line 11-12).
- The AI model's final output layer is a softmax, so we indicate this to the postprocessor (line 25).
- Since this model predicts between two classes, we need another entry in Holoviz's color lookup table (line 33). Note that the alpha value of the first color entry is `0.1` (line 32) so the mask for the background class may not be visible. The second entry we just added is a green color with an alpha value of `0.7` which will be easily visible.

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

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

- Input: This could be a video on disk, an input stream from a capture device, or other data stream.

- Preprocessing: You may need to preprocess the input stream to convert tensors into the shape and format that is expected by your AI model (e.g., converting datatype and resizing).
- Inference: Your model will need to be in onnx or trt format.
- Postprocessing: An operator that postprocesses the output of the model to a format that can be readily used by downstream operators.
- Output: The postprocessed stream can be displayed or used by other downstream operators.

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

7.6.5 Running the Application

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

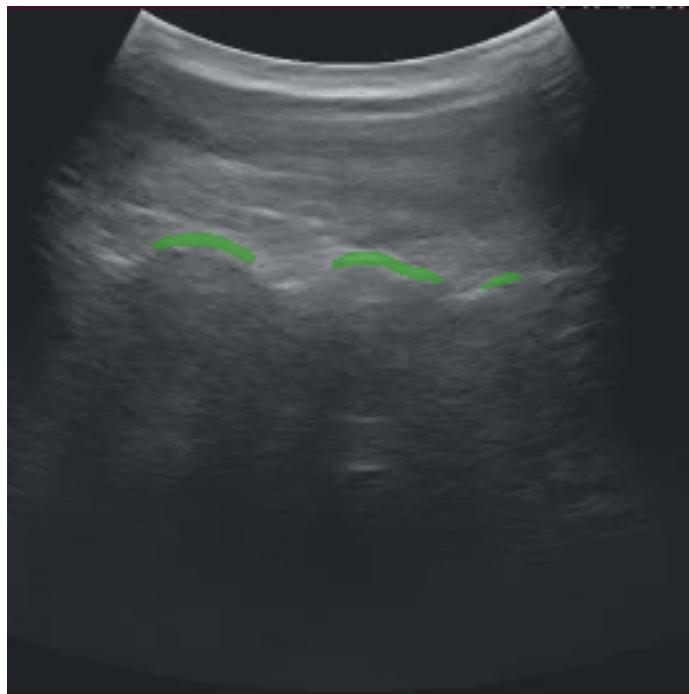


Fig. 7.6: Ultrasound Segmentation

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

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

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

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

7.6.6 Customizing the MultiAI Inference Operator

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

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

7.6.7 Common Pitfalls Deploying New Models

Color Channel Order

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

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

Normalizing Your Data

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

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

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

Network Output Type

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

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

```
network_output_type: softmax
```

CREATING AN APPLICATION

In this section, we'll address:

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

Note: At this time, the Holoscan SDK only supports a single fragment per application. This means that the application can have only one workflow and work on a single machine. We plan to support multiple fragments per application in a future release.

8.1 Defining an Application Class

The following code snippet shows an example Application code skeleton:

C++

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

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

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

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

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

Python

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

```
from holoscan.core import Application

class App(Application):

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

if __name__ == "__main__":
    app = App()
    app.run()
```

8.2 Configuring an Application

An application can be configured at different levels:

1. *providing the GXF extensions that need to be loaded* (when using `GXF operators`)
2. configuring parameters for your application, including `the operators` in the workflow

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

8.2.1 YAML Configuration support

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

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

Here is an example YAML configuration:

```
string_param: "test"
float_param: 0.50
bool_param: true
```

(continues on next page)

(continued from previous page)

```
dict_param:
  key_1: value_1
  key_2: value_2
```

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

C++

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

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

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

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

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

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

Python

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

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

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

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

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

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

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

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

8.2.2 Loading GXF extensions

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

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

YAML

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

C++

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

PYTHON

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

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

8.2.3 Configuring operators

Operators are instantiated in the `compose()` method of your application. They have three type of fields which can be configured: parameters, conditions, and resources.

Configuring operator parameters

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

C++

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

PYTHON

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

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

C++

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

PYTHON

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

Given this YAML configuration:

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

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

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

C++

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

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

PYTHON

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

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

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

C++

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

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

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

PYTHON

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

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

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

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

Configuring operator conditions

By default, operators will continuously run. To change that behavior, some condition classes (C++/Python) can be passed to the constructor of an operator to define when it (its `compute()` method) should execute. This includes:

- A `CountCondition` (C++/Python) can be used to only execute the operator a specific number of times.

C++

```
void compose() override {
    // Will only run 10 times
    auto op = make_operator<MyOp>("my_op", make_condition<CountCondition>(10));
}
```

PYTHON

```
def compose(self):
    # Will only run 10 times
    my_op = MyOp(self, CountCondition(self, 10), name="my_op")
```

- A `BooleanCondition` (C++/Python) can be used to configure when to disable or enable an operator.

C++

```
void compose() override {
    enable_op_condition = make_condition<BooleanCondition>("my_bool_condition")
    auto op = make_operator<MyOp>("my_op", enable_op_condition);
}
```

PYTHON

```
def compose(self):
    enable_op_condition = BooleanCondition(self, name="my_bool_condition")
    my_op = MyOp(self, enable_op_condition, name="my_op")
```

The condition object has two APIs - `enable_tick()` and `disable_tick()` - which will control whether the operator should execute or not. It can be called outside of the operator, or within the operator `compute()` method, based on any arbitrary condition. In the latter case, the name that is provided to the constructor ("`my_bool_condition`" here) must match the name used to retrieve it in the operator's `compute()` method. For example:

C++

```
void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override
{
    // ...
    if (<condition expression>) {           // e.g. if (index_ >= 10)
        auto my_bool_condition = condition<BooleanCondition>("my_bool_condition");
        if (my_bool_condition) {             // if condition exists (not true or
            my_bool_condition->disable_tick(); // this will stop the operator
        }
    }
    // ...
}
```

PYTHON

```
def compute(self, op_input, op_output, context):
    # ...
    if <condition expression>:           # e.g., self.index >= 10
        my_bool_condition = self.conditions.get("my_bool_condition")
        if my_bool_condition:             # if condition exists (not true or false)
            my_bool_condition.disable_tick() # this will stop the operator
    # ...
```

Configuring operator resources

Attention: This section still needs to be written.

8.3 Application Workflows

8.3.1 One-operator Workflow

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



Fig. 8.1: A one-operator workflow

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

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

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

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

CPP

```

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

```

PYTHON

```

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

```

8.3.2 Linear Workflow

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

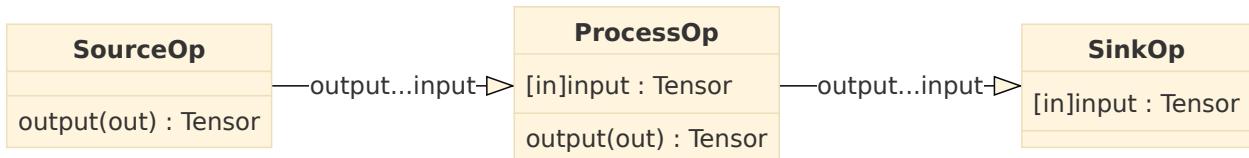


Fig. 8.2: A linear workflow

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

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

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

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

CPP

```

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

```

(continues on next page)

(continued from previous page)

```
12 }
13 };
```

PYTHON

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

12        self.add_flow(process, sink)    # same as `self.add_flow(process, sink, {"", ""})`  

13        ↪`
```

8.3.3 Complex Workflow (Multiple Inputs and Outputs)

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

CPP

```
1 class App : public holoscan::Application {
2     public:
3         void compose() override {
4             // Define Operators
5             auto reader1 = make_operator<Reader1>("reader1");
6             auto reader2 = make_operator<Reader2>("reader2");
7             auto processor1 = make_operator<Processor1>("processor1");
8             auto processor2 = make_operator<Processor2>("processor2");
9             auto processor3 = make_operator<Processor3>("processor3");
10            auto writer = make_operator<Writer>("writer");
11            auto notifier = make_operator<Notifier>("notifier");
12
13            // Define the workflow
14            add_flow(reader1, processor1, {"image", "image1"}, {"image", "image2"}, {"metadata",
15            ↪ "metadata"}});
16            add_flow(reader1, processor1, {"image", "image2"});
17            add_flow(reader2, processor2, {"roi", "roi"});
18            add_flow(processor1, processor2, {"image", "image"});
19            add_flow(processor1, writer, {"image", "image"});
20            add_flow(processor2, notifier);
21            add_flow(processor2, processor3);
22            add_flow(processor3, writer, {"seg_image", "seg_image"});
23        }
24    };
```

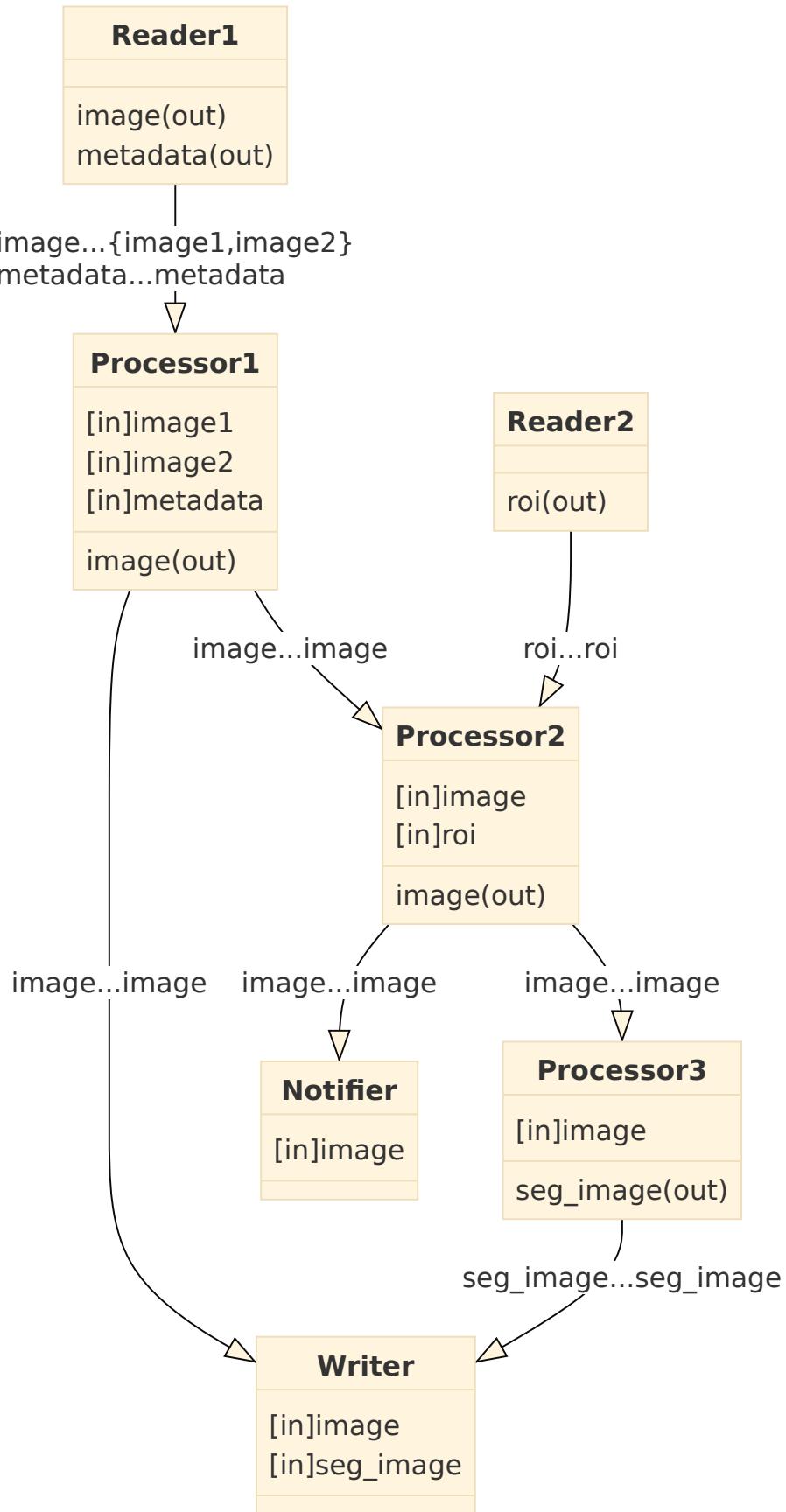


Fig. 8.3: A complex workflow (multiple inputs and outputs)
8.3. Application Workflows

PYTHON

```

1 class App(Application):
2
3     def compose(self):
4         # Define Operators
5         reader1 = Reader10p(self, name="reader1")
6         reader2 = Reader20p(self, name="reader2")
7         processor1 = Processor10p(self, name="processor1")
8         processor2 = Processor20p(self, name="processor2")
9         processor3 = Processor30p(self, name="processor3")
10        notifier = NotifierOp(self, name="notifier")
11        writer = WriterOp(self, name="writer")
12
13        # Define the workflow
14        self.add_flow(reader1, processor1, {("image", "image1"), ("image", "image2"), (
15            "metadata", "metadata")})
16        self.add_flow(reader2, processor2, {("roi", "roi")})
17        self.add_flow(processor1, processor2, {("image", "image")})
18        self.add_flow(processor1, writer, {("image", "image")})
19        self.add_flow(processor2, notifier)
20        self.add_flow(processor2, processor3)
21        self.add_flow(processor3, writer, {("seg_image", "seg_image")})

```

8.4 Building and running your Application

C++

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

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

Listing 8.1: <src_dir>/CMakeLists.txt

```

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

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

# Create an executable for your application
add_executable(my_app main.cpp)

```

(continues on next page)

(continued from previous page)

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

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

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

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

Python

Python applications do not require building. Simply ensure that:

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

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

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

CREATING OPERATORS

9.1 C++ Operators

When assembling a C++ application, two types of operators can be used:

1. *Native C++ operators*: custom operators defined in C++ without using the GXF API, by creating a subclass of `holoscan::Operator`. These C++ operators can pass arbitrary C++ shared objects around between operators.
2. *GXF Operators*: operators defined in the underlying C++ library by inheriting from the `holoscan::ops::GXFOperator` class. These operators wrap GXF codelets from GXF extensions. Examples are `VideoStreamReplayerOp` for replaying video files, `FormatConverterOp` for format conversions, and `HolovizOp` for visualization.

Note: It is possible to create an application using a mixture of GXF operators and native operators. In this case, some special consideration to cast the input and output tensors appropriately must be taken, as shown in *a section below*.

9.1.1 Native C++ Operators

Operator Lifecycle (C++)

The lifecycle of a `holoscan::Operator` is made up of three stages:

- `start()` is called once when the operator starts, and is used for initializing heavy tasks such as allocating memory resources and using parameters.
- `compute()` is called when the operator is triggered, which can occur any number of times throughout the operator lifecycle between `start()` and `stop()`.
- `stop()` is called once when the operator is stopped, and is used for deinitializing heavy tasks such as deallocating resources that were previously assigned in `start()`.

All operators on the workflow are scheduled for execution. When an operator is first executed, the `start()` method is called, followed by the `compute()` method. When the operator is stopped, the `stop()` method is called. The `compute()` method is called multiple times between `start()` and `stop()`.

If any of the scheduling conditions specified by *Conditions* are not met (for example, the `CountCondition` would cause the scheduling condition to not be met if the operator has been executed a certain number of times), the operator is stopped and the `stop()` method is called.

We will cover how to use Conditions in the *Specifying operator inputs and outputs (C++)* section of the user guide.

Typically, the `start()` and the `stop()` functions are only called once during the application's lifecycle. However, if the scheduling conditions are met again, the operator can be scheduled for execution, and the `start()` method will be called again.

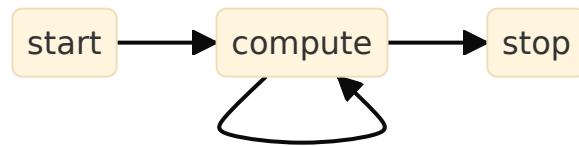


Fig. 9.1: The sequence of method calls in the lifecycle of a Holoscan Operator

We can override the default behavior of the operator by implementing the above methods. The following example shows how to implement a custom operator that overrides `start`, `stop` and `compute` methods.

Listing 9.1: The basic structure of a Holoscan Operator (C++)

```

1 #include "holoscan/holoscan.hpp"
2
3 using holoscan::Operator;
4 using holoscan::OperatorSpec;
5 using holoscan::InputContext;
6 using holoscan::OutputContext;
7 using holoscan::ExecutionContext;
8 using holoscan::Arg;
9 using holoscan::ArgList;
10
11 class MyOp : public Operator {
12 public:
13     HOLOSCAN_OPERATOR_FORWARD_ARGS(MyOp)
14
15     MyOp() = default;
16
17     void setup(OperatorSpec& spec) override {
18     }
19
20     void start() override {
21         HOLOSCAN_LOG_TRACE("MyOp::start()");
22     }
23
24     void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override {
25         HOLOSCAN_LOG_TRACE("MyOp::compute()");
26     };
27
28     void stop() override {
29         HOLOSCAN_LOG_TRACE("MyOp::stop()");
30     }
31 };
  
```

Creating a custom operator (C++)

To create a custom operator in C++ it is necessary to create a subclass of `holoscan::Operator`. The following example demonstrates how to use native operators (the operators that do not have an underlying, pre-compiled GXF Codelet).

Code Snippet: examples/native_operator/cpp/ping.cpp

Listing 9.2: examples/native_operator/cpp/ping.cpp

```

18 #include "holoscan/holoscan.hpp"
19
20 class ValueData {
21 public:
22     ValueData() = default;
23     explicit ValueData(int value) : data_(value) {
24         HOLOSCAN_LOG_TRACE("ValueData::ValueData(): {}", data_);
25     }
26     ~ValueData() {
27         HOLOSCAN_LOG_TRACE("ValueData::~ValueData(): {}", data_);
28     }
29
30     void data(int value) { data_ = value; }
31
32     int data() const { return data_; }
33
34 private:
35     int data_;
36 };
37
38 namespace holoscan::ops {
39
40 class PingTxOp : public Operator {
41 public:
42     HOLOSCAN_OPERATOR_FORWARD_ARGS(PingTxOp)
43
44     PingTxOp() = default;
45
46     void setup(OperatorSpec& spec) override {
47         spec.output<ValueData>("out1");
48         spec.output<ValueData>("out2");
49     }
50
51     void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override {
52         auto value1 = std::make_shared<ValueData>(index_++);
53         op_output.emit(value1, "out1");
54
55         auto value2 = std::make_shared<ValueData>(index_++);
56         op_output.emit(value2, "out2");
57     };
58     int index_ = 0;
59 };
60
61 class PingMiddleOp : public Operator {

```

(continues on next page)

(continued from previous page)

```

62 public:
63     HOLOSCAN_OPERATOR_FORWARD_ARGS(PingMiddleOp)
64
65     PingMiddleOp() = default;
66
67     void setup(OperatorSpec& spec) override {
68         spec.input<ValueData>("in1");
69         spec.input<ValueData>("in2");
70         spec.output<ValueData>("out1");
71         spec.output<ValueData>("out2");
72         spec.param(multiplier_, "multiplier", "Multiplier", "Multiply the input by this value"
73             , 2);
74     }
75
76     void compute(InputContext& op_input, OutputContext& op_output, ExecutionContext&)
77 override {
78         auto value1 = op_input.receive<ValueData>("in1");
79         auto value2 = op_input.receive<ValueData>("in2");
80
81         HOLOSCAN_LOG_INFO("Middle message received (count: {})", count_++);
82
83         HOLOSCAN_LOG_INFO("Middle message value1: {}", value1->data());
84         HOLOSCAN_LOG_INFO("Middle message value2: {}", value2->data());
85
86         // Multiply the values by the multiplier parameter
87         value1->data(value1->data() * multiplier_);
88         value2->data(value2->data() * multiplier_);
89
90         op_output.emit(value1, "out1");
91         op_output.emit(value2, "out2");
92     };
93
94     private:
95         int count_ = 1;
96         Parameter<int> multiplier_;
97     };
98
99     class PingRxOp : public Operator {
100     public:
101         HOLOSCAN_OPERATOR_FORWARD_ARGS(PingRxOp)
102
103         PingRxOp() = default;
104
105         void setup(OperatorSpec& spec) override {
106             spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.", {}
107             );
108         }
109
110         void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {
111             auto value_vector = op_input.receive<std::vector<ValueData>>("receivers");
112
113             HOLOSCAN_LOG_INFO("Rx message received (count: {}, size: {})", count_++, value_
114             vector.size());

```

(continues on next page)

(continued from previous page)

```

111
112     HOLOSCAN_LOG_INFO("Rx message value1: {}", value_vector[0]->data());
113     HOLOSCAN_LOG_INFO("Rx message value2: {}", value_vector[1]->data());
114 };
115
116 private:
117     Parameter<std::vector<IOSpec*>> receivers_;
118     int count_ = 1;
119 };
120
121 } // namespace holoscan::ops
122
123 class App : public holoscan::Application {
124     public:
125     void compose() override {
126         using namespace holoscan;
127
128         auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
129         auto mx = make_operator<ops::PingMiddleOp>("mx", from_config("mx"));
130         auto rx = make_operator<ops::PingRxOp>("rx");
131
132         add_flow(tx, mx, {{"out1", "in1"}, {"out2", "in2"}});
133         add_flow(mx, rx, {{"out1", "receivers"}, {"out2", "receivers"}});
134     }
135 };
136
137 int main(int argc, char** argv) {
138     holoscan::load_env_log_level();
139
140     auto app = holoscan::make_application<App>();
141
142     // Get the configuration
143     auto config_path = std::filesystem::canonical(argv[0]).parent_path();
144     config_path += "/app_config.yaml";
145     app->config(config_path);
146
147     app->run();
148
149     return 0;
150 }
```

Code Snippet: examples/native_operator/cpp/app_config.yaml

Listing 9.3: examples/native_operator/cpp/app_config.yaml

```

17 mx:
18   multiplier: 3

```

In this application, three operators are created: PingTxOp, PingMiddleOp, and PingRxOp

1. The PingTxOp operator is a source operator that emits two values every time it is invoked. The values are emitted on two different output ports, out1 (for even integers) and out2 (for odd integers).
2. The PingMiddleOp operator is a middle operator that receives two values from the PingTxOp operator and emits

two values on two different output ports. The values are multiplied by the `multiplier` parameter.

3. The `PingRxOp` operator is a sink operator that receives two values from the `PingMiddleOp` operator. The values are received on a single input, `receivers`, which is a vector of input ports. The `PingRxOp` operator receives the values in the order they are emitted by the `PingMiddleOp` operator.

As covered in more detail below, the inputs to each operator are specified in the `setup()` method of the operator. Then inputs are received within the `compute()` method via `op_input.receive()` and outputs are emitted via `op_output.emit()`.

Note that for native C++ operators as defined here, any shared pointer can be emitted or received. When transmitting between operators, a shared pointer to the object is transmitted rather than a copy. In some cases, such as sending the same tensor to more than one downstream operator, it may be necessary to avoid in-place operations on the tensor in order to avoid any potential race conditions between operators.

Specifying operator parameters (C++)

In the example `holoscan::ops::PingMiddleOp` operator above, we have a parameter `multiplier` that is declared as part of the class as a private member using the `param()` templated type:

```
Parameter<int> multiplier_;
```

It is then added to the `OperatorSpec` attribute of the operator in its `setup()` method, where an associated string key must be provided. Other properties can also be mentioned such as description and default value:

```
// Provide key, and optionally other information
spec.param(multiplier_, "multiplier", "Multiplier", "Multiply the input by this value", 2);
```

Note: If your parameter is of a custom type, you must register that type and provide a YAML encoder/decoder, as documented under [holoscan::Operator::register_converter](#)

See the [Configuring operator parameters](#) section to learn how an application can set these parameters.

Specifying operator inputs and outputs (C++)

To configure the input(s) and output(s) of C++ native operators, call the `spec.input()` and `spec.output()` methods within the `setup()` method of the operator.

The `spec.input()` and `spec.output()` methods should be called once for each input and output to be added. The `OperatorSpec` object and the `setup()` method will be initialized and called automatically by the `Application` class when its `run()` method is called.

These methods (`spec.input()` and `spec.output()`) return an `IOSpec` object that can be used to configure the input/output port.

By default, the `holoscan::MessageAvailableCondition` and `holoscan::DownstreamMessageAffordableCondition` conditions are applied (with a `min_size` of 1) to the input/output ports. This means that the operator's `compute()` method will not be invoked until a message is available on the input port and the downstream operator's input port (queue) has enough capacity to receive the message.

```
void setup(OperatorSpec& spec) override {
    spec.input<ValueData>("in");
    // Above statement is equivalent to:
```

(continues on next page)

(continued from previous page)

```
// spec.input<ValueData>("in")
//     .condition(ConditionType::kMessageAvailable, Arg("min_size") = 1);

spec.output<ValueData>("out");
// Above statement is equivalent to:
// spec.output<ValueData>("out")
//     .condition(ConditionType::kDownstreamMessageAffordable, Arg("min_size") =_
↪1);
...
}
```

In the above example, the `spec.input()` method is used to configure the input port to have the `holoscan::MessageAvailableCondition` with a minimum size of 1. This means that the operator's `compute()` method will not be invoked until a message is available on the input port of the operator. Similarly, the `spec.output()` method is used to configure the output port to have the `holoscan::DownstreamMessageAffordableCondition` with a minimum size of 1. This means that the operator's `compute()` method will not be invoked until the downstream operator's input port has enough capacity to receive the message.

If you want to change this behavior, use the `IOSpec::condition()` method to configure the conditions. For example, to configure the input and output ports to have no conditions, you can use the following code:

```
void setup(OperatorSpec& spec) override {
    spec.input<ValueData>("in")
        .condition(ConditionType::kNone);

    spec.output<ValueData>("out")
        .condition(ConditionType::kNone);
    // ...
}
```

The example code in the `setup()` method configures the input port to have no conditions, which means that the `compute()` method will be called as soon as the operator is ready to compute. Since there is no guarantee that the input port will have a message available, the `compute()` method should check if there is a message available on the input port before attempting to read it.

The `receive()` method of the `InputContext` object can be used to access different types of input data within the `compute()` method of your operator class, where its template argument (`DataT`) is the data type of the input. This method takes the name of the input port as an argument (which can be omitted if your operator has a single input port), and returns a shared pointer to the input data.

In the example code fragment below, the `PingRxOp` operator receives input on a port called "in" with data type `ValueData`. The `receive()` method is used to access the input data, and the `data()` method of the `ValueData` class is called to get the value of the input data.

```
// ...

class PingRxOp : public holoscan::ops::GXFOperator {
public:
    HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER(PingRxOp, holoscan::ops::GXFOperator)
    PingRxOp() = default;
    void setup(OperatorSpec& spec) override {
        spec.input<ValueData>("in");
    }
    void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {

```

(continues on next page)

(continued from previous page)

```
// The type of `value` is `std::shared_ptr<ValueData>`
auto value = op_input.receive<ValueData>("in");
if (value){
    HOLOSCAN_LOG_INFO("Message received (value: {})", value->data());
}
};
```

For GXF Entity objects (`holoscan::gxf::Entity` wraps underlying GXF `nvidia::gxf::Entity` class), the `receive()` method will return the GXF Entity object for the input of the specified name. In the example below, the `PingRxOp` operator receives input on a port called “in” with data type `holoscan::gxf::Entity`.

```
// ...

class PingRxOp : public holoscan::ops::GXFOperator {
public:
    HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER(PingRxOp, holoscan::ops::GXFOperator)
    PingRxOp() = default;
    void setup(OperatorSpec& spec) override {
        spec.input<holoscan::gxf::Entity>("in");
    }
    void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {
        // The type of `in_entity` is `holoscan::gxf::Entity`.
        auto in_entity = op_input.receive<holoscan::gxf::Entity>("in");
        if (in_entity) {
            // Process with `in_entity`.
            // ...
        }
    }
};
```

For objects of type `std::any`, the `receive()` method will return a `std::any` object containing the input of the specified name. In the example below, the `PingRxOp` operator receives input on a port called “in” with data type `std::any`. The `type()` method of the `std::any` object is used to determine the actual type of the input data, and the `std::any_cast<T>()` function is used to retrieve the value of the input data.

```
// ...

class PingRxOp : public holoscan::ops::GXFOperator {
public:
    HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER(PingRxOp, holoscan::ops::GXFOperator)
    PingRxOp() = default;
    void setup(OperatorSpec& spec) override {
        spec.input<std::any>("in");
    }
    void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {
        // The type of `in_any` is `std::any`.
        auto in_any = op_input.receive<std::any>("in");
        auto& in_any_type = in_any.type();

        if (in_any_type == typeid(holoscan::gxf::Entity)) {
            auto in_entity = std::any_cast<holoscan::gxf::Entity>(in_any);
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

// Process with `in_entity`.
// ...
} else if (in_any_type == typeid(std::shared_ptr<ValueData>)) {
    auto in_message = std::any_cast<std::shared_ptr<ValueData>>(in_any);
    // Process with `in_message`.
    // ...
} else if (in_any_type == typeid(nullptr_t)) {
    // No message is available.
} else {
    HOLOSCAN_LOG_ERROR("Invalid message type: {}", in_any_type.name());
    return;
}
}
};


```

The Holoscan SDK provides built-in data types called **Domain Objects**, defined in the `include/holoscan/core/domain` directory. For example, the `holoscan::Tensor` is a Domain Object class that is used to represent a multi-dimensional array of data, which can be used directly by `OperatorSpec`, `InputContext`, and `OutputContext`.

Tip: This `holoscan::Tensor` class is a wrapper around the `DLMangedTensorCtx` struct holding a `DLMangedTensor` object. As such, it provides a primary interface to access Tensor data and is interoperable with other frameworks that support the `DLPack` interface.

Warning: Passing `holoscan::Tensor` objects to/from *GXF operators* directly is not supported. Instead, they need to be passed through `holoscan::gxf::Entity` objects. See the *interoperability section* for more details.

Receiving any number of inputs (C++)

Instead of assigning a specific number of input ports, it may be desired to have the ability to receive any number of objects on a port in certain situations. This can be done by defining Parameter with `std::vector<IOSpec*>>` (`Parameter<std::vector<IOSpec*>> receivers_`) and calling `spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.", {});` as done for `PingRxOp` in the *native operator ping example*.

Listing 9.4: examples/native_operator/cpp/ping.cpp

```

97 class PingRxOp : public Operator {
98 public:
99     HOLOSCAN_OPERATOR_FORWARD_ARGS(PingRxOp)
100
101     PingRxOp() = default;
102
103     void setup(OperatorSpec& spec) override {
104         spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.", {});
105     }
106
107     void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override {
108         auto value_vector = op_input.receive<std::vector<ValueData>>("receivers");

```

(continues on next page)

(continued from previous page)

```

109     HOLOSCAN_LOG_INFO("Rx message received (count: {}, size: {})", count_++, value_
110     ↪vector.size());
111
112     HOLOSCAN_LOG_INFO("Rx message value1: {}", value_vector[0]->data());
113     HOLOSCAN_LOG_INFO("Rx message value2: {}", value_vector[1]->data());
114 };
115
116 private:
117     Parameter<std::vector<IOSpec*>> receivers_;
118     int count_ = 1;
119 };
120
121 } // namespace holoscan::ops
122
123 class App : public holoscan::Application {
124     public:
125         void compose() override {
126             using namespace holoscan;
127
128             auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10));
129             auto mx = make_operator<ops::PingMiddleOp>("mx", from_config("mx"));
130             auto rx = make_operator<ops::PingRxOp>("rx");
131
132             add_flow(tx, mx, {{"out1", "in1"}, {"out2", "in2"}});
133             add_flow(mx, rx, {{"out1", "receivers"}, {"out2", "receivers"}});
134         }
135     };

```

Then, once the following configuration is provided in the `compose()` method, the `PingRxOp` will receive two inputs on the `receivers` port.

```
133: add_flow(mx, rx, {{"out1", "receivers"}, {"out2", "receivers"}});
```

By using a parameter (`receivers`) with `std::vector<holoscan::IOSpec*>` type, the framework creates input ports (`receivers:0` and `receivers:1`) implicitly and connects them (and adds the references of the input ports to the `receivers` vector).

Building your C++ operator

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

Listing 9.5: <src_dir>/CMakeLists.txt

```

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

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

```

(continues on next page)

(continued from previous page)

```
# Create a library for your operator
add_library(my_operator SHARED my_operator.cpp)

# Link your operator against holoscan::core
target_link_libraries(my_operator
    PUBLIC holoscan::core
)
```

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

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

Using your C++ Operator in an Application

- If the application is configured in the same CMake project as the operator, you can simply add the operator CMake target library name under the application executable `target_link_libraries` call, as the operator CMake target is already defined.

```
# operator
add_library(my_op my_op.cpp)
target_link_libraries(my_operator PUBLIC holoscan::core)

# application
add_executable(my_app main.cpp)
target_link_libraries(my_operator
    PRIVATE
        holoscan::core
        my_op
)
```

- If the application is configured in a separate project as the operator, you need to export the operator in its own CMake project, and import it in the application CMake project, before being able to list it under `target_link_libraries` also. This is the same as what is done for the SDK *built-in operators*, available under the `holoscan::ops` namespace.

You can then include the headers to your C++ operator in your application code.

9.1.2 GXF Operators

With the Holoscan C++ API, we can also wrap *GXF Codelets* from GXF extensions as Holoscan Operators.

Note: If you do not have an existing GXF extension, we recommend developing native operators using the [C++](#) or [Python](#) APIs to skip the need for wrapping gxf codelets as operators. If you do need to create a GXF Extension, follow the [Creating a GXF Extension](#) section for a detailed explanation of the GXF extension development process.

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

The `MyRecorderOp` Holoscan Operator implementation below will wrap the `MyRecorder` GXF Codelet shown [here](#).

Operator definition

Listing 9.6: `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 */

```

The `holoscan::ops::MyRecorderOp` class wraps a `MyRecorder` GXF Codelet by inheriting from the `holoscan::ops::GXFOperator` class. The `HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER` macro is used to forward the arguments of the constructor to the base class.

We first need to define the fields of the `MyRecorderOp` class. You can see that fields with the same names are defined in both the `MyRecorderOp` class and the `MyRecorder` GXF codelet.

Listing 9.7: Parameter declarations in
gxf_extensions/my_recorder/my_recorder.hpp

```

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_;
nvidia::gxf::Parameter<bool> flush_on_tick_;

```

Comparing the `MyRecorderOp` holoscan parameter to the `MyRecorder` gxf codelet:

Holoscan Operator	GXF Codelet
<code>holoscan::Parameter</code>	<code>nvidia::gxf::Parameter</code>
<code>holoscan::IOSpec*</code>	<code>nvidia::gxf::Handle<nvidia::gxf::Receiver></code> or <code>nvidia::gxf::Handle<nvidia::gxf::Transmitter></code>
<code>std::shared_ptr<holoscan::Residue></code>	<code>gxf::Handle<T></code> example: <code>T</code> is <code>nvidia::gxf::EntitySerializer</code>

We then 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.

Setting up parameter specifications

The implementation of the `setup(OperatorSpec& spec)` function is as follows:

Listing 9.8: my_recorder_op.cpp

```

1 #include "./my_recorder_op.hpp"
2
3 #include "holoscan/core/fragment.hpp"
4 #include "holoscan/core/gxf/entity.hpp"
5 #include "holoscan/core/operator_spec.hpp"
6
7 #include "holoscan/core/resources/gxf/video_stream_serializer.hpp"
8
9 namespace holoscan::ops {
10
11 void MyRecorderOp::setup(OperatorSpec& spec) {
12     auto& input = spec.input<holoscan::gxf::Entity>("input");
13     // Above is same with the following two lines (a default condition is assigned to the
14     // input port if not specified):
15     //     auto& input = spec.input<holoscan::gxf::Entity>("input")

```

(continues on next page)

(continued from previous page)

```

16 // .condition(ConditionType::kMessageAvailable, Arg("min_size") =_
17 // 1);
18
19 spec.param(receiver_, "receiver", "Entity receiver", "Receiver channel to log", &
20 // input);
21 spec.param(my_serializer_,
22     "serializer",
23     "Entity serializer",
24     "Serializer for serializing input data");
25 spec.param(directory_, "out_directory", "Output directory path", "Directory path to_"
26 // store received output");
27 spec.param(basename_, "basename", "File base name", "User specified file name without_"
28 // extension");
29 spec.param(flush_on_tick_,
30     "flush_on_tick",
31     "Boolean to flush on tick",
32     "Flushes output buffer on every `tick` when true",
33     false);
34 }

32 void MyRecorderOp::initialize() {...}

34 } // namespace holoscan::ops

```

Here, we set up the inputs/outputs and parameters of the Operator. Note how the content of this function is very similar to the `MyRecorder` GXF codelet's `registerInterface` function.

- In the C++ API, GXF Receiver and Transmitter components (such as `DoubleBufferReceiver` and `DoubleBufferTransmitter`) are considered as input and output ports of the Operator so we register the inputs/outputs of the Operator with `input<T>` and `output<T>` functions (where T is the data type of the port).
- Compared to the pure *GXF application* that does the same job, the *SchedulingTerm* of an Entity in the *GXF Application YAML* are specified as Conditions on the input/output ports (e.g., `holoscan::MessageAvailableCondition` and `holoscan::DownstreamMessageAffordableCondition`).

The highlighted lines in `MyRecorderOp::setup` above match the following highlighted statements of *GXF Application YAML*:

Listing 9.9: A part of `apps/my_recorder_app_gxf/my_recorder_gxf.yaml`

```

35 name: recorder
36 components:
37 - name: input
38   type: nvidia::gxf::DoubleBufferReceiver
39 - name: allocator
40   type: nvidia::gxf::UnboundedAllocator
41 - name: component_serializer
42   type: nvidia::gxf::StdComponentSerializer
43   parameters:
44     allocator: allocator
45 - name: entity_serializer
46   type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from_
// nvidia::gxf::EntitySerializer

```

(continues on next page)

(continued from previous page)

```

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

```

Initializing the operator

Next, the implementation of the `initialize()` function is as follows:

Listing 9.10: my_recorder_op.cpp

```

1 #include "./my_recorder_op.hpp"
2
3 #include "holoscan/core/fragment.hpp"
4 #include "holoscan/core/gxf/entity.hpp"
5 #include "holoscan/core/operator_spec.hpp"
6
7 #include "holoscan/core/resources/gxf/video_stream_serializer.hpp"
8
9 namespace holoscan::ops {
10
11 void MyRecorderOp::setup(OperatorSpec& spec) {...}
12
13 void MyRecorderOp::initialize() {
14   // Set up prerequisite parameters before calling GXFOperator::initialize()
15   auto frag = fragment();
16   auto serializer =
17     frag->make_resource<holoscan::VideoStreamSerializer>("serializer");
18   add_arg(Arg("serializer") = serializer);
19
20   GXFOperator::initialize();
21 }

```

(continues on next page)

(continued from previous page)

```
22
23 } // namespace holoscan::ops
```

Here we set up the pre-defined parameters such as the `serializer`. The highlighted lines above matches the highlighted statements of [GXF Application YAML](#):

Listing 9.11: Another part of apps/my_recorder_app_gxf/my_recorder_gxf.yaml

```
35 name: recorder
36 components:
37   - name: input
38     type: nvidia::gxf::DoubleBufferReceiver
39   - name: allocator
40     type: nvidia::gxf::UnboundedAllocator
41   - name: component_serializer
42     type: nvidia::gxf::StdComponentSerializer
43     parameters:
44       allocator: allocator
45   - name: entity_serializer
46     type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
47     ↵nvidia::gxf::EntitySerializer
48     parameters:
49       component_serializers: [component_serializer]
50   - type: MyRecorder
51     parameters:
52       receiver: input
53       serializer: entity_serializer
54       out_directory: "/tmp"
55       basename: "tensor_out"
56   - type: nvidia::gxf::MessageAvailableSchedulingTerm
57     parameters:
58       receiver: input
       min_size: 1
```

Note: The Holoscan C++ API already provides the `holoscan::VideoStreamSerializer` class which wraps the `nvidia::holoscan::stream_playback::VideoStreamSerializer` GXF component, used here as `serializer`.

Building your GXF operator

There are no differences in CMake between building a GXF operator and [building a native C++ operator](#), since the GXF codelet is actually loaded through a GXF extension as a plugin, and does not need to be added to `target_link_libraries(my_operator ...)`.

Using your GXF Operator in an Application

There are no differences in CMake between using a GXF operator and [using a native C++ operator in an application](#). However, the application will need to load the GXF extension library which holds the wrapped GXF codelet symbols, so the application needs to be configured to find the extension library in its yaml configuration file, as documented [here](#).

9.1.3 Interoperability between GXF and native C++ operators

GXF passes `nvidia::gxf::Tensor` types between its codelets through a `nvidia::gxf::Entity` message. To support sending or receiving tensors to and from a GXF codelet (wrapped in a GXF operator) the Holoscan SDK provides the C++ classes below:

- `holoscan::gxf::GXFTensor`: inherits from `nvidia::gxf::Tensor`, and holds a `DLManagedTensorCtx` struct, making it interchangeable with the `holoscan::Tensor` class [mentioned above](#).
- `holoscan::gxf::Entity`: inherits from `nvidia::gxf::Entity`, handles the conversion from `holoscan::gxf::GXFTensor` to `holoscan::Tensor` under the hood.

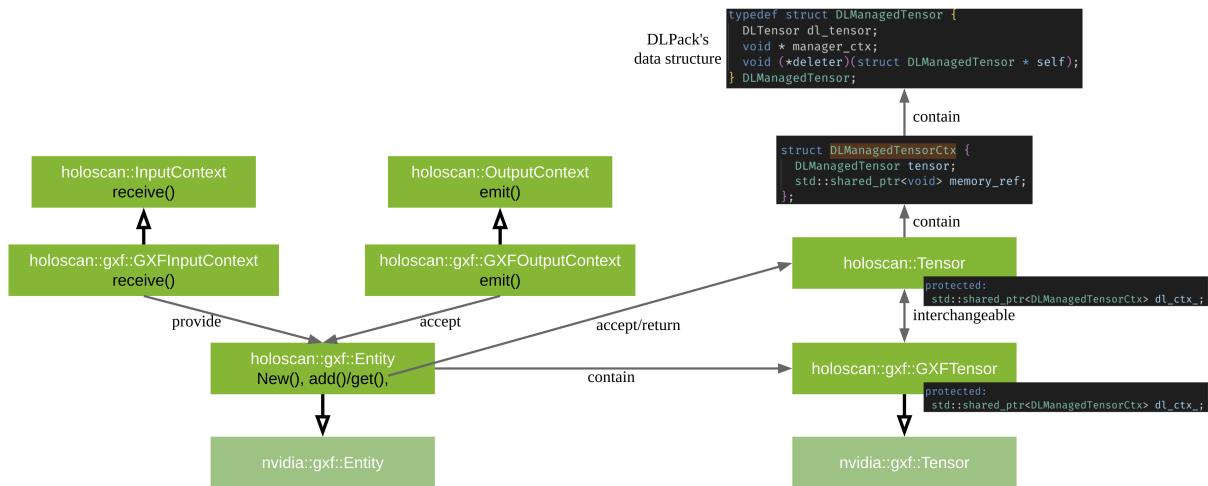


Fig. 9.2: Supporting Tensor Interoperability

Consider the following example, where `GXFSendTensorOp` and `GXFReceiveTensorOp` are GXF operators, and where `ProcessTensorOp` is a C++ native operator:

The following code shows how to implement `ProcessTensorOp`'s `compute()` method as a C++ native operator communicating with GXF operators. Focus on the use of the `holoscan::gxf::Entity`:

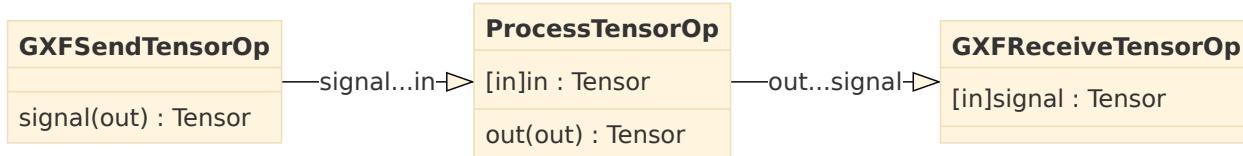


Fig. 9.3: The tensor interoperability between C++ native operator and GXF operator

Listing 9.12: examples/tensor_interop/cpp/tensor_interop.cpp

```

81 void compute(InputContext& op_input, OutputContext& op_output,
82               ExecutionContext& context) override {
83     // The type of `in_message` is `holoscan::gxf::Entity`.
84     auto in_message = op_input.receive<holoscan::gxf::Entity>("in");
85     // The type of `tensor` is `std::shared_ptr<holoscan::Tensor>`.
86     auto tensor = in_message.get<Tensor>();
87
88     // Process with `tensor` here.
89     cudaError_t cuda_status;
90
91     size_t data_size = tensor->nbytes();
92     std::vector<uint8_t> in_data(data_size);
93     CUDA_TRY(cudaMemcpy(in_data.data(), tensor->data(), data_size, ↴
94     cudaMemcpyDeviceToHost));
95
96     for (size_t i = 0; i < data_size; i++) { in_data[i] *= 2; }
97
98     CUDA_TRY(cudaMemcpy(tensor->data(), in_data.data(), data_size, ↴
99     cudaMemcpyHostToDevice));
100
101     // Create a new message (Entity)
102     auto out_message = holoscan::gxf::Entity::New(&context);
103     out_message.add(tensor, "tensor");
104
105     // Send the processed message.
106     op_output.emit(out_message);
107 };
  
```

- The `op_input.receive()` method call returns a `holoscan::gxf::Entity` object. That object has a `get()` method that returns the `holoscan::Tensor` object.
- The `holoscan::Tensor` object is copied on the host as `in_data`.
- The data is process (values multiplied by 2)
- The data is moved back to the `holoscan::Tensor` object on the GPU.
- A new `holoscan::gxf::Entity` object is created to be sent to the next operator with `op_output.emit()`. The `holoscan::Tensor` object is added to it using the `add()` method.

Note: A complete example of the C++ native operator that supports interoperability with GXF operators is available in the `examples/tensor_interop/cpp` directory.

You can add multiple tensors to a single `holoscan::gxf::Entity` object by calling the `add()` method multiple times

with a unique name for each tensor, as in the example below:

Operator sending a message:

```
auto out_message = holoscan::gxf::Entity::New(&context);
// Tensors and tensor names
out_message.add(output_tensor1, "video");
out_message.add(output_tensor2, "labels");
out_message.add(output_tensor3, "bbox_coords");
// Entity and port name
op_output.emit(out_message, "outputs");
```

Operator receiving the message, assuming the `outputs` port above is connected to the `inputs` port below with `add_flow()`:

```
// Entity and port name
auto in_message = op_input.receive<holoscan::gxf::Entity>("inputs");
// Tensors and tensor names
auto video = in_message.get<Tensor>("video");
auto labels = in_message.get<Tensor>("labels");
auto bbox_coords = in_message.get<Tensor>("bbox_coords");
```

Note: Some existing operators allow *configuring* the name of the tensors they send/receive. An example is the `tensors` parameter of `HolovizOp`, where the name for each tensor maps to the names of the tensors in the `Entity` (see the `holoviz` entry in `apps/endoscopy_tool_tracking/cpp/app_config.yaml`).

9.2 Python Operators

When assembling a Python application, two types of operators can be used:

1. **Native Python operators:** custom operators defined in Python, by creating a subclass of `holoscan.core.Operator`. These Python operators can pass arbitrary Python objects around between operators and are not restricted to the stricter parameter typing used for C++ API operators.
2. **Python wrappings of C++ Operators:** operators defined in the underlying C++ library by inheriting from the `holoscan::Operator` class. These operators have Python bindings available within the `holoscan.operators` module. Examples are `VideoStreamReplayerOp` for replaying video files, `FormatConverterOp` for format conversions, and `HolovizOp` for visualization.

Note: It is possible to create an application using a mixture of Python wrapped C++ operators and native Python operators. In this case, some special consideration to cast the input and output tensors appropriately must be taken, as shown in *a section below*.

9.2.1 Native Python Operator

Operator Lifecycle (Python)

The lifecycle of a `holoscan.core.Operator` is made up of three stages:

- `start()` is called once when the operator starts, and is used for initializing heavy tasks such as allocating memory resources and using parameters.
- `compute()` is called when the operator is triggered, which can occur any number of times throughout the operator lifecycle between `start()` and `stop()`.
- `stop()` is called once when the operator is stopped, and is used for deinitializing heavy tasks such as deallocating resources that were previously assigned in `start()`.

All operators on the workflow are scheduled for execution. When an operator is first executed, the `start()` method is called, followed by the `compute()` method. When the operator is stopped, the `stop()` method is called. The `compute()` method is called multiple times between `start()` and `stop()`.

If any of the scheduling conditions specified by [Conditions](#) are not met (for example, the `CountCondition` would cause the scheduling condition to not be met if the operator has been executed a certain number of times), the operator is stopped and the `stop()` method is called.

We will cover how to use [Conditions](#) in the [Specifying operator inputs and outputs \(Python\)](#) section of the user guide.

Typically, the `start()` and the `stop()` functions are only called once during the application's lifecycle. However, if the scheduling conditions are met again, the operator can be scheduled for execution, and the `start()` method will be called again.



Fig. 9.4: The sequence of method calls in the lifecycle of a Holoscan Operator

We can override the default behavior of the operator by implementing the above methods. The following example shows how to implement a custom operator that overrides `start`, `stop` and `compute` methods.

Listing 9.13: The basic structure of a Holoscan Operator (Python)

```

1  from holoscan.core import (
2      ExecutionContext,
3      InputContext,
4      Operator,
5      OperatorSpec,
6      OutputContext,
7  )
8
9
10 class MyOp(Operator):
11
12     def __init__(self, fragment, *args, **kwargs):
13         super().__init__(fragment, *args, **kwargs)
14
15     def setup(self, spec: OperatorSpec):
  
```

(continues on next page)

(continued from previous page)

```

16     pass
17
18     def start(self):
19         pass
20
21     def compute(self, op_input: InputContext, op_output: OutputContext, context: ExecutionContext):
22         pass
23
24     def stop(self):
25         pass

```

Creating a custom operator (Python)

To create a custom operator in Python it is necessary to create a subclass of `holoscan.core.Operator`. A simple example of an operator that takes a time-varying 1D input array named “signal” and applies convolution with a boxcar (i.e. rect) kernel.

For simplicity, this operator assumes that the “signal” that will be received on the input is already a `numpy.ndarray` or is something that can be cast to one via (`np.asarray`). We will see more details in a later section on how we can interoperate with various tensor classes, including the GXF Tensor objects used by some of the C++-based operators.

Code Snippet: `examples/numpy_native/convolve.py`

Listing 9.14: `examples/numpy_native/convolve.py`

```

16 import os
17
18 from holoscan.conditions import CountCondition
19 from holoscan.core import Application, Operator, OperatorSpec
20 from holoscan.logger import LogLevel, set_log_level
21
22 import numpy as np
23
24
25 class SignalGeneratorOp(Operator):
26     """Generate a time-varying impulse.
27
28     Transmits an array of zeros with a single non-zero entry of a
29     specified `height`. The position of the non-zero entry shifts
30     to the right (in a periodic fashion) each time `compute` is
31     called.
32
33     Parameters
34     -----
35     fragment : holoscan.core.Fragment
36         The Fragment (or Application) the operator belongs to.
37     height : number
38         The height of the signal impulse.
39     size : number
40         The total number of samples in the generated 1d signal.
41     dtype : numpy.dtype or str

```

(continues on next page)

(continued from previous page)

```

42     The data type of the generated signal.
43     """
44
45     def __init__(self, fragment, *args, height=1, size=10, dtype=np.int32, **kwargs):
46         self.count = 0
47         self.height = height
48         self.dtype = dtype
49         self.size = size
50         super().__init__(fragment, *args, **kwargs)
51
52     def setup(self, spec: OperatorSpec):
53         spec.output("signal")
54
55     def compute(self, op_input, op_output, context):
56
57         # single sample wide impulse at a time-varying position
58         signal = np.zeros((self.size,), dtype=self.dtype)
59         signal[self.count % signal.size] = self.height
60         self.count += 1
61
62         op_output.emit(signal, "signal")
63
64
65 class ConvolveOp(Operator):
66     """Apply convolution to a tensor.
67
68     Convolves an input signal with a "boxcar" (i.e. "rect") kernel.
69
70     Parameters
71     -----
72     fragment : holoscan.core.Fragment
73         The Fragment (or Application) the operator belongs to.
74     width : number
75         The width of the boxcar kernel used in the convolution.
76     unit_area : bool, optional
77         Whether or not to normalize the convolution kernel to unit area.
78         If False, all samples have implitude one and the dtype of the
79         kernel will match that of the signal. When True the sum over
80         the kernel is one and a 32-bit floating point data type is used
81         for the kernel.
82     """
83
84     def __init__(self, fragment, *args, width=4, unit_area=False, **kwargs):
85         self.count = 0
86         self.width = width
87         self.unit_area = unit_area
88         super().__init__(fragment, *args, **kwargs)
89
90     def setup(self, spec: OperatorSpec):
91         spec.input("signal_in")
92         spec.output("signal_out")
93

```

(continues on next page)

(continued from previous page)

```

94     def compute(self, op_input, op_output, context):
95
96         signal = op_input.receive("signal_in")
97         assert isinstance(signal, np.ndarray)
98
99         if self.unit_area:
100             kernel = np.full((self.width,), 1/self.width, dtype=np.float32)
101         else:
102             kernel = np.ones((self.width,), dtype=signal.dtype)
103
104         convolved = np.convolve(signal, kernel, mode='same')
105
106         op_output.emit(convolved, "signal_out")
107
108
109 class PrintSignalOp(Operator):
110     """Print the received signal to the terminal."""
111
112     def setup(self, spec: OperatorSpec):
113         spec.input("signal")
114
115     def compute(self, op_input, op_output, context):
116         signal = op_input.receive("signal")
117         print(signal)
118
119
120 class ConvolveApp(Application):
121     """Minimal signal processing application.
122
123     Generates a time-varying impulse, convolves it with a boxcar kernel, and
124     prints the result to the terminal.
125
126     A `CountCondition` is applied to the generate to terminate execution
127     after a specific number of steps.
128     """
129
130     def compose(self):
131         signal_generator = SignalGeneratorOp(
132             self,
133             CountCondition(self, count=24),
134             name="generator",
135             **self.kwargs("generator"),
136         )
137         convolver = ConvolveOp(self, name="conv", **self.kwargs("convolve"))
138         printer = PrintSignalOp(self, name="printer")
139         self.add_flow(signal_generator, convolver)
140         self.add_flow(convolver, printer)
141
142
143 if __name__ == "__main__":
144     set_log_level(LogLevel.WARN)

```

(continues on next page)

(continued from previous page)

```

146     app = ConvolveApp()
147     config_file = os.path.join(os.path.dirname(__file__), 'convolve.yaml')
148     app.config(config_file)
149     app.run()

```

Code Snippet: examples/numpy_native/convolve.yaml

Listing 9.15: examples/numpy_native/convolve.yaml

```

17 signal_generator:
18   height: 1
19   size: 20
20   dtype: int32
21
22 convolve:
23   width: 4
24   unit_area: false

```

In this application, three native Python operators are created: `SignalGeneratorOp`, `ConvolveOp` and `PrintSignalOp`. The `SignalGeneratorOp` generates a synthetic signal such as `[0, 0, 1, 0, 0, 0]` where the position of the non-zero entry varies each time it is called. `ConvolveOp` performs a 1D convolution with a boxcar (i.e. rect) function of a specified width. `PrintSignalOp` just prints the received signal to the terminal.

As covered in more detail below, the inputs to each operator are specified in the `setup()` method of the operator. Then inputs are received within the `compute` method via `op_input.receive()` and outputs are emitted via `op_output.emit()`.

Note that for native Python operators as defined here, any Python object can be emitted or received. When transmitting between operators, a shared pointer to the object is transmitted rather than a copy. In some cases, such as sending the same tensor to more than one downstream operator, it may be necessary to avoid in-place operations on the tensor in order to avoid any potential race conditions between operators.

Specifying operator parameters (Python)

In the example `SignalGeneratorOp` operator above, we added three keyword arguments in the operator's `__init__` method, used inside the `compose()` method of the operator to adjust its behavior:

```

def __init__(self, fragment, *args, width=4, unit_area=False, **kwargs):
    # Internal counter for the time-dependent signal generation
    self.count = 0

    # Parameters
    self.width = width
    self.unit_area = unit_area

    # To forward remaining arguments to any underlying C++ Operator class
    super().__init__(fragment, *args, **kwargs)

```

Note: As an alternative closer to C++, these parameters can be added through the `OperatorSpec` attribute of the operator in its `setup()` method, where an associated string key must be provided as well as a default value:

```
def setup(self, spec: OperatorSpec):
    spec.param("width", 4)
    spec.param("unit_area", False)
```

Other `kwarg`s properties can also be passed to `spec.param` such as `headline`, `description` (used by GXF applications), or `kind` (used when [Receiving any number of inputs \(Python\)](#)).

See the [Configuring operator parameters](#) section to learn how an application can set these parameters.

Specifying operator inputs and outputs (Python)

To configure the input(s) and output(s) of Python native operators, call the `spec.input()` and `spec.output()` methods within the `setup()` method of the operator.

The `spec.input()` and `spec.output()` methods should be called once for each input and output to be added. The `holoscan.core.OperatorSpec` object and the `setup()` method will be initialized and called automatically by the Application class when its `run()` method is called.

These methods (`spec.input()` and `spec.output()`) return an `IOSpec` object that can be used to configure the input/output port.

By default, the `holoscan.conditions.MessageAvailableCondition` and `holoscan.conditions.DownstreamMessageAffordableCondition` conditions are applied (with a `min_size` of 1) to the input/output ports. This means that the operator's `compute()` method will not be invoked until a message is available on the input port and the downstream operator's input port (queue) has enough capacity to receive the message.

```
def setup(self, spec: OperatorSpec):
    spec.input("in")
    # Above statement is equivalent to:
    #   spec.input("in")
    #     .condition(ConditionType.MESSAGE_AVAILABLE, min_size = 1)
    spec.output("out")
    # Above statement is equivalent to:
    #   spec.output("out")
    #     .condition(ConditionType.DOWNSTREAM_MESSAGE_AFFORDABLE, min_size = 1)
```

In the above example, the `spec.input()` method is used to configure the input port to have the `holoscan.conditions.MessageAvailableCondition` with a minimum size of 1. This means that the operator's `compute()` method will not be invoked until a message is available on the input port of the operator. Similarly, the `spec.output()` method is used to configure the output port to have a `holoscan.conditions.DownstreamMessageAffordableCondition` with a minimum size of 1. This means that the operator's `compute()` method will not be invoked until the downstream operator's input port has enough capacity to receive the message.

If you want to change this behavior, use the `IOSpec.condition()` method to configure the conditions. For example, to configure the input and output ports to have no conditions, you can use the following code:

```
from holoscan.core import ConditionType, OperatorSpec
#
# ...
def setup(self, spec: OperatorSpec):
    spec.input("in").condition(ConditionType.NONE)
    spec.output("out").condition(ConditionType.NONE)
```

The example code in the `setup()` method configures the input port to have no conditions, which means that the `compute()` method will be called as soon as the operator is ready to compute. Since there is no guarantee that the

input port will have a message available, the `compute()` method should check if there is a message available on the input port before attempting to read it.

The `receive()` method of the `InputContext` object can be used to access different types of input data within the `compute()` method of your operator class. This method takes the name of the input port as an argument (which can be omitted if your operator has a single input port).

For standard Python objects, `receive()` will directly return the Python object for input of the specified name.

The Holoscan SDK also provides built-in data types called **Domain Objects**, defined in the `include/holoscan/core/domain` directory. For example, the `Tensor` is a Domain Object class that is used to represent a multi-dimensional array of data, which can be used directly by `OperatorSpec`, `InputContext`, and `OutputContext`.

Tip: This `holoscan.core.Tensor` class supports both `DLPack` and NumPy's array interface (`__array_interface__` and `__cuda_array_interface__`) so that it can be used with other Python libraries such as CuPy, PyTorch, JAX, TensorFlow, and Numba.

Warning: Passing `holoscan.core.Tensor` objects to/from *Python wrapped C++ operators* (both C++ native and GXF-based) directly is not yet supported. At this time, they need to be passed through `holoscan.gxf.Entity` objects. See the [interoperability section](#) for more details. This won't be necessary in the future for native C++ operators.

In both cases, it will return `None` if there is no message available on the input port:

```
# ...
def compute(self, op_input, op_output, context):
    msg = op_input.receive("in")
    if msg:
        # Do something with msg
```

Receiving any number of inputs (Python)

Instead of assigning a specific number of input ports, it may be desired to have the ability to receive any number of objects on a port in certain situations. This can be done by calling `spec.param(port_name, kind='receivers')` as done for `PingRxOp` in the native operator ping example located at `examples/native_operator/python/ping.py`:

Code Snippet: examples/native_operator/python/ping.py

Listing 9.16: examples/native_operator/python/ping.py

```
124 class PingRxOp(Operator):
125     """Simple receiver operator.
126
127     This operator has:
128         input: "receivers"
129
130     This is an example of a native operator that can dynamically have any
131     number of inputs connected to its "receivers" port.
132     """
133
134     def __init__(self, fragment, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```

135     self.count = 1
136     # Need to call the base class constructor last
137     super().__init__(fragment, *args, **kwargs)
138
139     def setup(self, spec: OperatorSpec):
140         spec.param("receivers", kind="receivers")
141
142     def compute(self, op_input, op_output, context):
143         values = op_input.receive("receivers")
144         print(f"Rx message received (count: {self.count}, size: {len(values)})")
145         self.count += 1
146         print(f"Rx message value1: {values[0].data}")
147         print(f"Rx message value2: {values[1].data}")

```

and in the `compose` method of the application, two parameters are connected to this “receivers” port:

```
self.add_flow(mx, rx, {"out1", "receivers"), ("out2", "receivers")})
```

This line connects both the `out1` and `out2` ports of operator `mx` to the `receivers` port of operator `rx`.

Here, `values` as returned by `op_input.receive("receivers")` will be a tuple of python objects.

9.2.2 Python wrapping of a C++ operator

For convenience while maintaining highest performance, *operators written in C++* can be wrapped in Python. In the Holoscan SDK, we’ve used pybind11 to wrap all the built-in operators in `python/src/operators`. We’ll highlight the main components below:

1. Create a subclass in C++ that inherits the C++ Operator class to wrap, to define a new constructor which takes a `Fragment`, an explicit list of parameters with potential default values (`argA`, `argB` below...), and an operator name, in order to then fully initialize the operator like is done in `Fragment::make_operator`:

```

#include <holoscan/core/fragment.hpp>
#include <holoscan/core/operator.hpp>
#include <holoscan/core/operator_spec.hpp>

#include "my_op.hpp"

class PyMyOp : public MyOp {
public:
    using MyOp::MyOp;

    PyMyOp(
        Fragment* fragment,
        TypeA argA, TypeB argB = 0, ...,
        const std::string& name = "my_op"
    ) : MyOp(ArgList{
            Arg{"argA", argA},
            Arg{"argB", argB},
            ...
        })
        # If you have arguments you can't pass directly to the `MyOp` constructor as
        # an `Arg`,

```

(continues on next page)

(continued from previous page)

```
# do the conversion and call `this->add_arg` before setting up the spec below.
name_ = name;
fragment_ = fragment;
spec_ = std::make_shared<OperatorSpec>(fragment);
setup(*spec_.get());
initialize();
}
```

2. Prepare documentation for your python class. Below we use a PYDOC macro defined in the SDK [here](#). See note *below* for HoloHub.

```
#include "../macros.hpp"

namespace doc::MyOp {

PYDOC(cls, R"doc(
My operator.
)doc")

PYDOC(constructor, R"doc(
Create the operator.

Parameters
-----
fragment : holoscan.core.Fragment
    The fragment that the operator belongs to.
argA : TypeA
    argA description
argB : TypeB, optional
    argB description
name : str, optional
    The name of the operator.
)doc")

PYDOC(initialize, R"doc(
Initialize the operator.

This method is called only once when the operator is created for the first time,
and uses a light-weight initialization.
)doc")

PYDOC(setup, R"doc(
Define the operator specification.

Parameters
-----
spec : holoscan.core.OperatorSpec
    The operator specification.
)doc")

}
```

3. Call `py::class_` within `PYBIND11_MODULE` to define your operator python class:

```
#include <pybind11/pybind11.h>

using pybind11::literals::operator""_a;

#define STRINGIFY(x) #x
#define MACRO_STRINGIFY(x) STRINGIFY(x)

namespace py = pybind11;

// The name used as the first argument to the PYBIND11_MODULE macro here
// must match the name passed to the pybind11_add_module CMake function
PYBIND11_MODULE(_my_python_module, m) {
    m.doc() = R"pbdoc(
        Holoscan Python Bindings
        -----
        .. currentmodule:: _my_python_module
        .. autosummary::
            :toctree: _generate
            add
            subtract
    )pbdoc";

#ifdef VERSION_INFO
    m.attr("__version__") = MACRO_STRINGIFY(VERSION_INFO);
#else
    m.attr("__version__") = "dev";
#endif

py::class_<MyOp, PyMyOp, Operator, std::shared_ptr<MyOp>>(
    m, "MyOp", doc::MyOp::doc_cls)
.def(py::init<Fragment*, TypeA, TypeB, ...,
        const std::string&>(),
    "fragment"_a,
    "argA"_a,
    "argB"_a = 0,
    ...,
    "name"_a = "my_op",
    doc::MyOp::doc_constructor)
.def("initialize",
    &MyOp::initialize,
    doc::MyOp::doc_initialize)
.def("setup",
    &MyOp::setup,
    "spec"_a,
    doc::MyOp::doc_setup);
}
```

4. In CMake, use the `pybind11_add_module` macro ([official doc](#)) with the cpp files containing the code above, and link against `holoscan::core` and the library that exposes your C++ operator to wrap. In the SDK, this is done [here](#). See note *below* for HoloHub. For a simple standalone project/operator, it could look like this:

```
pybind11_add_module(my_python_module my_op_pybind.cpp)
target_link_libraries(my_python_module
```

(continues on next page)

(continued from previous page)

```
PRIVATE holoscan::core
PUBLIC my_op
)
```

5. The c++ module will need to be loaded in Python to expose the python class. This can be done with an `__init__.py` file like below. It uses `from .` assuming the python file and the generated `my_python_module` c++ library are in the same folder.

```
import holoscan.core

from ._my_python_module import MyOp
```

Note: We've added utilities to facilitate steps 2, 4 and 5 within HoloHub, using the `pybind11_add_holohub_module` CMake utility. An example of its use can be found [here](#).

9.2.3 Interoperability between wrapped and native Python operators

As described in the *Interoperability between GXF and native C++ operators* section, `holoscan::Tensor` objects can only be passed to GXF operators using a `holoscan::gxf::Entity` message that holds the tensor(s). In Python, this is done with the wrapped methods, `holoscan.core.Tensor` and `holoscan.gxf.Entity`.

Warning: At this time, using `holoscan.gxf.Entity` is required when communicating with **any** Python wrapped C++ operator. That includes native C++ operators and GXF operators. This will be addressed in future versions to only require a `holoscan.gxf.Entity` for Python wrapped GXF operators.

Consider the following example, where `VideoStreamReplayerOp` and `HolovizOp` are Python wrapped C++ operators, and where `ImageProcessingOp` is a Python native operator:



Fig. 9.5: The tensor interoperability between Python native operator and C++-based Python GXF operator

The following code shows how to implement `ImageProcessingOp`'s `compute()` method as a Python native operator communicating with C++ operators:

Listing 9.17: examples/tensor_interop/python/tensor_interop.py

```
81 def compute(self, op_input, op_output, context):
82     message = op_input.receive("input_tensor")
83
84     input_tensor = message.get()
85
86     print(f"message received (count: {self.count})")
87     self.count += 1
88
89     cp_array = cp.asarray(input_tensor)
```

(continues on next page)

(continued from previous page)

```

90
91     # smooth along first two axes, but not the color channels
92     sigma = (self.sigma, self.sigma, 0)
93
94     # process cp_array
95     cp_array = ndi.gaussian_filter(cp_array, sigma)
96
97     out_message = Entity(context)
98     output_tensor = hs.as_tensor(cp_array)
99
100    out_message.add(output_tensor)
101    op_output.emit(out_message, "output_tensor")

```

- The `op_input.receive()` method call returns a `holoscan.gxf.Entity` object. That object has a `get()` method that returns a `holoscan.core.Tensor` object.
- The `holoscan.core.Tensor` object is converted to a CuPy array by using `cupy.asarray()` method call.
- The CuPy array is used as an input to the `ndi.gaussian_filter()` function call with a parameter `sigma`. The result of the `ndi.gaussian_filter()` function call is a CuPy array.
- The CuPy array is converted to a `holoscan.core.Tensor` object by using `holoscan.as_tensor()` function call.
- Finally, a new `holoscan.gxf.Entity` object is created to be sent to the next operator with `op_output.emit()`. The `holoscan.core.Tensor` object is added to it using the `add()` method.

Note: A complete example of the Python native operator that supports interoperability with Python wrapped C++ operators is available in the `examples/tensor_interop/python` directory.

You can add multiple tensors to a single `holoscan.gxf.Entity` object by calling the `add()` method multiple times with a unique name for each tensor, as in the example below:

Operator sending a message:

```

out_message = Entity(context)
# Tensors and tensor names
out_message.add(output_tensor1, "video")
out_message.add(output_tensor2, "labels")
out_message.add(output_tensor3, "bbox_coords")
# Entity and port name
op_output.emit(out_message, "outputs")

```

Operator receiving the message, assuming the `outputs` port above is connected to the `inputs` port below with `add_flow()`:

```

# Entity and port name
in_message = op_input.receive("inputs")
# Tensors and tensor names
video = in_message.get("video")
labels = in_message.get("labels")
bbox_coords = in_message.get("bbox_coords")

```

Note: Some existing operators allow *configuring* the name of the tensors they send/receive. An example is the

tensors parameter of HolovizOp, where the name for each tensor maps to the names of the tensors in the Entity (see the `holoviz` entry in `apps/endoscopy_tool_tracking/python/endoscopy_tool_tracking.yaml`).

A complete example of a Python native operator that emits multiple tensors to a downstream C++ operator is available in the `examples/holoviz/python` directory.

BUILT-IN OPERATORS AND EXTENSIONS

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

10.1 Operators

The operators below are defined under the `holoscan::ops` namespace for C++ and CMake, and under the `holoscan.operators` module in Python.

Class	CMake target/lib	Documentation
<code>AJASourceOp</code>	<code>aja</code>	C++/Python
<code>BayerDemosaicOp</code>	<code>bayer_demosaic</code>	C++/Python
<code>FormatConverterOp</code>	<code>format_converter</code>	C++/Python
<code>HolovizOp</code>	<code>holoviz</code>	C++/Python
<code>MultiAIIInferenceOp</code>	<code>multiai_inference</code>	C++/Python
<code>MultiAIPostprocessorOp</code>	<code>multiai_postprocessor</code>	C++/Python
<code>PingRxOp</code>	<code>ping_rx</code>	C++/Python
<code>PingTxOp</code>	<code>ping_tx</code>	C++/Python
<code>SegmentationPostprocessorOp</code>	<code>segmentation_postprocessor</code>	C++/Python
<code>TensorRTInferenceOp</code> [†]	<code>tensor_rt</code>	C++/Python
<code>VideoStreamRecorderOp</code>	<code>video_stream_recorder</code>	C++/Python
<code>VideoStreamReplayerOp</code>	<code>video_stream_replayer</code>	C++/Python

[†] *deprecated*

Given an instance of an operator class, you can print a human-readable description of its specification to inspect the inputs, outputs, and parameters that can be configured on that operator class:

C++

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

Python

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

Note: The Holoscan SDK uses meta-programming with templating and `std::any` to support arbitrary data types. Because of this, some type information (and therefore values) might not be retrievable by the `description` API. If more details are needed, we recommend inspecting the list of `Parameter` members in the operator `header` to identify their type.

10.2 Extensions

The Holoscan SDK also includes some GXF extensions with GXF codelets, which are typically wrapped as operators, or present for legacy reasons. In addition to the core GXF extensions (std, cuda, serialization, multimedia) listed [here](#), the Holoscan SDK includes the following GXF extensions:

- `bayer_demosaic`
- `gxf_holoscan_wrapper`
- `opengl`
- `stream_playback`
- `tensor_rt`
- `v4l2`

10.2.1 Bayer Demosaic

The `bayer_demosaic` extension includes the `nvidia::holoscan::BayerDemosaic` codelet. It performs color filter array (CFA) interpolation for 1-channel inputs of 8 or 16-bit unsigned integer and outputs an RGB or RGBA image. It is wrapped by the `nvidia::holoscan::ops::BayerDemosaicOp` operator.

Note: The `BayerDemosaicOp` will be converted to a native operator in future releases.

10.2.2 GXF Holoscan Wrapper

The `gxf_holoscan_wrapper` extension includes the `holoscan::gxf::OperatorWrapper` codelet. It is used as a utility base class to wrap a holoscan operator to interface with the GXF framework.

Learn more about it in the [Using Holoscan Operators in GXF Applications](#) section.

10.2.3 OpenGL

The `opengl_renderer` extension includes the `nvidia::holoscan::OpenGLRenderer` codelet. It displays a `VideoBuffer`, leveraging OpenGL/CUDA interop.

Note: There is no operator currently wrapping this codelet. It is only use to demonstrate the V4L2 example.

Warning: This codelet is deprecated, and will be removed in a future release in favor of a native operator using the [visualization module](#).

Parameter	Description	Type
<code>signal</code>	Input Channel	<code>gxf::Handle<gxf::Receiver></code>
<code>width</code>	Width of the rendering window	<code>unsigned int</code>
<code>height</code>	Height of the rendering window	<code>unsigned int</code>
<code>win-dow_close_scheduling_term</code>	BooleanSchedulingTerm to stop the codelet from ticks after all messages are published	<code>gxf::Handle<gxf::BooleanSchedulingTerm></code>

10.2.4 Stream Playback

The `stream_playback` extension includes the `nvidia::holoscan::stream_playback::VideoStreamSerializer` entity serializer to/from a Tensor Object. This extension does not include any codelets: reading and writing video stream (gxf entity files) from the disk was implemented as native operators with `VideoStreamRecorderOp` and `VideoStreamReplayerOp`, though they leverage the `VideoStreamSerializer` from this extension.

Note: 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`.)

10.2.5 TensorRT

The `tensor_rt` extension includes the `nvidia::holoscan::TensorRtInference` codelet. It takes input tensors and feeds them into TensorRT for inference. It is wrapped by the `nvidia::holoscan::ops::TensorRTInferenceOp` operator.

Note: This codelet is based on `nvidia::gxf::TensorRtInference` (by GXF), 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.

Warning: This codelet is deprecated, and will be removed in Holoscan 0.6 in favor of a native operator using the *inference module*.

10.2.6 V4L2

The v4l2_source extension includes the `nvidia::holoscan::V4L2Source` codelet. It uses V4L2 to get image frames from USB cameras. The output is a VideoBuffer object.

Note: There is no operator currently wrapping this codelet. A native operator also supporting HDMI IN will replace this codelet in future releases,

Parameter	Description	Type	Default
signal	Output channel	<code>gxf::Handle<gxf::Transmitter></code>	
allocator	Output Allocator	<code>gxf::Handle<gxf::Allocator></code>	
device	Path to the V4L2 device	<code>std::string</code>	<code>/dev/video0</code>
width	Width of the V4L2 image	<code>uint32_t</code>	640
height	Height of the V4L2 image	<code>uint32_t</code>	480
numBuffers	Number of V4L2 buffers to use	<code>uint32_t</code>	2

10.2.7 HoloHub

Visit the HoloHub repository to find a collection of additional Holoscan operators and extensions.

LOGGING

11.1 Defining the log level

The `load_env_log_level()` (C++/Python) function loads the logging level from the environment variable `HOLOSCAN_LOG_LEVEL` (default: `INFO`).

CPP

```
1 #include <holoscan/ holoscan.hpp>
2
3 int main() {
4     holoscan::load_env_log_level();
5     // ...
6     return 0;
7 }
```

PYTHON

```
1 from holoscan.logger import load_env_log_level
2
3 def main():
4     load_env_log_level()
5     # ...
6
7 if __name__ == "__main__":
8     main()
```

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

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

```
export HOLOSCAN_LOG_LEVEL=TRACE
```

11.2 Calling the logger

The **C++ API** uses the HOLOSCAN_LOG_XXX() macros to log messages in the application. These macros use the [fmtlib format string syntax](#) for their format strings.

Users of the **Python API** should use the built-in [logging module](#) to log messages. Users can control the logging of any underlying C++ Operators used via their Python bindings via the functions in the `holoscan.logger` module. Specifically, calling `load_env_log_level()` will load the logging level from the `HOLOSCAN_LOG_LEVEL` environment variable described above. This log level can also be changed at runtime via `set_log_level()` using the `LogLevel` enum class (e.g. `set_log_level(LogLevel.WARN)`).

VISUALIZATION MODULE

12.1 Overview

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

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

12.2 Concepts

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 fast and easy to make visualization changes in a Holoscan application.

12.3 Usage

The code below creates a window and displays an image.

First Holoviz needs to be initialized. This is done by calling `viz::Init()`.

The elements to display are defined in the render loop, termination of the loop is checked with `viz::WindowShouldClose()`.

The definition of the displayed content starts with `viz::Begin()` and ends with `viz::End()`. `viz::End()` starts the rendering and displays the rendered result.

Finally Holoviz is shutdown with `viz::Shutdown()`.

```
#include "holoviz/holoviz.hpp"

namespace viz = holoscan::viz;

viz::Init("Holoviz Example");

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

(continues on next page)

(continued from previous page)

```
}
```

```
viz::Shutdown();
```

Result:

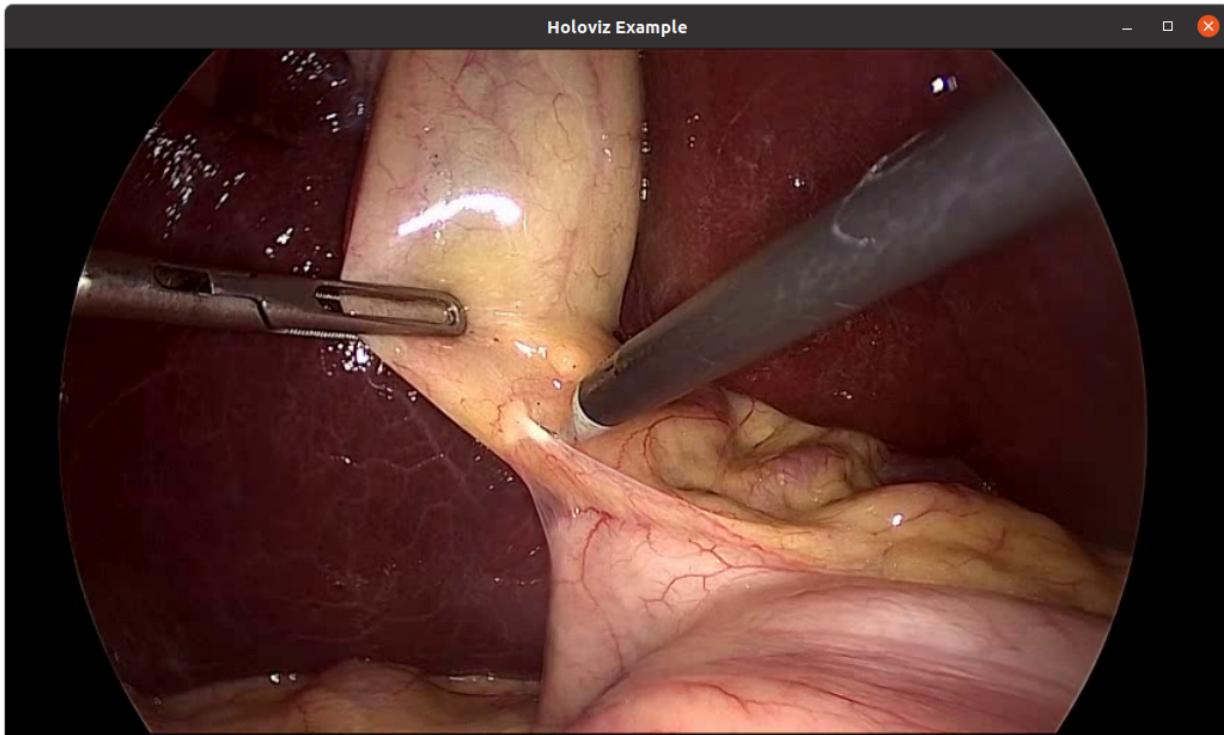


Fig. 12.1: Holoviz example app

12.4 Layers

The core entity of 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 Holoviz:

- image layer
- geometry layer
- GUI layer

The definition of a layer is started by calling one of the layer begin functions `viz::BeginImageLayer()`, `viz::BeginGeometryLayer()` or `viz::BeginImGuiLayer()`. The layer definition ends with `viz::EndLayer()`.

All layers have common attributes, these are priority and opacity. The start of a layer definition is resetting these values to their defaults.

The priority determines the rendering order of the layers. Before rendering the layers they are sorted by priority, the layers with the lowest priority are rendered first so that the layer with the highest priority is rendered on top of all other

layers. If layers have the same priority then the render order of these layers is undefined. Priority is set by calling `viz::LayerPriority()`.

Opacity is used to blend transparent layers over other layers. Opacity is set by calling `viz::LayerOpacity()`.

The code below draws a transparent geometry layer on top of an image layer (layer details are omitted). Although the geometry layer is specified first, it is drawn last because it has a higher priority (1) than the image layer (0). As mentioned the start of a layer is resetting layer attributes, so for the image layer, there is no need to set the opacity to 1.0 since the default is already 1.0.

```
namespace viz = holoscan::viz;

viz::Begin();

viz::BeginGeometryLayer();
viz::LayerPriority(1);
viz::LayerOpacity(0.5);
/// details omitted
viz::EndLayer();

viz::BeginImageLayer();
viz::LayerPriority(0);
/// details omitted
viz::EndLayer();

viz::End();
```

12.4.1 Image Layers

The function `viz::BeginImageLayer()` starts an image layer. An image layer displays a rectangular 2D image.

The image data is defined by calling `viz::ImageCudaDevice()` and `viz::ImageHost()`. Various input formats are supported, see `viz::ImageFormat`. For single channel image formats image colors can be looked up by defining a lookup table with `viz::LUT()`.

12.4.2 Geometry Layers

The function `viz::BeginGeometryLayer()` starts a geometry layer. A geometry layer is used to draw geometric primitives such as points, lines, rectangles, ovals or text. See `viz::PrimitiveTopology` for supported geometry primitive topologies. Coordinates start with (0, 0) in the top left and end with (1, 1) in the bottom right.

There are functions to set attributes for geometric primitives like color (`viz::Color()`), line width (`viz::LineWidth()`) and point size (`viz::PointSize()`).

The code below draws a red rectangle and a green text.

```
namespace viz = holoscan::viz;

viz::BeginGeometryLayer();

// draw a red rectangle
viz::Color(1.f, 0.f, 0.f, 0.f);
const float data[] {0.1f, 0.1f, 0.9f, 0.9f};
viz::Primitive(viz::PrimitiveTopology::RECTANGLE_LIST, 1, sizeof(data) / sizeof(data[0]),
    data);
```

(continues on next page)

(continued from previous page)

```
// draw green text
viz::Color(0.f, 1.f, 0.f, 0.f);
viz::Text(0.5f, 0.5f, 0.2f, "Text");

viz::EndLayer();
```

12.4.3 ImGui Layers

Holoviz support user interface layers created with [Dear ImGui](#).

If using Dear ImGui, create a context and pass it to Holoviz using `viz::ImGuiSetCurrentContext()`, do this before calling `viz::Init()`. Background: the Dear ImGui context is a global variable. Global variables are not shared across so/DLL boundaries. Therefore the app needs to create the Dear ImGui context first and then provide the pointer to Holoviz like this:

```
ImGui::CreateContext();
holoscan::viz::ImGuiSetCurrentContext(ImGui::GetCurrentContext());
```

Calls to the Dear ImGui API are allowed between `viz::BeginImGuiLayer()` and `viz::EndImGuiLayer()` are used to draw to the ImGui layer. The ImGui layer behaves like other layers and is rendered with the layer opacity and priority.

The code below creates a Dear ImGui window with a checkbox used to conditionally show a image layer.

```
namespace viz = holoscan::viz;

bool show_image_layer = false;
while (!viz::WindowShouldClose()) {
    viz::Begin();

    viz::BeginImGuiLayer();

    ImGui::Begin("Options");
    ImGui::Checkbox("Image layer", &show_image_layer);
    ImGui::End();

    viz::EndLayer();

    if (show_image_layer) {
        viz::BeginImageLayer();
        viz::ImageHost(...);
        viz::EndLayer();
    }

    viz::End();
}
```

12.4.4 Depth Map Layers

A depth map is single channel 2d array where each element represents a depth value. The data is specified with `viz::DepthMap()` and rendered as a 3d object using points, lines or triangles. The color for the elements can also be specified.

Supported format for the depth map:

- 8-bit unsigned normalized format that has a single 8-bit depth component

Supported format for the depth color map:

- 32-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3

Depth maps are rendered in 3D and support camera movement.

The camera is operated using the mouse.

- Orbit (LMB)
- Pan (LMB + CTRL | MMB)
- Dolly (LMB + SHIFT | RMB | Mouse wheel)
- Look Around (LMB + ALT | LMB + CTRL + SHIFT)
- Zoom (Mouse wheel + SHIFT)

12.5 Using a display in exclusive mode

Usually Holoviz opens a normal window on the Linux desktop. In that case the desktop compositor is combining the Holoviz image with all other elements on the desktop. To avoid this extra compositing step, Holoviz can render to a display directly.

12.5.1 Configure a display for exclusive use

Single display

SSH into the machine and stop the X server:

```
sudo systemctl stop display-manager
```

To resume the display manager, run:

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

Multiple displays

The display to be used in exclusive mode needs to be disabled in the NVIDIA Settings application (`nvidia-settings`): open the `X Server Display Configuration` tab, select the display and under `Configuration` select `Disabled`. Press `Apply`.

12.5.2 Enable exclusive display in Holoviz

Provide the name of the display and desired display mode properties to `viz::Init()`.

For example, here are parameters to pass to the Holoviz Operator:

```
# required
use_exclusive_display: true
# optional
display_name: DP-2
width: 2560
height: 1440
framerate: 240
```

The name of the display can either be the EDID name as displayed in the NVIDIA Settings, or the output name used by `xrandr`. If the name is `nullptr` then the first display is selected.

Tip: In this example output of `xrandr`, DP-2 would be an adequate display name to use:

```
Screen 0: minimum 8 x 8, current 4480 x 1440, maximum 32767 x 32767
DP-0 disconnected (normal left inverted right x axis y axis)
DP-1 disconnected (normal left inverted right x axis y axis)
DP-2 connected primary 2560x1440+1920+0 (normal left inverted right x axis y axis) 600mm
  ↗x 340mm
    2560x1440      59.98 + 239.97* 199.99    144.00    120.00      99.95
      1024x768      60.00
      800x600       60.32
      640x480       59.94
USB-C-0 disconnected (normal left inverted right x axis y axis)
```

12.6 Cuda streams

When providing Cuda resources to Holoviz through e.g. `viz::ImageCudaDevice()` Holoviz is using Cuda operations to use that memory. The Cuda stream used by this operations can be set by calling `viz::SetCudaStream()`. The stream can be changed at any time.

12.7 Reading the framebuffer

The rendered framebuffer can be read back using `viz::ReadFramebuffer()`.

INFERENCE MODULE

13.1 Overview

The Holoscan Inference Module in the Holoscan SDK is a framework that facilitates designing and executing inference and processing applications through its APIs. All parameters required by the Holoscan Inference Module are passed through a parameter set in the configuration file of an application. Detailed features and their corresponding parameter sets are explained in the section below.

13.2 Parameters and related Features

Required parameters and related features available with the Holoscan Inference Module are listed below, along with the limitations in the current release.

- Data Buffer Parameters: Parameters are provided in the inference settings to enable data buffer locations at several stages of the inference. As shown in the figure below, three parameters `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` can be set by the user.
 - `input_on_cuda` refers to the location of the data going into the inference.
 - * If value is `true`, it means the input data is on the device
 - * If value is `false`, it means the input data is on the host
 - `output_on_cuda` refers to the data location of the inferred data.
 - * If value is `true`, it means the inferred data is on the device
 - * If value is `false`, it means the inferred data is on the host
 - `transmit_on_cuda` refers to the data transmission.
 - * If value is `true`, it means the data transmission from the inference extension will be on **Device**
 - * If value is `false`, it means the data transmission from the inference extension will be on **Host**
- Inference Parameters
 - `backend` parameter is set to either `trt` for TensorRT, or `onnxrt` for Onnx runtime.
 - * TensorRT:
 - CUDA-based inference supported both on x86 and aarch64
 - End-to-end CUDA-based data buffer parameters supported. `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` will all be true for end-to-end CUDA-based data movement.
 - `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` can be either `true` or `false`.

- * Onnx runtime:
 - Data flow via host only. `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` must be `false`.
 - CUDA or CPU based inference on x86, only CPU based inference on aarch64
- `infer_on_cpu` parameter is set to `true` if CPU based inference is desired.

The tables below demonstrate the supported features related to the data buffer and the inference with `trt` and `onnxrt` based backend, on x86 and aarch64 system respectively.

x86	<code>input_on_cuda</code>	<code>output_on_cuda</code>	<code>transmit_on_cuda</code>	<code>infer_on_cpu</code>
Supported values for <code>trt</code>	<code>true</code> or <code>false</code>	<code>true</code> or <code>false</code>	<code>true</code> or <code>false</code>	<code>false</code>
Supported values for <code>onnxrt</code>	<code>false</code>	<code>false</code>	<code>false</code>	<code>true</code> or <code>false</code>

Aarch64	<code>input_on_cuda</code>	<code>output_on_cuda</code>	<code>transmit_on_cuda</code>	<code>infer_on_cpu</code>
Supported values for <code>trt</code>	<code>true</code> or <code>false</code>	<code>true</code> or <code>false</code>	<code>true</code> or <code>false</code>	<code>false</code>
Supported values for <code>onnxrt</code>	<code>false</code>	<code>false</code>	<code>false</code>	<code>true</code>

- `model_path_map`: User can design single or multi AI inference pipeline by populating `model_path_map` in the config file.
 - * With a single entry it is single inference and with more than one entry, multi AI inference is enabled.
 - * Each entry in `model_path_map` has a unique keyword as key (used as an identifier by the Holoscan Inference Module), and the path to the model as value.
 - * All model entries must have the models either in **onnx** or **tensorrt engine file** format.
- `pre_processor_map`: input tensor to the respective model is specified in `pre_processor_map` in the config file.
 - * The Holoscan Inference Module supports same input for multiple models or unique input per model.
 - * Each entry in `pre_processor_map` has a unique keyword representing the model (same as used in `model_path_map`), and the tensor name as the value.
 - * The Holoscan Inference Module supports one input tensor per model.
- `inference_map`: output tensor per model after inference is specified in `inference_map` in the config file.
 - * Each entry in `inference_map` has a unique keyword representing the model (same as used in `model_path_map` and `pre_processor_map`), and the tensor name as the value.
- `parallel_inference`: Parallel or Sequential execution of inferences.
 - * If multiple models are input, then user can execute models in parallel.
 - * Parameter `parallel_inference` can be either `true` or `false`.
 - * Inferences are launched in parallel without any check of the available GPU resources, user must make sure that there is enough memory and compute available to run all the inferences in parallel.
- `enable_fp16`: Generation of the TensorRT engine files with FP16 option
 - * If `backend` is set to `trt`, and if the input models are in **onnx** format, then users can generate the engine file with fp16 option to accelerate inferencing.
 - * It takes few minutes to generate the engine files for the first time.

- `is_engine_path`: if the input models are specified in **trt engine format** in `model_path_map`, this flag must be set to `true`.
- `in_tensor_names`: Input tensor names to be used by `pre_processor_map`.
- `out_tensor_names`: Output tensor names to be used by `inference_map`.
- Other features: Table below illustrates other features and supported values in the current release.

Feature	Supported values
Data type	<code>float32</code>
Inference Backend	<code>trt</code> or <code>onnxrt</code>
Inputs per model	1
GPU(s) supported	1
Inferred data size format	NHWC, NC (classification)
Model Type	All onnx or All trt engine type

- Multi Receiver and Single Transmitter support
 - The Holoscan Inference Module provides an API to extract the data from multiple receivers.
 - The Holoscan Inference Module provides an API to transmit multiple tensors via a single transmitter.

13.3 Usage

Following are the steps to be followed in sequence for creating an inference application using the Holoscan Inference Module in the Holoscan SDK.

13.3.1 Parameter Specification

All required inference parameters of the inference application must be specified. Specification are provided in the application configuration file in C++ API based application in the Holoscan SDK. Inference parameter set from the sample multi AI application using C++ APIs in the Holoscan SDK is shown below.

```
multiai_inference:
  backend: "trt"
  model_path_map:
    "plax_chamber": "../data/multiai_ultrasound/models/plax_chamber.onnx"
    "aortic_stenosis": "../data/multiai_ultrasound/models/aortic_stenosis.onnx"
    "bmode_perspective": "../data/multiai_ultrasound/models/bmode_perspective.onnx"
  pre_processor_map:
    "plax_chamber": ["plax_cham_pre_proc"]
    "aortic_stenosis": ["aortic_pre_proc"]
    "bmode_perspective": ["bmode_pre_proc"]
  inference_map:
    "plax_chamber": "plax_cham_infer"
    "aortic_stenosis": "aortic_infer"
    "bmode_perspective": "bmode_infer"
  in_tensor_names: ["plax_cham_pre_proc", "aortic_pre_proc", "bmode_pre_proc"]
  out_tensor_names: ["plax_cham_infer", "aortic_infer", "bmode_infer"]
  parallel_inference: true
  infer_on_cpu: false
  enable_fp16: false
```

(continues on next page)

(continued from previous page)

```
input_on_cuda: true
output_on_cuda: true
transmit_on_cuda: true
is_engine_path: false
```

13.3.2 Inference workflow

Inference workflow is the core inference unit in the inference application. This section provides steps to be followed to create an inference workflow.

In Holoscan SDK, the Multi AI Inference operator is designed using the Holoscan Inference Module APIs.

Arguments in the code sections below are referred to as

- Parameter Validity Check: Input inference parameters via the configuration (from step 1) are verified for correctness.

```
auto status = HoloInfer::multiai_inference_validity_check(...);
```

- Multi AI specification creation: For a single AI, only one entry is passed into the required entries in the parameter set. There is no change in the API calls below. Single AI or multi AI is enabled based on the number of entries in the parameter specifications from the configuration (in step 1).

```
// Declaration of multi AI inference specifications
std::shared_ptr<HoloInfer::MultiAISpecs> multiai_specs_;
```



```
// Creation of multi AI specification structure
multiai_specs_ = std::make_shared<HoloInfer::MultiAISpecs>(...);
```

- Inference context creation.

```
// Pointer to inference context.
std::unique_ptr<HoloInfer::InferContext> holoscan_infer_context_;
// Create holoscan inference context
holoscan_infer_context_ = std::make_unique<HoloInfer::InferContext>();
```

- Parameter setup with inference context: All required parameters of the Holoscan Inference Module are transferred in this step, and relevant memory allocations are initiated in the multi AI specification.

```
// Set and transfer inference specification to inference context
auto status = holoscan_infer_context_->set_inference_params(multiai_specs_);
```

- Data extraction and allocation: The following API is used from the Holoscan Inference Module to extract and allocate data for the specified tensor.

```
// Extract relevant data from input, and update multi AI specifications
gxf_result_t stat = HoloInfer::multiai_get_data_per_model(...);
```

- Map data from per tensor to per model: This step is required in this release. This step maps data per tensor to data per model. As mentioned above, current release supports only one input tensor per model.

```
auto status = HoloInfer::map_data_to_model_from_tensor(...);
```

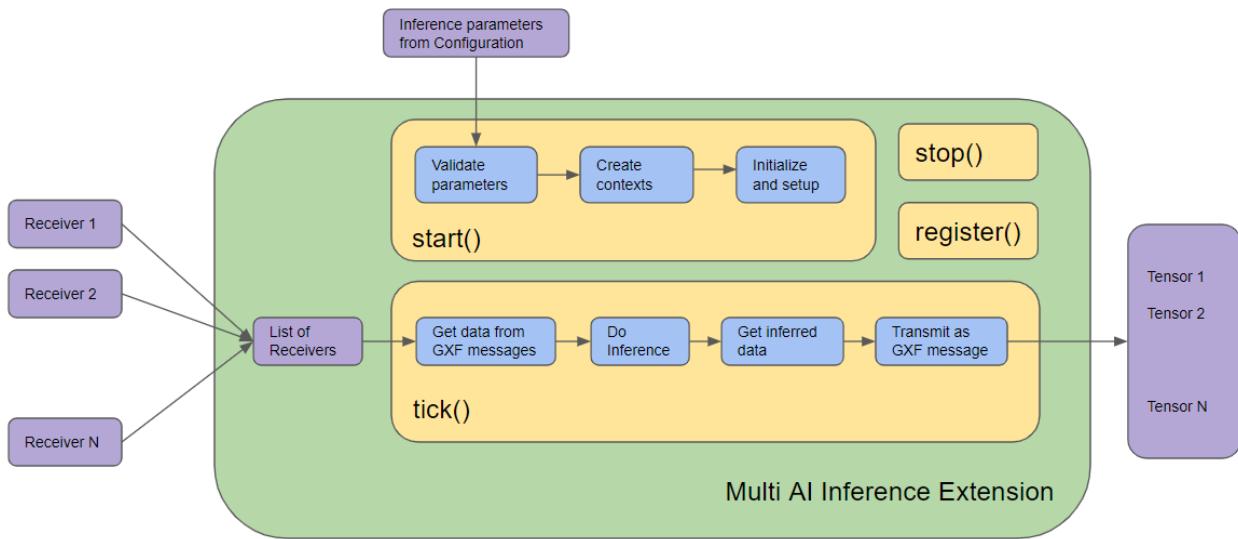
- Inference execution

```
// Execute inference and populate output buffer in multiai specifications
auto status = holoscan_infer_context_->execute_inference(multiai_specs_->data_per_
    ↵model_,
                                multiai_specs_->output_per_
    ↵model_);
```

- Transmit inferred data:

```
// Transmit output buffers
auto status = HoloInfer::multiai_transmit_data_per_model(...);
```

Figure below demonstrates the Multi AI Inference operator in the Holoscan SDK. All blocks with blue color are the API calls from the the Holoscan Inference Module.



13.3.3 Application creation

After creation of an inference workflow, an application creation is required to connect input data, pre-processors, inference workflow, post-processors and visualizers for end-to-end application creation. A sample multi AI pipeline from iCardio.ai's Multi AI application is part of Holoscan SDK, provided in both C++ and Python.

13.3.4 Application Execution

After a Holoscan SDK application has been successfully created, built and installed, execution is performed as described here for a sample Multi AI application

CHAPTER
FOURTEEN

GXF CORE CONCEPTS

Here is a list of the key GXF terms used in this section:

- **Applications** are built as compute graphs.
- **Entities** are nodes of the graph. They are nothing more than a unique identifier.
- **Components** are parts of an entity and provide their functionality.
- **Codelets** are special components which allow the execution of custom code. They can be derived by overriding the C++ functions `initialize`, `start`, `tick`, `stop`, `deinitialize`, and `registerInterface` (for defining configuration parameters).
- **Connections** are edges of the graph, which connect components.
- **Scheduler and Scheduling Terms**: components that determine how and when the `tick()` of a Codelet executes. This can be single or multithreaded, support conditional execution, asynchronous scheduling, and other custom behavior.
- **Memory Allocator**: provides a system for allocating a large contiguous memory pool up-front and then reusing regions as needed. Memory can be pinned to the device (enabling zero-copy between Codelets when messages are not modified) or host, or customized for other potential behavior.
- **Receivers, Transmitters, and Message Router**: a message passing system between Codelets that supports zero-copy.
- **Tensor**: the common message type is a tensor. It provides a simple abstraction for numeric data that can be allocated, serialized, sent between Codelets, etc. Tensors can be rank 1 to 7 supporting a variety of common data types like arrays, vectors, matrices, multi-channel images, video, regularly sampled time-series data, and higher dimensional constructs popular with deep learning flows.
- **Parameters**: configuration variables used by the Codelet. In GXF applications, they are loaded from the application YAML file and are modifiable without recompiling.

In comparison, the core concepts of the Holoscan SDK can be found [here](#).

HOLOSCAN AND GXF

15.1 Design differences

There are 2 main elements at the core of Holoscan and GXF designs:

1. How to define and execute application graphs
2. How to define nodes' functionality

How Holoscan SDK interfaces with GXF on those topics varies as Holoscan SDK evolves, as described below:

15.1.1 Holoscan SDK v0.2

Holoscan SDK was tightly coupled with GXF's existing interface:

1. GXF application graphs are defined in **YAML** configuration files. **GXE** (Graph Execution Engine) is used to execute AI application graphs. Its inputs are the YAML configuration file, and a list of GXF Extensions to load as plugins (manifest yaml file). This design allows entities to be swapped or updated without needing to recompile an application.
2. Components are made available by registering them within a **GXF extension**, each of which maps to a shared library and header(s).

Those concepts are illustrated in the *GXF by example* section.

The only additions that Holoscan provided on top of GXF were:

- domain specific reference applications
- new extensions
- CMake configurations for building extensions and applications

15.1.2 Holoscan SDK v0.3

The Holoscan SDK shifted to provide a more developer-friendly interface with C++:

1. GXF application graphs, memory allocation, scheduling, and message routing can be defined using a C++ API, with the ability to read parameters and required GXF extension names from a YAML configuration file. The backend used is still GXF as Holoscan uses the GXF C API, but this bypasses GXE and the full YAML definition.
2. The **C++ Operator** class was added to wrap and expose GXF extensions to that new application interface (See *dev guide*).

15.1.3 Holoscan SDK v0.4

The Holoscan SDK added Python wrapping and native operators to further increase ease of use:

1. The C++ API is also wrapped in Python. GXF is still used as the backend.
2. The Operator class supports **native operators**, i.e. operators that do not require to implement and register a GXF Extension. An important feature is the ability to support messaging between native and GXF operators without any performance loss (i.e. zero-copy communication of tensors).

15.1.4 Holoscan SDK v0.5

1. The built-in Holoscan GXF extensions are loaded automatically and don't need to be listed in the YAML configuration file of Holoscan applications. This allows Holoscan applications to be defined without requiring a YAML configuration file.
2. No significant changes to build operators. However, most built-in operators were switched to native implementations, with the ability to [convert native operators to GXF codelets](#) for GXF application developers.

15.2 Current limitations

Here is a list of GXF capabilities not yet available in the Holoscan SDK which are planned to be supported in future releases:

- [*Multithread scheduler*](#) (planned for 0.6)
- [*Periodic Scheduling Term*](#) (planned for 0.6)
- [*Asynchronous Scheduling Term*](#) (planned for 0.6)
- [*Job Statistics*](#)

The GXF capabilities below are not available in the Holoscan SDK either. There is no plan to support them at this time:

- [*Graph Composer*](#)
- [*Behavior Trees*](#)
- [*Epoch Scheduler*](#)
- [*Target Time Scheduling Term*](#)
- [*Multi-Message Available Scheduling Term*](#)
- [*Expiring Message Available Scheduling Term*](#)

GXF BY EXAMPLE

Warning: This section is legacy (0.2) as we recommend developing extensions and applications using the C++ or Python APIs. Refer to the developer guide for up-to-date recommendations.

16.1 Innerworkings of a GXF Entity

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

Listing 16.1: An example GXF Application YAML snippet

```
%YAML 1.2
---
# other entities declared
---

name: format_converter_entity
components:
  - name: in_tensor
    type: nvidia::gxf::DoubleBufferReceiver
  - type: nvidia::gxf::MessageAvailableSchedulingTerm
    parameters:
      receiver: in_tensor
      min_size: 1
  - name: out_tensor
    type: nvidia::gxf::DoubleBufferTransmitter
  - type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
    parameters:
      transmitter: out_tensor
      min_size: 1
  - name: pool
    type: nvidia::gxf::BlockMemoryPool
    parameters:
      storage_type: 1
      block_size: 4919040 # 854 * 480 * 3 (channel) * 4 (bytes per pixel)
      num_blocks: 2
  - name: format_converter_component
    type: nvidia::holoscan::formatconverter::FormatConverter
    parameters:
      in: in_tensor
```

(continues on next page)

(continued from previous page)

```

29      out: out_tensor
30      out_tensor_name: source_video
31      out_dtype: "float32"
32      scale_min: 0.0
33      scale_max: 255.0
34      pool: pool
35
36 # other entities declared
37
38 components:
39   - name: input_connection
40     type: nvidia::gxf::Connection
41     parameters:
42       source: upstream_entity/output
43       target: format_converter/in_tensor
44
45 components:
46   - name: output_connection
47     type: nvidia::gxf::Connection
48     parameters:
49       source: format_converter/out_tensor
50       target: downstream_entity/input
51
52 name: scheduler
53 components:
54   - type: nvidia::gxf::GreedyScheduler

```

Above:

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

The YAML snippet above can be visually represented as follows.

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 `DoubleBufferReceivernamedin_tensor`) and alerts the graph Executor that a message is available. This alert triggers `format_converter_component``.
- `DownstreamReceptiveSchedulingTerm` is a component that takes a `Transmitter` (in this case

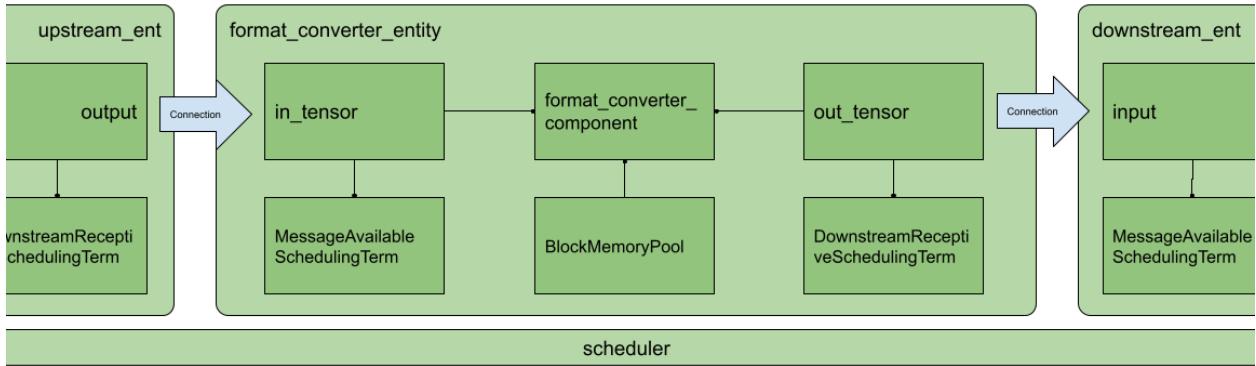


Fig. 16.1: Arrangement of components and entities in a Holoscan application

`DoubleBufferTransmitter` named `out_tensor`) and alerts the graph `Executor` that a message has been placed on the output.

- `BlockMemoryPool` provides two blocks of almost 5MB allocated on the GPU device and is used by `format_converter_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.

16.2 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 16.2: `MessageAvailableSchedulingTerm`

```

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

```

(continues on next page)

(continued from previous page)

```

3   receiver: in_tensor
4   min_size: 1

```

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

Listing 16.3: DownstreamReceptiveSchedulingTerm

```

1 - type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
2   parameters:
3     transmitter: out_tensor
4     min_size: 1

```

If we were to draw the entity in [Fig. 16.1](#) in greater detail it would look something like the following.

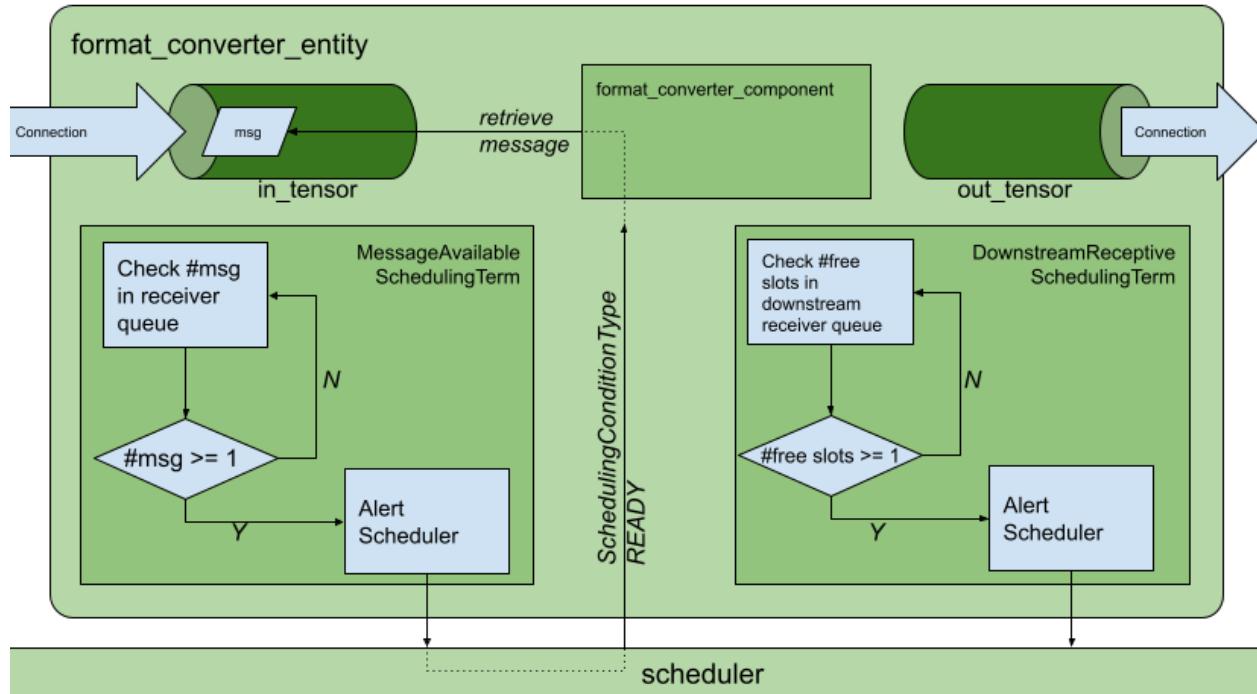


Fig. 16.2: Receive and transmit Queues and SchedulingTerms in entities.

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.

16.3 Creating a GXF Extension

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

16.3.1 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. 16.3*.

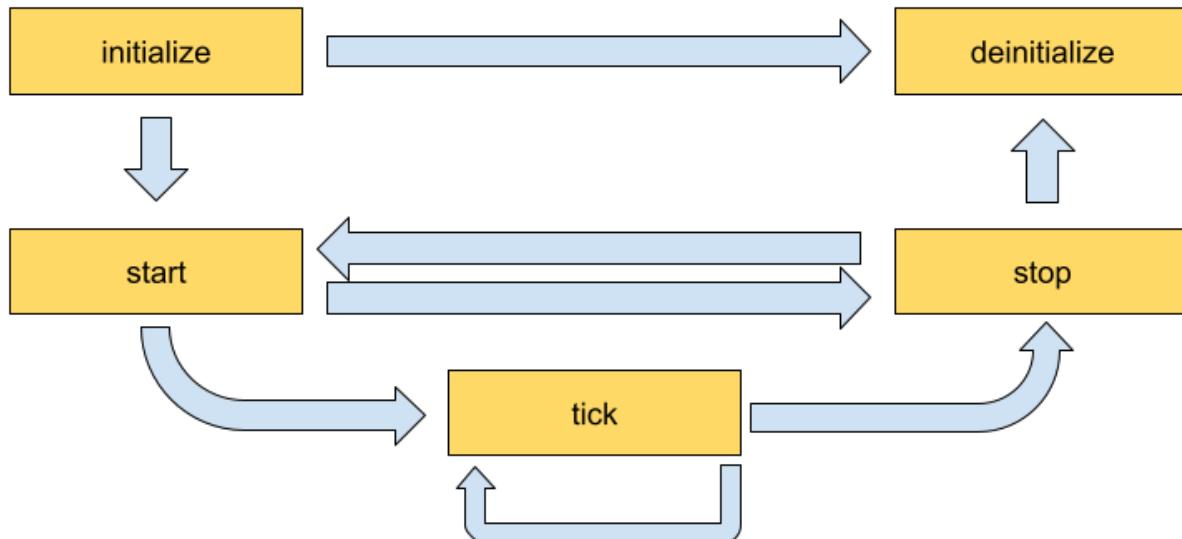


Fig. 16.3: Sequence of method calls in the lifecycle of a Holoscan extension

16.3.2 Implementing an Extension

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

Declare the Class That Will Implement the Extension Functionality

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

First, clone the Holoscan project from [here](#) and create a folder to develop our extension such as under `gxf_extensions/my_recorder`.

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

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

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

Listing 16.4: `gxf_extensions/my_recorder/my_recorder.hpp`

```
1 #include <string>
2
3 #include "gxf/core/handle.hpp"
4 #include "gxf/std/codelet.hpp"
5 #include "gxf/std/receiver.hpp"
6 #include "gxf/std/transmitter.hpp"
7 #include "gxf/serialization/file_stream.hpp"
8 #include "gxf/serialization/entity_serializer.hpp"
9
10
11 class MyRecorder : public nvidia::gxf::Codelet {
12 public:
13     gxf_result_t registerInterface(nvidia::gxf::Registrar* registrar) override;
14     gxf_result_t initialize() override;
15     gxf_result_t deinitialize() override;
16
17     gxf_result_t start() override;
18     gxf_result_t tick() override;
19     gxf_result_t stop() override;
20
21 private:
22     nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::Receiver>> receiver_;
23     nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::EntitySerializer>> my_
24     ↵ serializer_;
```

(continues on next page)

(continued from previous page)

```

24 nvidia::gxf::Parameter<std::string> directory_;
25 nvidia::gxf::Parameter<std::string> basename_;
26 nvidia::gxf::Parameter<bool> flush_on_tick_;
27
28 // File stream for data index
29 nvidia::gxf::FileStream index_file_stream_;
30 // File stream for binary data
31 nvidia::gxf::FileStream binary_file_stream_;
32 // Offset into binary file
33 size_t binary_file_offset_;
34 };

```

Declare the Parameters to Expose at the Application Level

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

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

Listing 16.5: registerInterface in `gxf_extensions/my_recorder/my_recorder.cpp`

```

1 #include "my_recorder.hpp"
2
3 gxf_result_t MyRecorder::registerInterface(nvidia::gxf::Registrar* registrar) {
4     nvidia::gxf::Expected<void> result;
5     result &= registrar->parameter(
6         receiver_, "receiver", "Entity receiver",
7         "Receiver channel to log");
8     result &= registrar->parameter(
9         my_serializer_, "serializer", "Entity serializer",
10        "Serializer for serializing input data");
11     result &= registrar->parameter(
12         directory_, "out_directory", "Output directory path",
13         "Directory path to store received output");
14     result &= registrar->parameter(
15         basename_, "basename", "File base name",
16         "User specified file name without extension",
17         nvidia::gxf::Registrar::NoDefaultParameter(), GXF_PARAMETER_FLAGS_OPTIONAL);
18     result &= registrar->parameter(
19         flush_on_tick_, "flush_on_tick", "Boolean to flush on tick",
20         "Flushes output buffer on every `tick` when true", false); // default value `false`
21     return nvidia::gxf::ToResultCode(result);
22 }

```

For pure GXF applications, our component's parameters can be specified in the following format in the YAML file:

Listing 16.6: Example parameters for MyRecorder component

```

1 name: my_recorder_entity
2 components:
3   - name: my_recorder_component

```

(continues on next page)

(continued from previous page)

```

4   type: MyRecorder
5   parameters:
6     receiver: receiver
7     serializer: my_serializer
8     out_directory: /home/user/out_path
9     basename: my_output_file # optional
10    # flush_on_tick: false # optional

```

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

Implement the Lifecycle Methods

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

Listing 16.7: initialize in gxf_extensions/my_recorder/my_recorder.cpp

```

24 gxf_result_t MyRecorder::initialize() {
25   // Create path by appending receiver name to directory path if basename is not provided
26   std::string path = directory_.get() + '/';
27   if (const auto& basename = basename_.try_get()) {
28     path += basename.value();
29   } else {
30     path += receiver_->name();
31   }
32
33   // Initialize index file stream as write-only
34   index_file_stream_ = nvidia::gxf::FileStream("", path +
35   ↵nvidia::gxf::FileStream::kIndexFileExtension);
36
37   // Initialize binary file stream as write-only
38   binary_file_stream_ = nvidia::gxf::FileStream("", path +
39   ↵nvidia::gxf::FileStream::kBinaryFileExtension);
40
41   // Open index file stream
42   nvidia::gxf::Expected<void> result = index_file_stream_.open();
43   if (!result) {
44     return nvidia::gxf::ToResultCode(result);
45   }
46
47   // Open binary file stream
48   result = binary_file_stream_.open();
49   if (!result) {
50     return nvidia::gxf::ToResultCode(result);
51   }
52   binary_file_offset_ = 0;
53
54   return GXF_SUCCESS;
55 }

```

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

Listing 16.8: deinitialize in gxf_extensions/my_recorder/my_recorder.cpp

```

55 gxf_result_t MyRecorder::deinitialize() {
56     // Close binary file stream
57     nvidia::gxf::Expected<void> result = binary_file_stream_.close();
58     if (!result) {
59         return nvidia::gxf::ToResultCode(result);
60     }
61
62     // Close index file stream
63     result = index_file_stream_.close();
64     if (!result) {
65         return nvidia::gxf::ToResultCode(result);
66     }
67
68     return 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 16.9: 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 16.10: tick in gxf_extensions/my_recorder/my_recorder.cpp

```

79 gxf_result_t MyRecorder::tick() {
80     // Receive entity
81     nvidia::gxf::Expected<nvidia::gxf::Entity> entity = receiver_>receive();
82     if (!entity) {
83         return nvidia::gxf::ToResultCode(entity);
84     }
85
86     // Write entity to binary file
87     nvidia::gxf::Expected<size_t> size = my_serializer_>serializeEntity(entity.value(), &
88     ↵binary_file_stream_);
89     if (!size) {
90         return nvidia::gxf::ToResultCode(size);
91     }
```

(continues on next page)

(continued from previous page)

```

91
92     // Create entity index
93     nvidia::gxf::EntityIndex index;
94     index.log_time = std::chrono::system_clock::now().time_since_epoch().count();
95     index.data_size = size.value();
96     index.data_offset = binary_file_offset_;
97
98     // Write entity index to index file
99     nvidia::gxf::Expected<size_t> result = index_file_stream_.writeTrivialType(&index);
100    if (!result) {
101        return nvidia::gxf::ToresultCode(result);
102    }
103    binary_file_offset_ += size.value();
104
105    if (flush_on_tick_) {
106        // Flush binary file output stream
107        nvidia::gxf::Expected<void> result = binary_file_stream_.flush();
108        if (!result) {
109            return nvidia::gxf::ToresultCode(result);
110        }
111
112        // Flush index file output stream
113        result = index_file_stream_.flush();
114        if (!result) {
115            return nvidia::gxf::ToresultCode(result);
116        }
117    }
118
119    return GXF_SUCCESS;
120}

```

Register the Extension as a Holoscan Component

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

Listing 16.11: `gxf_extensions/my_recorder/my_recorder_ext.cpp`

```

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

```

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

- UUID which can be generated using `scripts/generate_extension_uuids.py` which defines the `extension`

id

- extension name
- extension description
- author
- extension version
- license text

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

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

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

Listing 16.12: `gxf_extensions/my_recorder/CMakeLists.txt`

```

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

```

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

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

```
add_subdirectory(my_recorder)
```

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

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

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

16.4 Creating a GXF Application

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

Listing 16.13: `apps/my_recorder_app_gxf/my_recorder_gxf.yaml`

```
%YAML 1.2
---
name: replayer
components:
- name: output
  type: nvidia::gxf::DoubleBufferTransmitter
- name: allocator
  type: nvidia::gxf::UnboundedAllocator
- name: component_serializer
  type: nvidia::gxf::StdComponentSerializer
  parameters:
    allocator: allocator
- name: entity_serializer
  type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
  ↵nvidia::gxf::EntitySerializer
  parameters:
    component_serializers: [component_serializer]
- type: nvidia::holoscan::stream_playback::VideoStreamReplayer
  parameters:
    transmitter: output
    entity_serializer: entity_serializer
    boolean_scheduling_term: boolean_scheduling
    directory: "/workspace/data/endoscopy/video"
    basename: "surgical_video"
    frame_rate: 0      # as specified in timestamps
    repeat: false      # default: false
    realtime: true     # default: true
    count: 0          # default: 0 (no frame count restriction)
- name: boolean_scheduling
  type: nvidia::gxf::BooleanSchedulingTerm
- type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
  parameters:
    transmitter: output
    min_size: 1
```

(continues on next page)

(continued from previous page)

```

34
35 name: recorder
36 components:
37   - name: input
38     type: nvidia::gxf::DoubleBufferReceiver
39   - name: allocator
40     type: nvidia::gxf::UnboundedAllocator
41   - name: component_serializer
42     type: nvidia::gxf::StdComponentSerializer
43     parameters:
44       allocator: allocator
45   - name: entity_serializer
46     type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from
47     ↵nvidia::gxf::EntitySerializer
48     parameters:
49       component_serializers: [component_serializer]
50   - type: MyRecorder
51     parameters:
52       receiver: input
53       serializer: entity_serializer
54       out_directory: "/tmp"
55       basename: "tensor_out"
56   - type: nvidia::gxf::MessageAvailableSchedulingTerm
57     parameters:
58       receiver: input
59       min_size: 1
60
61 ---
62 components:
63   - name: input_connection
64     type: nvidia::gxf::Connection
65     parameters:
66       source: replayer/output
67       target: recorder/input
68
69 ---
70 name: scheduler
71 components:
72   - name: clock
73     type: nvidia::gxf::RealtimeClock
74   - name: greedy_scheduler
      type: nvidia::gxf::GreedyScheduler
      parameters:
        clock: clock

```

Above:

- The replayer reads data from /workspace/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 16.14: apps/my_recorder_app_gxf/CMakeLists.txt

```
1 create_gxe_application(
2   NAME my_recorder_gxf
3   YAML my_recorder_gxf.yaml
4   EXTENSIONS
5     GXF::std
6     GXF::cuda
7     GXF::multimedia
8     GXF::serialization
9     my_recorder
10    stream_playback
11 )
12
13 # Download the associated dataset if needed
14 if(HOLOSCAN_DOWNLOAD_DATASETS)
15   add_dependencies(my_recorder_gxf endoscopy_data)
16 endif()
```

In the declaration of `create_gxe_application` we list:

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

To make our newly built application discoverable by the build, in the root of the repository, we add the following line to `apps/CMakeLists.txt`:

```
add_subdirectory(my_recorder_app_gxf)
```

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

16.5 Running the GXF Recorder Application

To run our application in a local development container:

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

You can execute the following commands to build

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

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

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

from inside the development container.

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

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

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

(continues on next page)

(continued from previous page)

```
drwxrwxrwt 2 root root      4096 Aug 11 21:42 .X11-unix
-rw-r--r-- 1 1000 1000    729309 Aug 24 04:47 gxf_log
-rw-r--r-- 1 1000 1000 840054484 Aug 24 04:47 tensor_out.gxf_entities
-rw-r--r-- 1 1000 1000     16392 Aug 24 04:47 tensor_out.gxf_index

@LINUX:/workspace/holoscan-sdk/build$ ls -al ../data/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
```

USING HOLOSCAN OPERATORS IN GXF APPLICATIONS

For users who are familiar with the GXF development ecosystem (used in Holoscan SDK 0.2), we provide an export feature to leverage native Holoscan operators as GXF codelets to execute in GXF applications and GraphComposer.

We demonstrate how to wrap a native C++ holoscan operator as a GXF codelet in the [wrap_operator_as_gxf_extension](#) example on GitHub, as described below.

17.1 1. Creating compatible Holoscan Operators

Note: This section assumes you are already familiar with [how to create a native C++ operator](#).

To be compatible with GXF codelets, inputs and outputs specified in `Operator::setup(OperatorSpec& spec)` must be of type `holoscan::gxf::Entity`, as shown in the `PingTxNativeOp` and the `PingRxNativeOp` implementations of this example, in contrast to the `PingTxOp` and `PingRxOp` built-in operators of the SDK.

For more details regarding the use of `holoscan::gxf::Entity`, follow the documentation on [Interoperability between GXF and native C++ operators](#).

17.2 2. Creating the GXF extension that wraps the operator

To wrap the native operator as a GXF codelet in a GXF extension, we provide the CMake `wrap_operator_as_gxf_extension` function in the SDK. An example of how it wraps `PingTxNativeOp` and `PingRxNativeOp` can be found [here](#).

- It leverages the CMake target names of the operators defined in their respective `CMakeLists.txt` (`ping_tx_native_op`, `ping_rx_native_op`)
- The function parameters are documented at the top of the `WrapOperatorAsGXFEExtension.cmake` file (ignore implementation below).

Warning:

- A unique GXF extension is currently needed for each native operator to export (operators cannot be bundled in a single extension at this time).
- Wrapping other GXF entities than operators (as codelets) is not currently supported.

17.3 3. Using your wrapped operator in a GXF application

Note: This section assumes you are familiar with [how to create a GXF application](#).

As shown in the `gxf_app/CMakeLists.txt` [here](#), you need to list the following extensions in `create_gxe_application()` to use your wrapped codelets:

- `GXF::std`
- `gxf_holoscan_wrapper`
- the name of the CMake target for the created extension, defined by the `EXTENSION_TARGET_NAME` argument passed to `wrap_operator_as_gxf_extension` in the previous section

The codelet class name (defined by the `CODELET_NAMESPACE::CODELET_NAME` arguments passed to `wrap_operator_as_gxf_extension` in the previous section) can then be used as a component type in a GXF app node, as shown in the [YAML app definition](#) of the example, connecting the two ping operators.

GXF USER GUIDE

18.1 Graph Specification

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

18.1.1 Concepts

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

The section :doc: *GraphComposer_Dev_Workflow* talks more about this with a focus on developing extensions and components.

18.1.2 Graph File Format

Graph file stores list of entities. Each entity has a unique name and list of components. Each component has a name, a type and properties. Properties are stored as key-value pairs.

```
%YAML 1.2
---
name: source
components:
- name: signal
  type: sample::test::ping
- type: nvidia::gxf::CountSchedulingTerm
  parameters:
    count: 10
---
components:
- type: nvidia::gxf::GreedyScheduler
  parameters:
    realtime: false
    max_duration_ms: 1000000
```

18.2 Graph Execution Engine

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

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

18.3 Graph Specification TimeStamping

18.3.1 Message Passing

Once the graph is built, the communication between various entities occur by passing around messages (messages are entities themselves). Specifically, one component/codelet can publish a message entity and another can receive it. When publishing, a message should always have an associated `Timestamp` component with the name “`timestamp`”. A `Timestamp` component contains two different time values (See the `gxf/std/timestamp.hpp` header file for more information.):

1. **acqtime** - This is the time when the message entity is acquired, for instance, this would generally be the driver time of the camera when it captures an image. You must provide this timestamp if you are publishing a message in a codelet.
 2. **pubtime** - This is the time when the message entity is published by a node in the graph. This will automatically get updated using the clock of the scheduler.

In a codelet, when publishing message entities using a `Transmitter` (`tx`), there are two ways to add the required `Timestamp`:

1. `tx.publish(Entity message)`: You can manually add a component of type `Timestamp` with the name “`timestamp`” and set the `acqtime`. The `pubtime` in this case should be set to `Ø`. The message is published using the `publish(Entity message)`. **This will be deprecated in the next release.**
 2. `tx.publish(Entity message, int64_t acqtime)`: You can simply call `publish(Entity message, int64_t acqtime)` with the `acqtime`. `Timestamp` will be added automatically.

18.4 The GXF Scheduler

The execution of entities in a graph is governed by the scheduler and the scheduling terms associated with every entity. A scheduler is a component responsible for orchestrating the execution of all the entities defined in a graph. A scheduler typically keeps track of the graph entities and their current execution states and passes them on to a `nvidia::gfx::EntityExecutor` component when ready for execution. The following diagram depicts the flow for an entity execution.

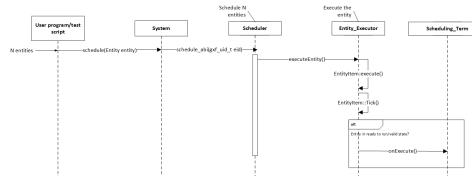


Figure: Entity execution sequence

As shown in the sequence diagram, the schedulers begin executing the graph entities via the `nvidia::gfx::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::gfx::SchedulingConditionType` described using `nvidia::gfx::SchedulingCondition`.

SchedulingConditionType	Description
NEVER	Entity will never execute again
READY	Entity is ready for execution
WAIT	Entity may execute in the future
WAIT_TIME	Entity will be ready for execution after specified duration
WAIT_EVENT	Entity is waiting on an asynchronous event with unknown time interval

Schedulers define deadlock as a condition when there are no entities which are in READY, WAIT_TIME or WAIT_EVENT 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

18.4.1 Greedy Scheduler

This is a basic single threaded scheduler which tests scheduling term greedily. It is great for simple use cases and predictable execution but may incur a large overhead of scheduling term execution, making it unsuitable for large applications. The scheduler requires a clock to keep track of time. Based on the choice of clock the scheduler will execute differently. If a Realtime clock is used the scheduler will execute in real-time. This means pausing execution - sleeping the thread, until periodic scheduling terms are due again. If a ManualClock is used scheduling will happen “time-compressed”. This means flow of time is altered to execute codelets in immediate succession.

The GreedyScheduler maintains a running count of entities which are in READY, WAIT_TIME and WAIT_EVENT states. The following activity diagram depicts the gist of the decision making for scheduling an entity by the greedy scheduler.

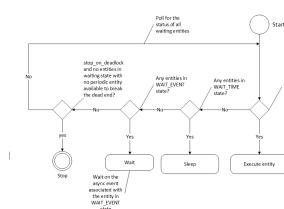


Figure: Greedy Scheduler Activity Diagram

Greedy Scheduler Configuration

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

Parameter name	Description
clock	The clock used by the scheduler to define the flow of time. Typical choices are 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.

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

```
name: scheduler
components:
- type: nvidia::gxf::GreedyScheduler
parameters:
    max_duration_ms: 3000
    clock: misc/clock
    stop_on_deadlock: true
---
name: misc
components:
- name: clock
  type: nvidia::gxf::ManualClock
```

18.4.2 Multithread Scheduler

The MultiThread scheduler is more suitable for large applications with complex execution patterns. The scheduler consists of a dispatcher thread which checks the status of an entity and dispatches it to a thread pool of worker threads responsible for executing them. Worker threads enqueue the entity back on to the dispatch queue upon completion of execution. The number of worker threads can be configured using `worker_thread_number` parameter. The MultiThread scheduler also manages a dedicated queue and thread to handle asynchronous events. The following activity diagram demonstrates the gist of the multithread scheduler implementation.

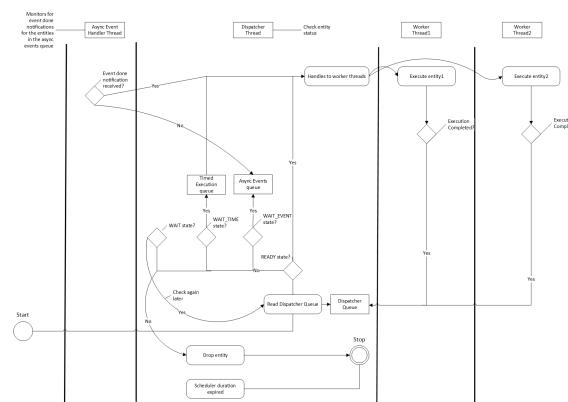


Figure: MultiThread Scheduler Activity Diagram

As depicted in the diagram, when an entity reaches 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.

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

```

name: scheduler
components:
- type: nvidia::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

```

18.4.3 Epoch Scheduler

The Epoch scheduler is used for running loads in externally managed threads. Each run is called an Epoch. The scheduler goes over all entities that are known to be active and executes them one by one. If the epoch budget is provided (in ms), it would keep running all codelets until the budget is consumed or no codelet is ready. It might run over budget since it guarantees to cover all codelets in epoch. In case the budget is not provided, it would go over all the codelets once and execute them only once.

The epoch scheduler takes in the following parameters from the configuration file -

Parameter name	Description
clock	The clock used by the scheduler to define the flow of time. Typical choice is a RealtimeClock.

Example usage - The following code snippet configures an Epoch scheduler -

```
name: scheduler
components:
- name: clock
  type: nvidia::gxf::RealtimeClock
- name: epoch
  type: nvidia::gxf::EpochScheduler
  parameters:
    clock: clock
```

Note that the epoch scheduler is intended to run from an external thread. The `runEpoch(float budget_ms);` can be used to set the budget_ms and run the scheduler from the external thread. If the specified budget is not positive, all the nodes are executed once.

18.4.4 SchedulingTerms

A SchedulingTerm defines a specific condition that is used by an entity to let the scheduler know when it's ready for execution. There are various scheduling terms currently supported by GXF.

PeriodicSchedulingTerm

An entity associated with `nvidia::gxf::PeriodicSchedulingTerm` is ready for execution after periodic time intervals specified using its `recess_period` parameter. The PeriodicSchedulingTerm can either be in READY or WAIT_TIME state.

Example usage -

```
- name: scheduling_term
  type: nvidia::gxf::PeriodicSchedulingTerm
  parameters:
    recess_period: 50000000
```

CountSchedulingTerm

An entity associated with `nvidia::gxf::CountSchedulingTerm` is executed for a specific number of times specified using its `count` parameter. The `CountSchedulingTerm` can either be in READY or NEVER state. The scheduling term reaches the NEVER state when the entity has been executed count number of times.

Example usage -

```
- name: scheduling_term
  type: nvidia::gxf::CountSchedulingTerm
  parameters:
    count: 42
```

MessageAvailableSchedulingTerm

An entity associated with `nvidia::gxf::MessageAvailableSchedulingTerm` is executed when the associated receiver queue has at least a certain number of elements. The receiver is specified using the `receiver` parameter of the scheduling term. The minimum number of messages that permits the execution of the entity is specified by `min_size`. An optional parameter for this scheduling term is `front_stage_max_size`, the maximum front stage message count. If this parameter is set, the scheduling term will only allow execution if the number of messages in the queue does not exceed this count. It can be used for codelets which do not consume all messages from the queue.

In the example shown below, the minimum size of the queue is configured to be 4. This means the entity will not be executed until there are at least 4 messages in the queue.

```
- type: nvidia::gxf::MessageAvailableSchedulingTerm
  parameters:
    receiver: tensors
    min_size: 4
```

MultiMessageAvailableSchedulingTerm

An entity associated with `nvidia::gxf::MultiMessageAvailableSchedulingTerm` is executed when a list of provided input receivers combined have at least a given number of messages. The `receivers` parameter is used to specify a list of the input channels/receivers. The minimum number of messages needed to permit the entity execution is set by `min_size` parameter.

Consider the example shown below. The associated entity will be executed when the number of messages combined for all the three receivers is at least the `min_size`, i.e. 5.

```
- name: input_1
  type: nvidia::gxf::test::MockReceiver
  parameters:
    max_capacity: 10
- name: input_2
  type: nvidia::gxf::test::MockReceiver
  parameters:
    max_capacity: 10
- name: input_3
  type: nvidia::gxf::test::MockReceiver
  parameters:
    max_capacity: 10
- type: nvidia::gxf::MultiMessageAvailableSchedulingTerm
```

(continues on next page)

(continued from previous page)

```
parameters:
  receivers: [input_1, input_2, input_3]
  min_size: 5
```

BooleanSchedulingTerm

An entity associated with `nvidia::gxf::BooleanSchedulingTerm` is executed when its internal state is set to tick. The parameter `enable_tick` is used to control the entity execution. The scheduling term also has two APIs `enable_tick()` and `disable_tick()` to toggle its internal state. The entity execution can be controlled by calling these APIs. If `enable_tick` is set to false, the entity is not executed (Scheduling condition is set to NEVER). If `enable_tick` is set to true, the entity will be executed (Scheduling condition is set to READY). Entities can toggle the state of the scheduling term by maintaining a handle to it.

Example usage -

```
- type: nvidia::gxf::BooleanSchedulingTerm
  parameters:
    enable_tick: true
```

AsynchronousSchedulingTerm

`AsynchronousSchedulingTerm` is primarily associated with entities which are working with asynchronous events happening outside of their regular execution performed by the scheduler. Since these events are non-periodic in nature, `AsynchronousSchedulingTerm` prevents the scheduler from polling the entity for its status regularly and reduces CPU utilization. `AsynchronousSchedulingTerm` can either be in READY, WAIT, WAIT_EVENT or NEVER states based on asynchronous event it's waiting on.

The state of an asynchronous event is described using `nvidia::gxf::AsynchronousEventState` and is updated using the `setEventState` API.

AsynchronousEventState	Description
READY	Init state, first tick is pending
WAIT	Request to async service yet to be sent, nothing to do but wait
EVENT_WAITING	Request sent to an async service, pending event done notification
EVENT_DONE	Event done notification received, entity ready to be ticked
EVENT_NEVER	Entity does not want to be ticked again, end of execution

Entities associated with this scheduling term most likely have an asynchronous thread which can update the state of the scheduling term outside of its regular execution cycle performed by the gxf scheduler. When the scheduling term is in WAIT state, the scheduler regularly polls for the state of the entity. When the scheduling term is in EVENT_WAITING state, schedulers will not check the status of the entity again until they receive an event notification which can be triggered using the `GxfEntityEventNotify` api. Setting the state of the scheduling term to EVENT_DONE automatically sends this notification to the scheduler. Entities can use the EVENT_NEVER state to indicate the end of its execution cycle.

Example usage -

```
- name: async_scheduling_term
  type: nvidia::gxf::AsynchronousSchedulingTerm
```

DownstreamReceptiveSchedulingTerm

This scheduling term specifies that an entity shall be executed if the receiver for a given transmitter can accept new messages.

Example usage -

```
- name: downstream_st
  type: nvidia::gxf::DownstreamReceptiveSchedulingTerm
  parameters:
    transmitter: output
    min_size: 1
```

TargetTimeSchedulingTerm

This scheduling term permits execution at a user-specified timestamp. The timestamp is specified on the clock provided.

Example usage -

```
- name: target_st
  type: nvidia::gxf::TargetTimeSchedulingTerm
  parameters:
    clock: clock/manual_clock
```

ExpiringMessageAvailableSchedulingTerm

This scheduling waits for a specified number of messages in the receiver. The entity is executed when the first message received in the queue is expiring or when there are enough messages in the queue. The `receiver` parameter is used to set the receiver to watch on. The parameters `max_batch_size` and `max_delay_ns` dictate the maximum number of messages to be batched together and the maximum delay from first message to wait before executing the entity respectively.

In the example shown below, the associated entity will be executed when the number of messages in the queue is greater than `max_batch_size`, i.e 5, or when the delay from the first message to current time is greater than `max_delay_ns`, i.e 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
```

BTSchedulingTerm

A BT (Behavior Tree) scheduling term is used to schedule a behavior tree entity itself and its child entities (if any) in a Behavior tree.

Example usage -

```
name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_codelet
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: [ child1/child1_st ]
    s_term: root_st
    controller: root_controller
```

18.5 Behavior Trees

Behavior tree codelets are one of the mechanisms to control the flow of tasks in GXF. They follow the same general behavior as classical behavior trees, with some useful additions for robotics applications. This document gives an overview of the general concept, the available behavior tree node types, and some examples of how to use them individually or in conjunction with each other.

18.5.1 General Concept

Behavior trees consist of n-ary trees of entities that can have zero or more children. The conditional execution of parent entity is based on the status of execution of the children. A behavior tree is graphically represented as a directed tree in which the nodes are classified as root, control flow nodes, or execution nodes (tasks). For each pair of connected nodes, the outgoing node is called parent and the incoming node is called child.

The execution of a behavior tree starts from the root which sends ticks with a certain frequency to its child. When the execution of a node in the behavior tree is allowed, it returns to the parent a status running if its execution has not finished yet, success if it has achieved its goal, or failure otherwise. The behavior tree also uses a controller component for controlling the entity's termination policy and the execution status. One of the controller behaviors currently implemented for Behavior Tree is `EntityCountFailureRepeatController`, which repeats the entity on failure up to `repeat_count` times before deactivating it.

GXF supports several behavior tree codelets which are explained in the following section.

18.5.2 Behavior Tree Codelets

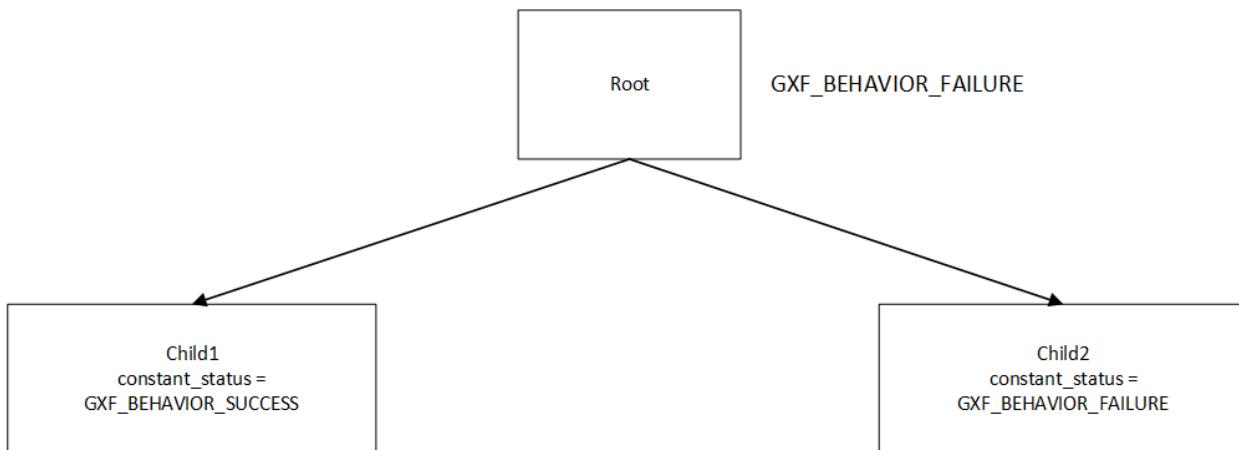
Each behavior tree codelet can have a set of parameters defining how it should behave. Note that in all the examples given below, the naming convention for configuring the `children` parameter for root codelets is `[child_codelet_name\child_codelet_scheduling_term]`.

Constant Behavior

After each tick period, switches its own status to the configured desired constant status.

Parameter	Description
<code>s_term</code>	scheduling term used for scheduling the entity itself
<code>constant_status</code>	The desired status to switch to during each tick time.

An example diagram depicting Constant behavior used in conjunction with a Sequence behavior defined for root entity is shown below



Here, the child1 is configured to return a constant status of success (GXF_BEHAVIOR_SUCCESS) and child2 returns failure (GXF_BEHAVIOR_FAILURE), resulting into the root node (configured to exhibit sequence behavior) returning GXF_BEHAVIOR_FAILURE.

The controller for each child can be configured to repeat the execution on failure. A code snippet of configuring the example described is shown below.

```

name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_codelet
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: [ child1/child1_st, child2/child2_st ]
    s_term: root_st
---

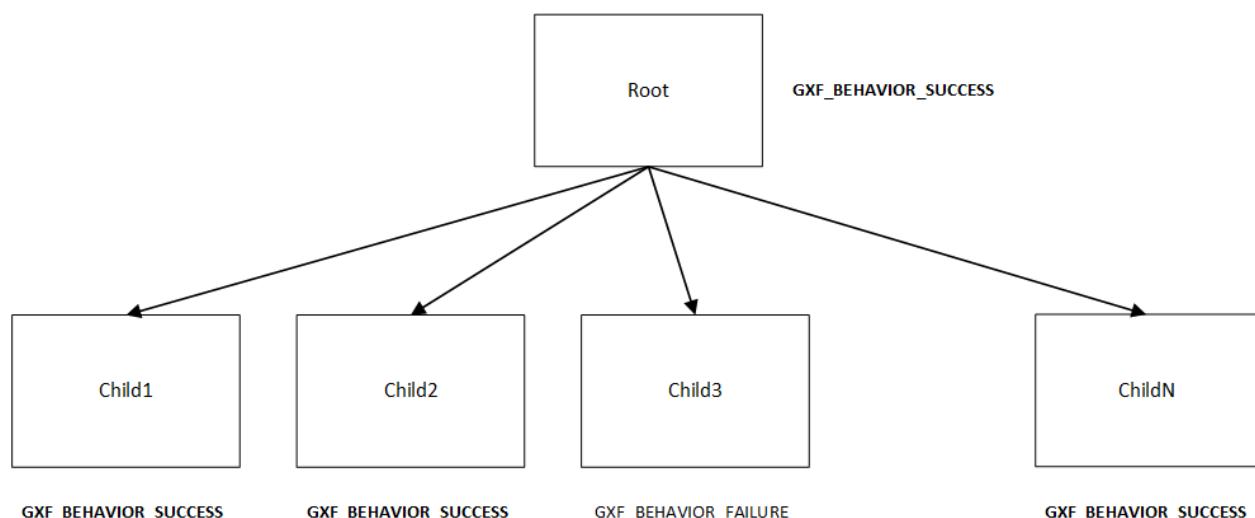
name: child2
components:
- name: child2_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 3
    return_behavior_running_if_failure_repeat: true
- name: child2_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: child2_codelet
  type: nvidia::gxf::ConstantBehavior
  parameters:
    s_term: child2_st
    constant_status: 1
  
```

Parallel Behavior

Runs its child nodes in parallel. By default, succeeds when all child nodes succeed, and fails when all child nodes fail. This behavior can be customized using the parameters below.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
children	Child entities
suc-cess_threshold	Number of successful children required for success. A value of -1 means all children must succeed for this node to succeed.
fail-ure_threshold	Number of failed children required for failure. A value of -1 means all children must fail for this node to fail.

The diagram below shows a graphical representation of a parallel behavior configured with failure_threshold configured as -1. Hence, the root node returns GXF_BEHAVIOR_SUCCESS even if one child returns a failure status.



A code snippet to configure the example described is shown below.

```

name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_codelet
  type: nvidia::gxf::ParallelBehavior
  parameters:
    children: [ child1/child1_st, child2/child2_st ]
    s_term: root_st
    success_threshold: 1
    failure_threshold: -1

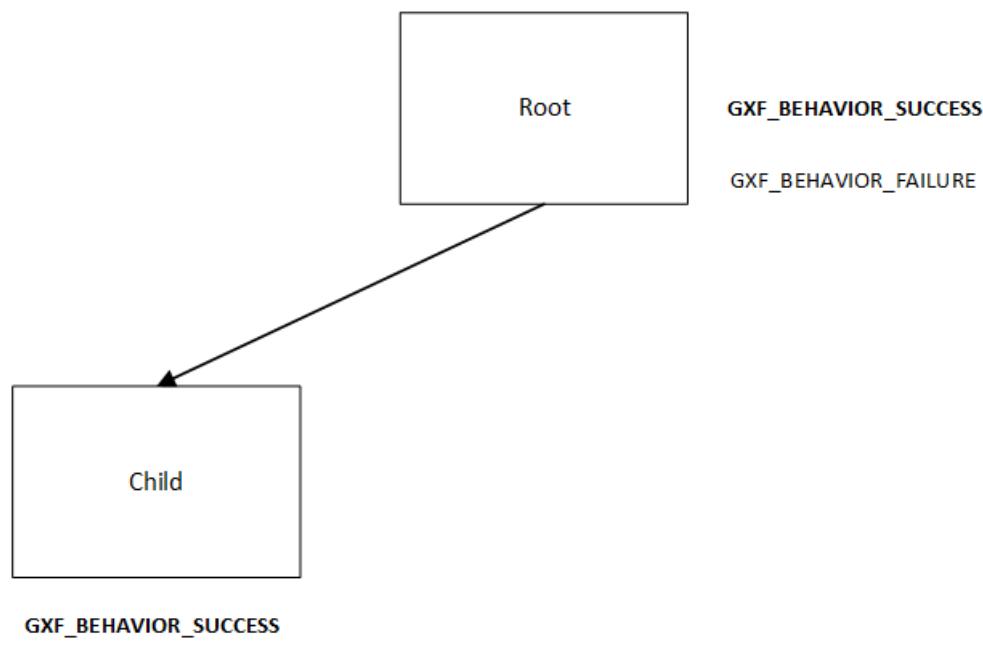
```

Repeat Behavior

Repeats its only child entity. By default, won't repeat when the child entity fails. This can be customized using the parameters below.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
repeat_after_failure	Denotes whether to repeat the child after it has failed.

The diagram below shows a graphical representation of a repeat behavior. The root entity can be configured to repeat the only child to repeat after failure. It succeeds when the child entity succeeds.



A code snippet to configure a repeat behavior is as shown below -

```

name: repeat_knock
components:
- name: repeat_knock_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: repeat_knock_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: repeat_codelet
  type: nvidia::gxf::RepeatBehavior
  parameters:
    s_term: repeat_knock_st
    children: [ knock_on_door/knock_on_door_st ]
    repeat_after_failure: true
  
```

(continues on next page)

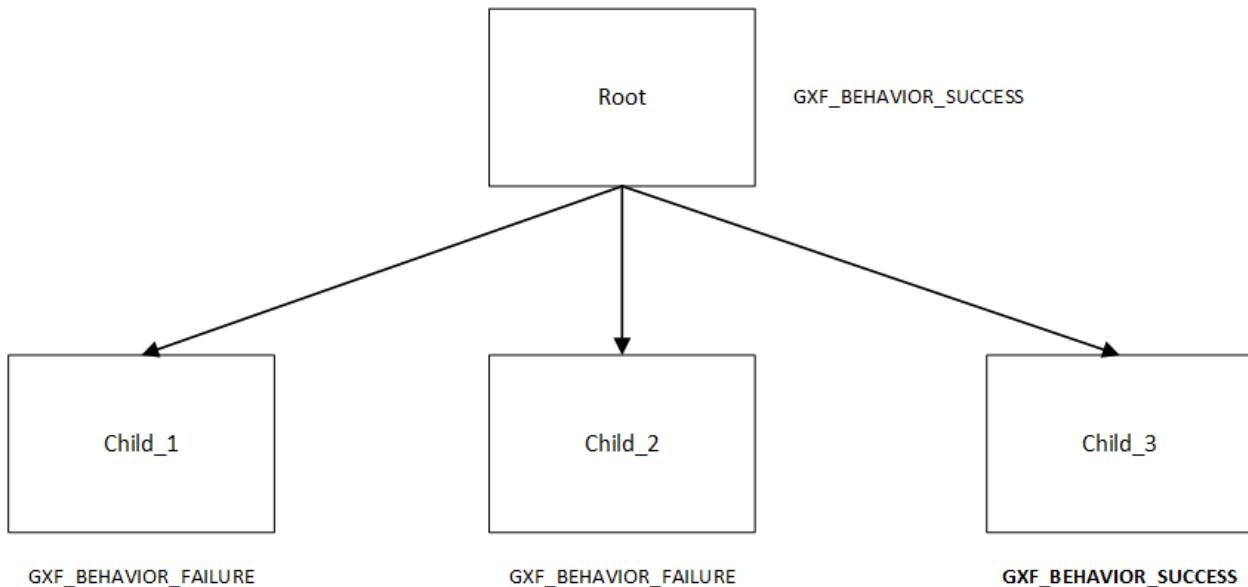
(continued from previous page)

Selector Behavior

Runs all child entities in sequence until one succeeds, then reports success. If all child entities fail (or no child entities are present), this codelet fails.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
children	Child entities

The diagram below shows a graphical representation of a Selector behavior. The root entity starts child_1, child_2 and child_3 in a sequence. Although child_1 and child_2 fail, the root entity will return success since child_3 returns successfully.



A code snippet to configure a selector behavior is as shown below -

```

name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_sel_codelet
  type: nvidia::gxf::SelectorBehavior
  parameters:
    children: [ door_distance/door_distance_st, door_detected/door_detected_st, knock/
    ↪knock_st ]
  
```

(continues on next page)

(continued from previous page)

```

s_term: root_st
---
name: door_distance
components:
- name: door_distance_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: door_distance_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: door_dist
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: []
  s_term: door_distance_st
---

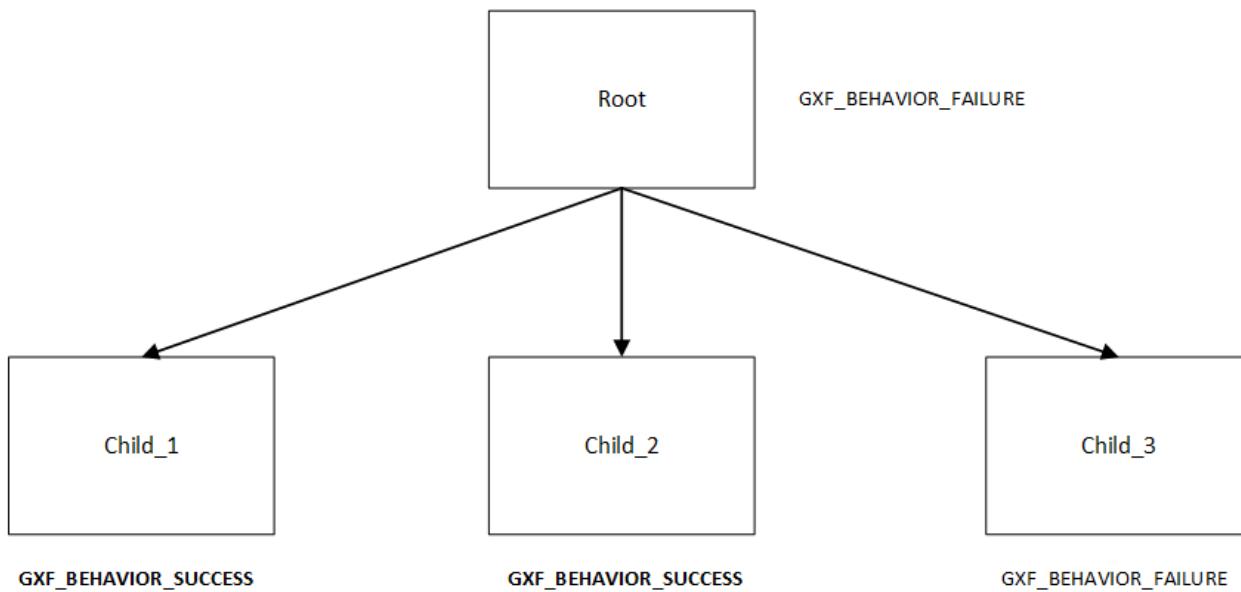
```

Sequence Behavior

Runs its child entities in sequence, in the order in which they are defined. Succeeds when all child entities succeed or fails as soon as one child entity fails.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
children	Child entities

The diagram below shows a graphical representation of a Sequence behavior. The root entity starts child_1, child_2 and child_3 in a sequence. Although child_1 and child_2 pass, the root entity will return failure since child_3 returns failure.



A code snippet to configure a sequence behavior is as shown below -

```
name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_codelet
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: [ child1/child1_st, child2/child2_st ]
    s_term: root_st
```

Switch Behavior

Runs the child entity with the index defined as desired_behavior.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
children	Child entities
desired_behavior	The index of child entity to switch to when this entity runs

In the code snippet shown below, the desired behavior of the root entity is designated to be the the child at index 1. (scene). Hence, that is the entity that is run.

```
name: root
components:
- name: root_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
    max_repeat_count: 0
- name: root_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: true
- name: root_switch_codelet
  type: nvidia::gxf::SwitchBehavior
  parameters:
    children: [ scene/scene_st, ref/ref_st ]
    s_term: root_st
    desired_behavior: 0
---

name: scene
components:
- name: scene_controller
  type: nvidia::gxf::EntityCountFailureRepeatController
  parameters:
```

(continues on next page)

(continued from previous page)

```

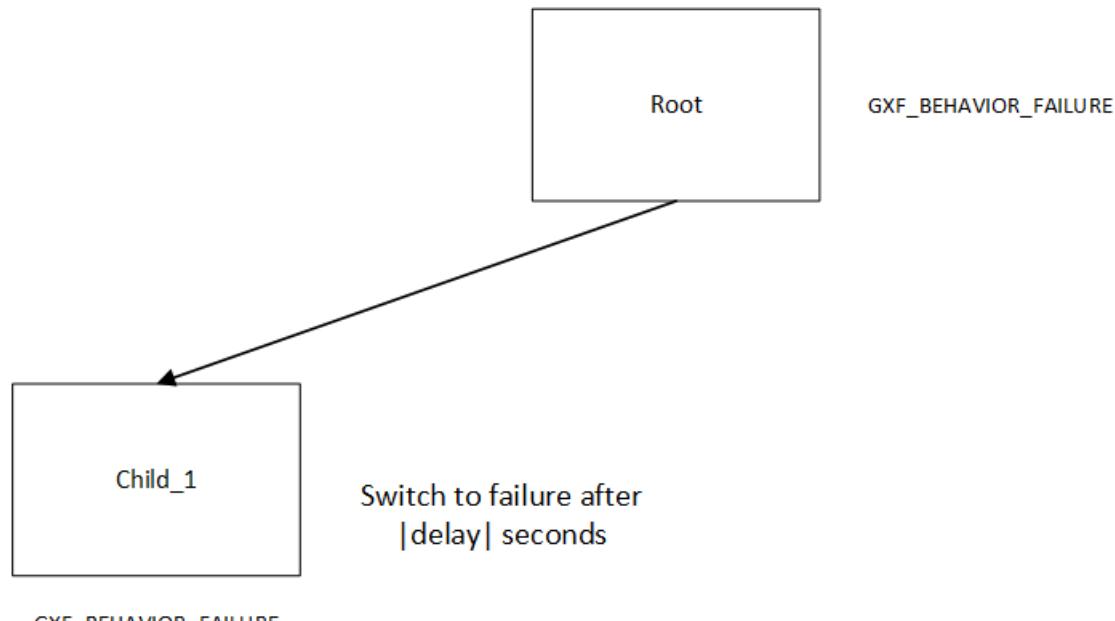
max_repeat_count: 0
- name: scene_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: scene_seq
  type: nvidia::gxf::SequenceBehavior
  parameters:
    children: [ pose/pose_st, det/det_st, seg/seg_st ]
    s_term: scene_st
---
```

Timer Behavior

Waits for a specified amount of time delay and switches to the configured result switch_status afterwards.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
clock	Clock
switch_status	Configured result to switch to after the specified delay
delay	Configured delay

In the diagram shown below, the child entity switches to failure after a configured delay period. The root entity hence returns failure.



A code snippet for the same shown below -

```

name: knock_on_door
components:
- name: knock_on_door_controller
```

(continues on next page)

(continued from previous page)

```

type: nvidia::gxf::EntityCountFailureRepeatController
parameters:
  max_repeat_count: 10
- name: knock_on_door_st
  type: nvidia::gxf::BTSchedulingTerm
  parameters:
    is_root: false
- name: knock
  type: nvidia::gxf::TimerBehavior
  parameters:
    switch_status: 1
    clock: sched/clock
    delay: 1
    s_term: knock_on_door_st
---

```

18.6 GXF Core C APIs

18.6.1 Context

Create context

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

Creates a new GXF context

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

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

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

Create a context from a shared context

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

Creates a new runtime context from shared context.

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

parameter: **shared** A valid GXF shared context.

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

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

Destroy context

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

Destroys a GXF context

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

parameter: `context` A valid GXF context.

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

18.6.2 Extensions

Maximum number of extensions in a context can be 1024.

Load Extensions from a file

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

Loads extension in the given context from file.

parameter: `context` A valid GXF context

parameter: `filename` A valid filename.

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

This function will be deprecated.

Load Extension libraries

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

Loads GXF extension libraries

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

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

Example: — START OF FILE — extensions: - gxf/std/libgxf_std.so - gxf/npp/libgxf_npp.so — END OF FILE

parameter: `context` A valid GXF context

parameter: `filename` A valid filename.

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

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

Loads extensions from manifest file.

parameter: `context` A valid GXF context.

parameter: `filename` A valid filename.

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

This function will be deprecated.

Load Metadata files

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

Loads an extension registration metadata file

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

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

parameter: `context` A valid GXF context.

parameter: `filenames` absolute paths of metadata files.

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

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

Register component

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

Registers a component with a GXF extension

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

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

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

parameter: `context` A valid GXF context

parameter: `tid` The chosen GXF tid

parameter: `name` The type name of the component

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

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

18.6.3 Graph Execution

Loads a list of entities from YAML file

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

parameter: `context` A valid GXF context

parameter: `filename` A valid YAML filename.

parameter: `params_override` An optional array of strings used for override parameters in yaml file.

parameter: `num_overrides` Number of optional override parameter strings.

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

Set the root folder for searching YAML files during loading

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

parameter: `context` A valid GXF context

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

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

Loads a list of entities from YAML text

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

parameter: `context` A valid GXF context

parameter: `text` A valid YAML text.

parameter: `params_override` An optional array of strings used for override parameters in yaml file.

parameter: `num_overrides` Number of optional override parameter strings.

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

Activate all system components

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

parameter: `context` A valid GXF context

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

Deactivate all System components

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

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

Starts the execution of the graph asynchronously

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

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

Interrupt the execution of the graph

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

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

Waits for the graph to complete execution

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

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

Runs all System components and waits for their completion

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

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

18.6.4 Entities

Create an entity

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

Creates a new entity and updates the `eid` to the unique identifier of the newly created entity.
This method will be deprecated.

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

Create a new GXF entity.

Entities are 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

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

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

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

the current execution is finished.

parameter: **context** A valid GXF context

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

returns: GXF error code

Destroy an entity

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

Destroys a previously created entity

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

Note: This function can block for the same reasons as ‘GxfEntityDeactivate’.

parameter: `context` A valid GXF context

parameter: `eid` The returned UID of the created entity

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

Find an entity

```
gxf_result_t GxfEntityFind(gxf_context_t context, const char* name, gxf_uid_t* eid);
```

Finds an entity by its name

parameter: `context` A valid GXF context

parameter: `name` A C string with the name of the entity. Ownership is not transferred.

parameter: `eid` The returned UID of the entity

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

Find all entities

```
gxf_result_t GxfEntityFindAll(gxf_context_t context, uint64_t* num_entities, gxf_uid_t* entities);
```

Finds all entities in the current application

Finds and returns all entity ids for the current application. If more than *max_entities* exist only *max_entities* will be returned. The order and selection of entities returned is arbitrary.

parameter: `context` A valid GXF context

parameter: `num_entities` In/Out: the max number of entities that can fit in the buffer/the number of entities that exist in the application

parameter: `entities` A buffer allocated by the caller for returned UIDs of all entities, with capacity for *num_entities*.

returns: GXF_SUCCESS if the operation was successful, GXF_QUERY_NOT_ENOUGH_CAPACITY if more entities exist in the application than *max_entities*, or otherwise one of the GXF error codes.

Increase reference count of an entity

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

Increases the reference count for an entity by 1.

By default reference counting is disabled for an entity. This means that entities created with ‘GxfEntityCreate’ are not automatically destroyed. If this function is called for an entity with disabled reference count, reference counting is enabled and the reference count is set to 1. Once reference counting is enabled an entity will be automatically destroyed if the reference count reaches zero, or if ‘GxfEntityCreate’ is called explicitly.

parameter: `context` A valid GXF context

parameter: `eid` The UID of a valid entity

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

Decrease reference count of an entity

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

Decreases the reference count for an entity by 1.

See ‘GxfEntityRefCountInc’ for more details on reference counting.

parameter: `context` A valid GXF context

parameter: `eid` The UID of a valid entity

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

Get status of an entity

```
gxf_result_t GxfEntityGetStatus(gxf_context_t context, gxf_uid_t eid,  
gxf_entity_status_t* entity_status);
```

Gets the status of the entity.

See ‘`gxf_entity_status_t`’ for the various status.

parameter: `context` A valid GXF context

parameter: `eid` The UID of a valid entity

parameter: `entity_status` output; status of an entity `eid`

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

Get state of an entity

```
gxf_result_t GxfEntityGetState(gxf_context_t context, gxf_uid_t eid, entity_state_t*  
entity_state);
```

Gets the state of the entity.

See ‘`gxf_entity_status_t`’ for the various status.

parameter: `context` A valid GXF context

parameter: `eid` The UID of a valid entity

parameter: `entity_state` output; behavior status of an entity `eid` used by the behavior tree parent codelet

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

Notify entity of an event

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

Notifies the occurrence of an event and inform the scheduler to check the status of the entity

The entity must have an ‘AsynchronousSchedulingTerm’ scheduling term component and it must be in “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.

18.6.5 Components

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

Get component type identifier

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

Gets the GXF unique type ID (TID) of a component

Get the unique type ID which was used to register the component with GXF. The function expects the fully qualified C++ type name of the component including namespaces.

Example of a valid component type name: “nvidia::gxf::test::PingTx”

parameter: **context** A valid GXF context

parameter: **name** The fully qualified C++ type name of the component

parameter: **tid** The returned TID of the component

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

Get component type name

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

Gets the fully qualified C++ type name GXF component typename

Get the unique typename of the component with which it was registered using one of the GXF_EXT_FACTORY_ADD*() macros

parameter: **context** A valid GXF context

parameter: **tid** The unique type ID (TID) of the component with which the component was registered

parameter: **name** The returned name of the component

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

Get component name

```
gxf_result_t GxfComponentName(gxf_context_t context, gxf_uid_t cid, const char** name);
```

Gets the name of a component

Each component has a user-defined name which was used in the call to ‘GxfComponentAdd’. Usually the name is specified in the GXF application file.

parameter: `context` A valid GXF context

parameter: `cid` The unique object ID (UID) of the component

parameter: `name` The returned name of the component

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

Get unique identifier of the entity of given component

```
gxf_result_t GxfComponentEntity(gxf_context_t context, gxf_uid_t cid, gxf_uid_t* eid);
```

Gets the unique object ID of the entity of a component

Each component has a unique ID with respect to the context and is stored in one entity. This function can be used to retrieve the ID of the entity to which a given component belongs.

parameter: `context` A valid GXF context

parameter: `cid` The unique object ID (UID) of the component

parameter: `eid` The returned UID of the entity

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

Add a new component

```
gxf_result_t GxfComponentAdd(gxf_context_t context, gxf_uid_t eid, gxf_tid_t tid, const char* name, gxf_uid_t* cid);
```

Adds a new component to an entity

An entity can contain multiple components and this function can be used to add a new component to an entity. A component must be added before an entity is activated, or after it was deactivated. Components must not be added to active entities. The order of components is stable and identical to the order in which components are added (see ‘GxfComponentFind’).

parameter: `context` A valid GXF context

parameter: `eid` The unique object ID (UID) of the entity to which the component is added.

parameter: `tid` The unique type ID (TID) of the component to be added to the entity.

parameter: `name` The name of the new component. Ownership is not transferred.

parameter: `cid` The returned UID of the created component

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

Add component to entity interface

```
gxf_result_t GxfComponentAddToInterface(gxf_context_t context, gxf_uid_t eid, gxf_uid_t cid, const char* name);
```

Adds an existing component to the interface of an entity

An entity can holds 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 has 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.

18.6.6 Primitive Parameters

64-bit floating point

Set

```
gxf_result_t GxfParameterSetFloat64(gxf_context_t context, gxf_uid_t uid, const char*  
key, double value);
```

parameter: **context** A valid GXF context.

parameter: **uid** A valid component identifier.

parameter: **key** A valid name of a component to set.

parameter: **value** a double value

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

Get

```
gxf_result_t GxfParameterGetFloat64(gxf_context_t context, gxf_uid_t uid, const char*  
key, double* value);
```

parameter: **context** A valid GXF context.

parameter: **uid** A valid component identifier.

parameter: **key** A valid name of a component to set.

parameter: **value** pointer to get the double value.

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

64-bit signed integer

Set

```
gxf_result_t GxfParameterSetInt64(gxf_context_t context, gxf_uid_t uid, const char* key,  
int64_t value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` 64-bit integer value to set.

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

Get

```
gxf_result_t GxfParameterGetInt64(gxf_context_t context, gxf_uid_t uid, const char* key,  
int64_t* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to get the 64-bit integer value.

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

64-bit unsigned integer

Set

```
gxf_result_t GxfParameterSetUInt64(gxf_context_t context, gxf_uid_t uid, const char* key,  
uint64_t value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` unsigned 64-bit integer value to set.

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

Get

```
gxf_result_t GxfParameterGetUInt64(gxf_context_t context, gxf_uid_t uid, const char* key,  
uint64_t* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to get the unsigned 64-bit integer value.

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

32-bit signed integer

Set

```
gxf_result_t GxfParameterSetInt32(gxf_context_t context, gxf_uid_t uid, const char* key,  
int32_t value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` 32-bit integer value to set.

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

Get

```
gxf_result_t GxfParameterGetInt32(gxf_context_t context, gxf_uid_t uid, const char* key,  
int32_t* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to get the 32-bit integer value.

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

String parameter

Set

```
gxf_result_t GxfParameterSetStr(gxf_context_t context, gxf_uid_t uid, const char* key,  
const char* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` A char array containing value to set.

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

Get

```
gxf_result_t GxfParameterGetStr(gxf_context_t context, gxf_uid_t uid, const char* key,  
const char** value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to a `char*` array to get the value.

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

Boolean

Set

```
gxf_result_t GxfParameterSetBool(gxf_context_t context, gxf_uid_t uid, const char* key,  
bool value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` A boolean value to set.

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

Get

```
gxf_result_t GxfParameterGetBool(gxf_context_t context, gxf_uid_t uid, const char* key,  
bool* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to get the boolean value.

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

Handle

Set

```
gxf_result_t GxfParameterSetHandle(gxf_context_t context, gxf_uid_t uid, const char* key,  
gxf_uid_t cid);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `cid` Unique identifier to set.

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

Get

```
gxf_result_t GxfParameterGetHandle(gxf_context_t context, gxf_uid_t uid, const char* key,  
gxf_uid_t* cid);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` Pointer to a unique identifier to get the value.

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

18.6.7 Vector Parameters

To set or get the vector parameters of a component, users can use the following C-APIs for various data types:

Set 1-D Vector Parameters

Users can call `gxf_result_t GxfParameterSet1D("DataType"Vector(gxf_context_t context, gxf_uid_t uid, const char* key, data_type* value, uint64_t length)`

`value` should point to an array of the data to be set of the corresponding type. The size of the stored array should match the `length` argument passed.

See the table below for all the supported data types and their corresponding function signatures.

parameter: `key` The name of the parameter

parameter: `value` The value to set of the parameter

parameter: `length` The length of the vector parameter

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

Table 18.1: Supported Data Types to Set 1D Vector Parameters

Function Name	data_type
GxfParameterSet1DFloat64Vector(...)	double
GxfParameterSet1DInt64Vector(...)	int64_t
GxfParameterSet1DUInt64Vector(...)	uint64_t
GxfParameterSet1DInt32Vector(...)	int32_t

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 18.2: Supported Data Types to Set 2D Vector Parameters

Function Name	data_type
GxfParameterSet2DFloat64Vector(...)	double
GxfParameterSet2DInt64Vector(...)	int64_t
GxfParameterSet2DUInt64Vector(...)	uint64_t
GxfParameterSet2DInt32Vector(...)	int32_t

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

Function Name	data_type
GxfParameterGet1DFloat64Vector(...)	double
GxfParameterGet1DInt64Vector(...)	int64_t
GxfParameterGet1DUInt64Vector(...)	uint64_t
GxfParameterGet1DInt32Vector(...)	int32_t

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 18.4: 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

18.6.8 Information Queries

Get Meta Data about the GXF Runtime

```
gxf_result_t GxfRuntimeInfo(gxf_context_t context, gxf_runtime_info* info);
```

parameter: `context` A valid GXF context.

parameter: `info` pointer to `gxf_runtime_info` object to get the meta data.

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

Get description and list of components in loaded Extension

```
gxf_result_t GxfExtensionInfo(gxf_context_t context, gxf_tid_t tid, gxf_extension_info_t* info);
```

parameter: `context` A valid GXF context.

parameter: `tid` The unique identifier of the extension.

parameter: `info` pointer to `gxf_extension_info_t` object to get the meta data.

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

Get description and list of parameters of Component

```
gxf_result_t GxfComponentInfo(gxf_context_t context, gxf_tid_t tid, gxf_component_info_t* info);
```

Note: Parameters are only available after at least one instance is created for the Component.

parameter: `context` A valid GXF context.

parameter: `tid` The unique identifier of the component.

parameter: `info` pointer to `gxf_component_info_t` object to get the meta data.

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

Get parameter type description

Gets a string describing the parameter type

```
const char* GxfParameterTypeStr(gxf_parameter_type_t param_type);
```

parameter: `param_type` Type of parameter to get info about.

returns: C-style string description of the parameter type.

Get flag type description

Gets a string describing the flag type

```
const char* GxfParameterFlagTypeStr(gxf_parameter_flags_t flag_type);
```

parameter: `flag_type` Type of flag to get info about.

returns: C-style string description of the flag type.

Get parameter description

Gets description of specific parameter. Fails if the component is not instantiated yet.

```
gxf_result_t GxfGetParameterInfo(gxf_context_t context, gxf_tid_t cid, const char* key, gxf_parameter_info_t* info);
```

parameter: `context` A valid GXF context.

parameter: `cid` The unique identifier of the component.

parameter: `key` The name of the parameter.

parameter: `info` Pointer to a `gxf_parameter_info_t` object to get the value.

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

Redirect logs to a file

Redirect console logs to the provided file.

```
gxf_result_t GxfGetParameterInfo(gxf_context_t context, FILE* fp);
```

parameter: `context` A valid `GXF` context.

parameter: `fp` File path for the redirected logs.

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

18.6.9 Miscellaneous

Get string description of error

```
const char* GxfResultStr(gxf_result_t result);
```

Gets a string describing an `GXF` error code.

The caller does not get ownership of the return C string and must not delete it.

parameter: `result` A `GXF` error code

returns: A pointer to a C string with the error code description.

18.7 CudaExtension

Extension for CUDA operations.

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

18.7.1 Components

nvidia::gxf::CudaStream

Holds and provides access to native `cudaStream_t`.

`nvidia::gxf::CudaStream` handle must be allocated by `nvidia::gxf::CudaStreamPool`. Its lifecycle is valid until explicitly recycled through `nvidia::gxf::CudaStreamPool.releaseStream()` or implicitly until `nvidia::gxf::CudaStreamPool` is deactivated.

You may call `stream()` to get the native `cudaStream_t` handle, and to submit GPU operations. After the submission, GPU takes over the input tensors/buffers and keeps them in use. To prevent host carelessly releasing these in-use buffers, CUDA Codelet needs to call `record(event, input_entity, sync_cb)` to extend `input_entity`'s lifecycle until GPU completely consumes it. Alternatively, you may call `record(event, event_destroy_cb)` for native `cudaEvent_t` operations and free in-use resource via `event_destroy_cb`.

It is required to have a `nvidia::gxf::CudaStreamSync` in the graph pipeline after all the CUDA operations. See more details in `nvidia::gxf::CudaStreamSync`

- Component ID: 5683d692-7884-11eb-9338-c3be62d576be
- Defined in: `gxf/cuda/cuda_stream.hpp`

nvidia::gxf::CudaStreamId

Holds CUDA stream Id to deduce `nvidia::gxf::CudaStream` handle.

`stream_cid` should be `nvidia::gxf::CudaStream` component id.

- Component ID: 7982aeac-37f1-41be-ade8-6f00b4b5d47c
- Defined in: `gxf/cuda/cuda_stream_id.hpp`

nvidia::gxf::CudaEvent

Holds and provides access to native `cudaEvent_t` handle.

When a `nvidia::gxf::CudaEvent` is created, you'll need to initialize a native `cudaEvent_t` through `init(flags, dev_id)`, or set third party event through `initWithEvent(event, dev_id, free_fnc)`. The event keeps valid until `deinit` is called explicitly otherwise gets recycled in destructor.

- Component ID: f5388d5c-a709-47e7-86c4-171779bc64f3
- Defined in: `gxf/cuda/cuda_event.hpp`

nvidia::gxf::CudaStreamPool

CudaStream allocation.

You must explicitly call `allocateStream()` to get a valid `nvidia::gxf::CudaStream` handle. This component would hold all the its allocated `nvidia::gxf::CudaStream` entities until `releaseStream(stream)` is called explicitly or the `CudaStreamPool` component is deactivated.

- Component ID: 6733bf8b-ba5e-4fae-b596-af2d1269d0e7
- Base Type: `nvidia::gxf::Allocator`

Parameters

dev_id

GPU device id.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_INT32`
- Default Value: 0

stream_flags

Flag values to create CUDA streams.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 0

stream_priority

Priority values to create CUDA streams.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 0

reserved_size

User-specified file name without extension.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 1

max_size

Maximum Stream Size.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 0, no limitation.

nvidia::gxf::CudaStreamSync

Synchronize all CUDA streams which are carried by message entities.

This codelet is required to get connected in the graph pipeline after all CUDA ops codelets. When a message entity is received, it would find all of the `nvidia::gxf::CudaStreamId` in that message, and extract out each `nvidia::gxf::CudaStream`. With each `CudaStream` handle, it synchronizes all previous `nvidia::gxf::CudaStream.record()` events, along with all submitted GPU operations before this point.

Note: `CudaStreamSync` must be set in the graph when `nvidia::gxf::CudaStream.record()` is used, otherwise it may cause memory leak.

- Component ID: 0d1d8142-6648-485d-97d5-277eed00129c

- Base Type: nvidia::gxf::Codelet

Parameters

rx

Receiver to receive all messages carrying nvidia::gxf::CudaStreamId.

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

tx

Transmitter to send messages to downstream.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Transmitter

18.8 MultimediaExtension

Extension for multimedia related data types, interfaces and components in GXF Core.

- UUID: 6f2d1afc-1057-481a-9da6-a5f61fed178e
- Version: 2.0.0
- Author: NVIDIA
- License: LICENSE

18.8.1 Components

nvidia::gxf::AudioBuffer

AudioBuffer is similar to Tensor component in the standard extension and holds memory and metadata corresponding to an audio buffer.

- Component ID: a914cac6-5f19-449d-9ade-8c5cdcebe7c3

AudioBufferInfo structure captures the following metadata:

Field	Description
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

18.9 NetworkExtension

Extension for communications external to a computation graph.

- UUID: f50665e5-ade2-f71b-de2a-2380614b1725
- Version: 1.0.0
- Author: NVIDIA
- License: LICENSE

18.9.1 Interfaces

18.9.2 Components

nvidia::gxf::TcpClient

Codelet that functions as a client in a TCP connection.

- Component ID: 9d5955c7-8fda-22c7-f18f-ea5e2d195be9
- Base Type: nvidia::gxf::Codelet

Parameters

receivers

List of receivers to receive entities from.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_CUSTOM`
- Custom Type: `std::vector<nvidia::gxf::Handle<nvidia::gxf::Receiver>>`

transmitters

List of transmitters to publish entities to.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_CUSTOM`
- Custom Type: `std::vector<nvidia::gxf::Handle<nvidia::gxf::Transmitter>>`

serializers

List of component serializers to serialize and de-serialize entities.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_CUSTOM`
- Custom Type: `std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>`

address

Address of TCP server.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_STRING`

port

Port of TCP server.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_INT32`

timeout_ms

Time in milliseconds to wait before retrying connection to TCP server.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

maximum_attempts

Maximum number of attempts for I/O operations before failing.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

nvidia::gxf::TcpServer

Codelet that functions as a server in a TCP connection.

- Component ID: a3e0e42d-e32e-73ab-ef83-fbb311310759
- Base Type: nvidia::gxf::Codelet

Parameters**receivers**

List of receivers to receive entities from.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::Receiver>>

transmitters

List of transmitters to publish entities to.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::Transmitter>>

serializers

List of component serializers to serialize and de-serialize entities.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: `std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>`

address

Address of TCP server.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

port

Port of TCP server.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32

timeout_ms

Time in milliseconds to wait before retrying connection to TCP client.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

maximum_attempts

Maximum number of attempts for I/O operations before failing.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

18.10 SerializationExtension

Extension for serializing messages.

- UUID: bc573c2f-89b3-d4b0-8061-2da8b11fe79a
- Version: 2.0.0
- Author: NVIDIA
- License: LICENSE

18.10.1 Interfaces

nvidia::gxf::ComponentSerializer

Interface for serializing components.

- Component ID: 8c76a828-2177-1484-f841-d39c3fa47613
- Base Type: nvidia::gxf::Component
- Defined in: gxf/serialization/component_serializer.hpp

18.10.2 Components

nvidia::gxf::EntityRecorder

Serializes incoming messages and writes them to a file.

- Component ID: 9d5955c7-8fda-22c7-f18f-ea5e2d195be9
- Base Type: nvidia::gxf::Codelet

Parameters

receiver

Receiver channel to log.

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

serializers

List of component serializers to serialize entities.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>

directory

Directory path for storing files.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

basename

User specified file name without extension.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_STRING

flush_on_tick

Flushes output buffer on every tick when true.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

nvidia::gxf::EntityReplayer

De-serializes and publishes messages from a file.

- Component ID: fe827c12-d360-c63c-8094-32b9244d83b6
- Base Type: nvidia::gxf::Codelet

Parameters

transmitter

Transmitter channel for replaying entities.

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

serializers

List of component serializers to serialize entities.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_CUSTOM
- Custom Type: `std::vector<nvidia::gxf::Handle<nvidia::gxf::ComponentSerializer>>`

directory

Directory path for storing files.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

batch_size

Number of entities to read and publish for one tick.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

ignore_corrupted_entities

If an entity could not be de-serialized, it is ignored by default; otherwise a failure is generated.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

nvidia::gxf::StdComponentSerializer

Serializer for Timestamp and Tensor components.

- Component ID: c0e6b36c-39ac-50ac-ce8d-702e18d8bff7
- Base Type: nvidia::gxf::ComponentSerializer

Parameters

allocator

Memory allocator for tensor components.

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

18.11 StandardExtension

Most commonly used interfaces and components in Gxf Core.

- UUID: 8ec2d5d6-b5df-48bf-8dee-0252606fdd7e
- Version: 2.1.0
- Author: NVIDIA
- License: LICENSE

18.11.1 Interfaces

nvidia::gxf::Codelet

Interface for a component which can be executed to run custom code.

- Component ID: 5c6166fa-6eed-41e7-bbf0-bd48cd6e1014
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/codelet.hpp

nvidia::gxf::Clock

Interface for clock components which provide time.

- Component ID: 779e61c2-ae70-441d-a26c-8ca64b39f8e7
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/clock.hpp

nvidia::gxf::System

Component interface for systems which are run as part of the application run cycle.

- Component ID: d1febca1-80df-454e-a3f2-715f2b3c6e69
- Base Type: nvidia::gxf::Component

nvidia::gxf::Queue

Interface for storing entities in a queue.

- Component ID: 792151bf-3138-4603-a912-5ca91828dea8
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/queue.hpp

nvidia::gxf::Router

Interface for classes which are routing messages in and out of entities.

- Component ID: 8b317aad-f55c-4c07-8520-8f66db92a19e
- Defined in: gxf/std/router.hpp

nvidia::gxf::Transmitter

Interface for publishing entities.

- Component ID: c30cc60f-0db2-409d-92b6-b2db92e02cce
- Base Type: nvidia::gxf::Queue
- Defined in: gxf/std/transmitter.hpp

nvidia::gxf::Receiver

Interface for receiving entities.

- Component ID: a47d2f62-245f-40fc-90b7-5dc78ff2437e
- Base Type: nvidia::gxf::Queue
- Defined in: gxf/std/receiver.hpp

nvidia::gxf::Scheduler

A simple poll-based single-threaded scheduler which executes codelets.

- Component ID: f0103b75-d2e1-4d70-9b13-3fe5b40209be
- Base Type: nvidia::gxf::System
- Defined in: nvidia/gxf/system.hpp

nvidia::gxf::SchedulingTerm

Interface for terms used by a scheduler to determine if codelets in an entity are ready to step.

- Component ID: 184d8e4e-086c-475a-903a-69d723f95d19
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/scheduling_term.hpp

nvidia::gxf::Allocator

Provides allocation and deallocation of memory.

- Component ID: 3cdd82d0-2326-4867-8de2-d565dbe28e03
- Base Type: nvidia::gxf::Component
- Defined in: nvidia/gxf/allocator.hpp

nvidia::gxf::Monitor

Monitors entities during execution.

- Component ID: 9ccf9421-b35b-8c79-e1f0-97dc23bd38ea
- Base Type: nvidia::gxf::Component
- Defined in: nvidia/gxf/monitor.hpp

18.11.2 Components

nvidia::gxf::RealtimeClock

A real-time clock which runs based off a system steady clock.

- Component ID: 7b170b7b-cf1a-4f3f-997c-bfea25342381
- Base Type: nvidia::gxf::Clock

Parameters

initial_time_offset

The initial time offset used until 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-ff63-4179-9200-3fc8dc38673
- 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-d7b5-6e7cadc704e8

float

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

double

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

bool

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

VIDEO PIPELINE LATENCY TOOL

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

19.1 Requirements

19.1.1 Hardware

The latency measurement tool requires the use of a Holoscan 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).

19.1.2 Software

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

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

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

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

The following is optional to enable AJA Video Systems support:

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

19.2 Installation

19.2.1 Downloading the Source

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

```
$ git clone https://github.com/nvidia-holoscan/holoscan-perf-tools.git
```

19.2.2 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
```

19.2.3 Building

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

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

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

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

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

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

Enabling DeepStream Support

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

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

Enabling AJA Support

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

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

19.3 Example Configurations

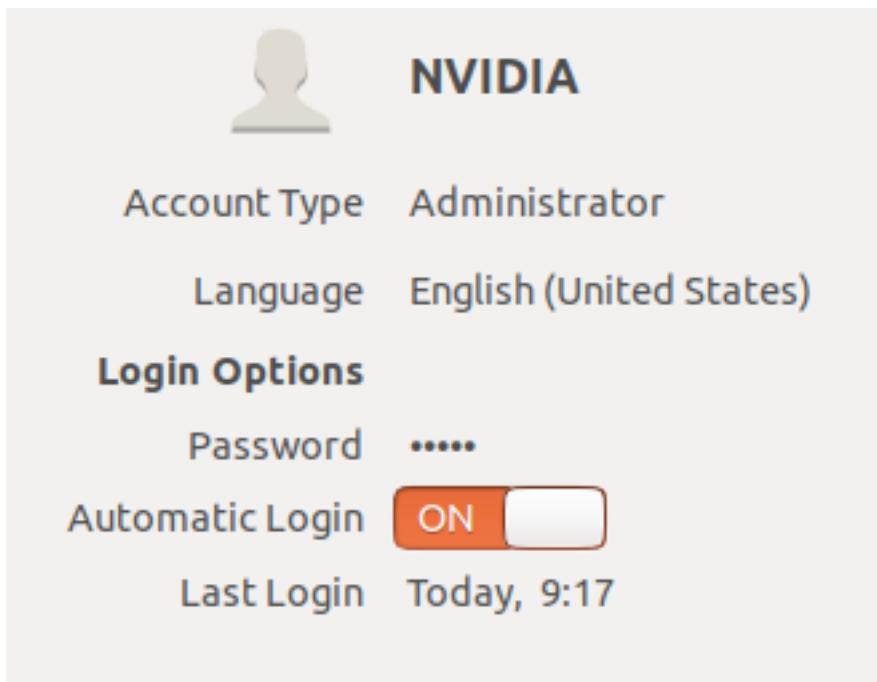
Note: When testing a configuration that outputs from the GPU, the tool currently only supports a display-less environment in which the loopback cable is the only cable attached to the GPU. Because of this, any tests that output from the GPU must be performed using a remote connection such as SSH from 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
```

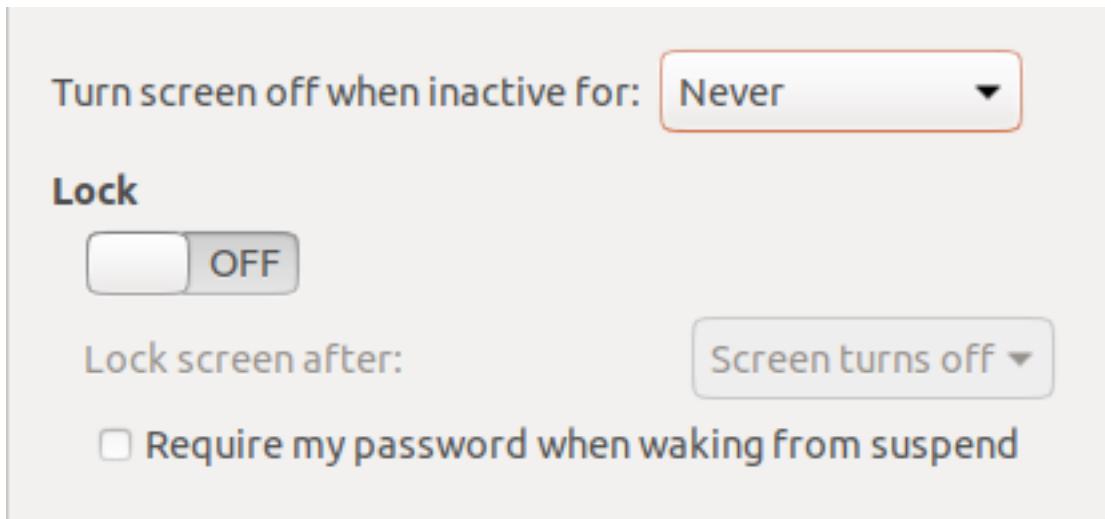
It is also required that the system is logged into the desktop and that the system does not sleep or lock when the latency tool is being used. This can be done by temporarily attaching a display to the system to do the following:

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](#)).

19.3.1 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. 19.1: 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
```

19.3.2 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. 19.2: 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
```

19.3.3 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. 19.3: 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
```

19.4 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.

19.4.1 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

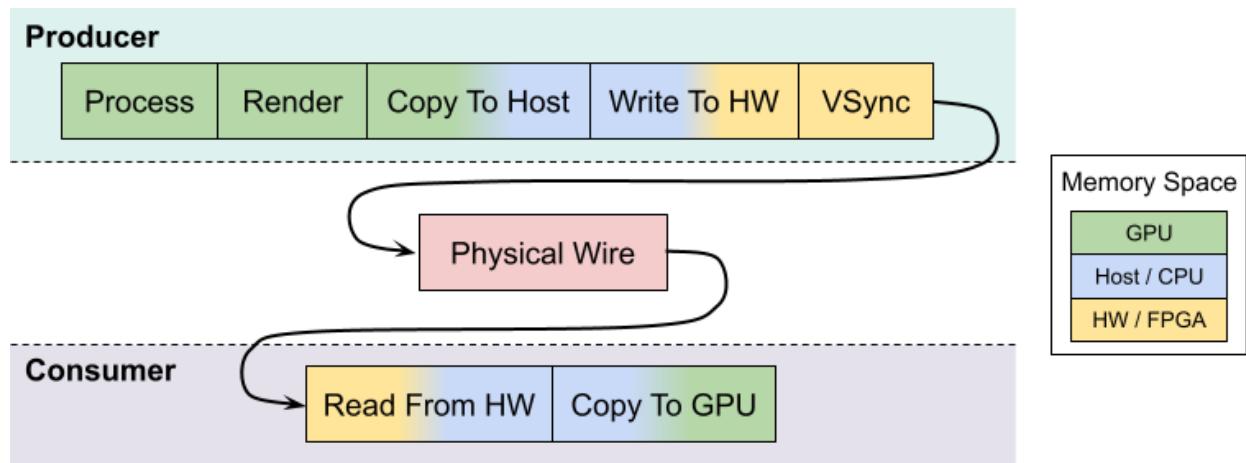


Fig. 19.4: Latency Tool Frame Lifespan (RDMA Disabled)

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.

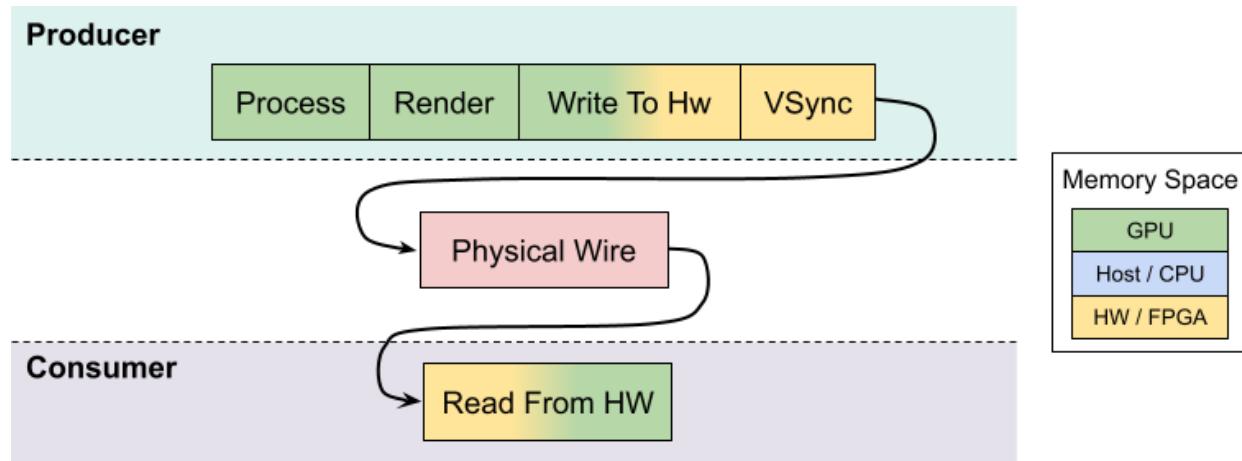


Fig. 19.5: Latency Tool Frame Lifespan (RDMA Enabled)

19.4.2 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)

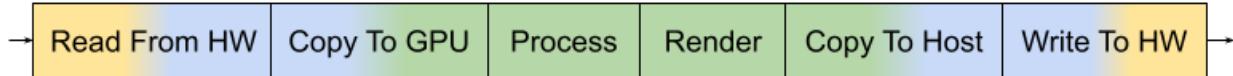


Fig. 19.6: Real Application Frame Lifespan

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)

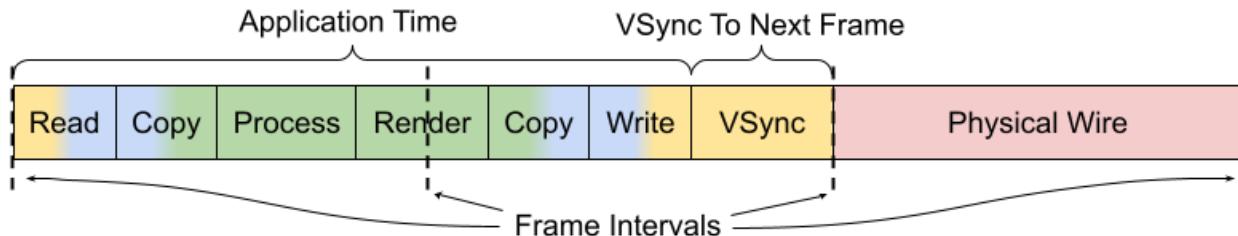


Fig. 19.7: Final Estimated Latency with VSync and Physical Wire Time

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

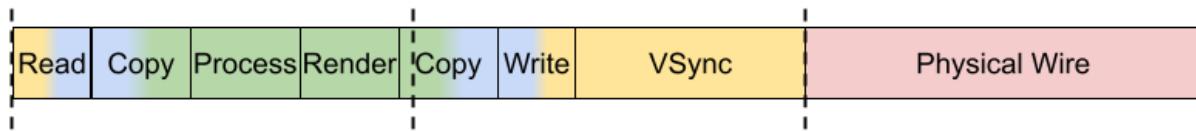
```

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).

19.4.3 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)

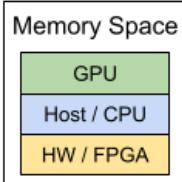
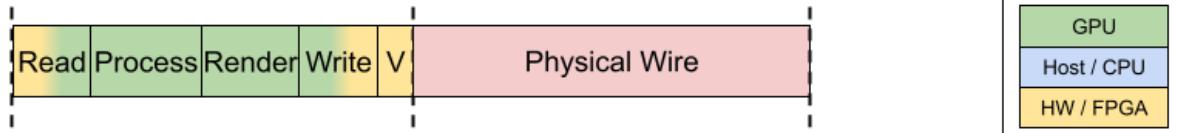


Fig. 19.8: Reducing Latency With RDMA

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

```

19.4.4 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.

19.5 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.

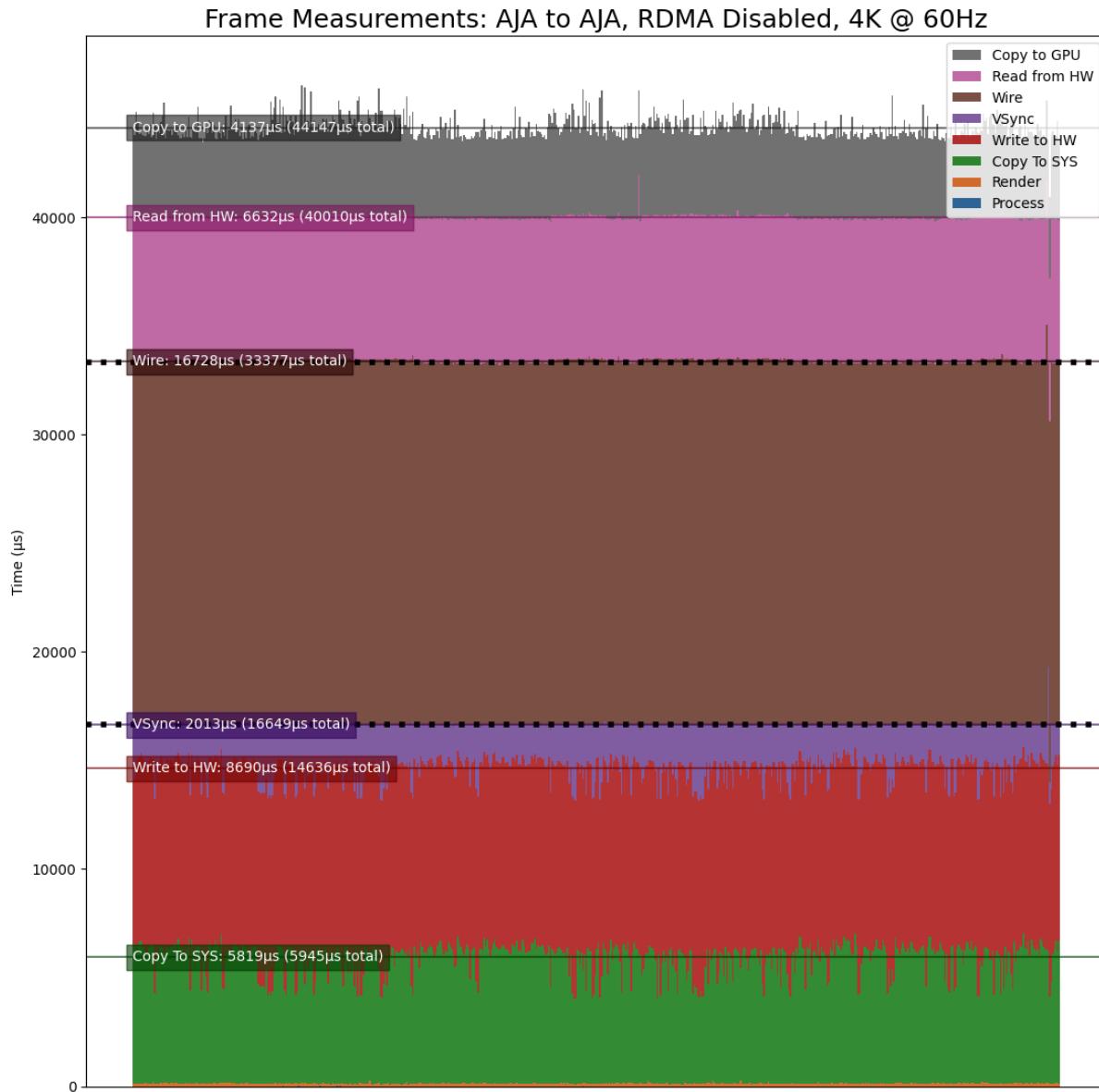
For example, if the latencies are measured using:

```
$ ./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
```

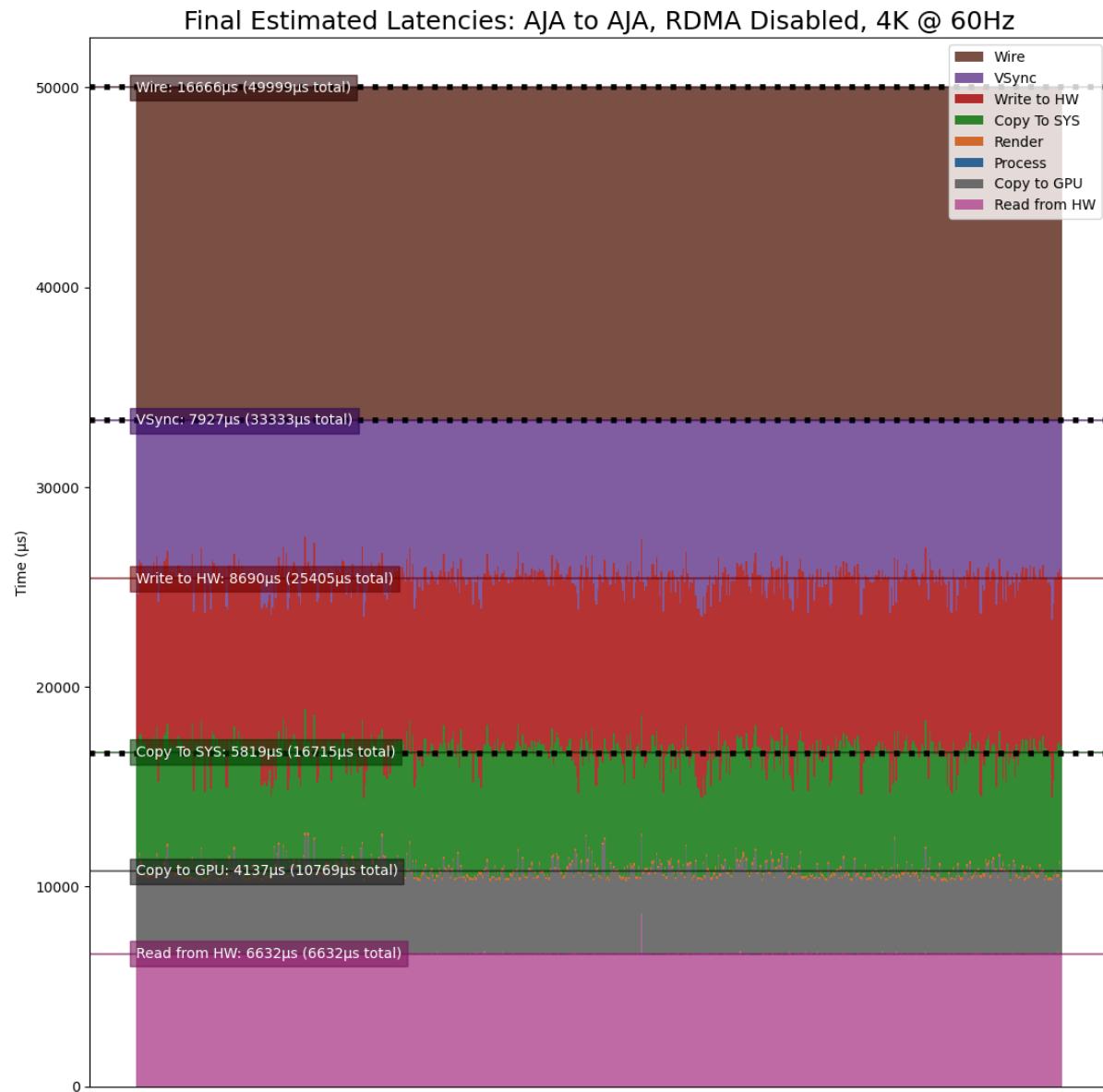
The graph can also be output to a PNG image file instead of opening a window on the desktop by providing the `--png {file}` option to the script. The following shows an example graph for an AJA to AJA measurement of a 4K @ 60Hz stream with RDMA disabled (as shown as an example in [Interpreting The Results](#),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



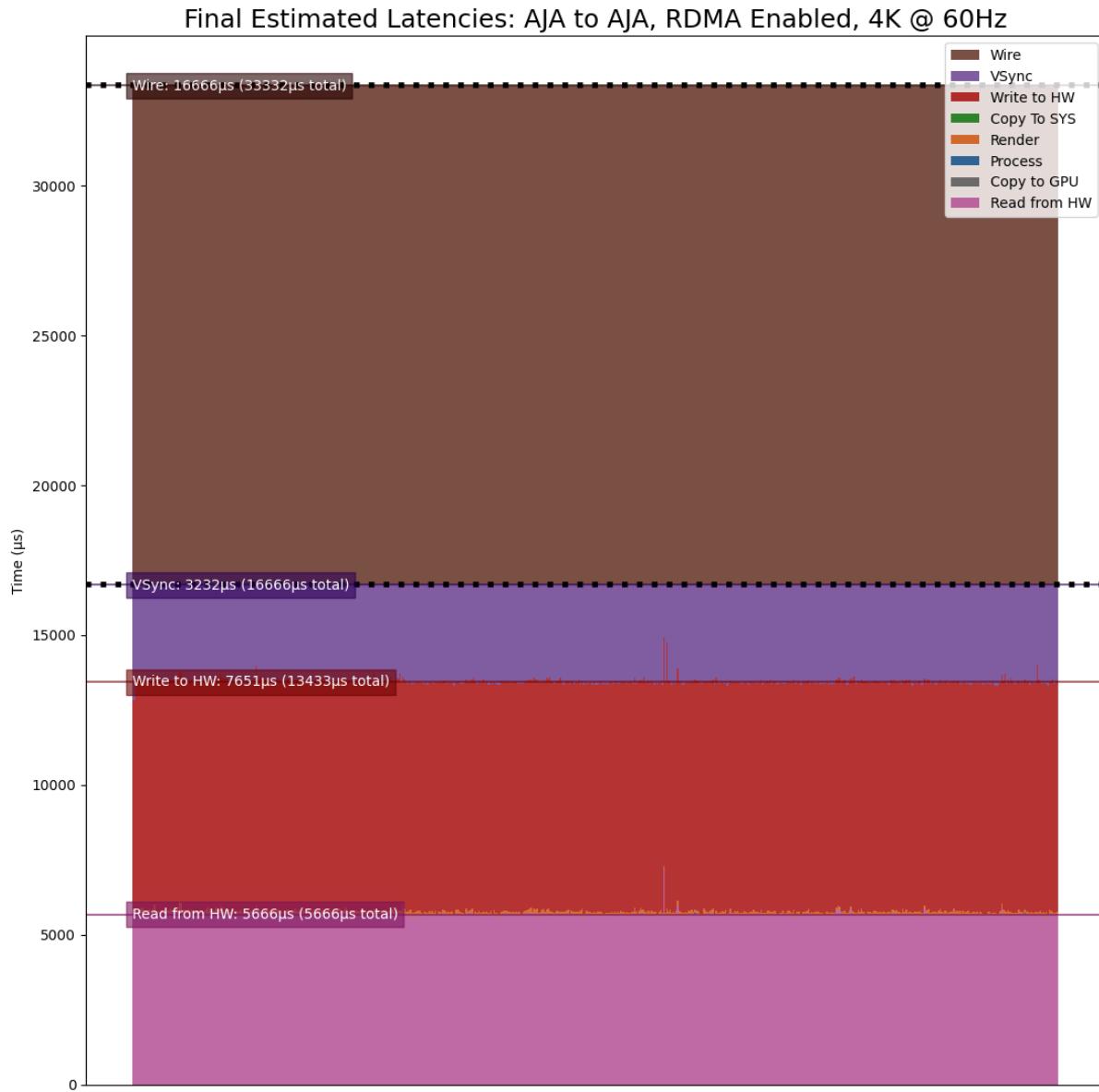
Note that this is showing the times for 600 frames, from left to right, with the life of each frame beginning at the bottom and ending at the top. The dotted black lines represent frame VSync intervals (every 16666us).

The above example graphs the times directly as measured by the tool. To instead generate a graph for the **Final Estimated Latencies** as described above in [Interpreting The Results](#), the `--estimate` flag can be provided to the script. As is done by the latency tool when it reports the estimated latencies, this reorders the producer and consumer steps then adds a VSync interval followed by the physical wire latency.

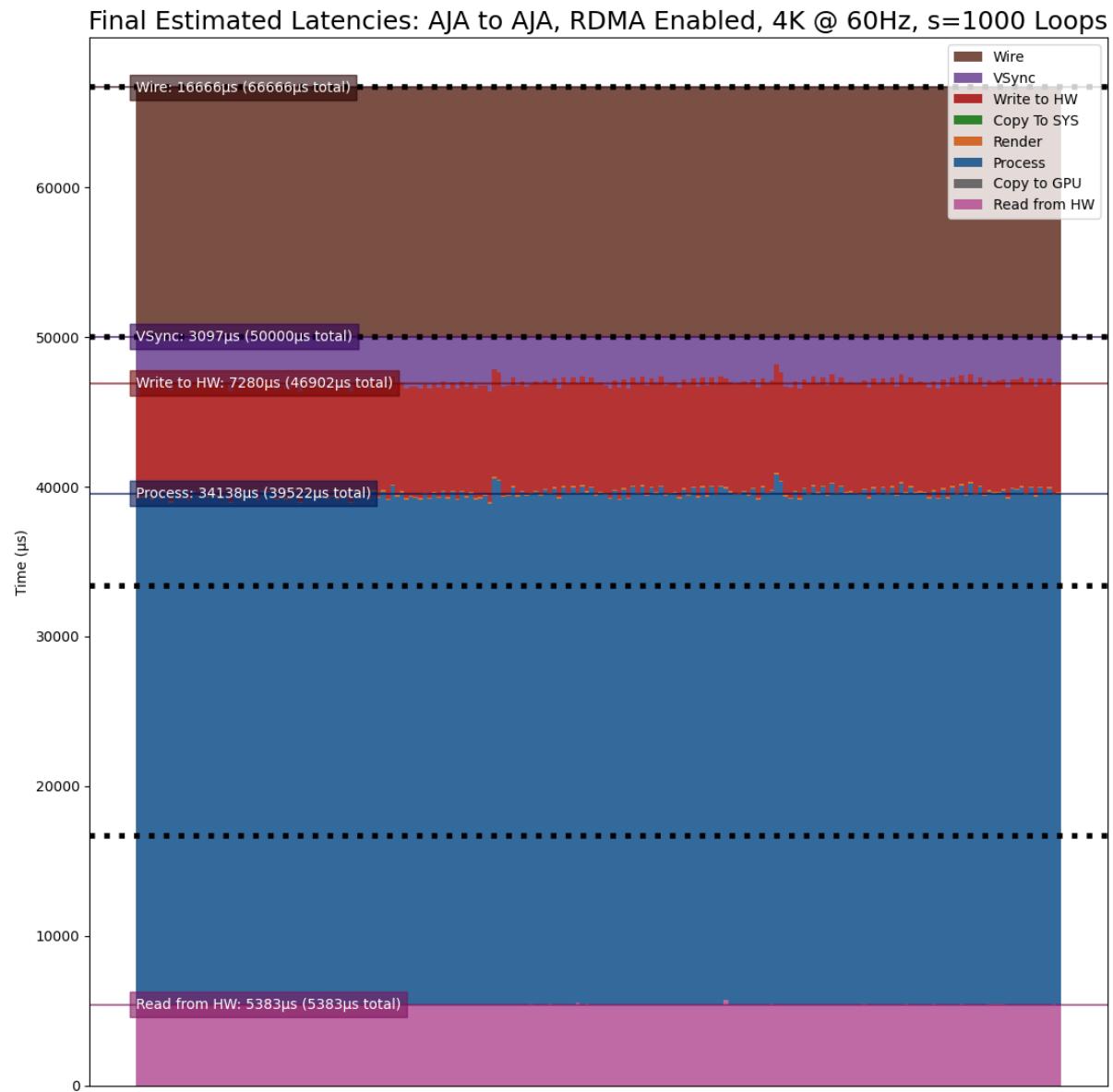
The following graphs the **Final Estimated Latencies** using the same data file as the graph above. Note that this shows a total of 3 frames of expected latency.



For the sake of comparison, the following graph shows the same test but with RDMA enabled. Note that the **Copy To GPU** and **Copy To SYS** times are now zero due to the use of RDMA, and this now shows just 2 frames of expected latency.



As a final example, the following graph duplicates the above test with RDMA enabled, but adds roughly 34ms of additional GPU processing time (-s 1000) to the pipeline to produce a final estimated latency of 4 frames.



19.6 Producers

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

19.6.1 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.

19.6.2 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.

19.6.3 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 \(SDI\)](#).

19.7 Consumers

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

19.7.1 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 Holoscan Developer Kits.

V4L2 Consumer Notes:

- The onboard HDMI capture card is locked to a specific frame resolution and frame rate (1080p @ 60Hz), and so `1080` is the only supported format when using this consumer.
- The `-c.device {device}` parameter can be used to specify the path to the device that is being used to capture the frames (defaults to `/dev/video0`).
- The V4L2 API does not support RDMA, and so the `c.rdma` option is ignored.

19.7.2 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.

19.7.3 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).

19.8 Troubleshooting

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

- 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.

- Problem:** An error like the following is output:

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

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

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

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

Solution: Try the following:

- Ensure that no other displays are connected to the GPU.
- 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*.