NVIDIA Holoscan SDK v2.0.0

Table of contents

List of Figures

List of Tables

Introduction

- Overview
- Relevant Technologies
- Getting Started

Setup

- SDK Installation
- Additional Setup
- Third Party Hardware Setup

Using the SDK

- Holoscan Core Concepts
- Holoscan by Example
- Creating an Application
- Creating a Distributed Application
- Packaging Holoscan Applications
- Creating Operators
- Logging
- Debugging

Operators

- Built-in Operators and Extensions
- Visualization
- Inference

Components

- Schedulers
- Conditions
- Resources

API

- Holoscan C++ API
- Holoscan Python API

App Packaging

- Holoscan Application Package Specification (HAP)
- Holoscan CLI
- Application Runner Configuration

Graph Execution Framework

- GXF Core concepts
- Holoscan and GXF
- GXF by Example
- Using Holoscan Operators in GXF Applications
- GXF User Guide

Performance Tools

- Data Flow Tracking
- Video Pipeline Latency Tool

Links

- Developer page
- Support Forum
- NGC Containers
- Github Repository
- User Guide PDF
- Previous releases

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 `<a href="https://developer.nvidia.com/industries/healthcare">Clara</a>` 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. **Repository of Operators and Applications**

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

6. **Tooling to Package and Deploy Applications**

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

7. **Performance tools**

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

- a Video Pipeline Latency Measurement Tool to measure and estimate the total end-to-end latency of a video streaming application including the video capture,

processing, and output using various hardware and software components that are supported by the NVIDIA Developer Kits.

- the Data Flow Tracking feature to profile your application and analyze the data flow between operators in its graph.

8. **Documentation**

The Holoscan SDK documentation is composed of:

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

Relevant Technologies

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

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

Rivermax and GPUDirect RDMA

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

> **Note**
>
> NVIDIA is 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.

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.

TensorRT Optimized Inference

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

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

Interoperability between CUDA and rendering frameworks

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

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

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.

Unified Communications X

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

Getting Started with Holoscan

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

1. Choose your platform

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

2. Setup the SDK and your platform

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

3. Learn the framework

1. Start with the Core Concepts to understand the technical terms used in this guide, and the overall behavior of the framework.

2. Learn how to use the SDK in one of two ways (or both) based on your preference:

    1. Going through the Holoscan by Example tutorial which will build your knowledge step-by-step by going over concrete minimal examples in the SDK. You can refer to each example source code and run instructions to inspect them and run them as you go.

    2. Going through the condensed documentations that should cover all capabilities of the SDK using minimal mock code snippets, including creating an application, creating a distributed application, and creating operators.

4. Understand the reusable capabilities of the SDK

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

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

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

- Just like the SDK operators, the HoloHub operators can be used in your own Holoscan applications.

- The HoloHub sample applications can be used as reference implementations to complete the examples available in the SDK.

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

5. Write and Run your own application

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

6. Master the details

- Expand your understanding of the framework with details on the logging utility or the data flow tracking benchmarking tool.

- Learn more details on the configurable components that control the execution of your application, like [Schedulers], [Conditions], and [Resources]. (Advanced) These components are part on the GXF execution backend, hence the **Graph Execution Framework** section at the bottom of this guide if deep understanding of the application execution is needed.

SDK Installation

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

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

Prerequisites

Ingested Tab Module

- For RDMA Support, follow the instructions in the Enabling RDMA section.
- Additional software dependencies might be needed based on how you choose to install the SDK (see section below).
- Refer to the Additional Setup and Third-Party Hardware Setup sections for additional prerequisites.

Install the SDK

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

Instructions

Ingested Tab Module

Not sure what to choose?

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

	NGC dev Container	Debian Package	Python Wheels
Runtime libraries	**Included**	**Included**	**Included**
Python module	3.10	3.10	**3.8 to 3.11**
C++ headers and CMake config	**Included**	**Included**	N/A
Examples (+ source)	**Included**	**Included**	retrieve from GitHub
Sample datasets	**Included**	retrieve from NGC	retrieve from NGC
CUDA runtime [1]	**Included**	automatically [2] installed	require manual installation

NPP support [3]	**Included**	automatically [2] installed	require manual installation
TensorRT support [4]	**Included**	automatically [2] installed	require manual installation
Vulkan support [5]	**Included**	automatically [2] installed	require manual installation
V4L2 support [6]	**Included**	automatically [2] installed	require manual installation
Torch support [7]	**Included**	require manual [8] installation	require manual [8] installation
ONNX Runtime support [9]	**Included**	require manual [10] installation	require manual [10] installation
MOFED support [11]	**User space included** Install kernel drivers on the host	require manual installation	require manual installation
CLI support	**Included**	needs docker w/ buildx plugin	needs docker w/ buildx plugin

Need more control over the SDK?

The Holoscan SDK source repository is **open-source** and provides reference implementations as well as infrastructure for building the SDK yourself.

> ⚠ **Attention**
>
> We only recommend building the SDK from source if you need to build it with debug symbols or other options not used as part of the published packages. If you want to write your own operator or application, you can use the SDK as a dependency (and contribute to

HoloHub). If you need to make other modifications to the SDK, file a feature or bug request.

Looking for a light runtime container image?

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

---

[1]

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

[2](1,2,3,4,5)

Debian installation on x86_64 requires the latest cuda-keyring package to automatically install all dependencies.

[3]

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

[4]

TensorRT 8.6.1+ and cuDNN needed for the Inference operator. Already installed on NVIDIA developer kits with IGX Software and JetPack.

[5]

Vulkan 1.3.204+ loader needed for the HoloViz operator (+ libegl1 for headless rendering). Already installed on NVIDIA developer kits with IGX Software and JetPack.

[6]

V4L2 1.22+ needed for the V4L2 operator. Already installed on NVIDIA developer kits with IGX Software and JetPack.
[7]

Torch support requires LibTorch 2.1+, TorchVision 0.16+, OpenBLAS 0.3.20+, OpenMPI (aarch64 only), MKL 2021.1.1 (x86_64 only), libpng and libjpeg.

[8](1,2)

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

[9]

ONNXRuntime 1.15.1+ needed for the Inference operator. Note that ONNX models are also supported through the TensoRT backend of the Inference Operator.

[10](1,2)

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

[11]

Tested with MOFED 23.10

Additional Setup

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

- Enabling RDMA
- Enabling G-SYNC
- Disabling Variable Backlight
- Enabling Exclusive Display Mode
- Use both Integrated and Discrete GPUs on NVIDIA Developer Kits
- Deployment Software Stack

Enabling RDMA

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

There are two parts to enabling RDMA for Holoscan:

- Enabling RDMA on the ConnectX SmartNIC
- Enabling GPUDirect RDMA

Enabling RDMA on the ConnectX SmartNIC

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

The NVIDIA IGX Orin developer kit comes with an embedded ConnectX Ethernet adapter to offer advanced hardware offloads and accelerations. You can also purchase an

individual ConnectX adapter and install it on other systems such as x86_64 workstations.

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

1. Install MOFED drivers

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

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

If not installed, or an older version, install the appropriate version from the MLNX_OFED download page, or use the script below:

```
# You can choose different versions/OS or download directly from the # Download Center in the webpage linked above MOFED_VERSION="23.10-2.1.3.1" OS="ubuntu22.04" MOFED_PACKAGE="MLNX_OFED_LINUX-${MOFED_VERSION}-${OS}-$(uname -m)" wget --progress=dot:giga https://www.mellanox.com/downloads/ofed/MLNX_OFED-${MOFED_VERSION}/${MOF tar xf ${MOFED_PACKAGE}.tgz sudo ./${MOFED_PACKAGE}/mlnxofedinstall # add the --force flag to force uninstallation if necessary: # sudo ./${MOFED_PACKAGE}/mlnxofedinstall --force rm -r ${MOFED_PACKAGE}*
```

2. Load MOFED drivers

Ensure the drivers are loaded:

```
sudo lsmod | grep ib_core
```

If nothing appears, run the following command:

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

3. Switch the board Link Layer to Ethernet

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

- Ethernet (ETH)
- Infiniband (IB)

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

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

```
$ ibstat CA 'mlx5_0' CA type: MT4129 Number of ports: 1 Firmware version: 28.37.0190 Hardware version: 0 Node GUID: 0x48b02d0300ee7a04 System image GUID: 0x48b02d0300ee7a04 Port 1: State: Down Physical state: Disabled Rate: 40 Base lid: 0 LMC: 0 SM lid: 0 Capability mask: 0x00010000 Port GUID: 0x4ab02dfffeee7a04 Link layer: Ethernet CA 'mlx5_1' CA type: MT4129 Number of ports: 1 Firmware version: 28.37.0190 Hardware version: 0 Node GUID: 0x48b02d0300ee7a05 System image GUID: 0x48b02d0300ee7a04 Port 1: State: Active Physical state: LinkUp Rate: 100 Base lid: 0 LMC: 0 SM lid: 0 Capability mask: 0x00010000 Port GUID: 0x4ab02dfffeee7a05 Link layer: InfiniBand
```

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

```
ib_core 425984 1 ib_uverbs
```

Consider running the following command or rebooting:

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

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

1. On IGX Orin developer kits, you can switch that setting through the BIOS: see IGX Orin documentation.

2. On any system with a ConnectX (including IGX Orin devkits), you can run the command below from a terminal (requires a reboot). `sudo ibdev2netdev -v` is used to identify the PCI address of the ConnectX (any of the two interfaces is fine to use), and `mlxconfig` is used to apply the changes.

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

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

   This is the output of the command above:

   ```
   Device #1: ---------- Device type: ConnectX7 Name: P3740-B0-QSFP_Ax Description: NVIDIA Prometheus P3740 ConnectX-7 VPI PCIe Switch Motherboard; 400Gb/s; dual-port QSFP; PCIe switch5.0 X8 SLOT0 ;X16 SLOT2; secure boot; Device: 0005:03:00.0 Configurations: Next Boot New LINK_TYPE_P1 ETH(2) ETH(2) LINK_TYPE_P2 IB(1) ETH(2) Apply new Configuration? (y/n) [n] :
   ```

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

   Apply with `y` and reboot afterwards:

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

4. Configure the IP addresses of the ethernet interfaces

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

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

Tip

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

- `0005:03.00.0` is the QSFP port closer to the PCI slots
- `0005:03.00.1` is the QSFP closer to the RJ45 ethernet ports

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

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

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

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

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

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

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

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

Note

If you are connecting the ConnectX to another ConnectX with a LinkX interconnect, do the same on the other system with an IP address on the same network segment.

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

Enabling GPUDirect RDMA

Note

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

Follow the instructions below to enable GPUDirect RDMA:

Ingested Tab Module

Testing with Rivermax

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

Note

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

Note

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

Running the Rivermax sample applications requires two systems, a sender and a receiver, connected via ConnectX network adapters. If two Developer Kits are used then the onboard ConnectX can be used on each system, 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/1.31.10
```

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

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

Install path might differ in future versions of Rivermax.

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

   ```
   $ sudo lshw -class network *-network:0 description: Ethernet interface product: MT28908 Family [ConnectX-6] vendor: Mellanox Technologies physical id: 0 bus info: pci@0000:09:00.0 <b>logical name: enp9s0f0</b> version: 00 serial: 48:b0:2d:13:9b:6b capacity: 10Gbit/s width: 64 bits clock: 33MHz capabilities: pciexpress vpd msix pm bus_master cap_list ethernet physical 1000bt-fd 10000bt-fd autonegotiation configuration: autonegotiation=on broadcast=yes driver=mlx5_core driverversion=5.4-1.0.3
   ```

```
duplex=full firmware=20.27.4006 (NVD0000000001) ip=10.0.0.2 latency=0
link=yes multicast=yes resources: iomemory:180-17f irq:33
memory:1818000000-1819fffffff
```

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

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

   a. Bring up the network:

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

   b. Build the sample apps:

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

   e. Launch the `generic_sender` application:

   ```
   $ sudo ./generic_sender -l 10.0.0.1 -d 10.0.0.2 -p 5001 -y 1462 -k 8192 -z 500 -v
   ... +############################################## | Sender
   index: 0 | Thread ID: 0x7fa1ffb1c0 | CPU core affinity: -1 | Number of streams
   in this thread: 1 | Memory address: 0x7f986e3010 | Memory length:
   59883520[B] | Memory key: 40308
   +############################################## | Stream
   index: 0 | Source IP: 10.0.0.1 | Destination IP: 10.0.0.2 | Destination port: 5001
   | Number of flows: 1 | Rate limit bps: 0 | Rate limit max burst in packets: 0 |
   Memory address: 0x7f986e3010 | Memory length: 59883520[B] | Memory key:
   40308 | Number of user requested chunks: 1 | Number of application chunks:
   5 | Number of packets in chunk: 8192 | Packet's payload size: 1462
   +**********************************************
   ```

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

a. Bring up the network:

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

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

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

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

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

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`:

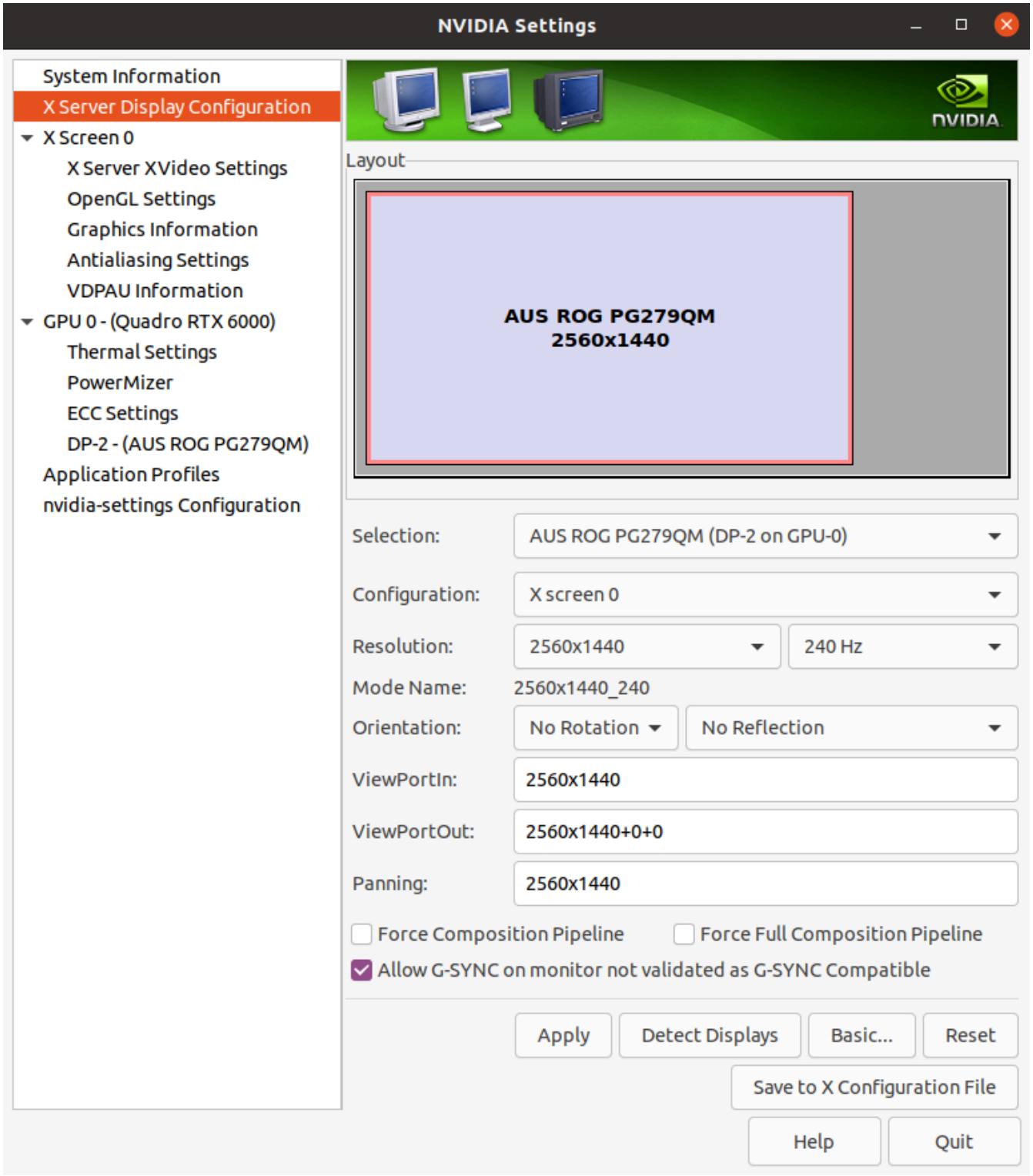


Fig. 1 *Enable G-SYNC for the current display*

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.

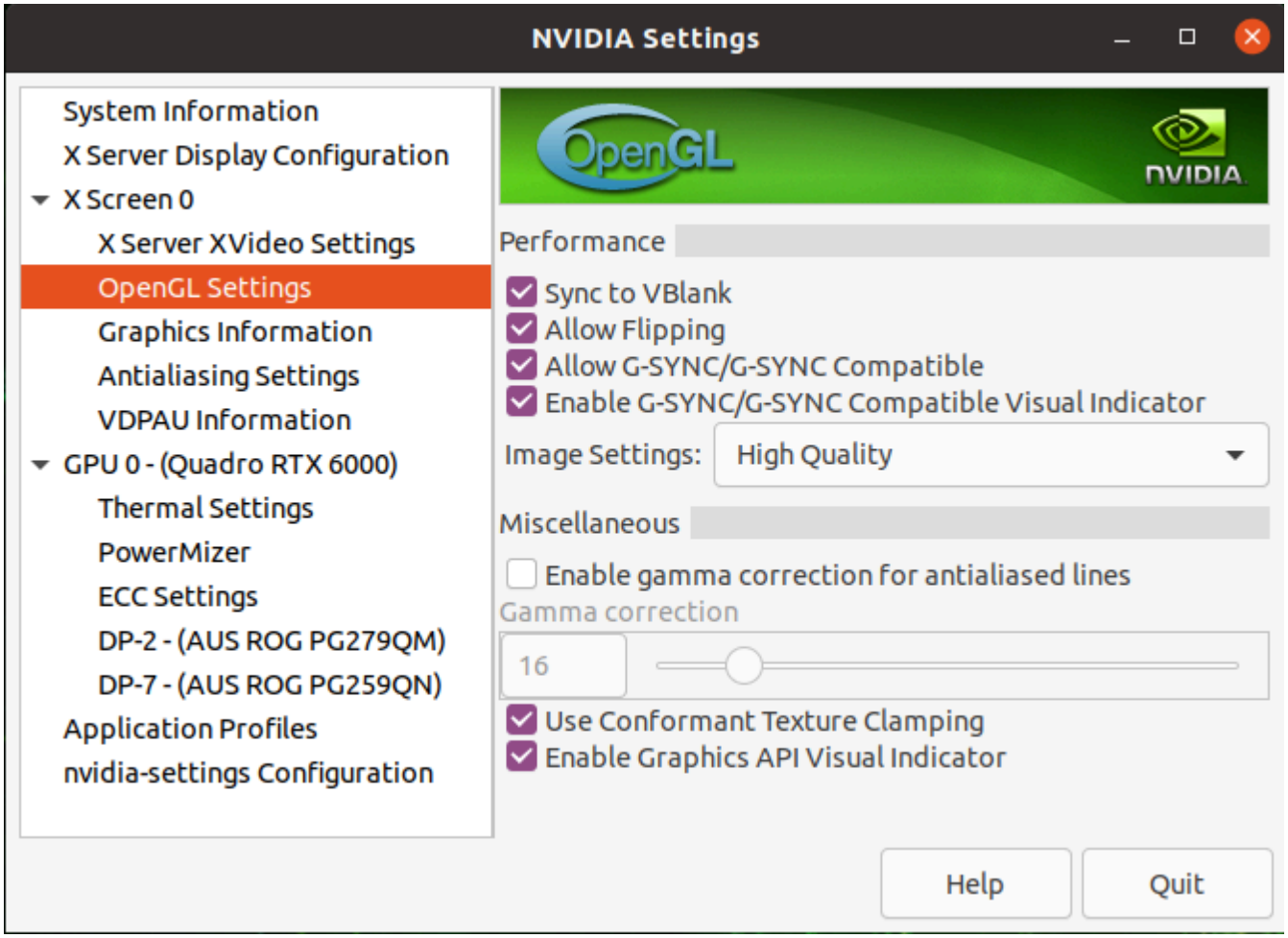


Fig. 2 *Enable Visual Indicator for the current display*

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`.

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.

Use both Integrated and Discrete GPUs on NVIDIA Developer Kits

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

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

1. Conflict between the dGPU kernel mode driver and the iGPU display kernel driver (both named `nvidia.ko`). This conflict is not addressable at this time, meaning that **the iGPU cannot be used for display while the dGPU is enabled**.

2. Conflicts between the user mode driver libraries (ex: `libcuda.so`) and the compute stack (ex: `libcuda_rt.so`) for dGPU and iGPU.

We provide utilities to work around the second conflict:

Ingested Tab Module

⚠ **Attention**

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

Deployment Software Stack

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

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

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

- 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.

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.

Table of Contents

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.

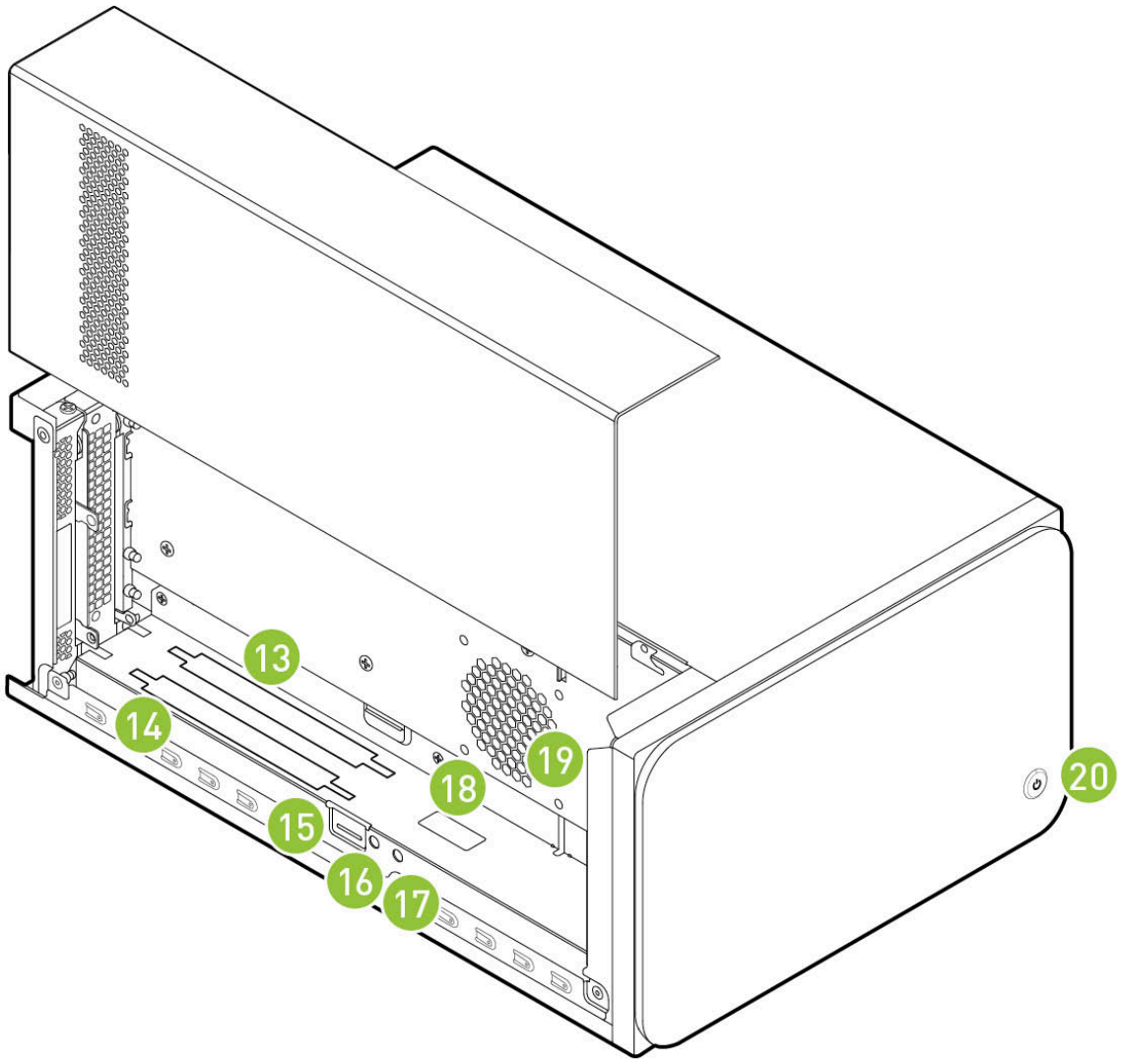
The following instructions describe the steps required to setup and use an AJA device with RDMA support on NVIDIA Developer Kits with a PCIe slot. Note that the AJA NTV2 SDK support for Holoscan includes all of the AJA Developer Products, though the following instructions have only been verified for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

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

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.

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

Note

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

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

```
$ sudo sh ${NTV2}/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) 47 video format(s): 1080i50, 1080i59.94, 1080i60, 720p59.94, 720p60, 1080p29.97, 1080p30, 1080p25, 1080p23.98, 1080p24, 2Kp23.98, 2Kp24, 720p50, 1080p50b, 1080p59.94b, 1080p60b, 1080p50a, 1080p59.94a, 1080p60a, 2Kp25, 525i59.94, 625i50, UHDp23.98, UHDp24, UHDp25, 4Kp23.98, 4Kp24, 4Kp25, UHDp29.97, UHDp30, 4Kp29.97, 4Kp30, UHDp50, UHDp59.94, UHDp60, 4Kp50, 4Kp59.94, 4Kp60, 4Kp47.95, 4Kp48, 2Kp60a, 2Kp59.94a, 2Kp29.97, 2Kp30, 2Kp50a, 2Kp47.95a, 2Kp48a
```

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

```
$ testrdma -t500 test device 0 start 0 end 7 size 8388608 count 500 frames/errors 500/0
```

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*.

Troubleshooting

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

   Solutions:

   1. Make sure the AJA card is properly installed and powered (see 2.a below)
   2. 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:

   1. 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)
      ```

   2. 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.

Emergent Vision Technologies (EVT)

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

Installing EVT Hardware

The EVT cameras can be connected to NVIDIA Developer Kits through a Mellanox ConnectX SmartNIC, with the most simple connection method being a single cable between a camera and the devkit. For 25 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 NVIDIA IGX Orin Developer Kit User Guide for the location of the QSFP28 connector on the device.

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

Tip

The EVT cameras require the user to buy the lens. Based on the application of camera, the lens can be bought from any online store.

Installing EVT Software

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

Note

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

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

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

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.

Troubleshooting

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

Solution:

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

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

Solutions:

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

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

Solution:

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

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

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

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

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

Holoscan Core Concepts

Note

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

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

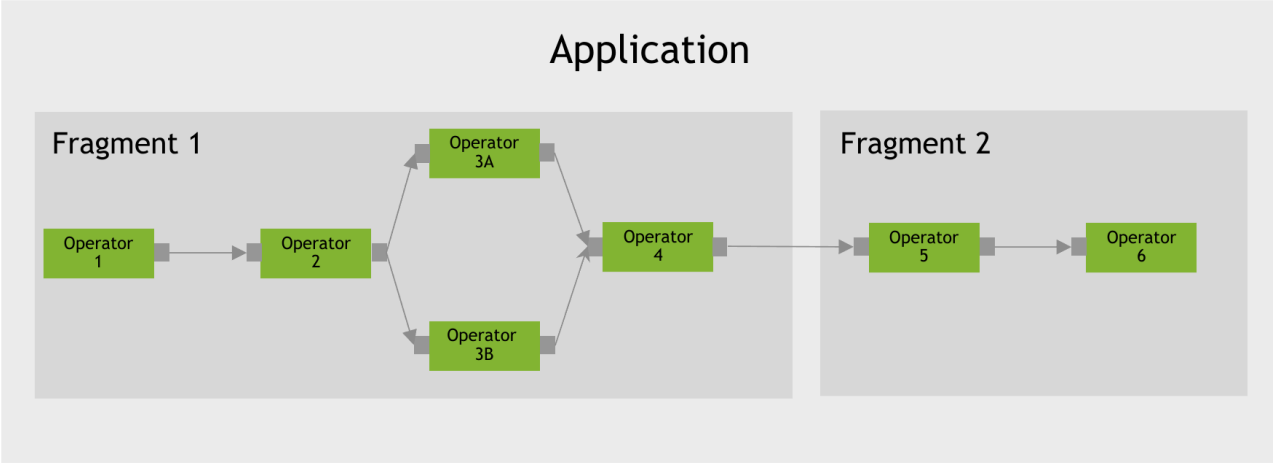


Fig. 3 *Core concepts: Application*

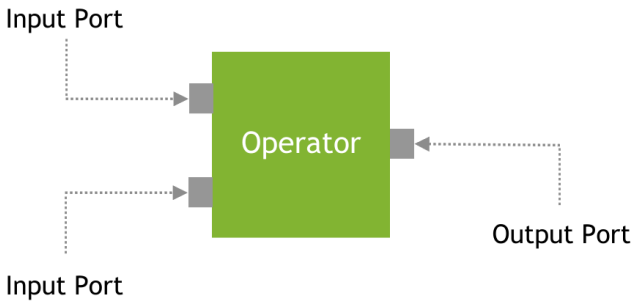


Fig. 4 *Core concepts: Port*

The core concepts of the Holoscan API are:

- **Application**: An application acquires and processes streaming data. An application is a collection of fragments where each fragment can be allocated to execute on a physical node of a Holoscan cluster.

- **Fragment**: A fragment is a building block of the Application. It is a directed graph of operators. A fragment can be assigned to a physical node of a Holoscan cluster during execution. The 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 `Receiver`s and `Transmitter`s of a GXF Entity as Input/Output `Port`s 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.

Table of Contents

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.

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.

Ingested Tab Module

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:

Ingested Tab Module

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()`.

Ingested Tab Module

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.

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!

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.

Operators and Workflow

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

digraph ping_simple { rankdir="LR" node [shape=record]; tx [label="PingTxOp| |out(out) : int"]; rx [label="PingRxOp|[in]in : int | "]; tx -> rx [label="out...in"] }

Fig. 5 *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.

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()`.

Ingested Tab Module

Running the Application

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

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

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.

Operators and Workflow

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

digraph custom_op { rankdir="LR" node [shape=record]; tx [label="PingTxOp| |out(out) : int"]; mx [label="PingMxOp| [in]in : int | out(out) : int "]; rx [label="PingRxOp| [in]in : int | "]; tx -> mx [label="out...in"] mx -> rx [label="out...in"] }

Fig. 6 *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.

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).

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.

Ingested Tab Module

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

Ingested Tab Module

Message Data Types

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

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

Ingested Tab Module

> ⚠ **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 `holoscan::TensorMap` type instead. See Interoperability between GXF and native C++ operators for more details. If you are writing a Python application which needs a mixture of Python wrapped C++ operators and native Python operators, see Interoperability between wrapped and native Python operators

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 Middle
```

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

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.

Operators and Workflow

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

digraph ping_multi_port { rankdir="LR" node [shape=record]; tx [label="PingTxOp| |out1(out) : ValueData\nout2(out) : ValueData"]; mx [label="PingMxOp|[in]in1 : ValueData\n[in]in2 : ValueData|out1(out) : ValueData\nout2(out) : ValueData"]; rx [label="PingRxOp|[in]receivers : ValueData | "]; tx -> mx [label="out1...in1"] tx -> mx [label="out2...in2"] mx -> rx [label="out1...receivers"] mx -> rx [label="out2...receivers"] }

Fig. 7 *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`.

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.

Ingested Tab Module

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`:

Ingested Tab Module

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

Ingested Tab Module

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

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()`.

Ingested Tab Module

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

Ingested Tab Module

- 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`.

Running the Application

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

```
[info] [gxf_executor.cpp:222] Creating context [info] [gxf_executor.cpp:1531] Loading extensions from configs... [info] [gxf_executor.cpp:1673] Activating Graph... [info] [gxf_executor.cpp:1703] Running Graph... [info] [gxf_executor.cpp:1705] Waiting for completion... [info] [gxf_executor.cpp:1706] Graph execution waiting. Fragment: [info] [greedy_scheduler.cpp:195] Scheduling 3 entities [info] [ping_multi_port.cpp:80] Middle message received (count: 1) [info] [ping_multi_port.cpp:82] Middle message value1: 1 [info] [ping_multi_port.cpp:83] Middle message value2: 2 [info] [ping_multi_port.cpp:112] Rx message received (count: 1, size: 2) [info] [ping_multi_port.cpp:114] Rx message value1: 3 [info] [ping_multi_port.cpp:115] Rx message value2: 6 [info] [ping_multi_port.cpp:80] Middle message received (count: 2) [info] [ping_multi_port.cpp:82] Middle message value1: 3 [info] [ping_multi_port.cpp:83] Middle message value2: 4 [info] [ping_multi_port.cpp:112] Rx message received (count: 2, size: 2) [info] [ping_multi_port.cpp:114] Rx message value1: 9 [info] [ping_multi_port.cpp:115] Rx message value2: 12 ... [info] [ping_multi_port.cpp:114] Rx message value1: 51 [info] [ping_multi_port.cpp:115] Rx message value2: 54 [info] [ping_multi_port.cpp:80] Middle message received (count: 10) [info] [ping_multi_port.cpp:82] Middle message
```

```
value1: 19 [info] [ping_multi_port.cpp:83] Middle message value2: 20 [info] [ping_multi_port.cpp:112] Rx message received (count: 10, size: 2) [info] [ping_multi_port.cpp:114] Rx message value1: 57 [info] [ping_multi_port.cpp:115] Rx message value2: 60 [info] [greedy_scheduler.cpp:374] Scheduler stopped: Some entities are waiting for execution, but there are no periodic or async entities to get out of the deadlock. [info] [greedy_scheduler.cpp:403] Scheduler finished. [info] [gxf_executor.cpp:1714] Graph execution deactivating. Fragment: [info] [gxf_executor.cpp:1715] Deactivating Graph... [info] [gxf_executor.cpp:1718] Graph execution finished. Fragment: [info] [gxf_executor.cpp:241] Destroying context
```

Note

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

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.

Operators and Workflow

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

digraph video_replayer { rankdir="LR" node [shape=record]; replayer [label="VideoStreamReplayerOp| |output(out) : Tensor"]; viz [label="HolovizOp| [in]receivers : Tensor | "]; replayer -> viz [label="output...receivers"] }

Fig. 8 *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.

Video Stream Replayer Operator

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

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

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

Holoviz Operator

The built-in Holoviz operator provides the functionality to composite 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 it's "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.

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/racerx" # Path to gxf entity video data basename: "racerx" # 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
```

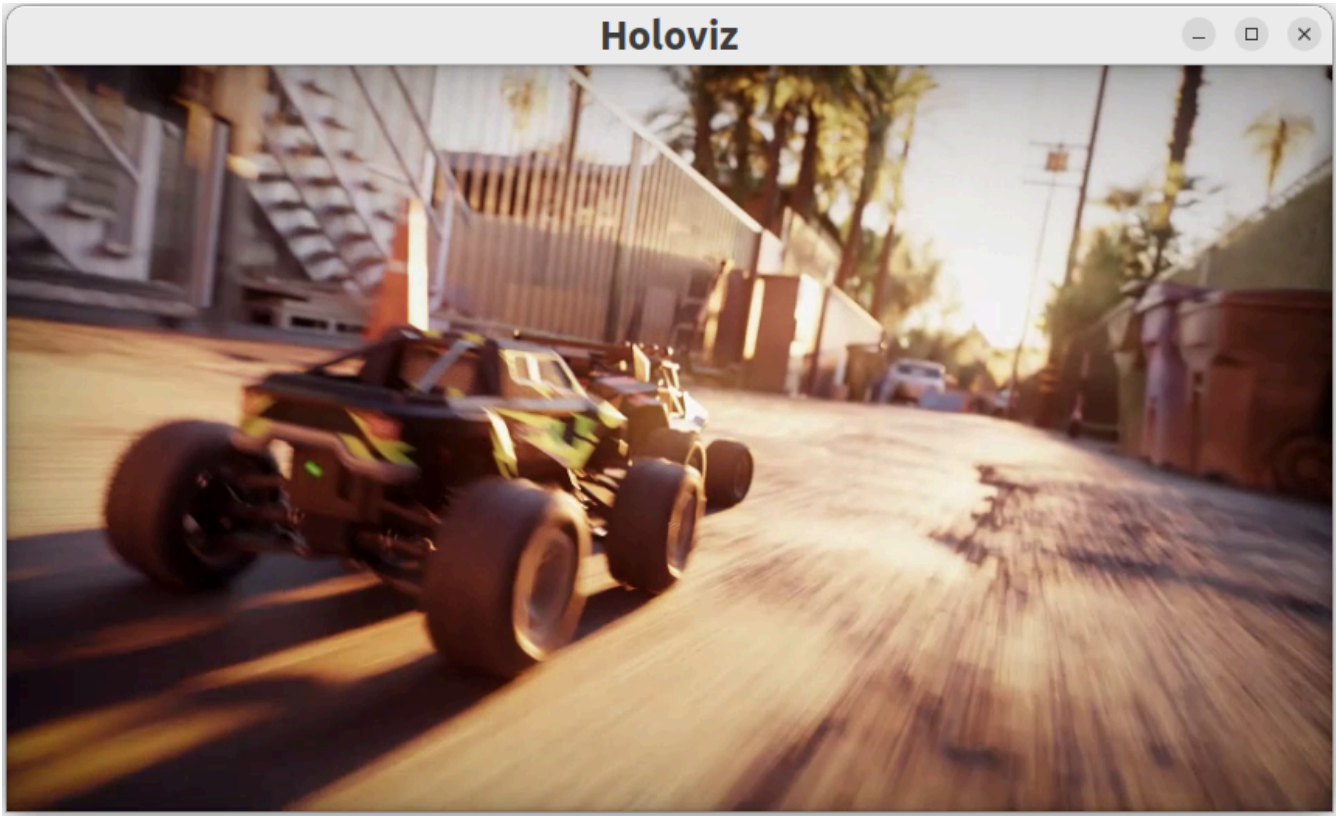
```
opacity: 1.0 # layer opacity priority: 0 # determines render order, higher priority layers are rendered on top
```

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

Ingested Tab Module

Running the Application

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



Video Replayer (Distributed)

In this example, we extend the previous video replayer application into a multi-node distributed application. A distributed application is made up of multiple Fragments (`C++` /

`Python`), each of which may run on its own node.

In the distributed case we will:

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

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

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

Operators and Workflow

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

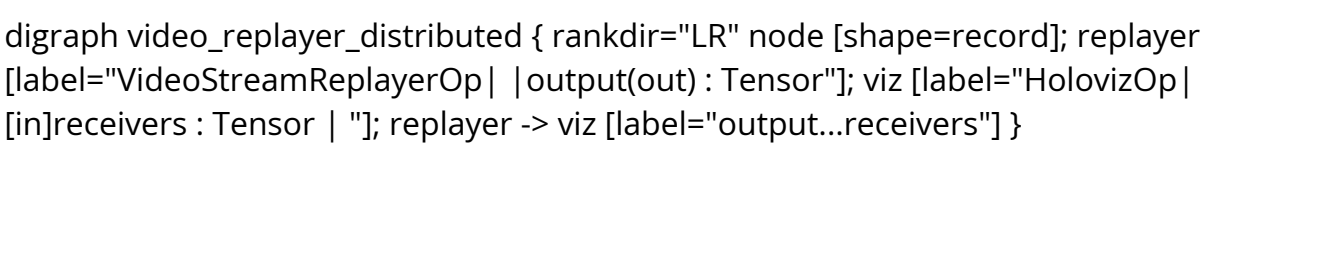


Fig. 9 *Workflow to load and display video from a file*

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

Defining and Connecting Fragments

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

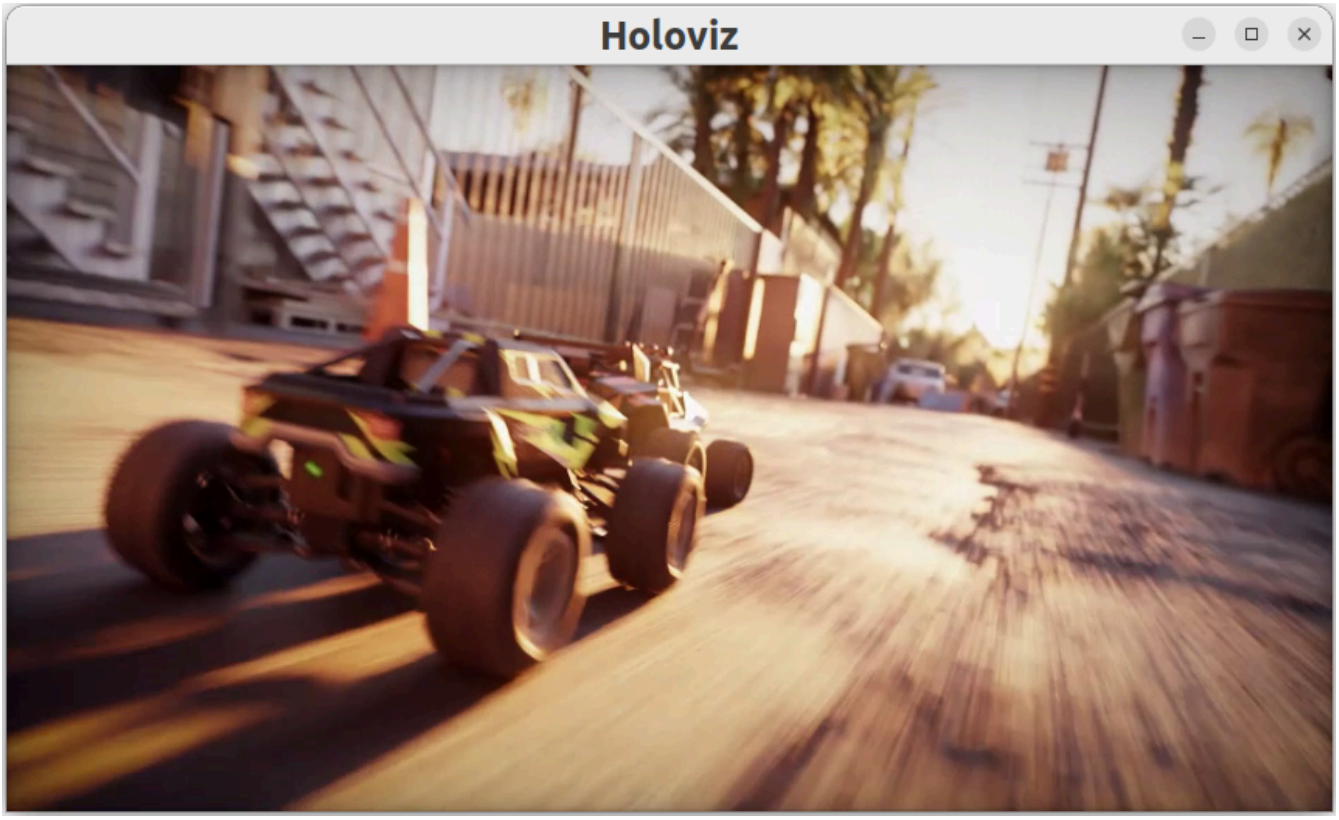
- We define two classes that inherit from `Fragment`:
    - **Fragment1** contains an instance of **VideoStreamReplayerOp** named "replayer".
    - **Fragment2** contains an instance of **HolovizOp** name "holoviz".
- We create an application, **DistributedVideoReplayerApp**. In its compose method:
    - we call **make_fragment** to initialize both fragments.
    - we then connect the "output" port of "replayer" operator in fragment1 to the "receivers" port of the "holoviz" operator in fragment2 to define the application workflow.
- The operators instantiated in the fragments can still be configured with parameters initialized from the YAML configuration ingested by the application using `from_config()` (C++) or `kwargs()` (Python).

Ingested Tab Module

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

Running the Application

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



Note

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

Tip

Refer to UCX Network Interface Selection when running a distributed application across multiple nodes.

Bring Your Own Model (BYOM)

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

In this example we'll cover:

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

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

Operators and Workflow

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

Fig. 10 *The BYOM inference workflow*

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

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

- **Format Converter**: The input video stream goes through a preprocessing stage to convert the tensors to the appropriate shape/format before being fed into the AI model. It is used here to convert the datatype of the image from `uint8` to `float32` and resized to match the model's expectations.

- **Inference**: This operator performs AI inferencing on the input video stream with the provided model. It supports inferencing of multiple input video streams and models.

- **Segmentation Postprocessor**: this postprocessing stage takes the output of inference, either with the final softmax layer (multiclass) or sigmoid (2-class), and emits a tensor with `uint8` values that contain the highest probability class index. The output of the segmentation postprocessor is then fed into the Holoviz visualizer to create the overlay.

Prerequisites

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

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

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

Input video

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

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

Input model

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

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

> **Tip**
>
> To get a better understanding of your model, and if this step is necessary, websites such as netron.app can be used.

Understanding the Application Code

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

Ingested Tab Module

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

Ingested Tab Module

Finally, we define the application and workflow.

Ingested Tab Module

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.

Ingested Tab Module

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.

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.

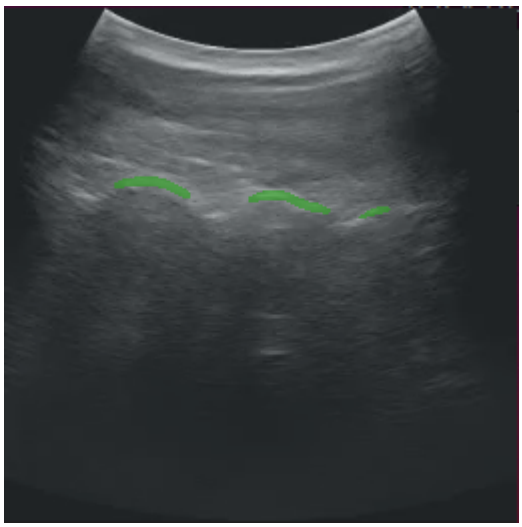


Fig. 11 *Ultrasound Segmentation*

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
```

Customizing the Inference Operator

The builtin `InferenceOp` operator provides the functionality of the Inference. This operator has a `receivers` port that can connect to any number of upstream ports to allow for multiai 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**

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

Defining an Application Class

The following code snippet shows an example Application code skeleton:

Ingested Tab Module

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

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

Ingested Tab Module

Tip

This is also illustrated in the ping_simple_run_async example.

Configuring an Application

An application can be configured at different levels:

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

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

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 dict_param: key_1: value_1 key_2: value_2
```

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

Ingested Tab Module

Tip

This is also illustrated in the video_replayer example.

⚠ **Attention**

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

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:

Ingested Tab Module

> **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.

Configuring operators

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

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

Configuring operator parameters

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

Ingested Tab Module

> **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:

Ingested Tab Module

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:

Ingested Tab Module

> **Tip**
>
> This is also illustrated in the ping_custom_op example.

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

Ingested Tab Module

Configuring operator conditions

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

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

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

Ingested Tab Module

Tip

This is also illustrated in the conditions examples.

Note

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

Note

Python operators that wrap an underlying C++ operator currently do not accept conditions as positional arguments. Instead one needs to call the

```
<a
href="api/python/holoscan_python_api_core.html#holoscan.core.Operator.add_ar
</a>
```

method after the object has been constructed to add the condition.

Configuring operator resources

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

For example:

Ingested Tab Module

Note

Python operators that wrap an underlying C++ operator currently do not accept resources as positional arguments. Instead one needs to call the

```
<a
href="api/python/holoscan_python_api_core.html#holoscan.core.Operator.add_ar
</a>
```

method after the object has been constructed to add the resource.

Configuring the scheduler

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

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

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

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

Ingested Tab Module

Tip

This is also illustrated in the multithread example.

Configuring runtime properties

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

Application Workflows

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

One-operator Workflow

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

digraph myop { rankdir="LR" node [shape=record]; myop [label="MyOp| | "]; }

Fig. 12 *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).

Ingested Tab Module

Linear Workflow

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

digraph linear_workflow { rankdir="LR" node [shape=record]; sourceop [label="SourceOp| |output(out) : Tensor"]; processop [label="ProcessOp| [in]input : Tensor | output(out) : Tensor "]; sinkop [label="SinkOp| [in]input : Tensor | "]; sourceop -> processop [label="output...input"] processop -> sinkop [label="output...input"] }

Fig. 13 *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).

Ingested Tab Module

Complex Workflow (Multiple Inputs and Outputs)

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

digraph complex_workflow { node [shape=record]; reader1 [label="{Reader1| |image(out)\nmetadata(out)}"]; reader2 [label="{Reader2| |roi(out)}"]; processor1 [label="{Processor1|[in]image1\n[in]image2\n[in]metadata|image(out)}"]; processor2 [label="{Processor2|[in]image\n[in]roi|image(out)}"]; processor3 [label="{Processor3|[in]image|seg_image(out)}"]; writer [label="{Writer|[in]image\n[in]seg_image| }"]; notifier [label="{Notifier|[in]image| }"]; reader1->processor1 [label="image...{image1,image2}\nmetadata...metadata"] reader2->processor2 [label="roi...roi"] processor1->processor2 [label="image...image"] processor1->writer [label="image...image"] processor2->notifier [label="image...image"] processor2->processor3 [label="image...image"] processor3->writer [label="seg_image...seg_image"] }

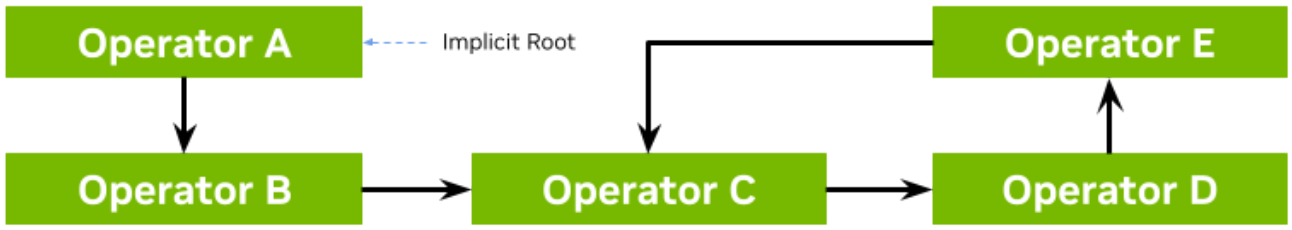
Fig. 14 *A complex workflow (multiple inputs and outputs)*

Ingested Tab Module

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

Ingested Tab Module

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



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

Building and running your Application

Ingested Tab Module

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

Creating a Distributed Application

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

In this section we'll describe:

- how to define a distributed Application
- how to build and run a distributed application

Defining a Distributed Application Class

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

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

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

Ingested Tab Module

Serialization of Custom Data Types for Distributed Applications

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

Building and running a Distributed Application

Ingested Tab Module

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

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

- The driver's IP (or hostname) **MUST** be set for each process (driver and worker(s)) when running distributed applications on multiple nodes (default: `0.0.0.0`). It can be set without the port (e.g., `--address 192.168.50.68`).

- In a single-node application, the driver's IP (or hostname) can be omitted, allowing any network interface (`0.0.0.0`) to be selected by the UCX library.

- The port is always optional (default: `8765`). It can be set without the IP (e.g., `--address :10000`).

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

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

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

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

- The application driver is launched at `192.168.50.68:10000` on the first node (A), with a worker running two fragments, "fragment1" and "fragment3".

- On a separate node (B), the application launches a worker for "fragment2", which will connect to the driver at the address above.

Ingested Tab Module

Note

UCX Network Interface Selection

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

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

```
export UCX_NET_DEVICES=eth0
```

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

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

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

Warning

Known limitations

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

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

The message
`Connection dropped with status -25 (Connection reset by remote peer)`
appears in the console even when the application is functioning properly. This is a known issue and will be addressed in future updates, ensuring that this message will only be displayed in the event of an actual connection error.

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

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

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

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

Note

GXF UCX Extension

Holoscan’s distributed application feature makes use of the GXF UCX Extension. Its documentation may provide useful additional context

into how data is transmitted between fragments.

Tip

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

Environment Variables for Distributed Applications

Holoscan SDK environment variables.

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

- **HOLOSCAN_ENABLE_HEALTH_CHECK** : determines if the health check service should be active, even without specifying `--driver` or `--worker` in the CLI. By default, initiating the AppDriver (`--driver`) or AppWorker (`--worker`) service automatically triggers the GRPC Health Checking Service so grpc-health-probe can monitor liveness/readiness. Interprets values like “true”, “1”, or “on” (case-insensitive) as true (to enable the health check). It defaults to false if left unspecified.

- **HOLOSCAN_HEALTH_CHECK_PORT** : designates the port number on which the Health Checking Service is launched. It must be an integer value representing a valid port number. If unspecified, it defaults to `8777`.

- **HOLOSCAN_DISTRIBUTED_APP_SCHEDULER** : controls which scheduler is used for distributed applications. It can be set to either `greedy`, `multi_thread` or `event_based`. `multithread` is also allowed as a synonym for `multi_thread` for backwards compatibility. If unspecified, the default scheduler is `multi_thread`.

- **HOLOSCAN_STOP_ON_DEADLOCK** : can be used in combination with `HOLOSCAN_DISTRIBUTED_APP_SCHEDULER` to control whether or not the application will automatically stop on deadlock. Values of “True”, “1” or “ON” will be

interpreted as true (enable stop on deadlock). It is true if unspecified. This environment variable is only used when `HOLOSCAN_DISTRIBUTED_APP_SCHEDULER` is explicitly set.

- **HOLOSCAN_STOP_ON_DEADLOCK_TIMEOUT** : controls the delay (in ms) without activity required before an application is considered to be in deadlock. It must be an integer value (units are ms).

- **HOLOSCAN_MAX_DURATION_MS** : sets the application to automatically terminate after the requested maximum duration (in ms) has elapsed. It must be an integer value (units are ms). This environment variable is only used when `HOLOSCAN_DISTRIBUTED_APP_SCHEDULER` is explicitly set.

- **HOLOSCAN_CHECK_RECESSION_PERIOD_MS** : controls how long (in ms) the scheduler waits before re-checking the status of operators in an application. It must be a floating point value (units are ms). This environment variable is only used when `HOLOSCAN_DISTRIBUTED_APP_SCHEDULER` is explicitly set.

- **HOLOSCAN_UCX_SERIALIZATION_BUFFER_SIZE** : can be used to override the default 7 kB serialization buffer size. This should typically not be needed as tensor types store only a small header in this buffer to avoid explicitly making a copy of their data. However, other data types do get directly copied to the serialization buffer and in some cases it may be necessary to increase it.

- **HOLOSCAN_UCX_DEVICE_ID** : The GPU ID of the device that will be used by UCX transmitter/receivers in distributed applications. If unspecified, it defaults to 0. A list of discrete GPUs available in a system can be obtained via `nvidia-smi -L`. GPU data sent between fragments of a distributed application must be on this device.

- **HOLOSCAN_UCX_PORTS** : This defines the preferred port numbers for the SDK when specific ports for UCX communication need to be predetermined, such as in a Kubernetes environment. If the distributed application requires three ports (UCX receivers) and the environment variable is unset, the SDK chooses three unused ports sequentially from the range 10000~32767. Specifying a value, for example, `HOLOSCAN_UCX_PORTS=10000`, results in the selection of ports 10000, 10001, and 10002. Multiple starting values can be comma-separated. The system increments from the last provided port if more ports are needed. Any unused specified ports are ignored.

- **HOLOSCAN_UCX_SOURCE_ADDRESS** : This environment variable specifies the local IP address (source) for the UCX connection. This variable is especially beneficial when a node has multiple network interfaces, enabling the user to determine which one should be utilized for establishing a UCX client (UCXTransmitter). If it is not explicitly specified, the default address is set to `0.0.0.0`, representing any available interface.

UCX-specific environment variables

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

- The `UCX_TLS` environment variable can be used to control which transport layers are enabled. By default, `UCX_TLS=all` and UCX will attempt to choose the optimal transport layer automatically.

- The `UCX_NET_DEVICES` environment variable is by default set to `all` meaning that UCX may choose to use any available network interface controller (NIC). In some cases it may be necessary to restrict UCX to a specific device or set of devices, which can be done by setting `UCX_NET_DEVICES` to a comma separated list of the device names (i.e. as obtained by linux command `ifconfig -a` or `ip link show`).

- Setting `UCX_TCP_CM_REUSEADDR=y` is recommended to enable ports to be reused without having to wait the full socket TIME_WAIT period after a socket is closed.

- The `UCX_LOG_LEVEL` environment variable can be used to control the logging level of UCX. The default is setting is WARN, but changing to a lower level such as INFO will provide more verbose output on which transports and devices are being used.

- By default, Holoscan SDK will automatically set `UCX_PROTO_ENABLE=y` upon application launch to enable the newer “v2” UCX protocols. If for some reason, the older v1 protocols are needed, one can set `UCX_PROTO_ENABLE=n` in the environment to override this setting. When the v2 protocols are enabled, one can optionally set `UCX_PROTO_INFO=y` to enable detailed logging of what protocols are being used at runtime.

- By default, Holoscan SDK will automatically set `UCX_MEMTYPE_CACHE=n` upon application launch to disable the UCX memory type cache (See UCX documentation for more information. It can cause about 0.2 microseconds of pointer type checking overhead with the cudacudaPointerGetAttributes() CUDA API). If for some reason, the memory type cache is needed, one can set `UCX_MEMTYPE_CACHE=y` in the environment to override this setting.

- By default, the Holoscan SDK will automatically set `UCX_CM_USE_ALL_DEVICES=n` at application startup to disable consideration of all devices for data transfer. If for some reason the opposite behavior is desired, one can set `UCX_CM_USE_ALL_DEVICES=y` in the environment to override this setting. Setting `UCX_CM_USE_ALL_DEVICES=n` can be used to workaround an issue where UCX sometimes defaults to a device that might not be the most suitable for data transfer based on the host's available devices. On a host with address 10.111.66.60, UCX, for instance, might opt for the `br-80572179a31d` (192.168.49.1) device due to its superior bandwidth as compared to `eno2` (10.111.66.60). With `UCX_CM_USE_ALL_DEVICES=n`, UCX will ensure consistency by using the same device for data transfer that was initially used to establish the connection. This ensures more predictable behavior and can avoid potential issues stemming from device mismatches during the data transfer process.

- Setting `UCX_TCP_PORT_RANGE=&lt;start&gt;-&lt;end&gt;` can be used to define a specific range of ports that UCX should utilize for data transfer. This is particularly useful in environments where ports need to be predetermined, such as in a Kubernetes setup. In such contexts, Pods often have ports that need to be exposed, and these ports must be specified ahead of time. Moreover, in scenarios where firewall configurations are stringent and only allow specified ports, having a predetermined range ensures that the UCX communication does not get blocked. This complements the `HOLOSCAN_UCX_SOURCE_ADDRESS`, which specifies the local IP address for the UCX connection, by giving further control over which ports on that specified address should be used. By setting a port range, users can ensure that UCX operates within the boundaries of the network and security policies of their infrastructure.

Tip

A list of all available UCX environment variables and a brief description of each can be obtained by running `ucx_info -f` from the

Holoscan SDK container. Holoscan SDK uses UCX's active message (AM) protocols, so environment variables related to other protocols such as tag-mat

Serialization

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

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

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

> ⚠ **Warning**
>
> If an operator transmitting both CPU and GPU tensors is to be used in distributed applications, the same output port cannot mix both GPU and CPU tensors. CPU and GPU tensor outputs should be placed on separate output ports. This is a limitation of the underlying UCX library being used for zero-copy tensor serialization between operators.
>
> As a concrete example, assume an operator, `MyOperator` with a single output port named “out” defined in it’s setup method. If the output port is only ever going to connect to other operators within a fragment, but never across fragments then it is okay to have a `TensorMap` with a mixture of host and device arrays on that single port.
>
> Ingested Tab Module
>
> However, this mixing of CPU and GPU arrays on a single port will not work for distributed apps and instead separate ports should be used if it is necessary for an operator to communicate across fragments.
>
> Ingested Tab Module

Python

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

> ⚠ **Warning**

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

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

C++

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

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

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

```
#include "holoscan/core/codec_registry.hpp" #include "holoscan/core/errors.hpp" #include "holoscan/core/expected.hpp" namespace holoscan { template <> struct codec<Coordinate> { static expected<size_t, RuntimeError> serialize(const Coordinate& value, Endpoint* endpoint) { return serialize_trivial_type<Coordinate>(value, endpoint); } static expected<Coordinate, RuntimeError> deserialize(Endpoint* endpoint) { return deserialize_trivial_type<Coordinate>(endpoint); } }; } // namespace holoscan
```

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

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

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

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

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

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

```
#include "holoscan/core/codec_registry.hpp" namespace holoscan::ops { void MyCoordinateOperator::initialize() { register_codec<Coordinate>("Coordinate"); // ... // parent class initialize() call must be after the argument additions above Operator::initialize(); } } // namespace holoscan::ops
```

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

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

```
namespace holoscan { CodecRegistry::get_instance().add_codec<Coordinate> ("Coordinate"); } // namespace holoscan
```

Tip

CLI arguments (such as `--driver`, `--worker`, `--fragments`) are parsed by the `Application` (

```
<a href="api/cpp/classholoscan_1_1Application.html#_CPPv4N8holoscan11Applicatio </a>
```

/
Pyt
) class and the remaining arguments are available as app.argv (
<a href="api/cpp/classholoscan_1_1Application.html#_CPPv4N8holoscan11Applicatio

/
<a href="api/python/holoscan_python_api_core.html#holoscan.core.Application.argv"
).

Ingested Tab Module

Adding user-defined command line arguments

When adding user-defined command line arguments to an application, one should avoid the use of any of the default command line argument names as `--help`, `--version`, `--config`, `--driver`, `--worker`, `--address`, `--worker-address`, `--fragments` as covered in the section on running a distributed application. It is recommended to parse user-defined arguments from the `argv` ((`C++` / `Python`)) method/property of the application as covered in the note above instead of using C++ `char* argv[]` or Python `sys.argv` directly. This way, only the new, user-defined arguments will need to be parsed.

A concrete example of this for both C++ and Python can be seen in the existing ping_distributed example where an application-defined boolean argument (`--gpu`) is specified in addition to the default set of application arguments.

Ingested Tab Module

Packaging Holoscan Applications

The Holoscan App Packager, included as part of the Holoscan CLI as the `package` command, allows you to package your Holoscan applications into a HAP-compliant container image for distribution and deployment.

Prerequisites

Dependencies

Ensure the following are installed in the environment where you want to run the CLI:

- **PIP dependencies** (automatically installed with the holoscan python wheel)
- **NVIDIA Container Toolkit with Docker**
    - Developer Kits (aarch64): already included in IGX Software and JetPack
    - x86_64: tested with NVIDIA Container Toolkit 1.13.3 w/Docker v24.0.1
- **Docker BuildX plugin**

    1. Check if it is installed:

    > $ docker buildx version github.com/docker/buildx v0.10.5 86bdced

    2. If not, run the following commands based on the official doc:

    > *# Install Docker dependencies* sudo apt-get update sudo apt-get install ca-certificates curl gnupg *# Add Docker Official GPG Key* sudo install -m 0755 -d /etc/apt/keyrings curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o

```
/etc/apt/keyrings/docker.gpg sudo chmod a+r
/etc/apt/keyrings/docker.gpg # Configure Docker APT Repository echo \
"deb [arch="$(dpkg --print-architecture)" signed-
by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \ "$(. /etc/os-release && echo
"$VERSION_CODENAME")" stable" | \ sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null # Install Docker BuildX Plugin
sudo apt-get update sudo apt-get install docker-buildx-plugin
```

- **QEMU** *(Optional)*
  - used for packaging container images of different architectures than the host (example: x86_64 -> arm64)

CLI Installation

The Holoscan CLI is installed as part of the Holoscan SDK and can be called with the following instructions depending on your installation:

Ingested Tab Module

Package an application

> **Tip**
>
> The packager feature is also illustrated in the cli_packager and video_replayer_distributed examples.

1. Ensure to use the HAP environment variables wherever possible when accessing data. For example:

   Let's take a look at the distributed video replayer example (`examples/video_replayer_distributed`).

   - **Using the Application Configuration File**

Ingested Tab Module

- **Using Environment Variable `HOLOSCAN_INPUT_PATH` for Data Input**

  Ingested Tab Module

2. Include a YAML configuration file as described in the Application Runner Configuration page.

3. Use the `holoscan package` command to create a HAP container image. For example:

   ```
   holoscan package --platform x64-workstation --tag my-awesome-app --config /path/to/my/awesome/application/config.yaml /path/to/my/awesome/application/
   ```

Run a packaged application

The packaged Holoscan application container image can run with the Holoscan App Runner:

```
holoscan run -i /path/to/my/input -o /path/to/application/generated/output my-application:1.0.1
```

Since the packaged Holoscan application container images are OCI-compliant, they're also compatible with Docker, Kubernetes, and containerd.

Each packaged Holoscan application container image includes tools inside for extracting the embedded application, manifest files, models, etc. To access the tool and to view all available options, run the following:

```
docker run -it my-container-image[:tag] help
```

The command should prints following:

```
USAGE: /var/holoscan/tools [command] [arguments]... Command List extract ----------
---------- Extract data based on mounted volume paths.
/var/run/holoscan/export/app extract the application
/var/run/holoscan/export/config extract app.json and pkg.json manifest files and
application YAML. /var/run/holoscan/export/models extract models
/var/run/holoscan/export/docs extract documentation files
/var/run/holoscan/export extract all of the above IMPORTANT: ensure the directory
to be mounted for data extraction is created first on the host system. and has the
correct permissions. If the directory had been created by the container previously
with the user and group being root, please delete it and manually create it again.
show ---------- Print manifest file(s): [app|pkg] to the terminal. app print
app.json pkg print pkg.json env ---------- Print all environment variables to
the terminal.
```

Note

The tools can also be accessed inside the Docker container via `/var/holoscan/tools`.

For example, run the following commands to extract the manifest files and the application configuration file:

```
# create a directory on the host system first mkdir -p config-files # mount the directory
created to /var/run/holoscan/export/config docker run -it --rm -v $(pwd)/config-
files:/var/run/holoscan/export/config my-container-image[:tag] extract # include -u
1000 if the above command reports a permission error docker run -it --rm -u 1000 -v
$(pwd)/config-files:/var/run/holoscan/export/config my-container-image[:tag]
extract # If the permission error continues to occur, please check if the mounted
directory has the correct permission. # If it doesn't, please recreate it or change the
permissions as needed. # list files extracted ls config-files/ # output: # app.json
app.yaml pkg.json
```

Creating Operators

Tip

Creating a custom operator is also illustrated in the ping_custom_op example.

C++ Operators

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

1. **Native C++ operators**: custom operators defined in C++ without using the GXF API, by creating a subclass of `holoscan::Operator`. These C++ operators can pass arbitrary C++ objects around between operators.

2. **GXF Operators**: operators defined in the underlying C++ library by inheriting from the `holoscan::ops::GXFOperator` class. These operators wrap GXF codelets from GXF extensions. Examples are `VideoStreamReplayerOp` for replaying video files, `FormatConverterOp` for format conversions, and `HolovizOp` for visualization.

Note

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

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.

digraph lifecycle { rankdir="LR" node [shape=Mrecord]; start [label="start"] compute [label="compute"] stop [label="stop"] start -> compute compute -> compute compute -> stop }

Fig. 15 *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 2 *The basic structure of a Holoscan Operator (C++)*

```
#include "holoscan/holoscan.hpp" using holoscan::Operator; using holoscan::OperatorSpec; using holoscan::InputContext; using holoscan::OutputContext; using holoscan::ExecutionContext; using holoscan::Arg; using holoscan::ArgList; class MyOp : public Operator { public: HOLOSCAN_OPERATOR_FORWARD_ARGS(MyOp) MyOp() = default; void setup(OperatorSpec& spec) override { } void start() override { HOLOSCAN_LOG_TRACE("MyOp::start()"); } void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override { HOLOSCAN_LOG_TRACE("MyOp::compute()"); }; void stop() override { HOLOSCAN_LOG_TRACE("MyOp::stop()"); } };
```

Creating a custom operator (C++)

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

Code Snippet: examples/ping_multi_port/cpp/ping_multi_port.cpp

Listing 3 *examples/ping_multi_port/cpp/ping_multi_port.cpp*

```
#include "holoscan/holoscan.hpp" class ValueData { public: ValueData() = default; explicit ValueData(int value) : data_(value) { HOLOSCAN_LOG_TRACE("ValueData::ValueData(): {}", data_); } ~ValueData() { HOLOSCAN_LOG_TRACE("ValueData::~ValueData(): {}", data_); } void data(int value) { data_ = value; } int data() const { return data_; } private: int data_; }; namespace holoscan::ops { class PingTxOp : public Operator { public: HOLOSCAN_OPERATOR_FORWARD_ARGS(PingTxOp) PingTxOp() = default; void setup(OperatorSpec& spec) override { spec.output<std::shared_ptr<ValueData>>("out1"); spec.output<std::shared_ptr<ValueData>>("out2"); } void compute(InputContext&, OutputContext& op_output, ExecutionContext&) override {
```

```
auto value1 = std::make_shared<ValueData>(index_++); op_output.emit(value1,
"out1"); auto value2 = std::make_shared<ValueData>(index_++);
op_output.emit(value2, "out2"); }; int index_ = 0; }; class PingMiddleOp : public
Operator { public: HOLOSCAN_OPERATOR_FORWARD_ARGS(PingMiddleOp)
PingMiddleOp() = default; void setup(OperatorSpec& spec) override {
spec.input<std::shared_ptr<ValueData>>("in1");
spec.input<std::shared_ptr<ValueData>>("in2");
spec.output<std::shared_ptr<ValueData>>("out1");
spec.output<std::shared_ptr<ValueData>>("out2"); spec.param(multiplier_,
"multiplier", "Multiplier", "Multiply the input by this value", 2); } void
compute(InputContext& op_input, OutputContext& op_output, ExecutionContext&)
override { auto value1 = op_input.receive<std::shared_ptr<ValueData>>
("in1").value(); auto value2 = op_input.receive<std::shared_ptr<ValueData>>
("in2").value(); HOLOSCAN_LOG_INFO("Middle message received (count: {})",
count_++); HOLOSCAN_LOG_INFO("Middle message value1: {}", value1->data());
HOLOSCAN_LOG_INFO("Middle message value2: {}", value2->data()); // Multiply the
values by the multiplier parameter value1->data(value1->data() * multiplier_); value2-
>data(value2->data() * multiplier_); op_output.emit(value1, "out1");
op_output.emit(value2, "out2"); }; private: int count_ = 1; Parameter<int> multiplier_;
}; class PingRxOp : public Operator { public:
HOLOSCAN_OPERATOR_FORWARD_ARGS(PingRxOp) PingRxOp() = default; void
setup(OperatorSpec& spec) override { spec.param(receivers_, "receivers", "Input
Receivers", "List of input receivers.", {}); } void compute(InputContext& op_input,
OutputContext&, ExecutionContext&) override { auto value_vector =
op_input.receive<std::vector<std::shared_ptr<ValueData>>>("receivers").value();
HOLOSCAN_LOG_INFO("Rx message received (count: {}, size: {})", count_++,
value_vector.size()); HOLOSCAN_LOG_INFO("Rx message value1: {}", value_vector[0]-
>data()); HOLOSCAN_LOG_INFO("Rx message value2: {}", value_vector[1]->data()); };
private: Parameter<std::vector<IOSpec*>> receivers_; int count_ = 1; }; } //
namespace holoscan::ops class App : public holoscan::Application { public: void
compose() override { using namespace holoscan; auto tx =
make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10)); auto
mx = make_operator<ops::PingMiddleOp>("mx", Arg("multiplier", 3)); auto rx =
make_operator<ops::PingRxOp>("rx"); add_flow(tx, mx, {{"out1", "in1"}, {"out2",
"in2"}}); add_flow(mx, rx, {{"out1", "receivers"}, {"out2", "receivers"}}); } }; int main(int
```

```
argc, char** argv) { auto app = holoscan::make_application<MyPingApp>(); app->run(); return 0; }
```

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

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

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

2. The `PingMxOp` operator is a middle operator that receives two values from the `PingTxOp` operator and emits two values on two different output ports. The values are multiplied by the `multiplier` parameter.

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

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

Note that for native C++ operators as defined here, any object including a shared pointer can be emitted or received. For large objects such as tensors it may be preferable from a performance standpoint to transmit a shared pointer to the object rather than making a copy. When shared pointers are used and the same tensor is sent to more than one downstream operator, one should avoid in-place operations on the tensor or race conditions between operators may occur.

Specifying operator parameters (C++)

In the example `holoscan::ops::PingMxOp` 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
>
> ```
> <a
> href="api/cpp/classholoscan_1_1Operator.html#_CPPv4I0EN8holoscan8Operator1
> </a>
> ```

See the Configuring operator parameters section to learn how an application can set these parameters.

Specifying operator inputs and outputs (C++)

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

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

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

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

```
void setup(OperatorSpec& spec) override { spec.input<std::shared_ptr<ValueData>>
("in"); // Above statement is equivalent to: // spec.input<std::shared_ptr<ValueData>>
("in") // .condition(ConditionType::kMessageAvailable, Arg("min_size") = 1);
spec.output<std::shared_ptr<ValueData>>("out"); // Above statement is equivalent to:
// spec.output<std::shared_ptr<ValueData>>("out") //
.condition(ConditionType::kDownstreamMessageAffordable, Arg("min_size") = 1); ... }
```

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

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

```
void setup(OperatorSpec& spec) override { spec.input<std::shared_ptr<ValueData>>
("in") .condition(ConditionType::kNone); spec.output<std::shared_ptr<ValueData>>
("out") .condition(ConditionType::kNone); // ... }
```

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

The `receive()` method of the `InputContext` object can be used to access different types of input data within the `compute()` method of your operator class, where its template argument (`DataT`) is the data type of the input. This method takes the name of the input port as an argument (which can be omitted if your operator has a single input port), and returns the input data. If input data is not available, the method returns an object of the `holoscan::RuntimeError` class which contains an error message describing the reason for the failure. The `holoscan::RuntimeError` class is a derived class of `std::runtime_error` and supports accessing more error information, for example, with `what()` method.

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. The `value` is checked to be valid or not with the `if` condition. If `value` is of `holoscan::RuntimeError` type, then `if` condition will be false. Otherwise, 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 { // The type of `value` is `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); // 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 `<a href="api/cpp/classholoscan_1_1Tensor.html#_CPPv4N8holoscan6TensorE">holos` class is a wrapper around the `DLManagedTensorCtx` struct holding a

DLManagedTensor object. As such, it provides a primary interface to access Tensor data and is interoperable with other frameworks that support the DLPack interface.

> ⚠ **Warning**
>
> Passing `<a href="api/cpp/classholoscan_1_1Tensor.html#_CPPv4N8holoscan6TensorE">holos` objects to/from GXF operators directly is not supported. Instead, they need to be passed through `<a href="api/cpp/classholoscan_1_1gxf_1_1Entity.html#_CPPv4N8holoscan3gxf6Entity` 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&lt;IOSpec*&gt;&gt;` (`Parameter&lt;std::vector&lt;IOSpec*&gt;&gt; receivers_`) and calling `spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.", {});` as done for `PingRxOp` in the native operator ping example.

Listing 4 *examples/ping_multi_port/cpp/ping_multi_port.cpp*

```
class PingRxOp : public Operator { public: HOLOSCAN_OPERATOR_FORWARD_ARGS(PingRxOp) PingRxOp() = default; void setup(OperatorSpec& spec) override { spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.", {}); } void compute(InputContext& op_input, OutputContext&, ExecutionContext&) override { auto value_vector = op_input.receive<std::vector<ValueData>>("receivers"); HOLOSCAN_LOG_INFO("Rx message received (count: {}, size: {})", count_++, value_vector.size()); HOLOSCAN_LOG_INFO("Rx message value1: {}", value_vector[0]->data());
```

```
HOLOSCAN_LOG_INFO("Rx message value2: {}", value_vector[1]->data()); }; private: Parameter<std::vector<IOSpec*>> receivers_; int count_ = 1; }; } // namespace holoscan::ops class App : public holoscan::Application { public: void compose() override { using namespace holoscan; auto tx = make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10)); auto mx = make_operator<ops::PingMiddleOp>("mx", Arg("multiplier", 3)); auto rx = make_operator<ops::PingRxOp>("rx"); add_flow(tx, mx, {{"out1", "in1"}, {"out2", "in2"}}); add_flow(mx, rx, {{"out1", "receivers"}, {"out2", "receivers"}}); } };
```

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

```
134: 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 5 */CMakeLists.txt*

```
# Your CMake project cmake_minimum_required(VERSION 3.20) project(my_project CXX) # Finds the holoscan SDK find_package(holoscan REQUIRED CONFIG PATHS "/opt/nvidia/holoscan") # Create a library for your operator add_library(my_operator SHARED my_operator.cpp) # Link your operator against holoscan::core target_link_libraries(my_operator PUBLIC holoscan::core )
```

Once your `CMakeLists.txt` is ready in `<src_dir>` , you can build in `<build_dir>` with the command line below. You can optionally pass `Holoscan_ROOT` if the SDK

installation you'd like to use differs from the `PATHS` given to `find_package(holoscan)` above.

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

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 6 *my_recorder_op.hpp*

```
#ifndef APPS_MY_RECORDER_APP_MY_RECORDER_OP_HPP #define
APPS_MY_RECORDER_APP_MY_RECORDER_OP_HPP #include
"holoscan/core/gxf/gxf_operator.hpp" namespace holoscan::ops { class
MyRecorderOp : public holoscan::ops::GXFOperator { public:
HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER(MyRecorderOp,
holoscan::ops::GXFOperator) MyRecorderOp() = default; const char* gxf_typename()
const override { return "MyRecorder"; } void setup(OperatorSpec& spec) override;
void initialize() override; private: Parameter<holoscan::IOSpec*> receiver_;
Parameter<std::shared_ptr<holoscan::Resource>> my_serializer_;
Parameter<std::string> directory_; Parameter<std::string> basename_;
Parameter<bool> flush_on_tick_; }; } // namespace holoscan::ops #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 7 *Parameter declarations in gxf_extensions/my_recorder/my_recorder.hpp*

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

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

Holoscan Operator	GXF Codelet
`holoscan::Parameter`	`nvidia::gxf::Parameter`
`holoscan::IOSpec*`	`nvidia::gxf::Handle&lt;nvidia::gxf::Receiver&gt;&gt;` or `nvidia::gxf::Handle&lt;nvidia::gxf::Transmitter&gt;&gt;`
`std::shared_ptr&lt;holoscan::Resource&gt;&gt;`	`nvidia::gxf::Handle&lt;T&gt;&gt;` example: `T` is `nvidia::gxf::EntitySerializer`

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 8 *my_recorder_op.cpp*

```
#include "./my_recorder_op.hpp" #include "holoscan/core/fragment.hpp" #include "holoscan/core/gxf/entity.hpp" #include "holoscan/core/operator_spec.hpp" #include "holoscan/core/resources/gxf/video_stream_serializer.hpp" namespace holoscan::ops { void MyRecorderOp::setup(OperatorSpec& spec) { auto& input = spec.input<holoscan::gxf::Entity>("input"); // Above is same with the following two lines (a default condition is assigned to the input port if not specified): // // auto& input = spec.input<holoscan::gxf::Entity>("input") // .condition(ConditionType::kMessageAvailable, Arg("min_size") = 1); spec.param(receiver_, "receiver", "Entity receiver", "Receiver channel to log", &input); spec.param(my_serializer_, "serializer", "Entity serializer", "Serializer for serializing input data"); spec.param(directory_, "out_directory", "Output directory path", "Directory path to store received output"); spec.param(basename_, "basename", "File base name", "User specified file name without extension"); spec.param(flush_on_tick_, "flush_on_tick", "Boolean to flush on tick", "Flushes output buffer on every `tick` when true", false); } void MyRecorderOp::initialize() {...} } // 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 `Condition`s 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 *A part of apps/my_recorder_app_gxf/my_recorder_gxf.yaml*

```
name: recorder components: - name: input type: nvidia::gxf::DoubleBufferReceiver - name: allocator type: nvidia::gxf::UnboundedAllocator - name: component_serializer type: nvidia::gxf::StdComponentSerializer parameters: allocator: allocator - name: entity_serializer type: nvidia::gxf::StdEntitySerializer parameters: component_serializers: [component_serializer] - type: MyRecorder parameters: receiver: input serializer: entity_serializer out_directory: "/tmp" basename: "tensor_out" - type: nvidia::gxf::MessageAvailableSchedulingTerm parameters: receiver: input 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 10 *my_recorder_op.cpp*

```
#include "./my_recorder_op.hpp" #include "holoscan/core/fragment.hpp" #include "holoscan/core/gxf/entity.hpp" #include "holoscan/core/operator_spec.hpp" #include "holoscan/core/resources/gxf/video_stream_serializer.hpp" namespace holoscan::ops { void MyRecorderOp::setup(OperatorSpec& spec) {...} void MyRecorderOp::initialize() { // Set up prerequisite parameters before calling GXFOperator::initialize() auto frag = fragment(); auto serializer = frag->make_resource<holoscan::StdEntitySerializer>("serializer"); add_arg(Arg("serializer") = serializer); GXFOperator::initialize(); } } // 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 11 *Another part of apps/my_recorder_app_gxf/my_recorder_gxf.yaml*

```
name: recorder components: - name: input type: nvidia::gxf::DoubleBufferReceiver - name: allocator type: nvidia::gxf::UnboundedAllocator - name: component_serializer type: nvidia::gxf::StdComponentSerializer parameters: allocator: allocator - name: entity_serializer type: nvidia::gxf::StdEntitySerializer parameters: component_serializers: [component_serializer] - type: MyRecorder parameters: receiver: input serializer: entity_serializer out_directory: "/tmp" basename: "tensor_out" - type: nvidia::gxf::MessageAvailableSchedulingTerm parameters: receiver: input min_size: 1
```

Note

The Holoscan C++ API already provides the `<a href="api/cpp/classholoscan_1_1StdEntitySerializer.html#_CPPv4N8holoscan19Std` class which wraps the `nvidia::gxf::StdEntitySerializer` GXF component, used here as `serializer`.

Building your GXF operator

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

Using your GXF Operator in an Application

There are no differences in CMake between using a GXF operator and using a native C++ operator in an application. However, the application will need to load the GXF extension library which holds the wrapped GXF codelet symbols, so the application needs to be

configured to find the extension library in its yaml configuration file, as documented here.

Interoperability between GXF and native C++ operators

To support sending or receiving tensors to and from operators (both GXF and native C++ operators), the Holoscan SDK provides the C++ classes below:

- A class template called `holoscan::MyMap` which inherits from `std::unordered_map<std::string, std::shared_ptr<T>>`. The template parameter `T` can be any type, and it is used to specify the type of the `std::shared_ptr` objects stored in the map.
-

A `holoscan::TensorMap` class defined as a specialization of `holoscan::Map` for the `holoscan::Tensor` type.

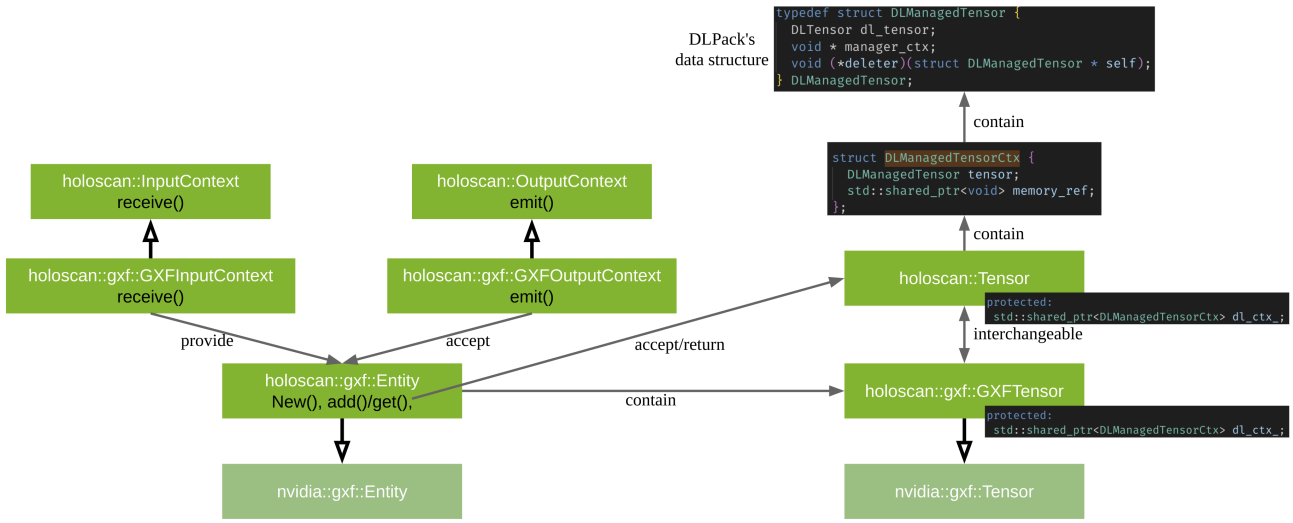


Fig. 16 *Supporting Tensor Interoperability*

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

digraph interop { rankdir="LR" node [shape=record]; source [label="GXFSendTensorOp| |signal(out) : Tensor"]; process [label="ProcessTensorOp| [in]in : TensorMap | out(out) :

TensorMap "]; sink [label="GXFReceiveTensorOp| [in]signal : Tensor | "]; source->process [label="signal...in"] process->sink [label="out...signal"] }

Fig. 17 *The tensor interoperability between C++ native operator and GXF operator*

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

Listing 12 *examples/tensor_interop/cpp/tensor_interop.cpp*

```
void compute(InputContext& op_input, OutputContext& op_output,
ExecutionContext& context) override { // The type of `in_message` is
'holoscan::TensorMap'. auto in_message = op_input.receive<holoscan::TensorMap>
("in").value(); // the type of out_message is TensorMap TensorMap out_message; for
(auto& [key, tensor] : in_message) { // Process with 'tensor' here. cudaError_t
cuda_status; size_t data_size = tensor->nbytes(); std::vector<uint8_t>
in_data(data_size); CUDA_TRY(cudaMemcpy(in_data.data(), tensor->data(), data_size,
cudaMemcpyDeviceToHost)); HOLOSCAN_LOG_INFO("ProcessTensorOp Before key:
'{}', shape: ({}), data: [{}]", key, fmt::join(tensor->shape(), ","), fmt::join(in_data, ","));
for (size_t i = 0; i < data_size; i++) { in_data[i] *= 2; }
HOLOSCAN_LOG_INFO("ProcessTensorOp After key: '{}', shape: ({}), data: [{}]", key,
fmt::join(tensor->shape(), ","), fmt::join(in_data, ","));
CUDA_TRY(cudaMemcpy(tensor->data(), in_data.data(), data_size,
cudaMemcpyHostToDevice)); out_message.insert({key, tensor}); } // Send the
processed message. op_output.emit(out_message); };
```

- The input message is of type `holoscan::TensorMap` object.
- Every `holoscan::Tensor` in the `TensorMap` object is copied on the host as `in_data`.
- The data is processed (values multiplied by 2)
- The data is moved back to the `holoscan::Tensor` object on the GPU.

- A new `holoscan::TensorMap` object `out_message` is created to be sent to the next operator with `op_output.emit()`.

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

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.

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.

digraph lifecycle2 { rankdir="LR" node [shape=Mrecord]; start [label="start"] compute [label="compute"] stop [label="stop"] start -> compute compute -> compute compute -> stop }

Fig. 18 *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 13 *The basic structure of a Holoscan Operator (Python)*

```
from holoscan.core import ( ExecutionContext, InputContext, Operator, OperatorSpec, OutputContext, ) class MyOp(Operator): def __init__(self, fragment, *args, **kwargs): super().__init__(fragment, *args, **kwargs) def setup(self, spec: OperatorSpec): pass def start(self): pass def compute(self, op_input: InputContext, op_output: OutputContext, context: ExecutionContext): pass def stop(self): pass
```

`setup` method vs `initialize` vs `__init__`

The `setup` method aims to get the "operator's spec" by providing `OperatorSpec` object as a spec param. When `__init__` is called, it calls C++'s `Operator::spec` method (and also sets `self.spec` class member), and calls `setup` method so that Operator's `spec` property holds the operator's specification. (See the source code for more details.)

Since the `setup` method can be called multiple times with other `OperatorSpec` object (e.g., to enumerate the operator's description), in the `setup` method, a user shouldn't initialize something in the `Operator` object. Such initialization needs to be done in `initialize` method. The `__init__` method is for creating the Operator object and it can be used for initializing the operator object itself by passing miscellaneous arguments. Still, it doesn't 'initialize' the corresponding GXF entity object.

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 14 *examples/numpy_native/convolve.py*

```
import os from holoscan.conditions import CountCondition from holoscan.core import Application, Operator, OperatorSpec from holoscan.logger import LogLevel, set_log_level import numpy as np class SignalGeneratorOp(Operator): """Generate a time-varying impulse. Transmits an array of zeros with a single non-zero entry of a specified `height`. The position of the non-zero entry shifts to the right (in a periodic fashion) each time `compute` is called. Parameters ---------- fragment : holoscan.core.Fragment The Fragment (or Application) the operator belongs to. height : number The height of the signal impulse. size : number The total number of samples in the generated 1d signal. dtype : numpy.dtype or str The data type of the generated signal. """ def __init__(self, fragment, *args, height=1, size=10, dtype=np.int32, **kwargs): self.count = 0 self.height = height self.dtype = dtype self.size = size super().__init__(fragment, *args, **kwargs) def setup(self, spec: OperatorSpec): spec.output("signal") def compute(self, op_input, op_output, context): # single sample wide impulse at a time-varying position signal = np.zeros((self.size,), dtype=self.dtype) signal[self.count % signal.size] = self.height self.count += 1 op_output.emit(signal, "signal") class ConvolveOp(Operator): """Apply convolution to a tensor. Convolves an input signal with a "boxcar" (i.e. "rect") kernel. Parameters ---------- fragment : holoscan.core.Fragment The Fragment (or Application) the operator belongs to. width : number The width of the boxcar kernel used in the convolution. unit_area : bool, optional Whether or not to normalize the convolution kernel to unit area. If False, all samples have implitude one and the dtype of the kernel will match that of the signal. When True the sum over the kernel is one and a 32-bit floating point data type is used for the kernel. """ def __init__(self, fragment, *args, width=4, unit_area=False, **kwargs): self.count = 0 self.width = width self.unit_area = unit_area super().__init__(fragment, *args, **kwargs) def setup(self, spec: OperatorSpec): spec.input("signal_in") spec.output("signal_out") def compute(self, op_input, op_output, context): signal = op_input.receive("signal_in") assert isinstance(signal, np.ndarray) if self.unit_area: kernel = np.full((self.width,), 1/self.width, dtype=np.float32) else: kernel = np.ones((self.width,), dtype=signal.dtype) convolved = np.convolve(signal, kernel, mode='same') op_output.emit(convolved, "signal_out") class PrintSignalOp(Operator): """Print the received signal to the terminal.""" def setup(self, spec: OperatorSpec): spec.input("signal") def compute(self, op_input, op_output, context): signal = op_input.receive("signal") print(signal) class ConvolveApp(Application): """Minimal
```

```
signal processing application. Generates a time-varying impulse, convolves it with a boxcar kernel, and prints the result to the terminal. A `CountCondition` is applied to the generate to terminate execution after a specific number of steps. """ def compose(self): signal_generator = SignalGeneratorOp( self, CountCondition(self, count=24), name="generator", **self.kwargs("generator"), ) convolver = ConvolveOp(self, name="conv", **self.kwargs("convolve")) printer = PrintSignalOp(self, name="printer") self.add_flow(signal_generator, convolver) self.add_flow(convolver, printer) def main(config_file): app = ConvolveApp() # if the --config command line argument was provided, it will override this config_file` app.config(config_file) app.run() if __name__ == "__main__": config_file = os.path.join(os.path.dirname(__file__), 'convolve.yaml') main(config_file=config_file)
```

Code Snippet: examples/numpy_native/convolve.yaml

Listing 15 *examples/numpy_native/convolve.yaml*

```
signal_generator: height: 1 size: 20 dtype: int32 convolve: width: 4 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

```
<a href="api/python/holoscan_python_api_core.html#holoscan.core.OperatorSpec">(
```

attribute of the operator in its

```
<a href="api/python/holoscan_python_api_core.html#holoscan.core.Operator.setup" </a>
```

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 `kwargs` properties can also be passed to `spec.param` such as `headline`, `description` (used by GXF applications), or `kind` (used when Receiving any number of inputs (Python)).

Note

Native operator parameters added via either of these methods must **not** have a name that overlaps with any of the existing attribute or method names of the base `<a href="api/python/holoscan_python_api_core.html#holoscan.core.Operator">Oper` class.

See the Configuring operator parameters section to learn how an application can set these parameters.

Specifying operator inputs and outputs (Python)

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

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

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

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

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

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

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

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

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

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

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

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

Tip

This <a href="api/python/holoscan_python_api_core.html#holoscan.core.Tensor">holosca class supports both DLPack and NumPy's array interface (<a href="https://numpy.org/doc/stable/reference/arrays.interface.html">__array_inte and <a href="https://numba.readthedocs.io/en/stable/cuda/cuda_array_interface.html">_) so that it can be used with other Python libraries such as CuPy, PyTorch, JAX, TensorFlow, and Numba. See the interoperability section for more details.

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

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

Receiving any number of inputs (Python)

Instead of assigning a specific number of input ports, it may be 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 16 *examples/native_operator/python/ping.py*

```
class PingRxOp(Operator): """Simple receiver operator. This operator has: input: "receivers" This is an example of a native operator that can dynamically have any number of inputs connected to is "receivers" port. """ def __init__(self, fragment, *args, **kwargs): self.count = 1 # Need to call the base class constructor last
```

```
super().__init__(fragment, *args, **kwargs) def setup(self, spec: OperatorSpec): spec.param("receivers", kind="receivers") def compute(self, op_input, op_output, context): values = op_input.receive("receivers") print(f"Rx message received (count: {self.count}, size:{len(values)})") self.count += 1 print(f"Rx message value1: {values[0].data}") 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.

Python wrapping of a C++ operator

> **Note**
>
> While we provide some utilities to simplify part of the process, this section is designed for advanced developers, since the wrapping of the C++ class using pybind11 is mostly manual and can vary greatly between each 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/holoscan/operators`. We’ll highlight the main components below:

Trampoline classes for handling Python kwargs

In a C++ file (`my_op_pybind.cpp` in our skeleton code below), create a subclass of the C++ Operator class to wrap. In the subclass, define a new constructor which takes a `Fragment`, an explicit list of parameters with potential default values (`argA`, `argB` below...), and an operator name to fully initialize the operator similar to what is done in `Fragment::make_operator`:

Listing 17 *my_op_python/my_op_pybind.cpp*

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

Example: Look at the implementation of `PyLSTMTensorRTInferenceOp` on HoloHub for a specific example, or any of the `Py*Op` classes used for the SDK built-in operators here. In the latter, you can find examples of `add_arg` used for less straightforward arguments.

Documentation strings

Prepare documentation strings (`const char*`) for your python class and its parameters, which we'll use in the next step.

> **Note**
>
> Below we use a `PYDOC` macro defined in the SDK and available in HoloHub as a utility to remove leading spaces. In this skeleton example, the documentation code is located in a header file named `my_op_pybind_docs.hpp`, under a custom `doc::MyOp` namespace.

None of this is required, you just need to make the strings available in some way for the next section.

Listing 18 *my_op_python/my_op_pybind_docs.hpp*

```
#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") }
```

Examples: Continuing with the `LSTMTensorRTInferenceOp` example on HoloHub, the documentation strings are defined in lstm_tensor_rt_inference_pydoc.hpp. The documentation strings for the SDK built-in operators are located in operators_pydoc.hpp.

Writing glue code

In the same C++ file as the first section, call `py::class_` within `PYBIND11_MODULE` to define your operator python class.

Note

- If you are implementing the python wrapping in Holohub, the `<module_name>` passed to `PYBIND_11_MODULE` **must** match `_<CPP_CMAKE_TARGET>` (covered in more details in the next section), in this case, `_my_op`.
- If you are implementing the python wrapping in a standalone CMake project,the `<module_name>` passed to

`PYBIND_11_MODULE` **must** match the name of the module passed to the pybind11-add-module CMake function.

Listing 19 *my_op_python/my_op_pybind.cpp (continued)*

```
#include <pybind11/pybind11.h> #include "my_op_pybind_docs.hpp" using pybind11::literals::operator""_a; namespace py = pybind11; #define STRINGIFY(x) #x #define MACRO_STRINGIFY(x) STRINGIFY(x) // See notes above, value of `<module_name>` is important PYBIND11_MODULE(<module_name>, m) { m.doc() = R"pbdoc( My Module Python Bindings ---------- .. currentmodule:: <module_name> .. 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); }
```

Examples: Like the trampoline class, the `PYBIND11_MODULE` implementation of the `LSTMTensorRTInferenceOp` example on HoloHub is located in lstm_tensor_rt_inference.cpp. For the SDK built-in operators, their class bindings are all implemented within a single `PYBIND11_MODULE` in operators.cpp.

Configuring with CMake

We use CMake to configure pybind11 and build the bindings for the C++ operator you wish to wrap. There are two approaches detailed below, one for HoloHub (recommended), one for standalone CMake projects.

Tip

To have your bindings built, ensure the CMake code below is executed as part of a CMake project which already defines the C++

operator as a CMake target, either built in your project (with `add_library`) or imported (with `find_package` or `find_library`).

Ingested Tab Module

Importing the class in Python

Ingested Tab Module

Interoperability between wrapped and native Python operators

As described in the Interoperability between GXF and native C++ operators section, `holoscan::Tensor` objects can be passed to GXF operators using a `holoscan::TensorMap` message that holds the tensor(s). In Python, this is done by sending `dict` type objects that have tensor names as the keys and holoscan Tensor or array-like objects as the values. Similarly, when a wrapped C++ operator that transmits a single `holoscan::Tensor` is connected to the input port of a Python native operator, calling `op_input.receive()` on that port will return a Python dict containing a single item. That item's key is the tensor name and its value is the corresponding `holoscan.core.Tensor`.

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

digraph interop2 { rankdir="LR" node [shape=record]; video [label="VideoStreamReplayerOp| |output_tensor(out) : Tensor"]; processop [label="ImageProcessingOp| [in]input_tensor : dict[str,Tensor] | output_tensor(out) : dict[str,Tensor]"]; viz [label="HolovizOp| [in]receivers : Tensor | "]; video->processop [label="output_tensor...input_tensor"] processop->viz [label="output_tensor...receivers"] }

Fig. 19 *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 22 *examples/tensor_interop/python/tensor_interop.py*

```
def compute(self, op_input, op_output, context): # in_message is of dict in_message = op_input.receive("input_tensor") # smooth along first two axes, but not the color channels sigma = (self.sigma, self.sigma, 0) # out_message is of dict out_message = dict() for key, value in in_message.items(): print(f"message received (count: {self.count})") self.count += 1 cp_array = cp.asarray(value) # process cp_array cp_array = ndi.gaussian_filter(cp_array, sigma) out_message[key] = cp_array op_output.emit(out_message, "output_tensor")
```

- The `op_input.receive()` method call returns a `dict` object.
- The `holoscan.core.Tensor` object is converted to a CuPy array by using `cupy.asarray()` method call.
- The CuPy array is used as an input to the `ndi.gaussian_filter()` function call with a parameter `sigma`. The result of the `ndi.gaussian_filter()` function call is a CuPy array.
- Finally, a new `dict` object is created , `out_message`, to be sent to the next operator with `op_output.emit()`. The CuPy array, `cp_array`, is added to it where the key is the tensor name. CuPy arrays do not have to explicitly be converted to a `holocan.core.Tensor` object first since they implement a DLPack (and `__cuda__array_interface__`) interface.

Note

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

You can add multiple tensors to a single `dict` object , as in the example below:

Operator sending a message:

```
out_message = { "video": output_array, "labels": labels, "bbox_coords": bbox_coords, } # emit the tensors op_output.emit(out_message, "outputs")
```

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

```
in_message = op_input.receive("inputs") # Tensors and tensor names video_tensor = in_message["video"] labels_tensor = in_message["labels"] bbox_coords_tensor = in_message["bbox_coords"]
```

Note

Some existing operators allow configuring the name of the tensors they send/receive. An example is the `tensors` parameter of `<a href="api/python/holoscan_python_api_operators.html#holoscan.operators.Holov`, where the name for each tensor maps to the names of the tensors in the `<a href="api/python/holoscan_python_api_gxf.html#holoscan.gxf.Entity">Entity</a>` (see the `holoviz` entry in apps/endoscopy_tool_tracking/python/endoscopy_tool_tracking.yaml).

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

There is a special serialization code for tensor types for emit/receive of tensor objects over a UCX connection that avoids copying the tensor data to an intermediate buffer. For distributed apps, we cannot just send the Python object as we do between operators in a single fragment app, but instead we need to cast it to `holoscan::Tensor` to use a special zero-copy code path. However, we also transmit a header indicating if the type was originally some other array-like object and attempt to return the same type again on the

other side so that the behavior remains more similar to the non-distributed case.

Transmitted object	Received Object
holoscan.Tensor	holoscan.Tensor
dict of array-like	dict of holoscan.Tensor
host array-like object (with `__array_interface__`)	numpy.ndarray
device array-like object (with `__cuda_array_interface__`)	cupy.ndarray

This avoids NumPy or CuPy arrays being serialized to a string via cloudpickle so that they can efficiently be transmitted and the same type is returned again on the opposite side. Worth mentioning is that ,if the type emitted was e.g. a PyTorch host/device tensor on emit, the received value will be a numpy/cupy array since ANY object implementing the interfaces returns those types.

Logging

Overview

The Holoscan SDK uses the Logger module to convey messages to the user. These messages are categorized into different severity levels (see below) to inform users of the severity of a message and as a way to control the number and verbosity of messages that are printed to the terminal. There are two settings which can be used for this purpose:

- Logger level
- Logger format

Logger Level

Messages that are logged using the Logger module have a severity level, e.g., messages can be categorized as INFO, WARN, ERROR, etc.

The default logging level for an application is to print out messages with severity INFO or above, i.e., messages that are categorized as INFO, WARN, ERROR, and CRITICAL. You can modify this default by calling `set_log_level()` (`C++` / `Python`) in the application code to override the SDK default logging level and give it one of the following log levels.

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

Ingested Tab Module

Additionally, at runtime, the user can set the `HOLOSCAN_LOG_LEVEL` environment variable to one of the values listed above. This provides users with the flexibility to enable printing of diagnostic information for debugging purposes when an issue occurs.

```
export HOLOSCAN_LOG_LEVEL=TRACE
```

Note

Under the hood, Holoscan SDK uses GXF to execute the computation graph. By default, this GXF layer uses the same logging level as Holoscan SDK. If it is desired to override the logging level of this executor independently of the Holoscan SDK logging level, environment variable `HOLOSCAN_EXECUTOR_LOG_LEVEL` can be used. It supports the same levels as `HOLOSCAN_LOG_LEVEL`.

Note

For distributed applications, it can sometimes be useful to also enable additional logging for the UCX library used to transmit data between fragments. This can be done by setting the UCX environment variable `UCX_LOG_LEVEL` to one of: fatal, error, warn, info, debug, trace, req, data, async, func, poll. These have the behavior as described here: UCX log levels.

Logger Format

When a message is printed out, the default message format shows the message severity level, filename:linenumber, and the message to be printed.

For example:

```
[info] [ping_multi_port.cpp:114] Rx message value1: 51 [info]
[ping_multi_port.cpp:115] Rx message value2: 54
```

You can modify this default by calling `set_log_pattern()` (`C++` / `Python`) in the application code to override the SDK default logging format.

The pattern string can be one of the following pre-defined values

- **SHORT** : prints message severity level, and message
- **DEFAULT** : prints message severity level, filename:linenumber, and message
- **LONG** : prints timestamp, application, message severity level, filename:linenumber, and message
- **FULL** : prints timestamp, thread id, application, message severity level, filename:linenumber, and message

Ingested Tab Module

With this logger format, the above application would display messages with the following format:

```
[info] Rx message value1: 51 [info] Rx message value2: 54
```

Alternatively, the pattern string can be a custom pattern to customize the logger format. Using this string pattern

```
"[%Y-%m-%d %H:%M:%S.%e] [%n] [%^%l%$] [%s:%#] %v";
```

would display messages with the following format:

```
[2023-06-27 14:22:36.073] [holoscan] [info] [ping_multi_port.cpp:114] Rx message
value1: 51 [2023-06-27 14:22:36.073] [holoscan] [info] [ping_multi_port.cpp:115] Rx
message value2: 54
```

For more details on custom formatting and details of each flag, please see the spdlog wiki page.

Additionally, at runtime, the user can also set the `HOLOSCAN_LOG_FORMAT` environment variable to modify the logger format. The accepted string pattern is the same as the string pattern for the `set_log_pattern()` api mentioned above.

Precedence of Logger Level and Logger Format

The `HOLOSCAN_LOG_LEVEL` environment variable takes precedence and overrides the application settings, such as `Logger::set_log_level()` (`C++` / `Python`).

When `HOLOSCAN_LOG_LEVEL` is set, it determines the logging level. If this environment variable is unset, the application settings are used if they are available. Otherwise, the SDK's default logging level of INFO is applied.

Similarly, the `HOLOSCAN_LOG_FORMAT` environment variable takes precedence and overrides the application settings, such as `Logger::set_log_pattern()` (`C++` / `Python`).

When `HOLOSCAN_LOG_FORMAT` is set, it determines the logging format. If this environment variable is unset, the application settings are used if they are available. Otherwise, the SDK's default logging format depending on the current log level (`FULL` format for `DEBUG` and `TRACE` log levels. `DEFAULT` format for other log levels) is applied.

Calling the Logger in Your Application

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

> **Note**
>
> Holoscan automatically checks `HOLOSCAN_LOG_LEVEL` environment variable and sets the log level when the Application class instance is created. However, those log level settings are for Holoscan core or C++ operator (C++)'s logging message (such as `HOLOSCAN_LOG_INFO` macro), not for Python's logging. Users of the

Python API should use the built-in `<a href="https://docs.python.org/3/howto/logging.html">logging</a>` module to log messages. The user needs to configure the logger before use (`logging.basicConfig(level=logging.INFO)`):

```
>>> import logging >>> logger = logging.getLogger("main") >>> logger.info('hello') >>> logging.basicConfig(level=logging.INFO) >>> logger.info('hello') INFO:main:hello
```

Debugging

Overview

The Holoscan SDK is designed to streamline the debugging process for developers working on advanced applications.

This comprehensive guide covers the SDK's debugging capabilities, with a focus on Visual Studio Code integration, and provides detailed instructions for various debugging scenarios.

It includes methods for debugging both the C++ and Python components of applications, utilizing tools like GDB, UCX, and Python-specific debuggers.

Visual Studio Code Integration

VSCode Dev Container

The Holoscan SDK can be effectively developed using Visual Studio Code, leveraging the capabilities of a development container. This container, defined in the `.devcontainer` folder, is pre-configured with all the necessary tools and libraries, as detailed in Visual Studio Code's documentation on development containers.

Launching VSCode with the Holoscan SDK

- **Local Development**: Use the `./run vscode` command to launch Visual Studio Code in a development container.
- **Remote Development**: For attaching to an existing dev container from a remote machine, use `./run vscode_remote`. Additional instructions can be accessed via `./run vscode_remote -h`.

Upon launching Visual Studio Code, the development container will automatically be built. This process also involves the installation of recommended extensions and the configuration of CMake.

Configuring CMake

For manual adjustments to the CMake configuration:

1. Open the command palette in VSCode (`Ctrl + Shift + P`).
2. Execute the `CMake: Configure` command.

Building the Source Code

To build the source code within the development container:

- Either press `Ctrl + Shift + B`.
- Or use the command palette (`Ctrl + Shift + P`) and run `Tasks: Run Build Task`.

Debugging Workflow

For debugging the source code:

1. Open the `Run and Debug` view in VSCode (`Ctrl + Shift + D`).
2. Select an appropriate debug configuration from the dropdown.
3. Press `F5` to start the debugging session.

The launch configurations are defined in `.vscode/launch.json` (link).

Please refer to Visual Studio Code's documentation on debugging for more information.

Integrated Debugging for C++ and Python in Holoscan SDK

The Holoscan SDK facilitates seamless debugging of both C++ and Python components within your applications. This is achieved through the integration of the `Python C++ Debugger` extension in Visual Studio Code, which can be found here.

This powerful extension is specifically designed to enable effective debugging of Python operators that are executed within the C++ runtime environment. Additionally, it provides robust capabilities for debugging C++ operators and various SDK components that are executed via the Python interpreter.

To utilize this feature, debug configurations for `Python C++ Debug` should be defined within the `.vscode/launch.json` file, available here.

Here's how to get started:

1. Open a Python file within your project, such as `examples/ping_vector/python/ping_vector.py`.

2. In the `Run and Debug` view of Visual Studio Code, select the `Python C++ Debug` debug configuration.

3. Set the necessary breakpoints in both your Python and C++ code.

4. Initiate the debugging session by pressing `F5`.

Upon starting the session, two separate debug terminals will be launched - one for Python and another for C++. In the C++ terminal, you will encounter a prompt regarding superuser access:

> Superuser access is required to attach to a process. Attaching as superuser can potentially harm your computer. Do you want to continue? [y/N]

Respond with `y` to proceed.

Following this, the Python application initiates, and the C++ debugger attaches to the Python process. This setup allows you to simultaneously debug both Python and C++ code. The `CALL STACK` tab in the `Run and Debug` view will display `Python: Debug Current File` and `(gdb) Attach`, indicating active debugging sessions for both languages.

By leveraging this integrated debugging approach, developers can efficiently troubleshoot and enhance applications that utilize both Python and C++ components within the Holoscan SDK.

Debugging an Application Crash

This section outlines the procedures for debugging an application crash.

Core Dump Analysis

In the event of an application crash, you might encounter messages like `Segmentation fault (core dumped)` or `Aborted (core dumped)`. These indicate the generation of a core dump file, which captures the application's memory state at the time of the crash. This file can be utilized for debugging purposes.

Enabling coredump

There are instances where core dumps might be disabled or not generated despite an application crash.

To activate core dumps, it's necessary to configure the `ulimit` setting, which determines the maximum size of core dump files. By default, `ulimit` is set to 0, effectively disabling core dumps. Setting `ulimit` to unlimited enables the generation of core dumps.

```
ulimit -c unlimited
```

Additionally, configuring the `core_pattern` value is required. This value specifies the naming convention for the core dump file. To view the current `core_pattern` setting, execute the following command:

```
cat /proc/sys/kernel/core_pattern # or sysctl kernel.core_pattern
```

To modify the `core_pattern` value, execute the following command:

```
echo "coredump_%e_%p" | sudo tee /proc/sys/kernel/core_pattern # or sudo sysctl -w kernel.core_pattern=coredump_%e_%p
```

where in this case we have requested both the executable name (`%e`) and the process id (`%p`) be present in the generated file's name. The various options available are documented in the core documentation.

If you encounter errors like `tee: /proc/sys/kernel/core_pattern: Read-only file system` or `sysctl: setting key "kernel.core_pattern", ignoring: Read-only file system` within a Docker container, it's advisable to set the `kernel.core_pattern` parameter on the host system instead of within the container.

As `kernel.core_pattern` is a system-wide kernel parameter, modifying it on the host should impact all containers. This method, however, necessitates appropriate permissions on the host machine.

Furthermore, when launching a Docker container using `docker run`, it's often essential to include the `--cap-add=SYS_PTRACE` option to enable core dump creation inside the container. Core dump generation typically requires elevated privileges, which are not automatically available to Docker containers.

Using GDB to Debug a coredump File

After the core dump file is generated, you can utilize GDB to debug the core dump file.

Consider a scenario where a segmentation fault is intentionally induced at line 29 in `examples/ping_simple/cpp/ping_simple.cpp` by adding the line `*(int*)0 = 0;` to trigger the fault.

```
--- a/examples/ping_simple/cpp/ping_simple.cpp +++
b/examples/ping_simple/cpp/ping_simple.cpp @@ -19,7 +19,6 @@ #include
<holoscan/operators/ping_tx/ping_tx.hpp> #include
<holoscan/operators/ping_rx/ping_rx.hpp> - class MyPingApp : public
holoscan::Application { public: void compose() override { @@ -27,6 +26,7 @@ class
MyPingApp : public holoscan::Application { // Define the tx and rx operators,
allowing the tx operator to execute 10 times auto tx =
make_operator<ops::PingTxOp>("tx", make_condition<CountCondition>(10)); auto
rx = make_operator<ops::PingRxOp>("rx"); + *(int*)0 = 0;
```

Upon running `./examples/ping_simple/cpp/ping_simple`, the following output is observed:

```
$ ./examples/ping_simple/cpp/ping_simple Segmentation fault (core dumped)
```

It's apparent that the application has aborted and a core dump file has been generated.

```
$ ls coredump* coredump_ping_simple_2160275
```

The core dump file can be debugged using GDB by executing `gdb &lt;application&gt; &lt;coredump_file&gt;`.

> $ gdb ./examples/ping_simple/cpp/ping_simple coredump_ping_simple_2160275 GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1 Copyright (C) 2022 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html> This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law. Type "show copying" and "show warranty" for details. This GDB was configured as "x86_64-linux-gnu". Type "show configuration" for configuration details. For bug reporting instructions, please see: <https://www.gnu.org/software/gdb/bugs/>. Find the GDB manual and other documentation resources online at: <http://www.gnu.org/software/gdb/documentation/>. For help, type "help". Type "apropos word" to search for commands related to "word"... Reading symbols from ./examples/ping_simple/cpp/ping_simple... [New LWP 2160275] [Thread debugging using libthread_db enabled] Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1". Core was generated by `./examples/ping_simple/cpp/ping_simple'. Program terminated with signal SIGSEGV, Segmentation fault. #0 MyPingApp::compose (this=0x563bd3a3de80) at ../examples/ping_simple/cpp/ping_simple.cpp:29 29 *(int*)0 = 0; (gdb)

It is evident that the application crashed at line 29 of `examples/ping_simple/cpp/ping_simple.cpp`.

To display the backtrace, the `bt` command can be executed.

> (gdb) bt *#0 MyPingApp::compose (this=0x563bd3a3de80) at ../examples/ping_simple/cpp/ping_simple.cpp:29 #1 0x00007f2a76cdb5ea in holoscan::Application::compose_graph (this=0x563bd3a3de80) at ../src/core/application.cpp:325 #2 0x00007f2a76c3d121 in holoscan::AppDriver::check_configuration (this=0x563bd3a42920) at ../src/core/app_driver.cpp:803 #3 0x00007f2a76c384ef in holoscan::AppDriver::run (this=0x563bd3a42920) at ../src/core/app_driver.cpp:168 #4 0x00007f2a76cda70c in holoscan::Application::run (this=0x563bd3a3de80) at ../src/core/application.cpp:207 #5*

> *0x0000563bd2ec4002 in main (argc=1, argv=0x7ffea82c4c28) at ../examples/ping_simple/cpp/ping_simple.cpp:38*

UCX Segmentation Fault Handler

In cases where a distributed application using the UCX library encounters a segmentation fault, you might see stack traces from UCX. This is a default configuration of the UCX library to output stack traces upon a segmentation fault. However, this behavior can be modified by setting the `UCX_HANDLE_ERRORS` environment variable:

- `UCX_HANDLE_ERRORS=bt` prints a backtrace during a segmentation fault (default setting).
- `UCX_HANDLE_ERRORS=debug` attaches a debugger if a segmentation fault occurs.
- `UCX_HANDLE_ERRORS=freeze` freezes the application on a segmentation fault.
- `UCX_HANDLE_ERRORS=freeze,bt` both freezes the application and prints a backtrace upon a segmentation fault.
- `UCX_HANDLE_ERRORS=none` disables backtrace printing during a segmentation fault.

While the default action is to print a backtrace on a segmentation fault, it may not always be helpful.

For instance, if a segmentation fault is intentionally caused at line 129 in `examples/ping_distributed/cpp/ping_distributed_ops.cpp` (by adding `*(int*)0 = 0;`), running `./examples/ping_distributed/cpp/ping_distributed` will result in the following output:

> [holoscan:2097261:0:2097311] Caught signal 11 (Segmentation fault: address not mapped to object at address (nil)) ==== backtrace (tid:2097311) ==== 0 /opt/ucx/1.15.0/lib/libucs.so.0(ucs_handle_error+0x2e4) [0x7f18db865264] 1 /opt/ucx/1.15.0/lib/libucs.so.0(+0x3045f) [0x7f18db86545f] 2 /opt/ucx/1.15.0/lib/libucs.so.0(+0x30746) [0x7f18db865746] 3 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x42520) [0x7f18da9ee520] 4 ./examples/ping_distributed/cpp/ping_distributed(+0x103d2b) [0x5651dafc7d2b] 5

```
/workspace/holoscan-sdk/build-debug-
x86_64/lib/libholoscan_core.so.1(_ZN8holoscan3gxf10GXFWrapper4tickEv+0x13d)
[0x7f18dcbfaafd] 6 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_core.so(_ZN6nvidia3gxf14EntityExecutor10EntityItem11tickCodeletEF
[0x7f18db2cb487] 7 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_core.so(_ZN6nvidia3gxf14EntityExecutor10EntityItem4tickElPNS0_6R
[0x7f18db2cde44] 8 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_core.so(_ZN6nvidia3gxf14EntityExecutor10EntityItem7executeElPNS0
[0x7f18db2ce859] 9 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_core.so(_ZN6nvidia3gxf14EntityExecutor13executeEntityEll+0x41b)
[0x7f18db2cf0cb] 10 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_serialization.so(_ZN6nvidia3gxf20MultiThreadScheduler20workerThr
[0x7f18daf0cc50] 11 /usr/lib/x86_64-linux-gnu/libstdc++.so.6(+0xdc253)
[0x7f18dacb0253] 12 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x94ac3)
[0x7f18daa40ac3] 13 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x126660)
[0x7f18daad2660] ================================= Segmentation fault
(core dumped)
```

Although a backtrace is provided, it may not always be helpful as it often lacks source code information. To obtain detailed source code information, using a debugger is necessary.

By setting the `UCX_HANDLE_ERRORS` environment variable to `freeze,bt` and running `./examples/ping_distributed/cpp/ping_distributed`, we can observe that the thread responsible for the segmentation fault is frozen, allowing us to attach a debugger to it for further investigation.

```
$ UCX_HANDLE_ERRORS=freeze,bt
./examples/ping_distributed/cpp/ping_distributed [holoscan:2127091:0:2127105]
Caught signal 11 (Segmentation fault: address not mapped to object at address (nil))
==== backtrace (tid:2127105) ==== 0
/opt/ucx/1.15.0/lib/libucs.so.0(ucs_handle_error+0x2e4) [0x7f9995850264] 1
/opt/ucx/1.15.0/lib/libucs.so.0(+0x3045f) [0x7f999585045f] 2
/opt/ucx/1.15.0/lib/libucs.so.0(+0x30746) [0x7f9995850746] 3 /usr/lib/x86_64-linux-
gnu/libc.so.6(+0x42520) [0x7f99949ee520] 4
./examples/ping_distributed/cpp/ping_distributed(+0x103d2b) [0x55971617fd2b] 5
```

```
/workspace/holoscan-sdk/build-debug-
x86_64/lib/libholoscan_core.so.1(_ZN8holoscan3gxf10GXFWrapper4tickEv+0x13d)
[0x7f9996bfaafd] 6 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_core.so(_ZN6nvidia3gxf14EntityExecutor10EntityItem11tickCodeletEF
[0x7f99952cb487] 7 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_core.so(_ZN6nvidia3gxf14EntityExecutor10EntityItem4tickElPNS0_6R
[0x7f99952cde44] 8 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_core.so(_ZN6nvidia3gxf14EntityExecutor10EntityItem7executeElPNS0
[0x7f99952ce859] 9 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_core.so(_ZN6nvidia3gxf14EntityExecutor13executeEntityEll+0x41b)
[0x7f99952cf0cb] 10 /workspace/holoscan-sdk/build-debug-
x86_64/lib/libgxf_serialization.so(_ZN6nvidia3gxf20MultiThreadScheduler20workerThr
[0x7f9994f0cc50] 11 /usr/lib/x86_64-linux-gnu/libstdc++.so.6(+0xdc253)
[0x7f9994cb0253] 12 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x94ac3)
[0x7f9994a40ac3] 13 /usr/lib/x86_64-linux-gnu/libc.so.6(+0x126660)
[0x7f9994ad2660] ================================
[holoscan:2127091:0:2127105] Process frozen, press Enter to attach a debugger...
```

It is observed that the thread responsible for the segmentation fault is 2127105 (`tid:2127105`). To attach a debugger to this thread, simply press Enter.

Upon attaching the debugger, a backtrace will be displayed, but it may not be from the thread that triggered the segmentation fault. To handle this, use the `info threads` command to list all threads, and the `thread &lt;thread_id&gt;` command to switch to the thread that caused the segmentation fault.

```
(gdb) info threads Id Target Id Frame * 1 Thread 0x7f9997b36000 (LWP 2127091)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 2 Thread 0x7f9992731000 (LWP 2127093)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 3 Thread 0x7f9991f30000 (LWP 2127094)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 4 Thread 0x7f999172f000 (LWP 2127095)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 5 Thread 0x7f99909ec000 (LWP 2127096)
"cuda-EvtHandlr" 0x00007f9994a96612 in __libc_pause () at
```

```
../sysdeps/unix/sysv/linux/pause.c:29 6 Thread 0x7f99891ff000 (LWP 2127097)
"async" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 7 Thread 0x7f997d7cd000 (LWP 2127098)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 8 Thread 0x7f997cfcc000 (LWP 2127099)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 9 Thread 0x7f995ffff000 (LWP 2127100)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 10 Thread 0x7f99577fe000 (LWP 2127101)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 11 Thread 0x7f995f3e5000 (LWP 2127103)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 12 Thread 0x7f995ebe4000 (LWP 2127104)
"ping_distribute" 0x00007f9994a96612 in __libc_pause () at
../sysdeps/unix/sysv/linux/pause.c:29 13 Thread 0x7f995e3e3000 (LWP 2127105)
"ping_distribute" 0x00007f9994a9642f in __GI___wait4 (pid=pid@entry=2127631,
stat_loc=stat_loc@entry=0x7f995e3ddd3c, options=options@entry=0,
usage=usage@entry=0x0) at ../sysdeps/unix/sysv/linux/wait4.c:30
```

It's evident that thread ID 13 is responsible for the segmentation fault (`LWP 2127105`). To investigate further, we can switch to this thread using the command `thread 13` in GDB:

```
(gdb) thread 13
```

After switching, we can employ the `bt` command to examine the backtrace of this thread.

```
(gdb) bt #0 0x00007f9994a9642f in __GI___wait4 (pid=pid@entry=2127631,
stat_loc=stat_loc@entry=0x7f995e3ddd3c, options=options@entry=0,
usage=usage@entry=0x0) at ../sysdeps/unix/sysv/linux/wait4.c:30 #1
0x00007f9994a963ab in __GI___waitpid (pid=pid@entry=2127631,
stat_loc=stat_loc@entry=0x7f995e3ddd3c, options=options@entry=0) at
./posix/waitpid.c:38 #2 0x00007f999584d587 in ucs_debugger_attach () at
/opt/ucx/src/contrib/../src/ucs/debug/debug.c:816 #3 0x00007f999585031d in
```

ucs_error_freeze (message=0x7f999586ec53 "address not mapped to object") at /opt/ucx/src/contrib/../src/ucs/debug/debug.c:919 #4 ucs_handle_error (message=0x7f999586ec53 "address not mapped to object") at /opt/ucx/src/contrib/../src/ucs/debug/debug.c:1089 #5 ucs_handle_error (message=0x7f999586ec53 "address not mapped to object") at /opt/ucx/src/contrib/../src/ucs/debug/debug.c:1077 #6 0x00007f999585045f in ucs_debug_handle_error_signal (signo=signo@entry=11, cause=0x7f999586ec53 "address not mapped to object", fmt=fmt@entry=0x7f999586ecf5 " at address %p") at /opt/ucx/src/contrib/../src/ucs/debug/debug.c:1038 #7 0x00007f9995850746 in ucs_error_signal_handler (signo=11, info=0x7f995e3de3f0, context=<optimized out>) at /opt/ucx/src/contrib/../src/ucs/debug/debug.c:1060 #8 <signal handler called> #9 holoscan::ops::PingTensorTxOp::compute (this=0x559716f26fa0, op_output=..., context=...) at ../examples/ping_distributed/cpp/ping_distributed_ops.cpp:129 #10 0x00007f9996bfaafd in holoscan::gxf::GXFWrapper::tick (this=0x559716f6f740) at ../src/core/gxf/gxf_wrapper.cpp:66 #11 0x00007f99952cb487 in nvidia::gxf::EntityExecutor::EntityItem::tickCodelet(nvidia::gxf::Handle<nvidia::gxf::Codelet> const&) () from /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_core.so #12 0x00007f99952cde44 in nvidia::gxf::EntityExecutor::EntityItem::tick(long, nvidia::gxf::Router) () from /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_core.so #13 0x00007f99952ce859 in nvidia::gxf::EntityExecutor::EntityItem::execute(long, nvidia::gxf::Router*, long&) () from /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_core.so #14 0x00007f99952cf0cb in nvidia::gxf::EntityExecutor::executeEntity(long, long) () from /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_core.so #15 0x00007f9994f0cc50 in nvidia::gxf::MultiThreadScheduler::workerThreadEntrance(nvidia::gxf::ThreadPool*, long) () from /workspace/holoscan-sdk/build-debug-x86_64/lib/libgxf_serialization.so #16 0x00007f9994cb0253 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6 #17 0x00007f9994a40ac3 in start_thread (arg=<optimized out>) at ./nptl/pthread_create.c:442 #18 0x00007f9994ad2660 in clone3 () at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:81*

Under the backtrace of thread 13, you will find:

> #8 <signal handler called> #9 holoscan::ops::PingTensorTxOp::compute (this=0x559716f26fa0, op_output=..., context=...) at ../examples/ping_distributed/cpp/ping_distributed_ops.cpp:129

This indicates that the segmentation fault occurred at line 129 in `examples/ping_distributed/cpp/ping_distributed_ops.cpp`.

To view the backtrace of all threads, use the `thread apply all bt` command.

> (gdb) thread apply all bt ... Thread 13 (Thread 0x7f995e3e3000 (LWP 2127105) "ping_distribute"): *#0 0x00007f9994a9642f in __GI___wait4 (pid=pid@entry=2127631, stat_loc=stat_loc@entry=0x7f995e3ddd3c, options=options@entry=0, usage=usage@entry=0x0) at ../sysdeps/unix/sysv/linux/wait4.c:30* ... Thread 12 (Thread 0x7f995ebe4000 (LWP 2127104) "ping_distribute"): *#0 0x00007f9994a96612 in __libc_pause () at ../sysdeps/unix/sysv/linux/pause.c:29* ...

Debugging Holoscan Python Application

The Holoscan SDK provides support for tracing and profiling tools, particularly focusing on the `compute` method of Python operators. Debugging Python operators using Python IDEs can be challenging since this method is invoked from the C++ runtime. This also applies to the `initialize`, `start`, and `stop` methods of Python operators.

Users can leverage IDEs like VSCode/PyCharm (which utilize the PyDev.Debugger) or other similar tools to debug Python operators:

- For VSCode, refer to VSCode Python Debugging.
- For PyCharm, consult PyCharm Python Debugging.

Subsequent sections will detail methods for debugging, profiling, and tracing Python applications using the Holoscan SDK.

pdb example

The following command initiates a Python application within a `pdb` session:

```
python python/tests/system/test_pytracing.py pdb # Type the following commands to check if the breakpoints are hit: # # b test_pytracing.py:76 # c # exit
```

```
This is an interactive session. Please type the following commands to check if the breakpoints are hit. (Pdb) b test_pytracing.py:76 Breakpoint 1 at /workspace/holoscan-sdk/python/tests/system/test_pytracing.py:76 (Pdb) c ... > /workspace/holoscan-sdk/python/tests/system/test_pytracing.py(76)start() -> print("Mx start") (Pdb) exit
```

For more details, please refer to the `pdb_main()` method in `test_pytracing.py`.

Profiling a Holoscan Python Application

For profiling, users can employ tools like cProfile or line_profiler for profiling Python applications/operators.

Note that when using a multithreaded scheduler, cProfile or the profile module might not accurately identify worker threads, or errors could occur.

In such cases with multithreaded schedulers, consider using multithread-aware profilers like pyinstrument, pprofile, or yappi.

For further information, refer to the test case at test_pytracing.py.

Using pyinstrument

pyinstrument is a call stack profiler for Python, designed to highlight performance bottlenecks in an easily understandable format directly in your terminal as the code executes.

```
python -m pip install pyinstrument pyinstrument python/tests/system/test_pytracing.py ## Note: With a multithreaded scheduler, the same method may appear multiple times across different threads. # pyinstrument python/tests/system/test_pytracing.py -s multithread
```

```
... 0.107 [root] None 0.088 MainThread <thread>:140079743820224 0.088 <module> ../../../bin/pyinstrument:1 0.088 main pyinstrument/__main__.py:28 [7 frames hidden] pyinstrument, <string>, runpy, <built... 0.087 _run_code
```

```
runpy.py:63         0.087 <module> test_pytracing.py:1          0.061 main
test_pytracing.py:153           0.057 MyPingApp.compose test_pytracing.py:141
    0.041 PingMxOp.__init__ test_pytracing.py:59                0.041 PingMxOp.__init__
../core/__init__.py:262           [35 frames hidden] .., numpy, re, sre_compile,
sre_parse...              0.015 [self] test_pytracing.py            0.002 [self]
test_pytracing.py         0.024 <module> ../__init__.py:1       [5 frames hidden] .., <built-
in>         0.001 <module> ../conditions/__init__.py:1    [2 frames hidden] .., <built-in>
    0.019 Dummy-1 <thread>:140078275749440      0.019 <module>
../../../bin/pyinstrument:1       0.019 main pyinstrument/__main__.py:28 [5 frames
hidden] pyinstrument, <string>, runpy 0.019 _run_code runpy.py:63       0.019
<module> test_pytracing.py:1       0.019 main test_pytracing.py:153       0.014 [self]
test_pytracing.py       0.004 PingRxOp.compute test_pytracing.py:118       0.004 print
<built-in>
```

Using pprofile

pprofile is a line-granularity, thread-aware deterministic and statistic pure-python profiler.

```
python -m pip install pprofile pprofile --include test_pytracing.py
python/tests/system/test_pytracing.py -s multithread
```

```
Total duration: 0.972872s File: python/tests/system/test_pytracing.py File duration:
0.542628s (55.78%) Line #| Hits| Time| Time per hit| %|Source code ------+----------+-
-----------+-------------+-------+----------- ... 33| 0| 0| 0| 0.00%| 34| 2| 2.86102e-06|
1.43051e-06| 0.00%| def setup(self, spec: OperatorSpec): 35| 1| 1.62125e-05|
1.62125e-05| 0.00%| spec.output("out") 36| 0| 0| 0| 0.00%| 37| 2| 3.33786e-06|
1.66893e-06| 0.00%| def initialize(self): 38| 1| 1.07288e-05| 1.07288e-05| 0.00%|
print("Tx initialize") 39| 0| 0| 0| 0.00%| 40| 2| 1.40667e-05| 7.03335e-06| 0.00%|
def start(self): 41| 1| 1.23978e-05| 1.23978e-05| 0.00%| print("Tx start") 42| 0| 0|
0| 0.00%| 43| 2| 3.09944e-05| 1.54972e-05| 0.00%| def stop(self): 44| 1|
2.88486e-05| 2.88486e-05| 0.00%| print("Tx stop") 45| 0| 0| 0| 0.00%| 46| 4|
4.05312e-05| 1.01328e-05| 0.00%| def compute(self, op_input, op_output, context):
47| 3| 2.57492e-05| 8.58307e-06| 0.00%| value = self.index 48| 3| 2.12193e-05|
7.07308e-06| 0.00%| self.index += 1
```

Using yappi

yappi is a tracing profiler that is multithreading, asyncio and gevent aware.

```
python -m pip install yappi # yappi requires setting a context ID callback function to specify the correct context ID for # Holoscan's worker threads. # For more details, please see `yappi_main()` in `test_pytracing.py`. python python/tests/system/test_pytracing.py yappi | grep test_pytracing.py ## Note: With a multithreaded scheduler, method hit counts are distributed across multiple threads. #python python/tests/system/test_pytracing.py yappi -s multithread | grep test_pytracing.py
```

```
... test_pytracing.py main:153 1 test_pytracing.py MyPingApp.compose:141 1 test_pytracing.py PingMxOp.__init__:59 1 test_pytracing.py PingTxOp.__init__:29 1 test_pytracing.py PingMxOp.setup:65 1 test_pytracing.py PingRxOp.__init__:99 1 test_pytracing.py PingRxOp.setup:104 1 test_pytracing.py PingTxOp.setup:34 1 test_pytracing.py PingTxOp.initialize:37 1 test_pytracing.py PingRxOp.stop:115 1 test_pytracing.py PingRxOp.initialize:109 1 test_pytracing.py PingMxOp.initialize:72 1 test_pytracing.py PingMxOp.stop:78 1 test_pytracing.py PingMxOp.compute:81 3 test_pytracing.py PingTxOp.compute:46 3 test_pytracing.py PingRxOp.compute:118 3 test_pytracing.py PingTxOp.start:40 1 test_pytracing.py PingMxOp.start:75 1 test_pytracing.py PingRxOp.start:112 1 test_pytracing.py PingTxOp.stop:43 1
```

Using profile/cProfile

profile/cProfile is a deterministic profiling module for Python programs.

```
python -m cProfile python/tests/system/test_pytracing.py 2>&1 | grep test_pytracing.py ## Executing a single test case #python python/tests/system/test_pytracing.py profile
```

```
1 0.001 0.001 0.107 0.107 test_pytracing.py:1(<module>) 1 0.000 0.000 0.000 0.000 test_pytracing.py:104(setup) 1 0.000 0.000 0.000 0.000 test_pytracing.py:109(initialize) 1 0.000 0.000 0.000 0.000 test_pytracing.py:112(start) 1 0.000 0.000 0.000 0.000 test_pytracing.py:115(stop) 3 0.000 0.000 0.000 0.000 test_pytracing.py:118(compute) 1 0.000 0.000 0.000 0.000 test_pytracing.py:140(MyPingApp) 1 0.014 0.014 0.073 0.073
```

```
test_pytracing.py:141(compose) 1 0.009 0.009 0.083 0.083
test_pytracing.py:153(main) 1 0.000 0.000 0.000 0.000
test_pytracing.py:28(PingTxOp) 1 0.000 0.000 0.000 0.000
test_pytracing.py:29(__init__) 1 0.000 0.000 0.000 0.000 test_pytracing.py:34(setup) 1
0.000 0.000 0.000 0.000 test_pytracing.py:37(initialize) 1 0.000 0.000 0.000 0.000
test_pytracing.py:40(start) 1 0.000 0.000 0.000 0.000 test_pytracing.py:43(stop) 3
0.000 0.000 0.000 0.000 test_pytracing.py:46(compute) 1 0.000 0.000 0.000 0.000
test_pytracing.py:58(PingMxOp) 1 0.000 0.000 0.058 0.058
test_pytracing.py:59(__init__) 1 0.000 0.000 0.000 0.000 test_pytracing.py:65(setup) 1
0.000 0.000 0.000 0.000 test_pytracing.py:72(initialize) 1 0.000 0.000 0.000 0.000
test_pytracing.py:75(start) 1 0.000 0.000 0.000 0.000 test_pytracing.py:78(stop) 3
0.001 0.000 0.001 0.000 test_pytracing.py:81(compute) 1 0.000 0.000 0.000 0.000
test_pytracing.py:98(PingRxOp) 1 0.000 0.000 0.000 0.000
test_pytracing.py:99(__init__)
```

Using line_profiler

line_profiler is a module for doing line-by-line profiling of functions.

```
python -m pip install line_profiler # Insert `@profile` before the function `def
compute(self, op_input, op_output, context):`. # The original file will be backed up as
`test_pytracing.py.bak`. file="python/tests/system/test_pytracing.py" pattern=" def
compute(self, op_input, op_output, context):" insertion=" @profile" if ! grep -q
"^$insertion" "$file"; then sed -i.bak "/^$pattern/i\\ $insertion" "$file" fi kernprof -lv
python/tests/system/test_pytracing.py # Remove the inserted `@profile` decorator. mv
"$file.bak" "$file"
```

```
... Wrote profile results to test_pytracing.py.lprof Timer unit: 1e-06 s Total time:
0.000304244 s File: python/tests/system/test_pytracing.py Function: compute at line
46 Line # Hits Time Per Hit % Time Line Contents
============================================================== 46
@profile 47 def compute(self, op_input, op_output, context): 48 3 2.3 0.8 0.8 value =
self.index 49 3 9.3 3.1 3.0 self.index += 1 50 51 3 0.5 0.2 0.2 output = [] 52 18 5.0 0.3
1.6 for i in range(0, 5): 53 15 4.2 0.3 1.4 output.append(value) 54 15 2.4 0.2 0.8 value
+= 1 55 56 3 280.6 93.5 92.2 op_output.emit(output, "out") ...
```

Measuring Code Coverage

The Holoscan SDK provides support for measuring code coverage using Coverage.py.

```
python -m pip install coverage coverage erase coverage run examples/ping_vector/python/ping_vector.py coverage report examples/ping_vector/python/ping_vector.py coverage html # Open the generated HTML report in a browser. xdg-open htmlcov/index.html
```

To record code coverage programmatically, please refer to the `coverage_main()` method in `test_pytracing.py`.

You can execute the example application with code coverage enabled by running the following command:

```
python -m pip install coverage python python/tests/system/test_pytracing.py coverage # python python/tests/system/test_pytracing.py coverage -s multithread
```

The following command starts a Python application using the `trace`:

```
python -m trace --trackcalls python/tests/system/test_pytracing.py | grep test_pytracing
```

```
... test_pytracing.main -> test_pytracing.MyPingApp.compose test_pytracing.main -> test_pytracing.PingMxOp.compute test_pytracing.main -> test_pytracing.PingMxOp.initialize test_pytracing.main -> test_pytracing.PingMxOp.start test_pytracing.main -> test_pytracing.PingMxOp.stop test_pytracing.main -> test_pytracing.PingRxOp.compute test_pytracing.main -> test_pytracing.PingRxOp.initialize test_pytracing.main -> test_pytracing.PingRxOp.start test_pytracing.main -> test_pytracing.PingRxOp.stop test_pytracing.main -> test_pytracing.PingTxOp.compute test_pytracing.main -> test_pytracing.PingTxOp.initialize test_pytracing.main -> test_pytracing.PingTxOp.start test_pytracing.main -> test_pytracing.PingTxOp.stop
```

A test case utilizing the `trace` module programmatically can be found in the `trace_main()` method in `test_pytracing.py`.

```
python python/tests/system/test_pytracing.py trace # python python/tests/system/test_pytracing.py trace -s multithread
```

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.

Operators

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

Class	CMake target/lib	Documentation
AJASourceOp	`aja`	`C++` / `Python`
BayerDemosaicOp	`bayer_demosaic`	`C++` / `Python`
FormatConverterOp	`format_converter`	`C++` / `Python`
HolovizOp	`holoviz`	`C++` / `Python`
InferenceOp	`inference`	`C++` / `Python`
InferenceProcessorOp	`inference_processor`	`C++` / `Python`
PingRxOp	`ping_rx`	`C++` / `Python`
PingTxOp	`ping_tx`	`C++` / `Python`
SegmentationPostprocessorOp	`segmentation_postprocessor`	`C++` / `Python`
VideoStreamRecorderOp	`video_stream_recorder`	`C++` / `Python`
VideoStreamReplayerOp	`video_stream_replayer`	`C++` / `Python`

V4L2VideoCaptureOp	v4l2	C++ / Python

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

Ingested Tab Module

> **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.

Extensions

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

- gxf_holoscan_wrapper
- ucx_holoscan

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.

UCX (Holoscan)

The `ucx_holoscan` extension includes `nvidia::holoscan::UcxHoloscanComponentSerializer` which is a `nvidia::gxf::ComponentSerializer` that handles serialization of `holoscan::Message` and `holoscan::Tensor` types for transmission using the Unified Communication X (UCX) library. UCX is the library used by Holoscan SDK to enable communication of data between fragments in distributed applications.

> **Note**
>
> The `UcxHoloscanComponentSerializer` is intended for use in combination with other UCX components defined in the GXF UCX extension. Specifically, it can be used by the `UcxEntitySerializer` where it can operate alongside the `UcxComponentSerializer` that serializes GXF-specific types (`nvidia::gxf::Tensor`, `nvidia::gxf::VideoBuffer`, etc.). This way both GXF and Holoscan types can be serialized by distributed applications.

HoloHub

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

Visualization

Overview

Holoviz provides the functionality to composite real time streams of frames with multiple different other layers like segmentation mask layers, geometry layers and GUI layers.

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

Holoscan provides the Holoviz operator which is sufficient for many, even complex visualization tasks. The Holoviz operator is used by multiple Holoscan example applications.

Additionally, for more advanced use cases, the Holoviz module can be used to create application specific visualization operators. The Holoviz module provides a C++ API and is also used by the Holoviz operator.

The term Holoviz is used for both the Holoviz operator and the Holoviz module below. Both the operator and the module roughly support the same features set. Where applicable information how to use a feature with the operator and the module is provided. It's explicitly mentioned below when features are not supported by the operator.

Layers

The core entity of Holoviz are layers. A layer is a two-dimensional image object. Multiple layers are composited to create the final output.

These layer types are supported by Holoviz:

- image layer
- geometry layer
- GUI layer

All layers have common attributes which define the look and also the way layers are finally composited.

The priority determines the rendering order of the layers. Before rendering the layers 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.

The example below draws a transparent geometry layer on top of an image layer (geometry data and image data creation is omitted in the code). Although the geometry layer is specified first, it is drawn last because it has a higher priority (`1`) than the image layer (`0`).

Ingested Tab Module

Image Layers

Ingested Tab Module

Supported Image Formats

Ingested Tab Module

Geometry Layers

A geometry layer is used to draw geometric primitives such as points, lines, rectangles, ovals or text.

Coordinates start with (0, 0) in the top left and end with (1, 1) in the bottom right.

Ingested Tab Module

ImGui Layers

> **Note**
>
> ImGui layers are not supported when using the Holoviz operator.

The Holoviz module supports user interface layers created with Dear ImGui.

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

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

```
namespace viz = holoscan::viz; bool show_image_layer = false; while (!viz::WindowShouldClose()) { viz::Begin(); viz::BeginImGuiLayer(); ImGui::Begin("Options"); ImGui::Checkbox("Image layer", &show_image_layer); ImGui::End(); viz::EndLayer(); if (show_image_layer) { viz::BeginImageLayer(); viz::ImageHost(...); viz::EndLayer(); } viz::End(); }
```

ImGUI is a static library and has no stable API. Therefore the application and Holoviz have to use the same ImGUI version. Therefore the link target `holoscan::viz::imgui` is exported, make sure to link your app against that target.

Depth Map Layers

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

Supported 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)

Ingested Tab Module

Views

By default a layer will fill the whole window. When using a view, the layer can be placed freely within the window.

Layers can also be placed in 3D space by specifying a 3D transformation matrix.

> **Note**
>
> For geometry layers there is a default matrix which allows coordinates in the range of [0 ... 1] instead of the Vulkan [-1 ... 1] range. When specifying a matrix for a geometry layer, this default matrix is overwritten.

When multiple views are specified the layer is drawn multiple times using the specified layer view.

It's possible to specify a negative term for height, which flips the image. When using a negative height, one should also adjust the y value to point to the lower left corner of the viewport instead of the upper left corner.

Ingested Tab Module

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.

Configure a display for exclusive use

Ingested Tab Module

Enable exclusive display in Holoviz

Ingested Tab Module

The name of the display can either be the EDID name as displayed in the NVIDIA Settings, or the output name used by `xrandr`.

> **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)
> ```

CUDA streams

By default Holoviz is using CUDA stream `0` for all CUDA operations. Using the default stream can affect concurrency of CUDA operations, see stream synchronization behavior for more information.

Ingested Tab Module

Reading the framebuffer

The rendered frame buffer can be read back. This is useful when when doing offscreen rendering or running Holoviz in a headless environment.

> **Note**
>
> Reading the depth buffer is not supported when using the Holoviz operator.

Ingested Tab Module

Holoviz operator

Class documentation

C++

Python.

Examples

There are multiple examples both in Python and C++ showing how to use various features of the Holoviz operator.

Holoviz module

Concepts

The Holoviz module uses the concept of the immediate mode design pattern for its API, inspired by the Dear ImGui library. The difference to the retained mode, for which most APIs are designed 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.

Instances

The Holoviz module uses a thread-local instance object to store its internal state. The instance object is created when calling the Holoviz module is first called from a thread. All Holoviz module functions called from that thread use this instance.

When calling into the Holoviz module from other threads other than the thread from which the Holoviz module functions were first called, make sure to call `viz::GetCurrent()` and `viz::SetCurrent()` in the respective threads.

There are usage cases where multiple instances are needed, for example, to open multiple windows. Instances can be created by calling `viz::Create()`. Call `viz::SetCurrent()` to make the instance current before calling the Holoviz module function to be executed for the window the instance belongs to.

Getting started

The code below creates a window and displays an image.

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

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

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

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

```
#include "holoviz/holoviz.hpp" namespace viz = holoscan::viz; viz::Init("Holoviz Example"); while (!viz::WindowShouldClose()) { viz::Begin(); viz::BeginImageLayer(); viz::ImageHost(width, height, viz::ImageFormat::R8G8B8A8_UNORM, image_data); viz::EndLayer(); viz::End(); } viz::Shutdown();
```

Result:

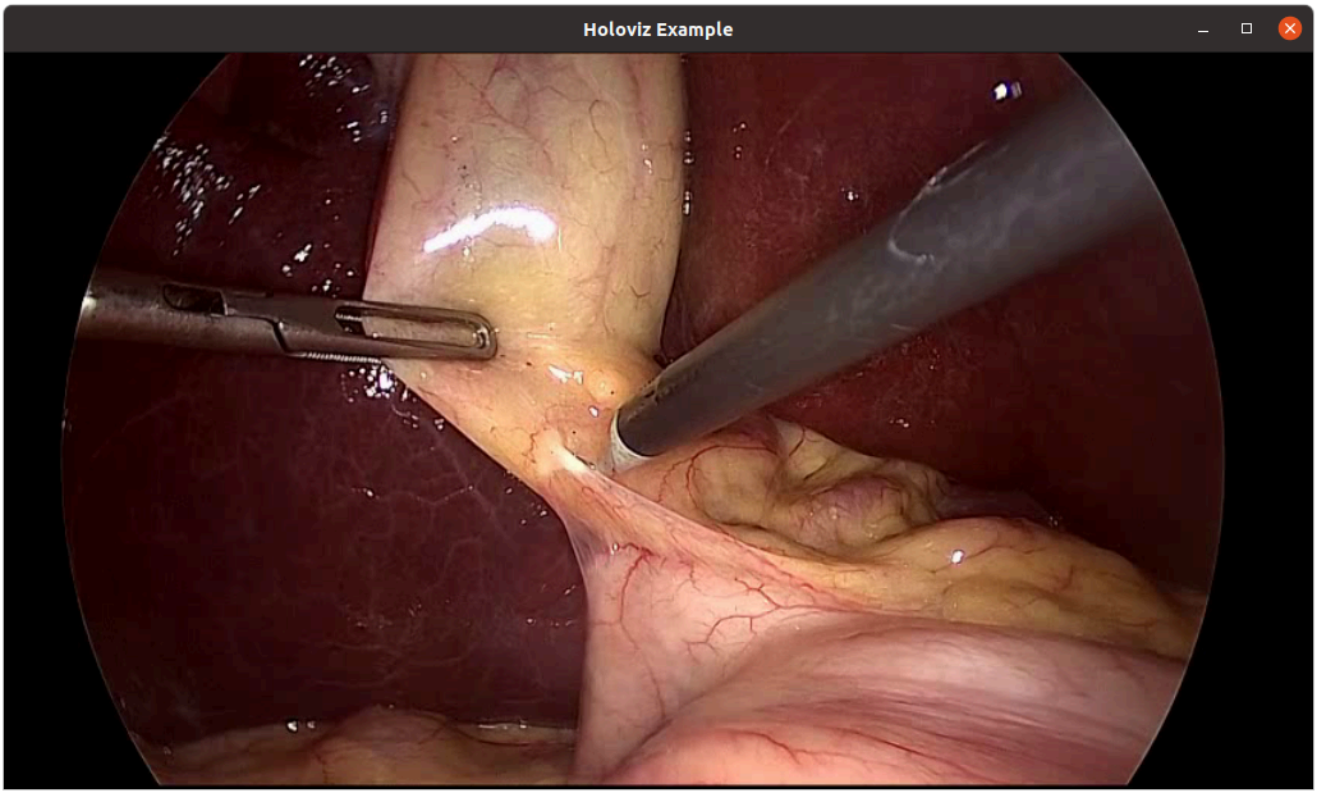


Fig. 20 *Holoviz example app*

API

Holoviz module API

Examples

There are multiple examples showing how to use various features of the Holoviz module.

Inference

Overview

A Holoscan application that needs to run inference will use an inference operator. The built-in Inference operator (`InferenceOp`) can be used, and several related use cases are documented in the Inference operator section below. The use cases are created using the parameter set that must be defined in the configuration file of the holoscan application. If the built-in `InferenceOp` doesn't cover a specific use case, users can create their own custom inference operator as documented in Creating an Inference operator section.

The core inference functionality in the Holoscan SDK is provided by the Inference Module which is a framework that facilitates designing and executing inference and processing applications through its APIs. It is used by the built-in `InferenceOp` which supports the same parameters as the Inference Module. All parameters required by the Holoscan Inference Module are passed through a parameter set in the configuration file of an application.

Parameters and related Features

Required parameters and related features available with the Holoscan Inference Module are listed below.

- Data Buffer Parameters: Parameters are provided in the inference settings to enable data buffer locations at several stages of the inference. As shown in the figure below, three parameters `input_on_cuda` , `output_on_cuda` and `transmit_on_cuda` can be set by the user.
    - `input_on_cuda` refers to the location of the data going into the inference.
        - If value is `true` , it means the input data is on the device
        - If value is `false` , it means the input data is on the host
        - Default value: `true`

- `output_on_cuda` refers to the data location of the inferred data.
  - If value is `true`, it means the inferred data is on the device
  - If value is `false`, it means the inferred data is on the host
  - Default value: `true`
- `transmit_on_cuda` refers to the data transmission.
  - If value is `true`, it means the data transmission from the inference extension will be on **Device**
  - If value is `false`, it means the data transmission from the inference extension will be on **Host**
  - Default value: `true`

- Inference Parameters
  - `backend` parameter is set to either `trt` for TensorRT, `onnxrt` for Onnx runtime, or `torch` for libtorch. If there are multiple models in the inference application, all models will use the same backend. If it is desired to use different backends for different models, specify the `backend_map` parameter instead.
    - TensorRT:
      - CUDA-based inference supported both on x86_64 and aarch64
      - End-to-end CUDA-based data buffer parameters supported. `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` will all be true for end-to-end CUDA-based data movement.
      - `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` can be either `true` or `false`.
      - TensorRT backend expects input models to be in `tensorrt engine file` format or `onnx` format.

        - if models are in `tensorrt engine file` format, parameter `is_engine_path` must be set to `true`.
        - if models are in `onnx` format, it will be automatically converted into `tensorrt engine file` by the Holoscan inference module.
- Torch:
    - CUDA and CPU based inference supported both on x86_64 and aarch64.
    - End-to-end CUDA-based data buffer parameters supported. `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` will all be true for end-to-end CUDA-based data movement.
    - `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` can be either `true` or `false`.
    - Libtorch and TorchVision are included in the Holoscan NGC container, initially built as part of the PyTorch NGC container. To use the Holoscan SDK torch backend outside of these containers, we recommend you download libtorch and torchvision binaries from Holoscan's third-party repository.
    - Torch backend expects input models to be in `torchscript` format.
        - It is recommended to use the same version of torch for `torchscript` model generation, as used in the HOLOSCAN SDK on the respective architectures.
        - Additionally, it is recommended to generate the `torchscript` model on the same architecture on which it will be executed. For example, `torchscript` model must be generated on `x86_64` to be executed in an application running on `x86_64` only.
- Onnx runtime:

        - Data flow via host only. `input_on_cuda`, `output_on_cuda` and `transmit_on_cuda` must be `false`.
        - CUDA based inference (supported on x86_64)
        - CPU based inference (supported on x86_64 and aarch64)
    - `infer_on_cpu` parameter is set to `true` if CPU based inference is desired.

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

x86	**input_on_cuda**	**output_on_cuda**	**transmit_on_cuda**	**infer_on_cpu**
Supported values for `trt`	`true` or `false`	`true` or `false`	`true` or `false`	`false`
Supported values for `torch`	`true` or `false`	`true` or `false`	`true` or `false`	`true` or `false`
Supported values for `onnxrt`	`false`	`false`	`true` or `false`	`true` or `false`

Aarch64	**input_on_cuda**	**output_on_cuda**	**transmit_on_cuda**	**infer_on_cpu**
Supported values for `trt`	`true` or `false`	`true` or `false`	`true` or `false`	`false`

Supported values for `torch`	`true` or `false`	`true` or `false`	`true` or `false`	`true` or `false`
Supported values for `onnxrt`	`false`	`false`	`true` or `false`	`true`

- `model_path_map`: User can design single or multi AI inference pipeline by populating `model_path_map` in the config file.
  - With a single entry it is single inference 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** or **torchscript** format.
- `pre_processor_map`: input tensor to the respective model is specified in `pre_processor_map` in the config file.
  - The Holoscan Inference Module supports same input for multiple models or unique input per model.
  - Each entry in `pre_processor_map` has a unique keyword representing the model (same as used in `model_path_map`), and a vector of tensor names as the value.
  - The Holoscan Inference Module supports multiple input tensors per model.

- `inference_map`: output tensors per model after inference is specified in `inference_map` in the config file.
    - Each entry in `inference_map` has a unique keyword representing the model (same as used in `model_path_map` and `pre_processor_map`), and a vector of the output tensor names as the value.
    - The Holoscan Inference Module supports multiple output tensors per model.
- `parallel_inference`: Parallel or Sequential execution of inferences.
    - If multiple models are input, then user can execute models in parallel.
    - Parameter `parallel_inference` can be either `true` or `false`. Default value is `true`.
    - Inferences are launched in parallel without any check of the available GPU resources, 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 mintues to generate the engine files for the first time.
    - It can be either `true` or `false`. Default value is `false`.
- `is_engine_path`: if the input models are specified in **trt engine format** in `model_path_map`, this flag must be set to `true`. Default value is `false`.
- `in_tensor_names`: Input tensor names to be used by `pre_processor_map`. This parameter is optional. If absent in the parameter map, values are derived from `pre_processor_map`.

- `out_tensor_names`: Output tensor names to be used by `inference_map`. This parameter is optional. If absent in the parameter map, values are derived from `inference_map`.
- `device_map`: Multi-GPU inferencing is enabled if `device_map` is populated in the parameter set.
  - Each entry in `device_map` has a unique keyword representing the model (same as used in `model_path_map` and `pre_processor_map`), and GPU identifier as the value. This GPU ID is used to execute the inference for the specified model.
  - GPUs specified in the `device_map` must have P2P (peer to peer) access and they must be connected to the same PCIE configuration. If P2P access is not possible among GPUs, the host (CPU memory) will be used to transfer the data.
  - Multi-GPU inferencing is supported for all backends.
- `backend_map`: Multiple backends can be used in the same application with this parameter.
  - Each entry in `backend_map` has a unique keyword representing the model (same as used in `model_path_map`), and the `backend` as the value.
  - A sample backend_map is shown below. In the example, model_1 uses the `tensorRT` backend, and model 2 and model 3 uses the `torch` backend for inference.

    ```
    backend_map: "model_1_unique_identifier": "trt"
    "model_2_unique_identifier": "torch" "model_3_unique_identifier":
    "torch"
    ```

- Other features: Table below illustrates other features and supported values in the current release.

Feature	Supported values

Data type	`float32` , `int32` , `int8`
Inference Backend	`trt` , `torch` , `onnxrt`
Inputs per model	Multiple
Outputs per model	Multiple
GPU(s) supported	Multi-GPU on same PCIE network
Tensor data dimension	2, 3, 4
Model Type	`All onnx` or `all torchscript` or `all trt engine` type or a `combination of torch and trt engine`

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

Parameter Specification

All required inference parameters of the inference application must be specified. Below is a sample parameter set for an application that uses three models for inferencing. User must populate all required fields with appropriate values.

```
inference: backend: "trt" model_path_map: "model_1_unique_identifier":
"path_to_model_1" "model_2_unique_identifier": "path_to_model_2"
"model_3_unique_identifier": "path_to_model_3" pre_processor_map:
"model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
"model_2_unique_identifier": ["input_tensor_1_model_2_unique_identifier"]
"model_3_unique_identifier": ["input_tensor_1_model_3_unique_identifier"]
inference_map: "model_1_unique_identifier":
["output_tensor_1_model_1_unique_identifier"] "model_2_unique_identifier":
```

```
["output_tensor_1_model_2_unique_identifier"] "model_3_unique_identifier":
["output_tensor_1_model_3_unique_identifier"] parallel_inference: true
infer_on_cpu: false enable_fp16: false input_on_cuda: true output_on_cuda: true
transmit_on_cuda: true is_engine_path: false
```

Inference Operator

In Holoscan SDK, the built-in Inference operator (`InferenceOp`) is designed using the Holoscan Inference Module APIs. The Inference operator ingests the inference parameter set (from the configuration file) and the data receivers (from previous connected operators in the application), executes the inference and transmits the inferred results to the next connected operators in the application.

`InferenceOp` is a generic operator that serves multiple use cases via the parameter set. Parameter sets for some key use cases are listed below:

Note: Some parameters have default values set for them in the `InferenceOp` . For any parameters not mentioned in the example parameter sets below, their default is used by the `InferenceOp` . These parameters are used to enable several use cases.

- Single model inference using `TensorRT` backend.

  ```
  backend: "trt" model_path_map: "model_1_unique_identifier":
  "path_to_model_1" pre_processor_map: "model_1_unique_identifier":
  ["input_tensor_1_model_1_unique_identifier"] inference_map:
  "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
  ```

  Value of `backend` can be modified for other supported backends, and other parameters related to each backend. User must ensure correct model type and model path is provided into the parameter set, along with supported values of all parameters for the respective backend.

  In this example, `path_to_model_1` must be an `onnx` file, which will be converted to a `tensorRT` engine file at first execution. During subsequent executions, the Holoscan inference module will automatically find the tensorRT engine file (if `path_to_model_1` has not changed). Additionally, if user has a pre-built `tensorRT`

engine file, `path_to_model_1` must be path to the engine file and the parameter `is_engine_path` must be set to `true` in the parameter set.

- Single model inference using `TensorRT` backend with multiple outputs.

  ```
  backend: "trt" model_path_map: "model_1_unique_identifier": "path_to_model_1" pre_processor_map: "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"] inference_map: "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier", "output_tensor_2_model_1_unique_identifier", "output_tensor_3_model_1_unique_identifier"]
  ```

  As shown in example above, Holoscan Inference module automatically maps the model outputs to the named tensors in the parameter set. Users must ensure to use the named tensors in the same sequence in which the model generates the output. Similar logic holds for multiple inputs.

- Single model inference using fp16 precision.

  ```
  backend: "trt" model_path_map: "model_1_unique_identifier": "path_to_model_1" pre_processor_map: "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"] inference_map: "model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier", "output_tensor_2_model_1_unique_identifier", "output_tensor_3_model_1_unique_identifier"] enable_fp16: true
  ```

  If a `tensorRT` engine file is not available for fp16 precision, it will be automatically generated by the Holoscan Inference module on the first execution. The file is cached for future executions.

- Single model inference on CPU.

  ```
  backend: "onnxrt" model_path_map: "model_1_unique_identifier": "path_to_model_1" pre_processor_map: "model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"] inference_map:
  ```

```
"model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
infer_on_cpu: true
```

Note that the backend can only be `onnxrt` or `torch` for CPU based inference.

- Single model inference with input/output data on Host.

```
backend: "trt" model_path_map: "model_1_unique_identifier":
"path_to_model_1" pre_processor_map: "model_1_unique_identifier":
["input_tensor_1_model_1_unique_identifier"] inference_map:
"model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
input_on_cuda: false output_on_cuda: false
```

Data in the core inference engine is passed through the host and is received on the host. Inference can happen on the GPU. Parameters `input_on_cuda` and `output_on_cuda` define the location of the data before and after inference respectively.

- Single model inference with data transmission via Host.

```
backend: "trt" model_path_map: "model_1_unique_identifier":
"path_to_model_1" pre_processor_map: "model_1_unique_identifier":
["input_tensor_1_model_1_unique_identifier"] inference_map:
"model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
transmit_on_host: true
```

Data from inference operator to the next connected operator in the application is transmitted via the host.

- Multi model inference with a single backend.

```
backend: "trt" model_path_map: "model_1_unique_identifier":
"path_to_model_1" "model_2_unique_identifier": "path_to_model_2"
"model_3_unique_identifier": "path_to_model_3" pre_processor_map:
"model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
"model_2_unique_identifier": ["input_tensor_1_model_2_unique_identifier"]
```

```
"model_3_unique_identifier": ["input_tensor_1_model_3_unique_identifier"]
inference_map: "model_1_unique_identifier":
["output_tensor_1_model_1_unique_identifier"] "model_2_unique_identifier":
["output_tensor_1_model_2_unique_identifier"] "model_3_unique_identifier":
["output_tensor_1_model_3_unique_identifier"]
```

By default multiple model inferences are launched in parallel. The backend specified via parameter `backend` is used for all models in the application.

- Multi model inference with sequential inference.

```
backend: "trt" model_path_map: "model_1_unique_identifier":
"path_to_model_1" "model_2_unique_identifier": "path_to_model_2"
"model_3_unique_identifier": "path_to_model_3" pre_processor_map:
"model_1_unique_identifier": ["input_tensor_1_model_1_unique_identifier"]
"model_2_unique_identifier": ["input_tensor_1_model_2_unique_identifier"]
"model_3_unique_identifier": ["input_tensor_1_model_3_unique_identifier"]
inference_map: "model_1_unique_identifier":
["output_tensor_1_model_1_unique_identifier"] "model_2_unique_identifier":
["output_tensor_1_model_2_unique_identifier"] "model_3_unique_identifier":
["output_tensor_1_model_3_unique_identifier"] parallel_inference: false
```

`parallel_inference` is set to `true` by default. To launch model inferences in sequence, `parallel_inference` must be set to `false`.

- Multi model inference with multiple backends.

```
backend_map: "model_1_unique_identifier": "trt" "model_2_unique_identifier":
"torch" "model_3_unique_identifier": "torch" model_path_map:
"model_1_unique_identifier": "path_to_model_1" "model_2_unique_identifier":
"path_to_model_2" "model_3_unique_identifier": "path_to_model_3"
pre_processor_map: "model_1_unique_identifier":
["input_tensor_1_model_1_unique_identifier"] "model_2_unique_identifier":
["input_tensor_1_model_2_unique_identifier"] "model_3_unique_identifier":
["input_tensor_1_model_3_unique_identifier"] inference_map:
```

```
"model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
"model_2_unique_identifier": ["output_tensor_1_model_2_unique_identifier"]
"model_3_unique_identifier": ["output_tensor_1_model_3_unique_identifier"]
```

In the above sample parameter set, the first model will do inference using the `tensorRT` backend, and model 2 and 3 will do inference using the `torch` backend.

Note: the combination of backends in `backend_map` must support all other parameters that will be used during the inference. For. e.g. `onnxrt` and `tensorRT` combination with CPU based inference will not be supported.

- Multi model inference with a single backend on multi-GPU.

```
backend: "trt" device_map: "model_1_unique_identifier": "1"
"model_2_unique_identifier": "0" "model_3_unique_identifier": "1"
model_path_map: "model_1_unique_identifier": "path_to_model_1"
"model_2_unique_identifier": "path_to_model_2" "model_3_unique_identifier":
"path_to_model_3" pre_processor_map: "model_1_unique_identifier":
["input_tensor_1_model_1_unique_identifier"] "model_2_unique_identifier":
["input_tensor_1_model_2_unique_identifier"] "model_3_unique_identifier":
["input_tensor_1_model_3_unique_identifier"] inference_map:
"model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
"model_2_unique_identifier": ["output_tensor_1_model_2_unique_identifier"]
"model_3_unique_identifier": ["output_tensor_1_model_3_unique_identifier"]
```

In the sample above, model 1 and model 3 will do inference on the GPU with ID 1 and model 2 will do inferene on the GPU with ID 0. GPUs must have P2P (peer to peer) access among them. If it is not enabled, the Holoscan inference module enables it by default. If P2P access is not possible between GPUs, then the data transfer will happen via the Host.

- Multi model inference with multiple backends on multiple GPUs.

```
backend_map: "model_1_unique_identifier": "trt" "model_2_unique_identifier":
"torch" "model_3_unique_identifier": "torch" device_map:
"model_1_unique_identifier": "1" "model_2_unique_identifier": "0"
```

```
"model_3_unique_identifier": "1" model_path_map:
"model_1_unique_identifier": "path_to_model_1" "model_2_unique_identifier":
"path_to_model_2" "model_3_unique_identifier": "path_to_model_3"
pre_processor_map: "model_1_unique_identifier":
["input_tensor_1_model_1_unique_identifier"] "model_2_unique_identifier":
["input_tensor_1_model_2_unique_identifier"] "model_3_unique_identifier":
["input_tensor_1_model_3_unique_identifier"] inference_map:
"model_1_unique_identifier": ["output_tensor_1_model_1_unique_identifier"]
"model_2_unique_identifier": ["output_tensor_1_model_2_unique_identifier"]
"model_3_unique_identifier": ["output_tensor_1_model_3_unique_identifier"]
```

In the sample above, three models are used during the inference. Model 1 uses the trt backend and runs on the GPU with ID 1, model 2 uses the torch backend and runs on the GPU with ID 0, and model 3 uses the torch backend and runs on the GPU with ID 1.

Creating an Inference operator

The Inference operator is the core inference unit in an inference application. The built-in Inference operator (`InferenceOp`) can be used for inference, or users can create their own custom inference operator as explained in this section. In Holoscan SDK, the inference operator can be designed using the Holoscan Inference Module APIs.

Arguments in the code sections below are referred to as **...**.

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

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

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

  ```
  // Declaration of inference specifications
  std::shared_ptr<HoloInfer::InferenceSpecs> inference_specs_; // Creation of
  ```

```
inference specification structure inference_specs_ =
std::make_shared<HoloInfer::InferenceSpecs>(...);
```

- Inference context creation.

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

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

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

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

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

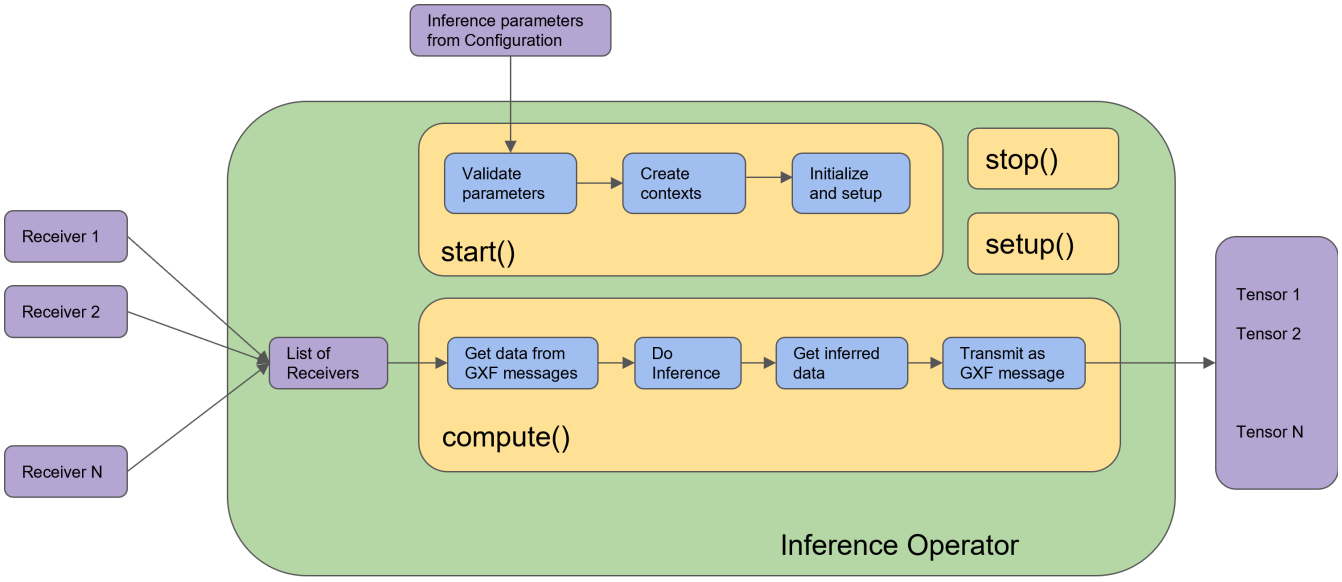
- Inference execution

```
// Execute inference and populate output buffer in inference specifications auto
status = holoscan_infer_context_->execute_inference(inference_specs_-
>data_per_model_, inference_specs_->output_per_model_);
```

- Transmit inferred data:

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

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


Inference parameters from Configuration
Validate parameters
Create contexts
Initialize and setup
start()
stop()
setup()
Receiver 1
Receiver 2
Receiver N
List of Receivers
Get data from GXF messages
Do Inference
Get inferred data
Transmit as GXF message
compute()
Tensor 1
Tensor 2
Tensor N
Inference Operator

Schedulers

The Scheduler component is a critical part of the system responsible for governing the execution of operators in a graph by enforcing conditions associated with each operator. Its primary responsibility includes orchestrating the execution of all operators defined in the graph while keeping track of their execution states.

The Holoscan SDK offers multiple schedulers that can cater to various use cases. These schedulers are:

1. Greedy Scheduler: This basic single-threaded scheduler tests conditions in a greedy manner. It is suitable for simple use cases and provides predictable execution. However, it may not be ideal for large-scale applications as it may incur significant overhead in condition execution.

2. MultiThread Scheduler: The multithread scheduler is designed to handle complex execution patterns in large-scale applications. This scheduler consists of a dispatcher thread that monitors the status of each operator and dispatches it to a thread pool of worker threads responsible for executing them. Once execution is complete, worker threads enqueue the operator back on the dispatch queue. The multithread scheduler offers superior performance and scalability over the greedy scheduler.

3. Event-Based Scheduler: The event-based scheduler is also a multi-thread scheduler, but as the name indicates it is event-based rather than polling based. Instead of having a thread that constantly polls for the execution readiness of each operator, it instead waits for an event to be received which indicates that an operator is ready to execute. The event-based scheduler will have a lower latency than using the multi-thread scheduler with a long polling interval (`check_recession_period_ms`), but without the high CPU usage seen for a multi-thread scheduler with a very short polling interval.

It is essential to select the appropriate scheduler for the use case at hand to ensure optimal performance and efficient resource utilization. Since most parameters of the schedulers overlap, it is easy to switch between them to test which may be most performant for a given application.

Note

Detailed APIs can be found here: C++/
`<a href="../api/python/holoscan_python_api_schedulers.html#module-holoscan.schedulers">Python</a>`
).

Greedy Scheduler

The greedy scheduler has a few parameters that the user can configure.

- The clock used by the scheduler can be set to either a `realtime` or `manual` clock.
    - The realtime clock is what should be used for applications as it pauses execution as needed to respect user specified conditions (e.g. operators with periodic conditions will wait the requested period before executing again).
    - The manual clock is of benefit mainly for testing purposes as it causes operators to run in a time-compressed fashion (e.g. periodic conditions are not respected and operators run in immediate succession).
- The user can specify a `max_duration_ms` that will cause execution of the application to terminate after a specified maximum duration. The default value of `-1` (or any other negative value) will result in no maximum duration being applied.
- This scheduler also has a boolean parameter, `stop_on_deadlock` that controls whether the application will terminate if a deadlock occurs. A deadlock occurs when all operators are in a `WAIT` state, but there is no periodic condition pending to break out of this state. This parameter is `true` by default.
- When setting the `stop_on_deadlock_timeout` parameter, the scheduler will wait this amount of time (in ms) before determining that it is in deadlock and should stop. It will reset if a job comes in during the wait. A negative value means no stop on deadlock. This parameter only applies when `stop_on_deadlock=true`.

Multithread Scheduler

The multithread scheduler has several parameters that the user can configure. These are a superset of the parameters available for the `GreedyScheduler` (described in the section above). Only the parameters unique to the multithread scheduler are described here. The multi-thread scheduler uses a dedicated thread to poll the status of operators and schedule any that are ready to execute. This will lead to high CPU usage by this polling thread when `check_recession_period_ms` is close to 0.

- The number of worker threads used by the scheduler can be set via `worker_thread_number`, which defaults to `1`. This should be set based on a consideration of both the workflow and the available hardware. For example, the topology of the computation graph will determine how many operators it may be possible to run in parallel. Some operators may potentially launch multiple threads internally, so some amount of performance profiling may be required to determine optimal parameters for a given workflow.

- The value of `check_recession_period_ms` controls how long the scheduler will sleep before checking a given condition again. In other words, this is the polling interval for operators that are in a `WAIT` state. The default value for this parameter is `5` ms.

Event-Based Scheduler

The event-based scheduler is also a multi-thread scheduler, but it is event-based rather than polling based. As such, there is no `check_recession_period_ms` parameter, and this scheduler will not have the high CPU usage that can occur when polling at a short interval. Instead, the scheduler only wakes up when an event is received indicating that an operator is ready to execute. The parameters of this scheduler are a superset of the parameters available for the `GreedyScheduler` (described above). Only the parameters unique to the event-based scheduler are described here.

- The number of worker threads used by the scheduler can be set via `worker_thread_number`, which defaults to `1`. This should be set based on a consideration of both the workflow and the available hardware. For example, the topology of the computation graph will determine how many operators it may be possible to run in parallel. Some operators may potentially launch multiple threads internally, so some amount of performance profiling may be required to determine optimal parameters for a given workflow.

Conditions

The following table shows various states of the scheduling status of an operator:

Scheduling Status	Description
NEVER	Operator will never execute again
READY	Operator is ready for execution
WAIT	Operator may execute in the future
WAIT_TIME	Operator will be ready for execution after specified duration
WAIT_EVENT	Operator is waiting on an asynchronous event with unknown time interval

Note

- A failure in execution of any single operator stops the execution of all the operators.
- Operators are naturally unscheduled from execution when their scheduling status reaches `NEVER` state.

By default, operators are always `READY`, meaning they are scheduled to continuously execute their `compute()` method. To change that behavior, some condition classes can be passed to the constructor of an operator. There are various conditions currently supported in the Holoscan SDK:

- MessageAvailableCondition
- DownstreamMessageAffordableCondition

- CountCondition
- BooleanCondition
- PperiodicCondition
- AsynchronousCondition

> **Note**
>
> Detailed APIs can be found here: C++/
> `<a href="../api/python/holoscan_python_api_conditions.html#module-holoscan.conditions">Python</a>`
>).

Conditions are AND-combined

An Operator can be associated with multiple conditions which define it's execution behavior. Conditions are AND combined to describe the current state of an operator. For an operator to be executed by the scheduler, all the conditions must be in `READY` state and conversely, the operator is unscheduled from execution whenever any one of the scheduling term reaches `NEVER` state. The priority of various states during AND combine follows the order `NEVER`, `WAIT_EVENT`, `WAIT`, `WAIT_TIME`, and `READY`.

MessageAvailableCondition

An operator associated with `MessageAvailableCondition` is executed when the associated queue of the input port has at least a certain number of elements. This condition is associated with a specific input port of an operator through the `condition()` method on the return value (IOSpec) of the OperatorSpec's `input()` method.

The minimum number of messages that permits the execution of the operator is specified by `min_size` parameter (default: `1`). An optional parameter for this condition is `front_stage_max_size`, the maximum front stage message count. If this parameter is

set, the condition will only allow execution if the number of messages in the queue does not exceed this count. It can be used for operators which do not consume all messages from the queue.

DownstreamMessageAffordableCondition

This condition specifies that an operator shall be executed if the input port of the downstream operator for a given output port can accept new messages. This condition is associated with a specific output port of an operator through the `condition()` method on the return value (IOSpec) of the OperatorSpec's `output()` method. The minimum number of messages that permits the execution of the operator is specified by `min_size` parameter (default: `1`).

CountCondition

An operator associated with `CountCondition` is executed for a specific number of times specified using its `count` parameter. The scheduling status of the operator associated with this condition can either be in `READY` or `NEVER` state. The scheduling status reaches the `NEVER` state when the operator has been executed `count` number of times.

BooleanCondition

An operator associated with `BooleanCondition` is executed when the associated boolean variable is set to `true`. The boolean variable is set to `true` / `false` by calling the `enable_tick()` / `disable_tick()` methods on the `BooleanCondition` object. The `check_tick_enabled()` method can be used to check if the boolean variable is set to `true` / `false`. The scheduling status of the operator associated with this condition can either be in `READY` or `NEVER` state. If the boolean variable is set to `true`, the scheduling status of the operator associated with this condition is set to `READY`. If the boolean variable is set to `false`, the scheduling status of the operator associated with this condition is set to `NEVER`. The `enable_tick()` / `disable_tick()` methods can be called from any operator in the workflow.

Ingested Tab Module

PeriodicCondition

An operator associated with `PeriodicCondition` is executed after periodic time intervals specified using its `recess_period` parameter. The scheduling status of the operator associated with this condition can either be in `READY` or `WAIT_TIME` state. For the first time or after periodic time intervals, the scheduling status of the operator associated with this condition is set to `READY` and the operator is executed. After the operator is executed, the scheduling status is set to `WAIT_TIME` and the operator is not executed until the `recess_period` time interval.

AsynchronousCondition

AsynchronousCondition is primarily associated with operators which are working with asynchronous events happening outside of their regular execution performed by the scheduler. Since these events are non-periodic in nature, AsynchronousCondition prevents the scheduler from polling the operator for its status regularly and reduces CPU utilization. The scheduling status of the operator associated with this condition can either be in `READY`, `WAIT`, `WAIT_EVENT` or `NEVER` states based on the asynchronous event it's waiting on.

The state of an asynchronous event is described using `AsynchronousEventState` and is updated using the `event_state()` API.

AsynchronousEventState	Description
READY	Init state, first execution of `compute()` method is pending
WAIT	Request to async service yet to be sent, nothing to do but wait
EVENT_WAITING	Request sent to an async service, pending event done notification
EVENT_DONE	Event done notification received, operator ready to be ticked
EVENT_NEVER	Operator does not want to be executed again, end of execution

Operators associated with this scheduling term most likely have an asynchronous thread which can update the state of the condition outside of it's regular execution cycle performed by the scheduler. When the asynchronous event state is in `WAIT` state, the

scheduler regularly polls for the scheduling state of the operator. When the asynchronous event state is in `EVENT_WAITING` state, schedulers will not check the scheduling status of the operator again until they receive an event notification. Setting the state of the asynchronous event to `EVENT_DONE` automatically sends the event notification to the scheduler. Operators can use the `EVENT_NEVER` state to indicate the end of its execution cycle.

Resources

Resource classes represent resources such as a allocators, clocks, transmitters or receivers that may be used as a parameter for operators or schedulers. The resource classes that are likely to be directly used by application authors are documented here.

> **Note**
>
> There are a number of other resources classes used internally which are not documented here, but appear in the API Documentation (C++/`<a href="../api/python/holoscan_python_api_resources.html#module-holoscan.resources">Python</a>`).

Allocator

UnboundedAllocator

An allocator that uses dynamic host or device memory allocation without an upper bound. This allocator does not take any user-specified parameters.

BlockMemoryPool

This is a memory pool which provides a user-specified number of equally sized blocks of memory.

- The `storage_type` parameter can be set to determine the memory storage type used by the operator. This can be 0 for page-locked host memory (allocated with `cudaMallocHost`), 1 for device memory (allocated with `cudaMalloc`) or 2 for system memory (allocated with C++ `new`).

- The `block_size` parameter determines the size of a single block in the memory pool in bytes. Any allocation requests made of this allocator must fit into this block size.
- The `num_blocks` parameter controls the total number of blocks that are allocated in the memory pool.
- The `dev_id` parameter is an optional parameter that can be used to specify the CUDA ID of the device on which the memory pool will be created.

CudaStreamPool

This allocator creates a pool of CUDA streams.

- The `stream_flags` parameter specifies the flags sent to cudaStreamCreateWithPriority when creating the streams in the pool.
- The `stream_priority` parameter specifies the priority sent to cudaStreamCreateWithPriority when creating the streams in the pool. Lower values have a higher priority.
- The `reserved_size` parameter specifies the initial number of CUDA streams created in the pool upon initialization.
- The `max_size` parameter is an optional parameter that can be used to specify a maximum number of CUDA streams that can be present in the pool. The default value of 0 means that the size of the pool is unlimited.
- The `dev_id` parameter is an optional parameter that can be used to specify the CUDA ID of the device on which the stream pool will be created.

Clock

Clock classes can be provided via a `clock` parameter to the `Scheduler` classes to manage the flow of time.

All clock classes provide a common set of methods that can be used at runtime in user applications.

- The `time()` method returns the current time in seconds (floating point).

- The `timestamp()` method returns the current time as an integer number of nanoseconds.
- The `sleep_for()` method sleeps for a specified duration in ns. An overloaded version of this method allows specifying the duration using a `std::chrono::duration<Rep, Period>` from the C++ API or a datetime.timedelta from the Python API.
- The `sleep_until()` method sleeps until a specified target time in ns.

Realtime Clock

The `RealtimeClock` respects the true duration of conditions such as `PeriodicCondition`. It is the default clock type and the one that would likely be used in user applications.

In addition to the general clock methods documented above:

- this class has a `set_time_scale()` method which can be used to dynamically change the time scale used by the clock.
- the parameter `initial_time_offset` can be used to set an initial offset in the time at initialization.
- the parameter `initial_time_scale` can be used to modify the scale of time. For instance, a scale of 2.0 would cause time to run twice as fast.
- the parameter `use_time_since_epoch` makes times relative to the POSIX epoch (`initial_time_offset` becomes an offset from epoch).

Manual Clock

The `ManualClock` compresses time intervals (e.g. `PeriodicCondition` proceeds immediately rather than waiting for the specified period). It is provided mainly for use during testing/development.

The parameter `initial_timestamp` controls the initial timestamp on the clock in ns.

Transmitter (advanced)

Typically users don't need to explicitly assign transmitter or receiver classes to the IOSpec ports of Holoscan SDK operators. For connections between operators a `DoubleBufferTransmitter` will automatically be used, while for connections between fragments in a distributed application, a `UcxTransmitter` will be used. When data frame flow tracking is enabled any `DoubleBufferTransmitter` will be replaced by an `AnnotatedDoubleBufferTransmitter` which also records the timestamps needed for that feature.

DoubleBufferTransmitter

This is the transmitter class used by output ports of operators within a fragment.

UcxTransmitter

This is the transmitter class used by output ports of operators that connect fragments in a distributed applications. It takes care of sending UCX active messages and serializing their contents.

Receiver (advanced)

Typically users don't need to explicitly assign transmitter or receiver classes to the IOSpec ports of Holoscan SDK operators. For connections between operators a `DoubleBufferReceiver` will be used, while for connections between fragments in a distributed application, the `UcxReceiver` will be used. When data frame flow tracking is enabled any `DoubleBufferReceiver` will be replaced by an `AnnotatedDoubleBufferReceiver` which also records the timestamps needed for that feature.

DoubleBufferReceiver

This is the receiver class used by input ports of operators within a fragment.

UcxReceiver

This is the receiver class used by input ports of operators that connect fragments in a distributed applications. It takes care of receiving UCX active messages and deserializing their contents.

Holoscan C++ API

Namespaces

Macros

Operator Definition

Resource Definition

Condition Definition

Scheduler Definition

- Define HOLOSCAN_SCHEDULER_FORWARD_ARGS
- Define HOLOSCAN_SCHEDULER_FORWARD_ARGS_SUPER

Logging

- Define HOLOSCAN_LOG_TRACE
- Define HOLOSCAN_LOG_DEBUG
- Define HOLOSCAN_LOG_INFO
- Define HOLOSCAN_LOG_WARN
- Define HOLOSCAN_LOG_ERROR
- Define HOLOSCAN_LOG_CRITICAL

Classes

Core

- Class Application
- Class Arg
- Class ArgList
- Class ArgType
- Class ArgumentSetter
- Struct CLIOptions
- Class Component
- Class ComponentSpec
- Class Condition
- Class Config

- Class DataFlowTracker
- Class ExecutionContext
- Class ExtensionManager
- Class Executor
- Template Class FlowGraph
- Class Fragment
- Template Class Graph
- Class InputContext
- Class IOSpec
- Class MessageLabel
- Template Class MetaParameter
- Class Operator
- Class OperatorSpec
- Struct OperatorTimestampLabel
- Class OutputContext
- Class ParameterWrapper
- Class Resource
- Class Scheduler

Operators

- Class AJASourceOp
- Class AsyncPingRxOp
- Class AsyncPingTxOp

- Class BayerDemosaicOp
- Class FormatConverterOp
- Class HolovizOp
- Class InferenceOp
- Class InferenceProcessorOp
- Class PingRxOp
- Class PingTxOp
- Class SegmentationPostprocessorOp
- Class V4L2VideoCaptureOp
- Class VideoStreamRecorderOp
- Class VideoStreamReplayerOp
- Struct BufferInfo
- Struct HolovizOp::InputSpec
- Struct InputSpec::View
- Struct InferenceOp::DataMap
- Struct InferenceOp::DataVecMap
- Struct InferenceProcessorOp::DataMap
- Struct InferenceProcessorOp::DataVecMap
- Struct V4L2VideoCaptureOp::Buffer
- Template Struct convert< NTV2Channel >
- Template Struct codec< ops::HolovizOp::InputSpec >
- Template Struct codec< ops::HolovizOp::InputSpec::View >

- Template Struct codec< std::vector< ops::HolovizOp::InputSpec > >
- Template Struct codec< std::vector< ops::HolovizOp::InputSpec::View > >

GXF Components

Conditions

- Class AsynchronousCondition
- Class BooleanCondition
- Class CountCondition
- Class DownstreamMessageAffordableCondition
- Class MessageAvailableCondition
- Class PeriodicCondition

Resources

- Class Allocator
- Class BlockMemoryPool
- Class Clock
- Class CudaStreamPool
- Class DoubleBufferReceiver
- Class DoubleBufferTransmitter
- Class ManualClock
- Class RealtimeClock
- Class Receiver
- Class SerializationBuffer

- Class StdComponentSerializer
- Class StdEntitySerializer
- Class Transmitter
- Class UcxComponentSerializer
- Class UcxEntitySerializer
- Class UcxReceiver
- Class UcxSerializationBuffer
- Class UcxTransmitter
- Class UnboundedAllocator

Schedulers

- Class EventBasedScheduler
- Class GreedyScheduler
- Class MultiThreadScheduler

Network Contexts

- Class UcxContext

Native Operator Support

- Class Message

Domain Objects

Tensor (interoperability with GXF Tensor and DLPack interface)

Class/Struct

- Class Tensor

- Class TensorMap
- exhale_struct_structholoscan_1_1DLManagedTensorCtx
- exhale_class_classholoscan_1_1DLManagedMemoryBuffer

Functions

- Function holoscan::calc_strides
- Function holoscan::dldatatype_from_typestr
- Function holoscan::dldevice_from_pointer
- Function holoscan::numpy_dtype

Utilities

Measurement

- Class Timer

Enums

- Enum MemoryStorageType
- Enum ArgElementType
- Enum ArgContainerType
- Enum ConditionType
- Enum DataFlowMetric
- Enum ErrorCode
- Enum LogLevel
- Enum ParameterFlag
- Enum SchedulerType

Operator-Specific Enums

- exhale_enum_include_2holoscan_2operators_2format__converter_2format__converte
- exhale_enum_include_2holoscan_2operators_2format__converter_2format__converte

Inference Module Enums

- Enum holoinfer_backend
- Enum holoinfer_data_processor
- Enum holoinfer_datatype
- Enum holoinfer_code

Visualization Module Enums

- Enum DepthMapRenderMode
- Enum ImageFormat
- Enum ComponentSwizzle
- Enum InitFlags
- Enum PrimitiveTopology

Functions

- Template Function holoscan::make_application
- Template Function holoscan::log_critical
- Template Function holoscan::log_error
- Template Function holoscan::log_warn
- Template Function holoscan::log_info
- Template Function holoscan::log_debug

- Template Function holoscan::log_trace
- Template Function holoscan::log_message
- Function holoscan::log_level
- Function holoscan::set_log_level
- Function holoscan::set_log_pattern

Typedefs

- Typedef holoscan::unexpect_t
- Typedef holoscan::bad_expected_access
- Typedef holoscan::unexpected
- Typedef holoscan::expected
- Typedef holoscan::OperatorFlowGraph
- Typedef holoscan::FragmentFlowGraph
- Typedef holoscan::Parameter
- Typedef holoscan::OperatorGraph
- Typedef holoscan::OperatorEdgeDataElementType
- Typedef holoscan::FragmentNodeType
- Typedef holoscan::FragmentEdgeDataElementType
- Typedef holoscan::FragmentGraph
- Typedef holoscan::OperatorNodeType
- Typedef holoscan::remove_pointer_t
- Typedef holoscan::is_yaml_convertable_t

- Typedef holoscan::is_shared_ptr_t
- Typedef holoscan::base_type_t
- Typedef holoscan::is_scalar_t
- Typedef holoscan::is_array_t
- Typedef holoscan::is_vector_t
- Typedef holoscan::type_info

Inference Module Typedefs

- Typedef holoscan::inference::processor_FP
- Typedef holoscan::inference::transforms_FP
- Typedef holoscan::inference::MultiMappings
- Typedef holoscan::inference::Mappings
- Typedef holoscan::inference::DataMap
- Typedef holoscan::inference::DimType
- Typedef holoscan::inference::byte
- Typedef holoscan::inference::TimePoint
- Typedef holoscan::inference::node_type

Visualization Module Typedefs

- Typedef GLFWwindow
- Typedef holoscan::viz::InstanceHandle

Variables

- Variable holoscan::kDefaultCpuMetrics

- Variable holoscan::metricToString
- Variable holoscan::kDefaultLogfileName
- Variable holoscan::kDefaultNumBufferedMessages
- Variable holoscan::kDefaultLatencyThreshold
- Variable holoscan::kDefaultNumLastMessagesToDiscard
- Variable holoscan::kDefaultNumStartMessagesToSkip
- Variable holoscan::unexpect
- Variable holoscan::kDefaultGpuMetrics
- Variable holoscan::kDefaultSerializationBufferSize
- Variable holoscan::dimension_of_v
- Variable holoscan::is_array_v
- Variable holoscan::is_one_of_derived_v
- Variable holoscan::is_scalar_v
- Variable holoscan::is_yaml_convertable_v
- Variable holoscan::is_shared_ptr_v
- Variable holoscan::is_one_of_v
- Variable holoscan::is_vector_v
- exhale_variable_ucx__receiver_8hpp_1ad86e6465bc051d691c6e6f92ae0ccafe
- Variable holoscan::kDefaultUcxSerializationBufferSize

Inference Module Variables

- exhale_variable_infer__manager_8hpp_1a4de612a9b562e15c3b767cccdee50b1e
- Variable holoscan::inference::kHoloInferDataTypeMap

- Variable holoscan::inference::process_manager
- Variable holoscan::inference::StreamDeleter
- Holoscan API
  - Holoviz

Holoscan Python API

Holoscan Python Submodules

holoscan.conditions

This module provides a Python API to underlying C++ API Conditions.

holoscan.conditions.BooleanCondition	Boolean condition class.
holoscan.conditions.CountCondition	Count condition class.

holoscan.conditions.DownstreamMessageAffordableCondition	Condition that permits execution when the downstream operator can accept new messages.
holoscan.conditions.MessageAvailableCondition	Condition that permits execution when an upstream message is available.
holoscan.conditions.PeriodicCondition	Condition class to support periodic execution of operators.

class holoscan.conditions.BooleanCondition

Bases: holoscan.gxf._gxf.GXFCondition

Boolean condition class.

Used to control whether an entity is executed.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the condition.
fragment	Fragment that the condition belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the condition.
id	The identifier of the component.
name	The name of the condition.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.

`check_tick_enabled`(self)	Check whether the condition is `True`.
`disable_tick`(self)	Set condition to `False`.
`enable_tick`(self)	Set condition to `True`.
`gxf_initialize`(self)	Initialize the component.
`initialize`(self)	Initialize the component.
`setup`(self, spec)	Define the component specification.

__init__(*self: holoscan.conditions._conditions.BooleanCondition*, *fragment: holoscan.core._core.Fragment*, *enable_tick: bool = True*, *name: str = 'noname_boolean_condition'*) None

Boolean condition.

Parameters

fragment

The fragment the condition will be associated with

enable_tick

Boolean value for the condition.

name

The name of the condition.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

check_tick_enabled(*self: holoscan.conditions._conditions.BooleanCondition*) bool

Check whether the condition is `True`.

property description

YAML formatted string describing the condition.

disable_tick(*self: holoscan.conditions._conditions.BooleanCondition*) None

Set condition to `False`.

enable_tick(*self: holoscan.conditions._conditions.BooleanCondition*) None

Set condition to `True`.

property fragment

Fragment that the condition belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self:* *holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the condition.

Returns

str

The GXF type name of the condition

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.gxf._gxf.GXFCondition*) None

Initialize the component.

property name

The name of the condition.

Returns

name

setup(*self:* *holoscan.conditions._conditions.BooleanCondition*, *spec:* *holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the condition.

property spec

class holoscan.conditions.CountCondition

Bases: holoscan.gxf._gxf.GXFCondition

Count condition class.

Attributes

args	The list of arguments associated with the component.
count	The execution count associated with the condition
description	YAML formatted string describing the condition.
fragment	Fragment that the condition belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the condition.
id	The identifier of the component.
name	The name of the condition.

spec	

Methods

add_arg (*args,	Overloaded function.

**kwargs)	
`gxf_initialize` (self)	Initialize the component.
`initialize` (self)	Initialize the component.
`setup` (self, arg0)	Define the component specification.

__init__(*self: holoscan.conditions._conditions.CountCondition*, *fragment: holoscan.core._core.Fragment*, *count: int = 1*, *name: str = 'noname_count_condition'*) None

Count condition.

Parameters

fragment

The fragment the condition will be associated with

count

The execution count value used by the condition.

name

The name of the condition.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property count

The execution count associated with the condition

property description

YAML formatted string describing the condition.

property fragment

Fragment that the condition belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the condition.

Returns

str

The GXF type name of the condition

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFCondition*) None

Initialize the component.

property name

The name of the condition.

Returns

name

setup(*self: holoscan.conditions._conditions.CountCondition*, *arg0: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the condition.

property spec

class holoscan.conditions.DownstreamMessageAffordableCondition

Bases: holoscan.gxf._gxf.GXFCondition

Condition that permits execution when the downstream operator can accept new messages.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the condition.
fragment	Fragment that the condition belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.

`gxf_eid`	The GXF entity ID.
`gxf_typename`	The GXF type name of the condition.
`id`	The identifier of the component.
`min_size`	The minimum number of free slots required for the downstream entity's back buffer.
`name`	The name of the condition.
`transmitter`	The transmitter associated with the condition.

spec	

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`gxf_initialize` (self)	Initialize the component.
`initialize` (self)	Initialize the condition
`setup` (self, spec)	Define the component specification.

__init__(*self: holoscan.conditions._conditions.DownstreamMessageAffordableCondition*, *fragment: holoscan.core._core.Fragment*, *min_size: int = 1*, *name: str = 'noname_downstream_affordable_condition'*) None

Condition that permits execution when the downstream operator can accept new messages.

Parameters

fragment

The fragment the condition will be associated with

min_size

The minimum number of free slots present in the back buffer.

name

The name of the condition.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the condition.

property fragment

Fragment that the condition belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the condition.

Returns

str

The GXF type name of the condition

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.conditions._conditions.DownstreamMessageAffordableCondition*)
None

Initialize the condition

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

property min_size

The minimum number of free slots required for the downstream entity's back buffer.

property name

The name of the condition.

Returns

name

setup(*self:* *holoscan.conditions._conditions.DownstreamMessageAffordableCondition*, *spec:* *holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the condition.

property spec

property transmitter

The transmitter associated with the condition.

class holoscan.conditions.MessageAvailableCondition

Bases: holoscan.gxf._gxf.GXFCondition

Condition that permits execution when an upstream message is available.

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 operators which do not consume all messages from the queue.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the condition.
fragment	Fragment that the condition belongs to.
front_stage_max_size	Threshold for the number of front stage messages.
gxf_cid	The GXF component ID.

gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the condition.
id	The identifier of the component.
min_size	The total number of messages over a set of input channels needed to permit execution.
name	The name of the condition.
receiver	The receiver associated with the condition.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
gxf_initialize (self)	Initialize the component.

initialize (self)	Initialize the condition
setup (self, arg0)	Define the component specification.

__init__(*self: holoscan.conditions._conditions.MessageAvailableCondition*, *fragment: holoscan.core._core.Fragment*, *min_size: int = 1*, *front_stage_max_size: int = 1*, *name: str = 'noname_message_available_condition'*) None

Condition that permits execution when an upstream message is available.

Parameters

fragment

The fragment the condition will be associated with

min_size

The total number of messages over a set of input channels needed to permit execution.

front_stage_max_size

Threshold for the number of front stage messages. Execution is only allowed if the number of front stage messages does not exceed this count.

name

The name of the condition.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the condition.

property fragment

Fragment that the condition belongs to.

Returns

name

property front_stage_max_size

Threshold for the number of front stage messages. Execution is only allowed if the number of front stage messages does not exceed this count.

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the condition.

Returns

str

The GXF type name of the condition

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.conditions._conditions.MessageAvailableCondition*) None

Initialize the condition

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

property min_size

The total number of messages over a set of input channels needed to permit execution.

property name

The name of the condition.

Returns

name

property receiver

The receiver associated with the condition.

setup(*self: holoscan.conditions._conditions.MessageAvailableCondition*, *arg0: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the condition.

property spec

class holoscan.conditions.PeriodicCondition

Bases: holoscan.gxf._gxf.GXFCondition

Condition class to support periodic execution of operators. The recess (pause) period indicates the minimum amount of time that must elapse before the *compute()* method can be executed again. The recess period can be specified as an integer value in nanoseconds.

For example: 1000 for 1 microsecond 1000000 for 1 millisecond, and 10000000000 for 1 second.

The recess (pause) period can also be specified as a *datetime.timedelta* object representing a duration. (see https://docs.python.org/3/library/datetime.html#timedelta-objects)

For example: datetime.timedelta(minutes=1), datetime.timedelta(seconds=1), datetime.timedelta(milliseconds=1) and datetime.timedelta(microseconds=1). Supported argument names are: weeks| days | hours | minutes | seconds | millisecons | microseconds This requires *import datetime*.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the condition.
fragment	Fragment that the condition belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the condition.
id	The identifier of the component.
name	The name of the condition.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the component.
last_run_timestamp(self)	Gets the integer representing the last run time stamp.
recess_period(*args, **kwargs)	Overloaded function.
recess_period_ns(self)	Gets the recess (pause) period value in nanoseconds.
setup(self, arg0)	Define the component specification.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: holoscan.conditions._conditions.PeriodicCondition, fragment: holoscan.core._core.Fragment, recess_period: int, name: str = 'noname_periodic_condition') -> None

2. __init__(self: holoscan.conditions._conditions.PeriodicCondition, fragment: holoscan.core._core.Fragment, recess_period: datetime.timedelta, name: str = 'noname_periodic_condition') -> None

Condition class to support periodic execution of operators.

Parameters

fragment

The fragment the condition will be associated with

recess_period

The recess (pause) period value used by the condition. If an integer is provided, the units are in nanoseconds.

name

The name of the condition.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the condition.

property fragment

Fragment that the condition belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the condition.

Returns

str

The GXF type name of the condition

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFCondition*) None

Initialize the component.

last_run_timestamp(*self: holoscan.conditions._conditions.PeriodicCondition*) int

Gets the integer representing the last run time stamp.

property name

The name of the condition.

Returns

name

recess_period(**args*, ***kwargs*)

Overloaded function.

1. recess_period(self: holoscan.conditions._conditions.PeriodicCondition, arg0: int) -> None

Sets the recess (pause) period associated with the condition. The recess period can be specified as an integer value in nanoseconds or a *datetime.timedelta* object representing a duration.

2. recess_period(self: holoscan.conditions._conditions.PeriodicCondition, arg0: datetime.timedelta) -> None

Sets the recess (pause) period associated with the condition. The recess period can be specified as an integer value in nanoseconds or a *datetime.timedelta* object representing a duration.

recess_period_ns(*self:* *holoscan.conditions._conditions.PeriodicCondition*) int

Gets the recess (pause) period value in nanoseconds.

setup(*self:* *holoscan.conditions._conditions.PeriodicCondition*, *arg0:* *holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the condition.

property spec

holoscan.core

This module provides a Python API for the core C++ API classes.

The *Application* class is the primary class that should be derived from to create a custom application.

holoscan.core.Application([argv])	Application class.

holoscan.core.Arg	Class representing a typed argument.
holoscan.core.ArgContainerType	Enum class for an *Arg*'s container type.
holoscan.core.ArgElementType	Enum class for an *Arg*'s element type.
holoscan.core.ArgList	Class representing a list of arguments.
holoscan.core.ArgType	Class containing argument type info.
holoscan.core.CLIOptions	Attributes
holoscan.core.Component	Base component class.

holoscan.core.ComponentSpec	Component specification class.
holoscan.core.ConditionType	Enum class for Condition types.
holoscan.core.Condition	Class representing a condition.
holoscan.core.Config	Configuration class.
holoscan.core.DataFlowMetric	Enum class for DataFlowMetric type.
holoscan.core.DataFlowTracker	Data Flow Tracker class.

holoscan.core.DLDevice	DLDevice class.
holoscan.core.DLDeviceType	Members:
holoscan.core.ExecutionContext	Class representing an execution context.
holoscan.core.Executor	Executor class.
holoscan.core.Fragment([app, name])	Fragment class.
holoscan.core.Graph	alias of holoscan.graphs._graphs.OperatorGraph

holoscan.core.InputContext	Class representing an input context.
holoscan.core.IOSpec	I/O specification class.
holoscan.core.Message	Class representing a message.
holoscan.core.NetworkContext	Class representing a network context.
holoscan.core.Operator(fragment, *args, **kwargs)	Operator class.
holoscan.core.OperatorSpec	alias of holoscan.core._core.PyOperatorSpec

holoscan.core.OutputContext	Class representing an output context.
holoscan.core.ParameterFlag	Enum class for parameter flags.
holoscan.core.Resource	Class representing a resource.
holoscan.core.Tensor	alias of holoscan.core._core.PyTensor
holoscan.core.Tracker(app, *[, filename, ...])	Context manager to add data flow tracking to an application.
holoscan.core.arg_to_py_object(arg)	Utility that converts an *Arg* to a corresponding Python object.

holoscan.core.arglist_to_kwargs(arglist)	Utility that converts an *ArgList* to a Python kwargs dictionary.
holoscan.core.kwargs_to_arglist(**kwargs)	Utility that converts a set of python keyword arguments to an *ArgList*.
holoscan.core.py_object_to_arg(obj[, name])	Utility that converts a single python argument to a corresponding *Arg* type.

class holoscan.core.Application(*argv=None*, **args*, ***kwargs*)

Bases: holoscan.core._core.Application

Application class.

This constructor parses the command line for flags that are recognized by App Driver/Worker, and removes all recognized flags so users can use the remaining flags for their own purposes.

If the arguments are not specified, the arguments are retrieved from sys.executable and sys.argv.

The arguments after processing arguments (parsing Holoscan-specific flags and removing them) are accessible through the argv attribute.

Parameters

argv

The command line arguments to parse. The first item should be the path to the python executable. If not specified, `[sys.executable, *sys.argv]` is used.

Examples

```
>>> from holoscan.core import Application >>> import sys >>>
Application().argv == sys.argv True >>> Application([]).argv == sys.argv True >>>
Application([sys.executable, *sys.argv]).argv == sys.argv True >>>
Application(["python3", "myapp.py", "--driver", "--address=10.0.0.1",
"my_arg1"]).argv ['myapp.py', 'my_arg1']
```

Attributes

`application`	The application associated with the fragment.
`argv`	The command line arguments after processing flags.
`description`	The application's description.
`executor`	Get the executor associated with the fragment.
`fragment_graph`	Get the computation graph (Graph node is a Fragment) associated with the application.
`graph`	Get the computation graph (Graph node is an Operator) associated with the fragment.
`name`	The fragment's name.

options	The reference to the CLI options.
version	The application's version.

Methods

add_flow (*args, **kwargs)	Overloaded function.
add_fragment (self, frag)	Add a fragment to the application.
add_operator (self, op)	Add an operator to the application.
compose (self)	The compose method of the application.
config (*args, **kwargs)	Overloaded function.
config_keys (self)	The set of keys present in the fragment's configuration file.

`from_config` (self, key)	Retrieve parameters from the associated configuration.
`kwargs` (self, key)	Retrieve a dictionary parameters from the associated configuration.
`network_context` (*args, **kwargs)	Overloaded function.
`run` (self)	The run method of the application.
`run_async` ()	Run the application asynchronously.
`scheduler` (*args, **kwargs)	Overloaded function.
`track` (self, num_start_messages_to_skip, ...)	The track method of the application.

__init__(*self: holoscan.core._core.Application*, *argv: List[str] = []*) None

Application class.

This constructor parses the command line for flags that are recognized by App Driver/Worker, and removes all recognized flags so users can use the remaining flags for their own purposes.

If the arguments are not specified, the arguments are retrieved from `sys.executable` and `sys.argv`.

The arguments after processing arguments (parsing Holoscan-specific flags and removing them) are accessible through the `argv` attribute.

Parameters

argv

The command line arguments to parse. The first item should be the path to the python executable. If not specified, `[sys.executable, *sys.argv]` is used.

Examples

```
>>> from holoscan.core import Application >>> import sys >>>
Application().argv == sys.argv True >>> Application([]).argv == sys.argv
True >>> Application([sys.executable, *sys.argv]).argv == sys.argv True
>>> Application(["python3", "myapp.py", "--driver", "--address=10.0.0.1",
"my_arg1"]).argv ['myapp.py', 'my_arg1']
```

add_flow(**args*, ***kwargs*)

Overloaded function.

1. add_flow(self: holoscan.core._core.Application, upstream_op: holoscan.core._core.Operator, downstream_op: holoscan.core._core.Operator) -> None

2. add_flow(self: holoscan.core._core.Application, upstream_op: holoscan.core._core.Operator, downstream_op: holoscan.core._core.Operator, port_pairs: Set[Tuple[str, str]]) -> None

Connect two operators associated with the fragment.

Parameters

upstream_op

Source operator.

downstream_op

Destination operator.

port_pairs

Sequence of ports to connect. The first element of each 2-tuple is a port from *upstream_op* while the second element is the port of *downstream_op* to which it connects.

Notes

This is an overloaded function. Additional variants exist:

1.) For the Application class there is a variant where the first two arguments are of type *holoscan.core.Fragment* instead of *holoscan.core.Operator*. This variant is used in building multi-fragment applications. 2.) There are also variants that omit the *port_pairs* argument that are applicable when there is only a single output on the upstream operator/fragment and a single input on the downstream operator/fragment.

3. add_flow(self: holoscan.core._core.Application, upstream_frag: holoscan.core._core.Fragment, downstream_frag: holoscan.core._core.Fragment, port_pairs: Set[Tuple[str, str]]) -> None

add_fragment(*self: holoscan.core._core.Application*, *frag: holoscan.core._core.Fragment*) None

Add a fragment to the application.

Parameters

frag

The fragment to add.

add_operator(*self: holoscan.core._core.Application*, *op: holoscan.core._core.Operator*) None

Add an operator to the application.

Parameters

op

The operator to add.

property application

The application associated with the fragment.

Returns

app

property argv

The command line arguments after processing flags. This does not include the python executable like *sys.argv* does.

Returns

argv

compose(*self: holoscan.core._core.Application*) None

The compose method of the application.

This method should be called after *config*, but before *run* in order to compose the computation graph.

config(**args*, ***kwargs*)

Overloaded function.

1. config(self: holoscan.core._core.Fragment, config_file: str, prefix: str = '') -> None

Configuration class.

Represents configuration parameters as read from a YAML file.

Parameters

config

The path to the configuration file (in YAML format) or a *holoscan.core.Config* object.

prefix

Prefix path for the` config` file. Only available in the overloaded variant that takes a string for *config*.

2. config(self: holoscan.core._core.Fragment, arg0: holoscan.core._core.Config) -> None
3. config(self: holoscan.core._core.Fragment) -> holoscan.core._core.Config

config_keys(*self: holoscan.core._core.Fragment*) Set[str]

The set of keys present in the fragment's configuration file.

property description

The application's description.

Returns

description

property executor

Get the executor associated with the fragment.

property fragment_graph

Get the computation graph (Graph node is a Fragment) associated with the application.

from_config(*self: holoscan.core._core.Fragment*, *key: str*) object

Retrieve parameters from the associated configuration.

Parameters

key

The key within the configuration file to retrieve. This can also be a specific component of the parameter via syntax *'key.sub_key'*.

Returns

args

An argument list associated with the key.

property graph

Get the computation graph (Graph node is an Operator) associated with the fragment.

kwargs(*self: holoscan.core._core.Fragment*, *key: str*) dict

Retrieve a dictionary parameters from the associated configuration.

Parameters

key

The key within the configuration file to retrieve. This can also be a specific component of the parameter via syntax *'key.sub_key'*.

Returns

kwargs

A Python dict containing the parameters in the configuration file under the specified key.

property name

The fragment's name.

Returns

name

network_context(**args*, ***kwargs*)

Overloaded function.

1. network_context(self: holoscan.core._core.Fragment, network_context: holoscan.core._core.NetworkContext) -> None

Assign a network context to the Fragment

Parameters

network_context

A network_context class instance to be used by the underlying GXF executor. If unspecified, no network context will be used.

2. network_context(self: holoscan.core._core.Fragment) -> holoscan.core._core.NetworkContext
Get the network context to be used by the Fragment

property options

The reference to the CLI options.

Returns

options

run(*self: holoscan.core._core.Application*) None

The run method of the application.

This method runs the computation. It must have first been initialized via *config* and *compose*.

run_async()

Run the application asynchronously.

This method is a convenience method that creates a thread pool with one thread and runs the application in that thread. The thread pool is created using *concurrent.futures.ThreadPoolExecutor*.

Returns

future : concurrent.futures.Future

scheduler(**args*, ***kwargs*)

Overloaded function.

1. scheduler(self: holoscan.core._core.Fragment, scheduler: holoscan.core._core.Scheduler) -> None

Assign a scheduler to the Fragment.

Parameters

scheduler

A scheduler class instance to be used by the underlying GXF executor. If unspecified, the default is a *holoscan.gxf.GreedyScheduler*.

2. scheduler(self: holoscan.core._core.Fragment) -> holoscan.core._core.Scheduler
Get the scheduler to be used by the Fragment.

track(*self: holoscan.core._core.Fragment*, *num_start_messages_to_skip: int = 10*, *num_last_messages_to_discard: int = 10*, *latency_threshold: int = 0*)
holoscan::DataFlowTracker

The track method of the application.

This method enables data frame flow tracking and returns a DataFlowTracker object which can be used to display metrics data for profiling an application.

Parameters

num_start_messages_to_skip

The number of messages to skip at the beginning.

num_last_messages_to_discard

The number of messages to discard at the end.

latency_threshold

The minimum end-to-end latency in milliseconds to account for in the end-to-end latency metric calculations

property version

The application's version.

Returns

version

class holoscan.core.Arg

Bases: pybind11_builtins.pybind11_object

Class representing a typed argument.

Attributes

arg_type	ArgType info corresponding to the argument.
description	YAML formatted string describing the argument.
has_value	Boolean flag indicating whether a value has been assigned to the argument.
name	The name of the argument.

__init__(*self: holoscan.core._core.Arg*, *name: str*) None

Class representing a typed argument.

Parameters

name

The argument's name.

property arg_type

ArgType info corresponding to the argument.

Returns

arg_type

property description

YAML formatted string describing the argument.

property has_value

Boolean flag indicating whether a value has been assigned to the argument.

property name

The name of the argument.

Returns

name

class holoscan.core.ArgContainerType

Bases: pybind11_builtins.pybind11_object

Enum class for an *Arg*'s container type.

Members:

NATIVE

VECTOR

ARRAY

Attributes

name	

value	

ARRAY = *<ArgContainerType.ARRAY: 2>*

NATIVE = *<ArgContainerType.NATIVE: 0>*

VECTOR = *<ArgContainerType.VECTOR: 1>*

__init__(*self: holoscan.core._core.ArgContainerType*, *value: int*) None

property name

property value

class holoscan.core.ArgElementType

Bases: pybind11_builtins.pybind11_object

Enum class for an *Arg*'s element type.

Members:

CUSTOM

BOOLEAN

INT8

UNSIGNED8

INT16

UNSIGNED16

INT32

UNSIGNED32

INT64

UNSIGNED64

FLOAT32

FLOAT64

STRING

HANDLE

YAML_NODE

IO_SPEC

CONDITION

RESOURCE

Attributes

name	

value	

BOOLEAN = *<ArgElementType.BOOLEAN: 1>*

CONDITION = *<ArgElementType.CONDITION: 18>*

CUSTOM = *<ArgElementType.CUSTOM: 0>*

FLOAT32 = *<ArgElementType.FLOAT32: 10>*

FLOAT64 = *<ArgElementType.FLOAT64: 11>*

HANDLE = *<ArgElementType.HANDLE: 15>*

INT16 = *<ArgElementType.INT16: 4>*

INT32 = *<ArgElementType.INT32: 6>*

INT64 = *<ArgElementType.INT64: 8>*

INT8 = *<ArgElementType.INT8: 2>*

IO_SPEC = *<ArgElementType.IO_SPEC: 17>*

RESOURCE = *<ArgElementType.RESOURCE: 19>*

STRING = *<ArgElementType.STRING: 14>*

UNSIGNED16 = *<ArgElementType.UNSIGNED16: 5>*

UNSIGNED32 = *<ArgElementType.UNSIGNED32: 7>*

UNSIGNED64 = *<ArgElementType.UNSIGNED64: 9>*

UNSIGNED8 = *<ArgElementType.UNSIGNED8: 3>*

YAML_NODE = *<ArgElementType.YAML_NODE: 16>*

__init__(*self: holoscan.core._core.ArgElementType*, *value: int*) None

property name

property value

class holoscan.core.ArgList

Bases: pybind11_builtins.pybind11_object

Class representing a list of arguments.

Attributes

args	The underlying list of *Arg* objects.
description	YAML formatted string describing the list.
name	The name of the argument list.
size	The number of arguments in the list.

Methods

add (*args, **kwargs)	Overloaded function.
clear (self)	Clear the argument list.

__init__(*self: holoscan.core._core.ArgList*) None

Class representing a list of arguments.

add(**args*, ***kwargs*)

Overloaded function.

1. add(self: holoscan.core._core.ArgList, arg: holoscan.core._core.Arg) -> None

Add an argument to the list.

2. add(self: holoscan.core._core.ArgList, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the list.

property args

The underlying list of *Arg* objects.

clear(*self: holoscan.core._core.ArgList*) None

Clear the argument list.

property description

YAML formatted string describing the list.

property name

The name of the argument list.

Returns

name

property size

The number of arguments in the list.

class holoscan.core.ArgType

Bases: pybind11_builtins.pybind11_object

Class containing argument type info.

Attributes

container_type	The container type of the argument.
dimension	The dimension of the argument container.
element_type	The element type of the argument.
to_string	String describing the argument type.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: holoscan.core._core.ArgType) -> None

Class containing argument type info.

2. __init__(self: holoscan.core._core.ArgType, element_type: holoscan.core._core.ArgElementType, container_type: holoscan.core._core.ArgContainerType) -> None

Class containing argument type info.

Parameters

element_type

Element type of the argument.

container_type

Container type of the argument.

property container_type

The container type of the argument.

property dimension

The dimension of the argument container.

property element_type

The element type of the argument.

property to_string

String describing the argument type.

class holoscan.core.CLIOptions

Bases: pybind11_builtins.pybind11_object

Attributes

config_path	The path to the configuration file.
driver_address	The address of the App Driver.

run_driver	The flag to run the App Driver.
run_worker	The flag to run the App Worker.
worker_address	The address of the App Worker.
worker_targets	The list of fragments for the App Worker.

Methods

print(self)	Print the CLI Options.

__init__(*self: holoscan.core._core.CLIOptions*, *run_driver: bool = False*, *run_worker: bool = False*, *driver_address: str = ''*, *worker_address: str = ''*, *worker_targets: List[str] = []*, *config_path: str = ''*) None

CLIOptions class.

property config_path

The path to the configuration file.

property driver_address

The address of the App Driver.

print(*self: holoscan.core._core.CLIOptions*) None

Print the CLI Options.

property run_driver

The flag to run the App Driver.

property run_worker

The flag to run the App Worker.

property worker_address

The address of the App Worker.

property worker_targets

The list of fragments for the App Worker.

Returns

worker_targets

class holoscan.core.Component

Bases: holoscan.core._core.ComponentBase

Base component class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the component.
fragment	The fragment containing the component.
id	The identifier of the component.
name	The name of the component.

Methods

add_arg(*args, **kwargs)	Overloaded function.
initialize(self)	Initialize the component.

__init__(*self: holoscan.core._core.Component*) None

Base component class.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the component.

property fragment

The fragment containing the component.

Returns

name

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.core._core.ComponentBase*) None

Initialize the component.

property name

The name of the component.

Returns

name

class holoscan.core.ComponentSpec

Bases: `pybind11_builtins.pybind11_object`

Component specification class.

Attributes

description	YAML formatted string describing the component spec.
fragment	The fragment that the component belongs to.
params	The parameters associated with the component.

__init__(*self: holoscan.core._core.ComponentSpec*, *fragment: holoscan::Fragment*) None

Component specification class.

Parameters

fragment

The fragment that the component belongs to.

property description

YAML formatted string describing the component spec.

property fragment

The fragment that the component belongs to.

Returns

name

property params

The parameters associated with the component.

class holoscan.core.Condition

Bases: holoscan.core._core.Component

Class representing a condition.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the condition.
fragment	Fragment that the condition belongs to.
id	The identifier of the component.
name	The name of the condition.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
initialize (self)	initialization method for the condition.
setup (self, arg0)	setup method for the condition.

__init__(*self: holoscan.core._core.Condition*, **args*, ***kwargs*) None

Class representing a condition.

Can be initialized with any number of Python positional and keyword arguments.

If a *name* keyword argument is provided, it must be a *str* and will be used to set the name of the Operator.

If a *fragment* keyword argument is provided, it must be of type *holoscan.core.Fragment* (or *holoscan.core.Application*). A single *Fragment* object can also be provided positionally instead.

Any other arguments will be cast from a Python argument type to a C++ *Arg* and stored in `self.args`. (For details on how the casting is done, see the *py_object_to_arg* utility).

Parameters

***args**

Positional arguments.

****kwargs**

Keyword arguments.

Raises

RuntimeError

If *name* kwarg is provided, but is not of *str* type. If multiple arguments of type *Fragment* are provided. If any other arguments cannot be converted to *Arg* type via *py_object_to_arg*.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the condition.

property fragment

Fragment that the condition belongs to.

Returns

name

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.core._core.Condition*) None

initialization method for the condition.

property name

The name of the condition.

Returns

name

setup(*self:* holoscan.core._core.Condition, *arg0:* holoscan.core._core.ComponentSpec) None

setup method for the condition.

property spec

class holoscan.core.ConditionType

Bases: pybind11_builtins.pybind11_object

Enum class for Condition types.

Members:

NONE

MESSAGE_AVAILABLE

DOWNSTREAM_MESSAGE_AFFORDABLE

COUNT

BOOLEAN

Attributes

name	

value	

BOOLEAN = <*ConditionType.BOOLEAN: 4*>

COUNT = <*ConditionType.COUNT: 3*>

DOWNSTREAM_MESSAGE_AFFORDABLE = <*ConditionType.DOWNSTREAM_MESSAGE_AFFORDABLE: 2*>

MESSAGE_AVAILABLE = <*ConditionType.MESSAGE_AVAILABLE: 1*>

NONE = <*ConditionType.NONE: 0*>

__init__(*self: holoscan.core._core.ConditionType*, *value: int*) None

property name

property value

class holoscan.core.Config

Bases: pybind11_builtins.pybind11_object

Configuration class.

Represents configuration parameters as read from a YAML file.

Attributes

config_file	The configuration file (in YAML format) associated with the Config object.
prefix	TODO

__init__(*self: holoscan.core._core.Config*, *config_file: str*, *prefix: str = ''*) None

Configuration class.

Represents configuration parameters as read from a YAML file.

Parameters

config_file

The path to the configuration file (in YAML format).

prefix

TODO

property config_file

The configuration file (in YAML format) associated with the Config object.

property prefix

TODO

class holoscan.core.DLDevice

Bases: pybind11_builtins.pybind11_object

DLDevice class.

Attributes

device_id	The device id (int).
device_type	The device type (*DLDeviceType*).

__init__(*self: holoscan.core._core.DLDevice*, *arg0: holoscan.core._core.DLDeviceType*, *arg1: int*) None

property device_id

The device id (int).

property device_type

The device type (*DLDeviceType*).

The following device types are supported:

- *DLDeviceType.DLCPU*: system memory (kDLCPU)
- *DLDeviceType.DLCUDA*: CUDA GPU memory (kDLCUDA)
- *DLDeviceType.DLCUDAHost*: CUDA pinned memory (kDLCUDAHost)
- *DLDeviceType.DLCUDAManaged*: CUDA managed memory (kDLCUDAManaged)

class holoscan.core.DLDeviceType

Bases: pybind11_builtins.pybind11_object

Members:

DLCPU

DLCUDA

DLCUDAHOST

DLCUDAMANAGED

Attributes

name	

value	

DLCPU = *<DLDeviceType.DLCPU: 1>*

DLCUDA = *<DLDeviceType.DLCUDA: 2>*

DLCUDAHOST = *<DLDeviceType.DLCUDAHOST: 3>*

DLCUDAMANAGED = *<DLDeviceType.DLCUDAMANAGED: 13>*

__init__(*self: holoscan.core._core.DLDeviceType*, *value: int*) None

property name

property value

class holoscan.core.DataFlowMetric

Bases: pybind11_builtins.pybind11_object

Enum class for DataFlowMetric type.

Members:

MAX_MESSAGE_ID

MIN_MESSAGE_ID

MAX_E2E_LATENCY

AVG_E2E_LATENCY

MIN_E2E_LATENCY

NUM_SRC_MESSAGES

NUM_DST_MESSAGES

Attributes

name	

value	

AVG_E2E_LATENCY = *<DataFlowMetric.AVG_E2E_LATENCY: 3>*

MAX_E2E_LATENCY = *<DataFlowMetric.MAX_E2E_LATENCY: 2>*

MAX_MESSAGE_ID = *<DataFlowMetric.MAX_MESSAGE_ID: 0>*

MIN_E2E_LATENCY = *<DataFlowMetric.MIN_E2E_LATENCY: 4>*

MIN_MESSAGE_ID = *<DataFlowMetric.MIN_MESSAGE_ID: 1>*

NUM_DST_MESSAGES = *<DataFlowMetric.NUM_DST_MESSAGES: 6>*

NUM_SRC_MESSAGES = *<DataFlowMetric.NUM_SRC_MESSAGES: 5>*

__init__(*self: holoscan.core._core.DataFlowMetric*, *value: int*) None

property name

property value

class holoscan.core.DataFlowTracker

Bases: pybind11_builtins.pybind11_object

Data Flow Tracker class.

The DataFlowTracker class is used to track the data flow metrics for different paths between the root and leaf operators. This class is used by developers to get data flow metrics either during the execution of the application and/or as a summary after the application ends.

Methods

enable_logging(self[, filename, ...])	Enable logging of frames at the end of the every execution of a leaf Operator.
end_logging(self)	Write out any remaining messages from the log buffer and close the file
get_metric(*args, **kwargs)	Overloaded function.
get_num_paths(self)	The number of tracked paths
get_path_strings(self)	Return an array of strings which are path names.

`print` (self)	Print the result of the data flow tracking in pretty-printed format to the standard output
`set_discard_last_messages` (self, arg0)	Set the number of messages to discard at the end of the execution.
`set_skip_latencies` (self, arg0)	Set the threshold latency for which the end-to-end latency calculations will be done.
`set_skip_starting_messages` (self, arg0)	Set the number of messages to skip at the beginning of the execution.

__init__(*self: holoscan.core._core.DataFlowTracker*) None

Data Flow Tracker class.

The DataFlowTracker class is used to track the data flow metrics for different paths between the root and leaf operators. This class is used by developers to get data flow metrics either during the execution of the application and/or as a summary after the application ends.

enable_logging(*self: holoscan.core._core.DataFlowTracker*, *filename: str = 'logger.log'*, *num_buffered_messages: int = 100*) None

Enable logging of frames at the end of the every execution of a leaf Operator.

A path consisting of an array of tuples in the form of (an Operator name, message receive timestamp, message publish timestamp) is logged in a file. The logging does not take into account the number of message to skip or discard or the threshold latency.

This function buffers a number of lines set by *num_buffered_messages* before flushing the buffer to the log file.

Parameters

filename

The name of the log file.

num_buffered_messages

The number of messages to be buffered before flushing the buffer to the log file.

end_logging(*self: holoscan.core._core.DataFlowTracker*) None

Write out any remaining messages from the log buffer and close the file

get_metric(**args*, ***kwargs*)

Overloaded function.

1. get_metric(self: holoscan.core._core.DataFlowTracker, pathstring: str, metric: holoscan.core._core.DataFlowMetric) -> float

Return the value of a metric for a given path.

If *metric* is DataFlowMetric::NUM_SRC_MESSAGES, then the function returns -1.

Parameters

pathstring

The path name string for which the metric is being queried

metric

The metric to be queried.

Returns

val

The value of the metric for the given path

Notes

There is also an overloaded version of this function that takes only the *metric* argument.

2. get_metric(self: holoscan.core._core.DataFlowTracker, metric: holoscan.core._core.DataFlowMetric = <DataFlowMetric.NUM_SRC_MESSAGES: 5>) -> Dict[str, int]

get_num_paths(*self: holoscan.core._core.DataFlowTracker*) int

The number of tracked paths

Returns

num_paths

The number of tracked paths

get_path_strings(*self: holoscan.core._core.DataFlowTracker*) List[str]

Return an array of strings which are path names. Each path name is a comma-separated list of Operator names in a path. The paths are agnostic to the edges between two Operators.

Returns

paths

A list of the path names.

print(*self: holoscan.core._core.DataFlowTracker*) None

Print the result of the data flow tracking in pretty-printed format to the standard output

set_discard_last_messages(*self: holoscan.core._core.DataFlowTracker*, *arg0: int*) None

Set the number of messages to discard at the end of the execution.

This does not affect the log file or the number of source messages metric.

Parameters

num

The number of messages to discard.

set_skip_latencies(*self: holoscan.core._core.DataFlowTracker*, *arg0: int*) None

Set the threshold latency for which the end-to-end latency calculations will be done. Any latency strictly less than the threshold latency will be ignored.

This does not affect the log file or the number of source messages metric.

Parameters

threshold

The threshold latency in milliseconds.

set_skip_starting_messages(*self: holoscan.core._core.DataFlowTracker*, *arg0: int*) None

Set the number of messages to skip at the beginning of the execution.

This does not affect the log file or the number of source messages metric.

Parameters

num

The number of messages to skip.

class holoscan.core.ExecutionContext

Bases: pybind11_builtins.pybind11_object

Class representing an execution context.

__init__(**args*, ***kwargs*)

class holoscan.core.Executor

Bases: pybind11_builtins.pybind11_object

Executor class.

Attributes

context	The corresponding GXF context.
context_uint64	The corresponding GXF context represented as a 64-bit unsigned integer address
fragment	The fragment that the executor belongs to.

Methods

run(self, arg0)	Method that can be called to run the executor.

__init__(*self: holoscan.core._core.Executor*, *fragment: holoscan::Fragment*) None

Executor class.

Parameters

fragment

The fragment that the executor is associated with.

property context

The corresponding GXF context. This will be an opaque PyCapsule object.

property context_uint64

The corresponding GXF context represented as a 64-bit unsigned integer address

property fragment

The fragment that the executor belongs to.

Returns

name

run(*self: holoscan.core._core.Executor*, *arg0: holoscan.graphs._graphs.OperatorGraph*) None

Method that can be called to run the executor.

class holoscan.core.Fragment(*app=None*, *name=''*, **args*, ***kwargs*)

Bases: holoscan.core._core.Fragment

Fragment class.

Attributes

application	The application associated with the fragment.
executor	Get the executor associated with the fragment.
graph	Get the computation graph (Graph node is an Operator) associated with the fragment.
name	The fragment's name.

Methods

add_flow(*args, **kwargs)	Overloaded function.

`add_operator` (self, op)	Add an operator to the fragment.
`compose` (self)	The compose method of the Fragment.
`config` (*args, **kwargs)	Overloaded function.
`config_keys` (self)	The set of keys present in the fragment's configuration file.
`from_config` (self, key)	Retrieve parameters from the associated configuration.
`kwargs` (self, key)	Retrieve a dictionary parameters from the associated configuration.
`network_context` (*args, **kwargs)	Overloaded function.
`run` (self)	The run method of the Fragment.

run_async()	Run the fragment asynchronously.
scheduler(*args, **kwargs)	Overloaded function.
track(self, num_start_messages_to_skip, ...)	The track method of the application.

__init__(*self: holoscan.core._core.Fragment*, *arg0: object*) None

Fragment class.

add_flow(**args*, ***kwargs*)

Overloaded function.

1. add_flow(self: holoscan.core._core.Fragment, upstream_op: holoscan.core._core.Operator, downstream_op: holoscan.core._core.Operator) -> None

2. add_flow(self: holoscan.core._core.Fragment, upstream_op: holoscan.core._core.Operator, downstream_op: holoscan.core._core.Operator, port_pairs: Set[Tuple[str, str]]) -> None

Connect two operators associated with the fragment.

Parameters

upstream_op

Source operator.

downstream_op

Destination operator.

port_pairs

Sequence of ports to connect. The first element of each 2-tuple is a port from *upstream_op* while the second element is the port of *downstream_op* to which it connects.

Notes

This is an overloaded function. Additional variants exist:

1.) For the Application class there is a variant where the first two arguments are of type *holoscan.core.Fragment* instead of *holoscan.core.Operator*. This variant is used in building multi-fragment applications. 2.) There are also variants that omit the *port_pairs* argument that are applicable when there is only a single output on the upstream operator/fragment and a single input on the downstream operator/fragment.

add_operator(*self: holoscan.core._core.Fragment*, *op: holoscan.core._core.Operator*) None

Add an operator to the fragment.

Parameters

op

The operator to add.

property application

The application associated with the fragment.

Returns

app

compose(*self: holoscan.core._core.Fragment*) None

The compose method of the Fragment.

This method should be called after *config*, but before *run* in order to compose the computation graph.

config(**args*, ***kwargs*)

Overloaded function.

1. config(self: holoscan.core._core.Fragment, config_file: str, prefix: str = ‘’) -> None

Configuration class.

Represents configuration parameters as read from a YAML file.

Parameters

config

The path to the configuration file (in YAML format) or a *holoscan.core.Config* object.

prefix

Prefix path for the` config` file. Only available in the overloaded variant that takes a string for *config*.

2. config(self: holoscan.core._core.Fragment, arg0: holoscan.core._core.Config) -> None
3. config(self: holoscan.core._core.Fragment) -> holoscan.core._core.Config

config_keys(*self: holoscan.core._core.Fragment*) Set[str]

The set of keys present in the fragment’s configuration file.

property executor

Get the executor associated with the fragment.

from_config(*self: holoscan.core._core.Fragment*, *key: str*) object

Retrieve parameters from the associated configuration.

Parameters

key

The key within the configuration file to retrieve. This can also be a specific component of the parameter via syntax *'key.sub_key'*.

Returns

args

An argument list associated with the key.

property graph

Get the computation graph (Graph node is an Operator) associated with the fragment.

kwargs(*self: holoscan.core._core.Fragment*, *key: str*) dict

Retrieve a dictionary parameters from the associated configuration.

Parameters

key

The key within the configuration file to retrieve. This can also be a specific component of the parameter via syntax *'key.sub_key'*.

Returns

kwargs

A Python dict containing the parameters in the configuration file under the specified key.

property name

The fragment's name.

Returns

name

network_context(**args*, ***kwargs*)

Overloaded function.

1. network_context(self: holoscan.core._core.Fragment, network_context: holoscan.core._core.NetworkContext) -> None

Assign a network context to the Fragment

Parameters

network_context

A network_context class instance to be used by the underlying GXF executor. If unspecified, no network context will be used.

2. network_context(self: holoscan.core._core.Fragment) -> holoscan.core._core.NetworkContext
Get the network context to be used by the Fragment

run(*self: holoscan.core._core.Fragment*) None

The run method of the Fragment.

This method runs the computation. It must have first been initialized via *config* and *compose*.

run_async()

Run the fragment asynchronously.

This method is a convenience method that creates a thread pool with one thread and runs the fragment in that thread. The thread pool is created using *concurrent.futures.ThreadPoolExecutor*.

Returns

future : concurrent.futures.Future

scheduler(**args*, ***kwargs*)

Overloaded function.

1. scheduler(self: holoscan.core._core.Fragment, scheduler: holoscan.core._core.Scheduler) -> None

Assign a scheduler to the Fragment.

Parameters

scheduler

A scheduler class instance to be used by the underlying GXF executor. If unspecified, the default is a *holoscan.gxf.GreedyScheduler*.

2. scheduler(self: holoscan.core._core.Fragment) -> holoscan.core._core.Scheduler
Get the scheduler to be used by the Fragment.

track(*self: holoscan.core._core.Fragment*, *num_start_messages_to_skip: int = 10*, *num_last_messages_to_discard: int = 10*, *latency_threshold: int = 0*) holoscan::DataFlowTracker

The track method of the application.

This method enables data frame flow tracking and returns a DataFlowTracker object which can be used to display metrics data for profiling an application.

Parameters

num_start_messages_to_skip

The number of messages to skip at the beginning.

num_last_messages_to_discard

The number of messages to discard at the end.

latency_threshold

The minimum end-to-end latency in milliseconds to account for in the end-to-end latency metric calculations

class holoscan.core.FragmentGraph

Bases: pybind11_builtins.pybind11_object

Abstract base class for all graphs

__init__(**args*, ***kwargs*)

holoscan.core.Graph

alias of holoscan.graphs._graphs.OperatorGraph

class holoscan.core.IOSpec

Bases: pybind11_builtins.pybind11_object

I/O specification class.

Attributes

conditions	List of Condition objects associated with this I/O specification.
connector_type	The receiver or transmitter type of the I/O specification class.
io_type	The type (input or output) of the I/O specification class.
name	The name of the I/O specification class.

Methods

condition (self, arg0, **kwargs)	Add a condition to this input/output.
connector (*args,	Overloaded function.

**kwargs)	

ConnectorType	
IOType	

class ConnectorType

Bases: pybind11_builtins.pybind11_object

Enum representing the receiver type (for input specs) or transmitter type (for output specs).

Members:

DEFAULT

DOUBLE_BUFFER

UCX

Attributes

name	

value	

DEFAULT = <*ConnectorType.DEFAULT: 0*>

DOUBLE_BUFFER = <*ConnectorType.DOUBLE_BUFFER: 1*>

UCX = <*ConnectorType.UCX: 2*>

__init__(*self: holoscan.core._core.IOSpec.ConnectorType*, *value: int*) None

property name

property value

class IOType

Bases: pybind11_builtins.pybind11_object

Enum representing the I/O specification type (input or output).

Members:

INPUT

OUTPUT

Attributes

name	

value	

INPUT = *<IOType.INPUT: 0>*

OUTPUT = *<IOType.OUTPUT: 1>*

__init__(*self: holoscan.core._core.IOSpec.IOType*, *value: int*) None

property name

property value

__init__(*self: holoscan.core._core.IOSpec*, *op_spec: holoscan::OperatorSpec*, *name: str*, *io_type: holoscan.core._core.IOSpec.IOType*) None

I/O specification class.

Parameters

op_spec

Operator specification class of the associated operator.

name

The name of the IOSpec object.

io_type

Enum indicating whether this is an input or output specification.

condition(*self: holoscan.core._core.IOSpec*, *arg0: holoscan.core._core.ConditionType*, ***kwargs*) holoscan.core._core.IOSpec

Add a condition to this input/output.

The following ConditionTypes are supported:

- *ConditionType.NONE*
- *ConditionType.MESSAGE_AVAILABLE*
- *ConditionType.DOWNSTREAM_MESSAGE_AFFORDABLE*
- *ConditionType.COUNT*
- *ConditionType.BOOLEAN*

Parameters

kind

The type of the condition.

****kwargs**

Python keyword arguments that will be cast to an *ArgList* associated with the condition.

Returns

obj

The self object.

property conditions

List of Condition objects associated with this I/O specification.

Returns

condition

connector(**args*, ***kwargs*)

Overloaded function.

1. connector(self: holoscan.core._core.IOSpec, arg0: holoscan.core._core.IOSpec.ConnectorType, **kwargs) -> holoscan.core._core.IOSpec

Add a connector (transmitter or receiver) to this input/output.

The following ConditionTypes are supported:

- *IOSpec.ConnectorType.DEFAULT*
- *IOSpec.ConnectorType.DOUBLE_BUFFER*
- *IOSpec.ConnectorType.UCX*

If this method is not been called, the IOSpec's *connector_type* will be *ConnectorType.DEFAULT* which will result in a DoubleBuffered receiver or or transmitter being used (or their annotated variant if flow tracking is enabled).

Parameters

kind

The type of the connector. For example for type *IOSpec.ConnectorType.DOUBLE_BUFFER*, a *holoscan.resources.DoubleBufferReceiver* will be used for an input port and a *holoscan.resources.DoubleBufferTransmitter* will be used for an output port.

****kwargs**

Python keyword arguments that will be cast to an *ArgList* associated with the resource (connector).

Returns

obj

The self object.

Notes

This is an overloaded function. Additional variants exist:

1.) A variant with no arguments will just return the *holoscan.core.Resource* corresponding to the transmitter or receiver used by this *IOSpec* object. If None was explicitly set, it will return `None`.

2.) A variant that takes a single *holoscan.core.Resource* corresponding to a transmitter or receiver as an argument. This sets the transmitter or receiver used by the *IOSpec* object.

2. connector(self: holoscan.core._core.IOSpec) -> holoscan.core._core.Resource

3. connector(self: holoscan.core._core.IOSpec, arg0: holoscan.core._core.Resource) -> None

property connector_type

The receiver or transmitter type of the I/O specification class.

Returns

connector_type

property io_type

The type (input or output) of the I/O specification class.

Returns

io_type

property name

The name of the I/O specification class.

Returns

name

class holoscan.core.InputContext

Bases: pybind11_builtins.pybind11_object

Class representing an input context.

Methods

receive(self, name)	

__init__(**args*, ***kwargs*)

receive(*self: holoscan.core._core.InputContext*, *name: str*) None

class holoscan.core.Message

Bases: pybind11_builtins.pybind11_object

Class representing a message.

A message is a data structure that is used to pass data between operators. It wraps a std::any object and provides a type-safe interface to access the data.

This class is used by the *holoscan::gxf::GXFWrapper* to support the Holoscan native operator. The *holoscan::gxf::GXFWrapper* will hold the object of this class and delegate the message to the Holoscan native operator.

__init__(**args*, ***kwargs*)

class holoscan.core.NetworkContext

Bases: holoscan.core._core.Component

Class representing a network context.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the component.
fragment	Fragment that the network context belongs to.
id	The identifier of the component.
name	The name of the network context.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
initialize (self)	initialization method for the network context.
setup (self, arg0)	setup method for the network context.

__init__(*self: holoscan.core._core.NetworkContext*, **args*, ***kwargs*) None

Class representing a network context.

Parameters

***args**

Positional arguments.

****kwargs**

Keyword arguments.

Raises

RuntimeError

If *name* kwarg is provided, but is not of *str* type. If multiple arguments of type *Fragment* are provided. If any other arguments cannot be converted to *Arg* type via *py_object_to_arg*.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the component.

property fragment

Fragment that the network context belongs to.

Returns

name

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.core._core.NetworkContext*) None

initialization method for the network context.

property name

The name of the network context.

Returns

name

setup(*self:* *holoscan.core._core.NetworkContext*, *arg0:* *holoscan.core._core.ComponentSpec*) None

setup method for the network context.

property spec

class holoscan.core.Operator(*fragment*, **args*, ***kwargs*)

Bases: `holoscan.core._core.Operator`

Operator class.

Can be initialized with any number of Python positional and keyword arguments.

If a *name* keyword argument is provided, it must be a *str* and will be used to set the name of the Operator.

Condition classes will be added to `self.conditions`, *Resource* classes will be added to `self.resources`, and any other arguments will be cast from a Python argument type to a C++ *Arg* and stored in `self.args`. (For details on how the casting is done, see the *py_object_to_arg* utility). When a Condition or Resource is provided via a kwarg, it's name will be automatically be updated to the name of the kwarg.

Parameters

fragment

The *holoscan.core.Fragment* (or *holoscan.core.Application*) to which this Operator will belong.

***args**

Positional arguments.

****kwargs**

Keyword arguments.

Raises

RuntimeError

If *name* kwarg is provided, but is not of *str* type. If multiple arguments of type *Fragment* are provided. If any other arguments cannot be converted to *Arg* type via *py_object_to_arg*.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.

`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`compute` (op_input, op_output, context)	Default implementation of compute
`initialize` ()	Default implementation of initialize

`setup`(spec)	Default implementation of setup method.
`start`()	Default implementation of start
`stop`()	Default implementation of stop

OperatorType	

class OperatorType

Bases: `pybind11_builtins.pybind11_object`

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

`name`	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.core._core.Operator*, *arg0: object*, *arg1: holoscan::Fragment*, **args*, ***kwargs*) None

Operator class.

Can be initialized with any number of Python positional and keyword arguments.

If a *name* keyword argument is provided, it must be a *str* and will be used to set the name of the Operator.

Condition classes will be added to `self.conditions`, *Resource* classes will be added to `self.resources`, and any other arguments will be cast from a Python argument type to a C++ *Arg* and stored in `self.args`. (For details on how the casting is done, see the *py_object_to_arg* utility). When a Condition or Resource is provided via a kwarg, it's name will be automatically be updated to the name of the kwarg.

Parameters

fragment

The *holoscan.core.Fragment* (or *holoscan.core.Application*) to which this Operator will belong.

***args**

Positional arguments.

****kwargs**

Keyword arguments.

Raises

RuntimeError

If *name* kwarg is provided, but is not of *str* type. If multiple arguments of type *Fragment* are provided. If any other arguments cannot be converted to *Arg* type via *py_object_to_arg*.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*op_input*, *op_output*, *context*)

Default implementation of compute

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1` , and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize()

Default implementation of initialize

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*spec: holoscan.core._core.PyOperatorSpec*)

Default implementation of setup method.

property spec

The operator spec (holoscan.core.OperatorSpec) associated with the operator.

start()

Default implementation of start

stop()

Default implementation of stop

class holoscan.core.OperatorGraph

Bases: pybind11_builtins.pybind11_object

Abstract base class for all graphs

__init__(**args*, ***kwargs*)

holoscan.core.OperatorSpec

alias of holoscan.core._core.PyOperatorSpec

class holoscan.core.OutputContext

Bases: pybind11_builtins.pybind11_object

Class representing an output context.

Methods

emit(self, data[, name])	

OutputType	

class OutputType

Bases: pybind11_builtins.pybind11_object

Members:

SHARED_POINTER

GXF_ENTITY

Attributes

name	

value	

GXF_ENTITY = <*OutputType.GXF_ENTITY: 1*>

SHARED_POINTER = <*OutputType.SHARED_POINTER: 0*>

__init__(*self: holoscan.core._core.OutputContext.OutputType*, *value: int*) None

property name

property value

__init__(**args*, ***kwargs*)

emit(*self: holoscan.core._core.OutputContext*, *data: object*, *name: str = ''*) None

class holoscan.core.ParameterFlag

Bases: pybind11_builtins.pybind11_object

Enum class for parameter flags.

The following flags are supported: - *NONE*: The parameter is mendatory and static. It cannot be changed at runtime. - *OPTIONAL*: The parameter is optional and might not be available at runtime. - *DYNAMIC*: The parameter is dynamic and might change at runtime.

Members:

NONE

OPTIONAL

DYNAMIC

Attributes

name	

value	

DYNAMIC = *<ParameterFlag.DYNAMIC: 2>*

NONE = *<ParameterFlag.NONE: 0>*

OPTIONAL = *<ParameterFlag.OPTIONAL: 1>*

__init__(*self: holoscan.core._core.ParameterFlag*, *value: int*) None

property name

property value

class holoscan.core.Resource

Bases: holoscan.core._core.Component

Class representing a resource.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
initialize (self)	initialization method for the resource.

`setup`(self, arg0)	setup method for the resource.

__init__(*self: holoscan.core._core.Resource*, **args*, ***kwargs*) None

Class representing a resource.

Can be initialized with any number of Python positional and keyword arguments.

If a *name* keyword argument is provided, it must be a *str* and will be used to set the name of the Operator.

If a *fragment* keyword argument is provided, it must be of type *holoscan.core.Fragment* (or *holoscan.core.Application*). A single *Fragment* object can also be provided positionally instead.

Any other arguments will be cast from a Python argument type to a C++ *Arg* and stored in `self.args`. (For details on how the casting is done, see the *py_object_to_arg* utility).

Parameters

***args**

Positional arguments.

****kwargs**

Keyword arguments.

Raises

RuntimeError

If *name* kwarg is provided, but is not of *str* type. If multiple arguments of type *Fragment* are provided. If any other arguments cannot be converted to *Arg* type via *py_object_to_arg*.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.core._core.Resource*) None

initialization method for the resource.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.core._core.Resource*, *arg0: holoscan.core._core.ComponentSpec*) None

setup method for the resource.

property spec

class holoscan.core.Scheduler

Bases: holoscan.core._core.Component

Class representing a scheduler.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the component.
fragment	Fragment that the scheduler belongs to.
id	The identifier of the component.
name	The name of the scheduler.

spec	

Methods

`add_arg`(*args, **kwargs)	Overloaded function.
`initialize`(self)	initialization method for the scheduler.
`setup`(self, arg0)	setup method for the scheduler.

__init__(*self: holoscan.core._core.Scheduler*, **args*, ***kwargs*) None

Class representing a scheduler.

Can be initialized with any number of Python positional and keyword arguments.

If a *name* keyword argument is provided, it must be a *str* and will be used to set the name of the Operator.

If a *fragment* keyword argument is provided, it must be of type *holoscan.core.Fragment* (or *holoscan.core.Application*). A single *Fragment* object can also be provided positionally instead.

Any other arguments will be cast from a Python argument type to a C++ *Arg* and stored in `self.args`. (For details on how the casting is done, see the *py_object_to_arg* utility).

Parameters

***args**

Positional arguments.

****kwargs**

Keyword arguments.

Raises

RuntimeError

If *name* kwarg is provided, but is not of *str* type. If multiple arguments of type *Fragment* are provided. If any other arguments cannot be converted to *Arg* type via *py_object_to_arg*.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the component.

property fragment

Fragment that the scheduler belongs to.

Returns

name

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.core._core.Scheduler*) None

initialization method for the scheduler.

property name

The name of the scheduler.

Returns

name

setup(*self: holoscan.core._core.Scheduler*, *arg0: holoscan.core._core.ComponentSpec*) None

setup method for the scheduler.

property spec

holoscan.core.Tensor

alias of `holoscan.core._core.PyTensor`

class holoscan.core.Tracker(*app*, *, *filename=None*, *num_buffered_messages=100*, *num_start_messages_to_skip=10*, *num_last_messages_to_discard=10*, *latency_threshold=0*)

Bases: object

Context manager to add data flow tracking to an application.

__init__(*app*, *, *filename=None*, *num_buffered_messages=100*, *num_start_messages_to_skip=10*, *num_last_messages_to_discard=10*, *latency_threshold=0*)

Parameters

app

on which flow tracking should be applied.

filename

If none, logging to file will be disabled. Otherwise, logging will write to the specified file.

num_buffered_messages

Controls the number of messages buffered between file writing when *filename* is not None.

num_start_messages_to_skip

The number of messages to skip at the beginning of the execution. This does not affect the log file or the number of source messages metric.

num_last_messages_to_discard

The number of messages to discard at the end of the execution. This does not affect the log file or the number of source messages metric.

latency_threshold

The minimum end-to-end latency in milliseconds to account for in the end-to-end latency metric calculations.

holoscan.core.arg_to_py_object(*arg: holoscan.core._core.Arg*) object

Utility that converts an *Arg* to a corresponding Python object.

Parameters

arg

The argument to convert.

Returns

obj

Python object corresponding to the provided argument. For example, an argument of any integer type will become a Python *int* while *std::vector<double>* would become a list of Python floats.

holoscan.core.arglist_to_kwargs(*arglist: holoscan.core._core.ArgList*) dict

Utility that converts an *ArgList* to a Python kwargs dictionary.

Parameters

arglist

The argument list to convert.

Returns

kwargs

Python dictionary with keys matching the names of the arguments in *ArgList*. The values will be converted as for *arg_to_py_object*.

holoscan.core.kwargs_to_arglist(***kwargs*) holoscan.core._core.ArgList

Utility that converts a set of python keyword arguments to an *ArgList*.

Parameters

****kwargs**

The python keyword arguments to convert.

Returns

arglist

ArgList class corresponding to the provided keyword values. The argument names will match the keyword names. Values will be converted as for *py_object_to_arg*.

holoscan.core.py_object_to_arg(*obj: object*, *name: str = ''*) holoscan.core._core.Arg

Utility that converts a single python argument to a corresponding *Arg* type.

Parameters

value

The python value to convert.

Returns

obj

Arg class corresponding to the provided value. For example a Python float will become an *Arg* containing a C++ double while a list of Python ints would become an *Arg* corresponding to a `std::vector<uint64_t>`.

name

A name to assign to the argument.

holoscan.executors

This module provides a Python API for the C++ API Executor classes.

holoscan.executors.GXFExecutor	GXF-based executor class.

class holoscan.executors.GXFExecutor

Bases: `holoscan.core._core.Executor`

GXF-based executor class.

Attributes

context	The corresponding GXF context.
context_uint64	The corresponding GXF context represented as a 64-bit unsigned integer address
fragment	The fragment that the executor belongs to.

Methods

run(self, arg0)	Method that can be called to run the executor.

__init__(*self: holoscan.executors._executors.GXFExecutor*, *app: holoscan.core._core.Fragment*) None

GXF-based executor class.

Parameters

app

The fragment associated with the executor.

property context

The corresponding GXF context. This will be an opaque PyCapsule object.

property context_uint64

The corresponding GXF context represented as a 64-bit unsigned integer address

property fragment

The fragment that the executor belongs to.

Returns

name

run(*self: holoscan.core._core.Executor*, *arg0: holoscan.graphs._graphs.OperatorGraph*)
None

Method that can be called to run the executor.

holoscan.graphs

This module provides a Python API for the C++ API Graph classes.

holoscan.graphs.FlowGraph	alias of holoscan.graphs._graphs.OperatorFlowGraph
holoscan.graphs.FragmentFlowGraph	Directed graph class.
holoscan.graphs.OperatorFlowGraph	Directed graph class.

holoscan.graphs.FlowGraph

alias of `holoscan.graphs._graphs.OperatorFlowGraph`

class holoscan.graphs.FragmentFlowGraph

Bases: `holoscan.graphs._graphs.FragmentGraph`

Directed graph class.

Attributes

`context`	The graph's context (as an opaque PyCapsule object)

Methods

`add_node`(self, node)	Add the node to the graph.
`get_next_nodes`(self, arg0)	Get the nodes immediately downstream of a given node.
`get_nodes`(self)	Get all nodes.
`get_port_map`(self, arg0, arg1)	

`get_previous_nodes`(self, arg0)	Get the nodes immediately upstream of a given node.
`get_root_nodes`(self)	Get all root nodes.
`is_leaf`(self, node)	Check if the node is a leaf node.
`is_root`(self, node)	Check if the node is a root node.

__init__(*self: holoscan.graphs._graphs.FragmentFlowGraph*) None

Directed graph class.

add_node(*self: holoscan.graphs._graphs.FragmentFlowGraph*, *node: holoscan::Fragment*) None

Add the node to the graph.

Parameters

node

The node to add.

property context

The graph's context (as an opaque PyCapsule object)

get_next_nodes(*self: holoscan.graphs._graphs.FragmentFlowGraph*, *arg0: holoscan::Fragment*) vector_of_node_type

Get the nodes immediately downstream of a given node.

Parameters

node

A node in the graph.

Returns

list of Operator or Fragment

A list containing the downstream nodes.

get_nodes(*self: holoscan.graphs._graphs.FragmentFlowGraph*) vector_of_node_type

Get all nodes.

The nodes are returned in the order they were added to the graph.

Returns

list of Operator or Fragment

A list containing all nodes.

get_port_map(*self: holoscan.graphs._graphs.FragmentFlowGraph*, *arg0: holoscan::Fragment*, *arg1: holoscan::Fragment*) dict

get_previous_nodes(*self: holoscan.graphs._graphs.FragmentFlowGraph*, *arg0: holoscan::Fragment*) vector_of_node_type

Get the nodes immediately upstream of a given node.

Parameters

node

A node in the graph.

Returns

list of Operator or Fragment

A list containing the upstream nodes.

get_root_nodes(*self: holoscan.graphs._graphs.FragmentFlowGraph*)
vector_of_node_type

Get all root nodes.

Returns

list of Operator or Fragment

A list containing all root nodes.

is_leaf(*self: holoscan.graphs._graphs.FragmentFlowGraph*, *node: holoscan::Fragment*)
bool

Check if the node is a leaf node.

Parameters

node

A node in the graph.

Returns

bool

Whether the node is a leaf node

is_root(*self: holoscan.graphs._graphs.FragmentFlowGraph*, *node: holoscan::Fragment*)
bool

Check if the node is a root node.

Parameters

node

A node in the graph.

Returns

bool

Whether the node is a root node

class holoscan.graphs.OperatorFlowGraph

Bases: holoscan.graphs._graphs.OperatorGraph

Directed graph class.

Attributes

context	The graph's context (as an opaque PyCapsule object)

Methods

add_node(self, node)	Add the node to the graph.
get_next_nodes(self, arg0)	Get the nodes immediately downstream of a given node.
get_nodes(self)	Get all nodes.
get_port_map(self, arg0, arg1)	

`get_previous_nodes`(self, arg0)	Get the nodes immediately upstream of a given node.
`get_root_nodes`(self)	Get all root nodes.
`is_leaf`(self, node)	Check if the node is a leaf node.
`is_root`(self, node)	Check if the node is a root node.

__init__(*self: holoscan.graphs._graphs.OperatorFlowGraph*) None

Directed graph class.

add_node(*self: holoscan.graphs._graphs.OperatorFlowGraph*, *node: holoscan::Operator*) None

Add the node to the graph.

Parameters

node

The node to add.

property context

The graph's context (as an opaque PyCapsule object)

get_next_nodes(*self: holoscan.graphs._graphs.OperatorFlowGraph*, *arg0: holoscan::Operator*) vector_of_node_type

Get the nodes immediately downstream of a given node.

Parameters

node

A node in the graph.

Returns

list of Operator or Fragment

A list containing the downstream nodes.

get_nodes(*self: holoscan.graphs._graphs.OperatorFlowGraph*) vector_of_node_type

Get all nodes.

The nodes are returned in the order they were added to the graph.

Returns

list of Operator or Fragment

A list containing all nodes.

get_port_map(*self: holoscan.graphs._graphs.OperatorFlowGraph*, *arg0: holoscan::Operator*, *arg1: holoscan::Operator*) dict

get_previous_nodes(*self: holoscan.graphs._graphs.OperatorFlowGraph*, *arg0: holoscan::Operator*) vector_of_node_type

Get the nodes immediately upstream of a given node.

Parameters

node

A node in the graph.

Returns

list of Operator or Fragment

A list containing the upstream nodes.

get_root_nodes(*self: holoscan.graphs._graphs.OperatorFlowGraph*)
vector_of_node_type

Get all root nodes.

Returns

list of Operator or Fragment

A list containing all root nodes.

is_leaf(*self: holoscan.graphs._graphs.OperatorFlowGraph*, *node: holoscan::Operator*)
bool

Check if the node is a leaf node.

Parameters

node

A node in the graph.

Returns

bool

Whether the node is a leaf node

is_root(*self: holoscan.graphs._graphs.OperatorFlowGraph*, *node: holoscan::Operator*)
bool

Check if the node is a root node.

Parameters

node

A node in the graph.

Returns

bool

Whether the node is a root node

holoscan.gxf

This module provides a Python API for GXF base classes in the C++ API.

holoscan.gxf.Entity	alias of holoscan.gxf._gxf.PyEntity
holoscan.gxf.GXFComponent	Base GXF-based component class.
holoscan.gxf.GXFCondition	Base GXF-based condition class.
holoscan.gxf.GXFExecutionContext	GXF execution context.

holoscan.gxf.GXFInputContext	GXF input context.
holoscan.gxf.GXFNetworkContext	Base GXF-based network context class.
holoscan.gxf.GXFOperator	Base GXF-based operator class.
holoscan.gxf.GXFOutputContext	GXF output context.
holoscan.gxf.GXFResource	Base GXF-based resource class.
holoscan.gxf.GXFScheduler	Base GXF-based scheduler class.

holoscan.gxf.Entity

alias of holoscan.gxf._gxf.PyEntity

class holoscan.gxf.GXFComponent

Bases: pybind11_builtins.pybind11_object

Base GXF-based component class.

Attributes

gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.

Methods

gxf_initialize (self)	Initialize the component.

__init__(*self: holoscan.gxf._gxf.GXFComponent*) None

Base GXF-based component class.

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

class holoscan.gxf.GXFCondition

Bases: holoscan.core._core.Condition, holoscan.gxf._gxf.GXFComponent

Base GXF-based condition class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the condition.
fragment	Fragment that the condition belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.

`gxf_eid`	The GXF entity ID.
`id`	The identifier of the component.
`name`	The name of the condition.

spec	

Methods

`add_arg`(*args, **kwargs)	Overloaded function.
`gxf_initialize`(self)	Initialize the component.
`initialize`(self)	Initialize the component.
`setup`(self, arg0)	setup method for the condition.

__init__(*self: holoscan.gxf._gxf.GXFCondition*) None

Base GXF-based condition class.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the condition.

property fragment

Fragment that the condition belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFCondition*) None

Initialize the component.

property name

The name of the condition.

Returns

name

setup(*self: holoscan.core._core.Condition*, *arg0: holoscan.core._core.ComponentSpec*) None

setup method for the condition.

property spec

class holoscan.gxf.GXFExecutionContext

Bases: `holoscan.core._core.ExecutionContext`

GXF execution context.

__init__(*self: holoscan.gxf._gxf.GXFExecutionContext*, *context: capsule*, *op: holoscan.gxf._gxf.GXFOperator*) None

Execution context for an operator using GXF.

Parameters

op

The GXF operator that owns this context.

class holoscan.gxf.GXFInputContext

Bases: holoscan.core._core.InputContext

GXF input context.

Methods

receive(self, name)	

__init__(*self: holoscan.gxf._gxf.GXFInputContext*, *context: holoscan.core._core.ExecutionContext*, *op: holoscan.gxf._gxf.GXFOperator*) None

GXF input context.

Parameters

op

The GXF operator that owns this context.

receive(*self: holoscan.core._core.InputContext*, *name: str*) None

class holoscan.gxf.GXFNetworkContext

Bases: holoscan.core._core.NetworkContext, holoscan.gxf._gxf.GXFComponent

Base GXF-based network context class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the component.
fragment	Fragment that the network context belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
id	The identifier of the component.
name	The name of the network context.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.

`gxf_initialize`(self)	Initialize the component.
`initialize`(self)	Initialize the network context.
`setup`(self, arg0)	setup method for the network context.

__init__(**args*, ***kwargs*)

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the component.

property fragment

Fragment that the network context belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFNetworkContext*) None

Initialize the network context.

property name

The name of the network context.

Returns

name

setup(*self: holoscan.core._core.NetworkContext*, *arg0: holoscan.core._core.ComponentSpec*) None

setup method for the network context.

property spec

class holoscan.gxf.GXFOperator

Bases: holoscan.core._core.Operator

Base GXF-based operator class.

Attributes

args	The list of arguments associated with the component.
conditions	Conditions associated with the operator.
description	YAML formatted string describing the operator.
fragment	The fragment (holoscan.core.Fragment) that the operator belongs to.
gxf_cid	The GXF component ID.
gxf_context	The GXF context of the component.

`gxf_eid`	The GXF entity ID.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`compute` (self, arg0, arg1, arg2)	Operator compute method.
`initialize` (self)	Operator initialization method.
`setup` (self, arg0)	Operator setup method.

start (self)	Operator start method.
stop (self)	Operator stop method.

OperatorType	

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = <*OperatorType.NATIVE: 0*>

VIRTUAL = <*OperatorType.VIRTUAL: 2*>

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.gxf._gxf.GXFOperator*) None

Base GXF-based operator class.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property gxf_cid

The GXF component ID.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.core._core.Operator*) None

Operator initialization method.

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.OperatorSpec*) None

Operator setup method.

property spec

The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.gxf.GXFOutputContext

Bases: holoscan.core._core.OutputContext

GXF output context.

Methods

emit(self, data[, name])	

OutputType	

class OutputType

Bases: pybind11_builtins.pybind11_object

Members:

SHARED_POINTER

GXF_ENTITY

Attributes

name	

value	

GXF_ENTITY = <*OutputType.GXF_ENTITY: 1*>

SHARED_POINTER = <*OutputType.SHARED_POINTER: 0*>

__init__(*self: holoscan.core._core.OutputContext.OutputType*, *value: int*) None

property name

property value

__init__(*self: holoscan.gxf._gxf.GXFOutputContext*, *context: holoscan.core._core.ExecutionContext*, *op: holoscan.gxf._gxf.GXFOperator*) None

GXF input context.

Parameters

op

The GXF operator that owns this context.

emit(*self: holoscan.core._core.OutputContext*, *data: object*, *name: str = ''*) None

class holoscan.gxf.GXFResource

Bases: holoscan.core._core.Resource, holoscan.gxf._gxf.GXFComponent

Base GXF-based resource class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.

gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the component.
setup(self, arg0)	setup method for the resource.

__init__(*self: holoscan.gxf._gxf.GXFResource*) None

Base GXF-based resource class.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.core._core.Resource*, *arg0: holoscan.core._core.ComponentSpec*) None

setup method for the resource.

property spec

class holoscan.gxf.GXFScheduler

Bases: holoscan.core._core.Scheduler, holoscan.gxf._gxf.GXFComponent

Base GXF-based scheduler class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the component.
fragment	Fragment that the scheduler belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
id	The identifier of the component.
name	The name of the scheduler.

clock	
spec	

Methods

`add_arg`(*args, **kwargs)	Overloaded function.
`gxf_initialize`(self)	Initialize the component.
`initialize`(self)	Initialize the scheduler.
`setup`(self, arg0)	setup method for the scheduler.

__init__(**args*, ***kwargs*)

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property clock

property description

YAML formatted string describing the component.

property fragment

Fragment that the scheduler belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFScheduler*) None

Initialize the scheduler.

property name

The name of the scheduler.

Returns

name

setup(*self: holoscan.core._core.Scheduler*, *arg0: holoscan.core._core.ComponentSpec*) None

setup method for the scheduler.

property spec

holoscan.gxf.load_extensions(*context: int*, *extension_filenames: List[str] = []*, *manifest_filenames: List[str] = []*) None

Loads GXF extension libraries

holoscan.logger

This module provides a Python interface to the Holoscan SDK logger.

holoscan.logger.LogLevel	Enum class for the logging level.

holoscan.logger.log_level()	Get the global logging level.
holoscan.logger.set_log_level(arg0)	Set the global logging level.
holoscan.logger.set_log_pattern(arg0)	Set the format pattern for the logger.

class holoscan.logger.LogLevel

Bases: pybind11_builtins.pybind11_object

Enum class for the logging level.

Members:

TRACE

DEBUG

INFO

WARN

ERROR

CRITICAL

OFF

Attributes

name	

value	

CRITICAL = *<LogLevel.CRITICAL: 5>*

DEBUG = *<LogLevel.DEBUG: 1>*

ERROR = *<LogLevel.ERROR: 4>*

INFO = *<LogLevel.INFO: 2>*

OFF = *<LogLevel.OFF: 6>*

TRACE = *<LogLevel.TRACE: 0>*

WARN = *<LogLevel.WARN: 3>*

__init__(*self: holoscan.logger._logger.LogLevel*, *value: int*) None

property name

property value

holoscan.logger.log_level() holoscan.logger._logger.LogLevel

Get the global logging level.

holoscan.logger.set_log_level(*arg0: holoscan.logger._logger.LogLevel*) None

Set the global logging level.

Parameters

level

The logging level to set

holoscan.logger.set_log_pattern(*arg0: str*) None

Set the format pattern for the logger.

Parameters

pattern

The pattern to use for logging messages. Uses the spdlog format specified at [1]. The default pattern used by spdlog is “[%Y-%m-%d %H:%M:%S.%e] [%l] [%n] %v”.

References

[1]

https://spdlog.docsforge.com/v1.x/3.custom-formatting/

holoscan.operators

This module provides a Python API to underlying C++ API Operators.

holoscan.operators.AJASourceOp	Operator to get a video stream from an AJA capture card.
holoscan.operators.BayerDemosaicOp	Bayer Demosaic operator.

holoscan.operators.FormatConverterOp	Format conversion operator.
holoscan.operators.HolovizOp(fragment[, ...])	Holoviz visualization operator using Holoviz module.
holoscan.operators.InferenceOp	Inference operator.
holoscan.operators.InferenceProcessorOp	Holoinfer Processing operator.
holoscan.operators.PingRxOp(fragm	Simple receiver operator.

ent, *args, ...)	
holoscan.operators.PingTxOp(fragment, *args, ...)	Simple transmitter operator.
holoscan.operators.SegmentationPostprocessorOp	Operator carrying out post-processing operations on segmentation outputs.
holoscan.operators.V4L2VideoCaptureOp	Operator to get a video stream from a V4L2 source.
holoscan.operators.VideoStreamRecorderOp	Operator class to record a video stream to a file.

holoscan.operators.VideoStreamReplayerOp	Operator class to replay a video stream from a file.

class holoscan.operators.AJASourceOp

Bases: holoscan.core._core.Operator

Operator to get a video stream from an AJA capture card.

==Named Inputs==

overlay_buffer_inputnvidia::gxf::VideoBuffer (optional)

The operator does not require a message on this input port in order for `compute` to be called. If a message is found, and `enable_overlay` is `True`, the image will be mixed with the image captured by the AJA card. If `enable_overlay` is `False`, any message on this port will be ignored.

==Named Outputs==

video_buffer_outputnvidia::gxf::VideoBuffer

The output video frame from the AJA capture card. If `overlay_rdma` is `True`, this video buffer will be on the device, otherwise it will be in pinned host memory.

overlay_buffer_outputnvidia::gxf::VideoBuffer (optional)

This output port will only emit a video buffer when `enable_overlay` is `True`. If `overlay_rdma` is `True`, this video buffer will be on the device, otherwise it will be in pinned host memory.

Parameters

fragment

The fragment that the operator belongs to.

device

The device to target (e.g., “0” for device 0). Default value is `"0"`.

channel

The camera `NTV2Channel` to use for output (e.g., `NTV2Channel.NTV2_CHANNEL1` (`0`) or “NTV2_CHANNEL1” (in YAML) for the first channel). Default value is `NTV2Channel.NTV2_CHANNEL1` (`"NTV2_CHANNEL1"` in YAML).

width

Width of the video stream. Default value is `1920`.

height

Height of the video stream. Default value is `1080`.

framerate

Frame rate of the video stream. Default value is `60`.

rdma

Boolean indicating whether RDMA is enabled. Default value is `False` (`"false"` in YAML).

enable_overlay

Boolean indicating whether a separate overlay channel is enabled. Default value is `False` (`"false"` in YAML).

overlay_channel

The camera NTV2Channel to use for overlay output. Default value is `NTV2Channel.NTV2_CHANNEL2` (`"NTV2_CHANNEL2"` in YAML).

overlay_rdma

Boolean indicating whether RDMA is enabled for the overlay. Default value is `False` (`"false"` in YAML).

name

The name of the operator. Default value is `"aja_source"`.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

add_arg(*args, **kwargs)	Overloaded function.
compute(self, arg0, arg1, arg2)	Operator compute method.
initialize(self)	Initialize the operator.
setup(self, spec)	Define the operator specification.
start(self)	Operator start method.
stop(self)	Operator stop method.

OperatorType	

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.

- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.operators.aja_source._aja_source.AJASourceOp*, *fragment: holoscan.core._core.Fragment*, *device: str = '0'*, *channel: holoscan.operators.aja_source._aja_source.NTV2Channel = <NTV2Channel.NTV2_CHANNEL1: 0>*, *width: int = 1920*, *height: int = 1080*, *framerate: int = 60*, *rdma: bool = False*, *enable_overlay: bool = False*, *overlay_channel: holoscan.operators.aja_source._aja_source.NTV2Channel = <NTV2Channel.NTV2_CHANNEL2: 1>*, *overlay_rdma: bool = True*, *name: str = 'aja_source'*) None

Operator to get a video stream from an AJA capture card.

==Named Inputs==

overlay_buffer_inputnvidia::gxf::VideoBuffer (optional)

The operator does not require a message on this input port in order for `compute` to be called. If a message is found, and `enable_overlay` is `True`, the image will be mixed with the image captured by the AJA card. If `enable_overlay` is `False`, any message on this port will be ignored.

==Named Outputs==

video_buffer_outputnvidia::gxf::VideoBuffer

The output video frame from the AJA capture card. If `overlay_rdma` is `True`, this video buffer will be on the device, otherwise it will be in pinned host memory.

overlay_buffer_outputnvidia::gxf::VideoBuffer (optional)

This output port will only emit a video buffer when `enable_overlay` is `True`. If `overlay_rdma` is `True`, this video buffer will be on the device, otherwise it will be in pinned host memory.

Parameters

fragment

The fragment that the operator belongs to.

device

The device to target (e.g., “0” for device 0). Default value is `"0"`.

channel

The camera `NTV2Channel` to use for output (e.g., `NTV2Channel.NTV2_CHANNEL1` (`0`) or “NTV2_CHANNEL1” (in YAML) for

the first channel). Default value is `NTV2Channel.NTV2_CHANNEL1` (`"NTV2_CHANNEL1"` in YAML).

width

Width of the video stream. Default value is `1920`.

height

Height of the video stream. Default value is `1080`.

framerate

Frame rate of the video stream. Default value is `60`.

rdma

Boolean indicating whether RDMA is enabled. Default value is `False` (`"false"` in YAML).

enable_overlay

Boolean indicating whether a separate overlay channel is enabled. Default value is `False` (`"false"` in YAML).

overlay_channel

The camera NTV2Channel to use for overlay output. Default value is `NTV2Channel.NTV2_CHANNEL2` (`"NTV2_CHANNEL2"` in YAML).

overlay_rdma

Boolean indicating whether RDMA is enabled for the overlay. Default value is `False` (`"false"` in YAML).

name

The name of the operator. Default value is `"aja_source"`.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*)

None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1` , and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.operators.aja_source._aja_source.AJASourceOp*) None

Initialize the operator.

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

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.operators.aja_source._aja_source.AJASourceOp*, *spec: holoscan.core._core.OperatorSpec*) None

Define the operator specification.

Parameters

spec

The operator specification.

property spec

The operator spec (holoscan.core.OperatorSpec) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.operators.BayerDemosaicOp

Bases: holoscan.core._core.Operator

Bayer Demosaic operator.

==Named Inputs==

receivernvidia::gxf::Tensor or nvidia::gxf::VideoBuffer

The input video frame to process. If the input is a VideoBuffer it must be an 8-bit unsigned grayscale video (nvidia::gxf::VideoFormat::GXF_VIDEO_FORMAT_GRAY). The video buffer may be in either host or device memory (a host->device copy is performed if needed). If a video buffer is not found, the input port message is searched for a tensor with the name specified by `in_tensor_name`. This must be a device tensor in either 8-bit or 16-bit unsigned integer format.

==Named Outputs==

transmitternvidia::gxf::Tensor

The output video frame after demosaicing. This will be a 3-channel RGB image if `alpha_value` is `True`, otherwise it will be a 4-channel RGBA image. The data type will be either 8-bit or 16-bit unsigned integer (matching the bit depth of the input). The name of the tensor that is output is controlled by `out_tensor_name`.

Parameters

fragment

The fragment that the operator belongs to.

pool

Memory pool allocator used by the operator.

cuda_stream_pool

`holoscan.resources.CudaStreamPool` instance to allocate CUDA streams. Default value is `None`.

in_tensor_name

The name of the input tensor. Default value is `""` (empty string).

out_tensor_name

The name of the output tensor. Default value is `""` (empty string).

interpolation_mode

The interpolation model to be used for demosaicing. Values available at: https://docs.nvidia.com/cuda/npp/nppdefs.html?highlight=Two%20parameter%20cubic%20filter#c.NppiInterpolationMode

- NPPI_INTER_UNDEFINED (`0`): Undefined filtering interpolation mode.
- NPPI_INTER_NN (`1`): Nearest neighbor filtering.
- NPPI_INTER_LINEAR (`2`): Linear interpolation.
- NPPI_INTER_CUBIC (`4`): Cubic interpolation.
- NPPI_INTER_CUBIC2P_BSPLINE (`5`): Two-parameter cubic filter (B=1, C=0)
- NPPI_INTER_CUBIC2P_CATMULLROM (`6`): Two-parameter cubic filter (B=0, C=1/2)
- NPPI_INTER_CUBIC2P_B05C03 (`7`): Two-parameter cubic filter (B=1/2, C=3/10)
- NPPI_INTER_SUPER (`8`): Super sampling.
- NPPI_INTER_LANCZOS (`16`): Lanczos filtering.
- NPPI_INTER_LANCZOS3_ADVANCED (`17`): Generic Lanczos filtering with order 3.
- NPPI_SMOOTH_EDGE (`0x8000000`): Smooth edge filtering.

Default value is `0` (NPPI_INTER_UNDEFINED).

bayer_grid_pos

The Bayer grid position. Values available at: https://docs.nvidia.com/cuda/npp/nppdefs.html?highlight=Two%20parameter%20cubic%20filter#c.NppiBayerGridPosition

- NPPI_BAYER_BGGR (`0`): Default registration position BGGR.
- NPPI_BAYER_RGGB (`1`): Registration position RGGB.
- NPPI_BAYER_GBRG (`2`): Registration position GBRG.
- NPPI_BAYER_GRBG (`3`): Registration position GRBG.

Default value is `2` (NPPI_BAYER_GBRG).

generate_alpha

Generate alpha channel. Default value is `False`.

alpha_value

Alpha value to be generated if `generate_alpha` is set to `True`. Default value is `255`.

name

The name of the operator. Default value is `"bayer_demosaic"`.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.

`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`compute` (self, arg0, arg1, arg2)	Operator compute method.
`initialize` (self)	Initialize the operator.
`setup` (self, spec)	Define the operator specification.
`start` (self)	Operator start method.
`stop` (self)	Operator stop method.

OperatorType	

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1` , and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.operators.bayer_demosaic._bayer_demosaic.BayerDemosaicOp*) None

Initialize the operator.

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

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.operators.bayer_demosaic._bayer_demosaic.BayerDemosaicOp*, *spec: holoscan.core._core.OperatorSpec*) None

Define the operator specification.

Parameters

spec

The operator specification.

property spec

The operator spec (holoscan.core.OperatorSpec) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.operators.FormatConverterOp

Bases: holoscan.core._core.Operator

Format conversion operator.

==Named Inputs==

source_videonvidia::gxf::Tensor or nvidia::gxf::VideoBuffer

The input video frame to process. If the input is a VideoBuffer it must be in format GXF_VIDEO_FORMAT_RGBA, GXF_VIDEO_FORMAT_RGB or GXF_VIDEO_FORMAT_NV12. This video buffer may be in either host or device memory (a host->device copy is performed if needed). If a video buffer is not found, the input port message is searched for a tensor with the name specified by `in_tensor_name`. This must be a device tensor in one of several supported formats (unsigned 8-bit int or float32 graycale, unsigned 8-bit int RGB or RGBA, YUV420 or NV12).

==Named Outputs==

tensornvidia::gxf::Tensor

The output video frame after processing. The shape, data type and number of channels of this output tensor will depend on the specific parameters that were set for this operator. The name of the Tensor transmitted on this port is determined by `out_tensor_name`.

Parameters

fragment

The fragment that the operator belongs to.

pool

Memory pool allocator used by the operator.

out_dtype

Destination data type. The available options are:

- `"rgb888"`
- `"uint8"`

- `"float32"`
- `"rgba8888"`
- `"yuv420"`
- `"nv12"`

in_dtype

Source data type. The available options are:

- `"rgb888"`
- `"uint8"`
- `"float32"`
- `"rgba8888"`
- `"yuv420"`
- `"nv12"`

in_tensor_name

The name of the input tensor. Default value is `""` (empty string).

out_tensor_name

The name of the output tensor. Default value is `""` (empty string).

scale_min

Output will be clipped to this minimum value. Default value is `0.0`.

scale_max

Output will be clipped to this maximum value. Default value is `1.0`.

alpha_value

Unsigned integer in range [0, 255], indicating the alpha channel value to use when converting from RGB to RGBA. Default value is `255`.

resize_height

Desired height for the (resized) output. Height will be unchanged if `resize_height` is `0`. Default value is `0`.

resize_width

Desired width for the (resized) output. Width will be unchanged if `resize_width` is `0`. Default value is `0`.

resize_mode

Resize mode enum value corresponding to NPP's NppiInterpolationMode. Values available at: https://docs.nvidia.com/cuda/npp/nppdefs.html?highlight=Two%20parameter%20cubic%20filter#c.NppiInterpolationMode

- NPPI_INTER_UNDEFINED (`0`): Undefined filtering interpolation mode.
- NPPI_INTER_NN (`1`): Nearest neighbor filtering.
- NPPI_INTER_LINEAR (`2`): Linear interpolation.
- NPPI_INTER_CUBIC (`4`): Cubic interpolation.
- NPPI_INTER_CUBIC2P_BSPLINE (`5`): Two-parameter cubic filter (B=1, C=0)
- NPPI_INTER_CUBIC2P_CATMULLROM (`6`): Two-parameter cubic filter (B=0, C=1/2)
- NPPI_INTER_CUBIC2P_B05C03 (`7`): Two-parameter cubic filter (B=1/2, C=3/10)
- NPPI_INTER_SUPER (`8`): Super sampling.
- NPPI_INTER_LANCZOS (`16`): Lanczos filtering.

- NPPI_INTER_LANCZOS3_ADVANCED (`17`): Generic Lanczos filtering with order 3.
- NPPI_SMOOTH_EDGE (`0x8000000`): Smooth edge filtering.

Default value is `0` (NPPI_INTER_UNDEFINED) which would be equivalent to `4` (NPPI_INTER_CUBIC).

channel_order

Sequence of integers describing how channel values are permuted. Default value is `[0, 1, 2]` for 3-channel images and `[0, 1, 2, 3]` for 4-channel images.

cuda_stream_pool

holoscan.resources.CudaStreamPool instance to allocate CUDA streams. Default value is `None`.

name

The name of the operator. Default value is `"format_converter"`.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.

`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`compute` (self, arg0, arg1, arg2)	Operator compute method.
`initialize` (self)	Initialize the operator.
`setup` (self, spec)	Define the operator specification.
`start` (self)	Operator start method.
`stop` (self)	Operator stop method.

OperatorType	

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.operators.format_converter._format_converter.FormatConverterOp*, *fragment: holoscan.core._core.Fragment*, *pool: holoscan.resources._resources.Allocator*, *out_dtype: str*, *in_dtype: str = ''*, *in_tensor_name: str = ''*, *out_tensor_name: str = ''*, *scale_min: float = 0.0*, *scale_max: float = 1.0*, *alpha_value: int = 255*, *resize_height: int = 0*, *resize_width: int = 0*, *resize_mode: int = 0*, *out_channel_order: List[int] = []*, *cuda_stream_pool: holoscan.resources._resources.CudaStreamPool = None*, *name: str = 'format_converter'*)

None

Format conversion operator.

==Named Inputs==

source_videonvidia::gxf::Tensor or nvidia::gxf::VideoBuffer

The input video frame to process. If the input is a VideoBuffer it must be in format GXF_VIDEO_FORMAT_RGBA, GXF_VIDEO_FORMAT_RGB or GXF_VIDEO_FORMAT_NV12. This video buffer may be in either host or device memory (a host->device copy is performed if needed). If a video buffer is not found, the input port message is searched for a tensor with the name specified by `in_tensor_name`. This must be a device tensor in one of several supported formats (unsigned 8-bit int or float32 graycale, unsigned 8-bit int RGB or RGBA, YUV420 or NV12).

==Named Outputs==

tensornvidia::gxf::Tensor

The output video frame after processing. The shape, data type and number of channels of this output tensor will depend on the specific parameters that were set for this operator. The name of the Tensor transmitted on this port is determined by `out_tensor_name`.

Parameters

fragment

The fragment that the operator belongs to.

pool

Memory pool allocator used by the operator.

out_dtype

Destination data type. The available options are:

- `"rgb888"`
- `"uint8"`
- `"float32"`
- `"rgba8888"`
- `"yuv420"`
- `"nv12"`

in_dtype

Source data type. The available options are:

- `"rgb888"`
- `"uint8"`
- `"float32"`
- `"rgba8888"`
- `"yuv420"`
- `"nv12"`

in_tensor_name

The name of the input tensor. Default value is `""` (empty string).

out_tensor_name

The name of the output tensor. Default value is `""` (empty string).

scale_min

Output will be clipped to this minimum value. Default value is `0.0`.

scale_max

Output will be clipped to this maximum value. Default value is `1.0`.

alpha_value

Unsigned integer in range [0, 255], indicating the alpha channel value to use when converting from RGB to RGBA. Default value is `255`.

resize_height

Desired height for the (resized) output. Height will be unchanged if `resize_height` is `0`. Default value is `0`.

resize_width

Desired width for the (resized) output. Width will be unchanged if `resize_width` is `0`. Default value is `0`.

resize_mode

Resize mode enum value corresponding to NPP's NppiInterpolationMode. Values available at: https://docs.nvidia.com/cuda/npp/nppdefs.html?highlight=Two%20parameter%20cubic%20filter#c.NppiInterpolationMode

- NPPI_INTER_UNDEFINED (`0`): Undefined filtering interpolation mode.
- NPPI_INTER_NN (`1`): Nearest neighbor filtering.
- NPPI_INTER_LINEAR (`2`): Linear interpolation.

- NPPI_INTER_CUBIC (`4`): Cubic interpolation.
- NPPI_INTER_CUBIC2P_BSPLINE (`5`): Two-parameter cubic filter (B=1, C=0)
- NPPI_INTER_CUBIC2P_CATMULLROM (`6`): Two-parameter cubic filter (B=0, C=1/2)
- NPPI_INTER_CUBIC2P_B05C03 (`7`): Two-parameter cubic filter (B=1/2, C=3/10)
- NPPI_INTER_SUPER (`8`): Super sampling.
- NPPI_INTER_LANCZOS (`16`): Lanczos filtering.
- NPPI_INTER_LANCZOS3_ADVANCED (`17`): Generic Lanczos filtering with order 3.
- NPPI_SMOOTH_EDGE (`0x8000000`): Smooth edge filtering.

Default value is `0` (NPPI_INTER_UNDEFINED) which would be equivalent to `4` (NPPI_INTER_CUBIC).

channel_order

Sequence of integers describing how channel values are permuted. Default value is `[0, 1, 2]` for 3-channel images and `[0, 1, 2, 3]` for 4-channel images.

cuda_stream_pool

holoscan.resources.CudaStreamPool instance to allocate CUDA streams. Default value is `None`.

name

The name of the operator. Default value is `"format_converter"`.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1` , and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.operators.format_converter._format_converter.FormatConverterOp*) None

Initialize the operator.

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

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.operators.format_converter._format_converter.FormatConverterOp*, *spec: holoscan.core._core.OperatorSpec*) None

Define the operator specification.

Parameters

spec

The operator specification.

property spec

The operator spec (holoscan.core.OperatorSpec) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.operators.HolovizOp(*fragment*, *allocator=None*, *receivers=()*, *tensors=()*, *color_lut=()*, *window_title='Holoviz'*, *display_name='DP-0'*, *width=1920*, *height=1080*, *framerate=60*, *use_exclusive_display=False*, *fullscreen=False*, *headless=False*, *enable_render_buffer_input=False*, *enable_render_buffer_output=False*, *enable_camera_pose_output=False*, *font_path=''*, *cuda_stream_pool=None*, *name='holoviz_op'*)

Bases: holoscan.operators.holoviz._holoviz.HolovizOp

Holoviz visualization operator using Holoviz module.

This is a Vulkan-based visualizer.

==Named Inputs==

receiversmulti-receiver accepting nvidia::gxf::Tensor and/or nvidia::gxf::VideoBuffer

Any number of upstream ports may be connected to this `receivers` port. This port can accept either VideoBuffers or Tensors. These inputs can be in either host or device memory. Each tensor or video buffer will result in a layer. The operator autodetects the layer type for certain input types (e.g. a video buffer will result in an image layer). For other input types or more complex use cases, input specifications can be provided either at initialization time as a parameter or dynamically at run time (via `input_specs`). On each call to `compute`, tensors corresponding to all names specified in the `tensors` parameter must be found or an exception will be raised. Any extra, named tensors not present in the `tensors` parameter specification (or optional, dynamic `input_specs` input) will be ignored.

input_specslist[holoscan.operators.HolovizOp.InputSpec] (optional)

A list of `InputSpec` objects. This port can be used to dynamically update the overlay specification at run time. No inputs are required on this port in order for the operator to `compute`.

render_buffer_inputnvidia::gxf::VideoBuffer (optional)

An empty render buffer can optionally be provided. The video buffer must have format GXF_VIDEO_FORMAT_RGBA and be in device memory. This input port only exists if `enable_render_buffer_input` was set to `True`, in which case `compute` will only be called when a message arrives on this input.

==Named Outputs==

render_buffer_outputnvidia::gxf::VideoBuffer (optional)

Output for a filled render buffer. If an input render buffer is specified, it is using that one, else it allocates a new buffer. The video buffer will have format GXF_VIDEO_FORMAT_RGBA and will be in device memory. This output is useful for offline rendering or headless mode. This output port only exists if `enable_render_buffer_output` was set to `True`.

camera_pose_outputstd::array<float, 16> (optional)

The camera pose. The parameters returned represent the values of a 4x4 row major projection matrix. This output port only exists if `enable_camera_pose_output` was set to `True`.

Parameters

fragment

The fragment that the operator belongs to.

allocator

Allocator used to allocate render buffer output. If `None`, will default to `holoscan.core.UnboundedAllocator`.

receivers

List of input receivers.

tensors

List of input tensors. Each tensor is defined by a dictionary where the `"name"` key must correspond to a tensor sent to the operator's input. See the notes section below for further details on how the tensor dictionary is defined.

color_lut

Color lookup table for tensors of type `color_lut`. Should be shape `(n_colors, 4)`.

window_title

Title on window canvas. Default value is `"Holoviz"`.

display_name

In exclusive mode, name of display to use as shown with xrandr. Default value is `"DP-0"`.

width

Window width or display resolution width if in exclusive or fullscreen mode. Default value is `1920`.

height

Window height or display resolution width if in exclusive or fullscreen mode. Default value is `1080`.

framerate

Display framerate in Hz if in exclusive mode. Default value is `60.0`.

use_exclusive_display

Enable exclusive display. Default value is `False`.

fullscreen

Enable fullscreen window. Default value is `False`.

headless

Enable headless mode. No window is opened, the render buffer is output to port `render_buffer_output`. Default value is `False`.

enable_render_buffer_input

If `True`, an additional input port, named `"render_buffer_input"` is added to the operator. Default value is `False`.

enable_render_buffer_output

If `True`, an additional output port, named `"render_buffer_output"` is added to the operator. Default value is `False`.

enable_camera_pose_output

If `True`, an additional output port, named `"camera_pose_output"` is added to the operator. Default value is `False`.

font_path

File path for the font used for rendering text. Default value is `""`.

cuda_stream_pool

`holoscan.resources.CudaStreamPool` instance to allocate CUDA streams. Default value is `None`.

name

The name of the operator. Default value is `"holoviz_op"`.

Notes

The `tensors` argument is used to specify the tensors to display. Each tensor is defined using a dictionary, that must, at minimum include a 'name' key that corresponds to a tensor found on the operator's input. A 'type' key should also be provided to indicate the type of entry to display. The 'type' key will be one of { `"color"`, `"color_lut"`, `"crosses"`, `"lines"`, `"lines_3d"`, `"line_strip"`, `"line_strip_3d"`, `"ovals"`, `"points"`, `"points_3d"`, `"rectangles"`, `"text"`, `"triangles"`, `"triangles_3d"`, `"depth_map"`, `"depth_map_color"`, `"unknown"` }. The default type is `"unknown"` which will attempt to guess the corresponding type based on the tensor dimensions. Concrete examples are given below.

To show a single 2D RGB or RGBA image, use a list containing a single tensor of type `"color"`.

```
tensors = [dict(name="video", type="color", opacity=1.0, priority=0)]
```

Here, the optional key `opacity` is used to scale the opacity of the tensor. The `priority` key is used to specify the render priority for layers. Layers with a higher priority will be rendered on top of those with a lower priority.

If we also had a `"boxes"` tensor representing rectangular bounding boxes, we could display them on top of the image like this.

```
tensors = [ dict(name="video", type="color", priority=0), dict(name="boxes", type="rectangles", color=[1.0, 0.0, 0.0], line_width=2, priority=1), ]
```

where the `color` and `line_width` keys specify the color and line width of the bounding box.

The details of the dictionary is as follows:

- **name**: name of the tensor containing the input data to display
  - type: `str`
- **type**: input type (default `"unknown"`)
  - type: `str`
  - possible values:
    - **unknown**: unknown type, the operator tries to guess the type by inspecting the tensor.
    - **color**: RGB or RGBA color 2d image.
    - **color_lut**: single channel 2d image, color is looked up.
    - **points**: point primitives, one coordinate (x, y) per primitive.
    - **lines**: line primitives, two coordinates (x0, y0) and (x1, y1) per primitive.
    - **line_strip**: line strip primitive, a line primitive i is defined by each coordinate (xi, yi) and the following (xi+1, yi+1).
    - **triangles**: triangle primitive, three coordinates (x0, y0), (x1, y1) and (x2, y2) per primitive.

        - **crosses**: cross primitive, a cross is defined by the center coordinate and the size (xi, yi, si).
        - **rectangles**: axis aligned rectangle primitive, each rectangle is defined by two coordinates (xi, yi) and (xi+1, yi+1).
        - **ovals**: oval primitive, an oval primitive is defined by the center coordinate and the axis sizes (xi, yi, sxi, syi).
        - **text**: text is defined by the top left coordinate and the size (x, y, s) per string, text strings are defined by InputSpec member **text**.
        - **depth_map**: single channel 2d array where each element represents a depth value. The data is rendered as a 3d object using points, lines or triangles. The color for the elements can be specified through `depth_map_color`. Supported format: 8-bit unsigned normalized format that has a single 8-bit depth component.
        - **depth_map_color**: RGBA 2d image, same size as the depth map. One color value for each element of the depth map grid. Supported format: 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.
- **opacity**: layer opacity, 1.0 is fully opaque, 0.0 is fully transparent (default: `1.0`)
    - type: `float`
- **priority**: layer priority, determines the render order, layers with higher priority values are rendered on top of layers with lower priority values (default: `0`)
    - type: `int`
- **color**: RGBA color of rendered geometry (default: `[1.f, 1.f, 1.f, 1.f]`)
    - type: `List[float]`
- **line_width**: line width for geometry made of lines (default: `1.0`)

    - type: `float`

- **point_size**: point size for geometry made of points (default: `1.0`)
    - type: `float`
- **text**: array of text strings, used when `type` is text (default: `[]`)
    - type: `List[str]`
- **depth_map_render_mode**: depth map render mode (default: `points`)
    - type: `str`
    - possible values:
        - **points**: render as points
        - **lines**: render as lines
        - **triangles**: render as triangles

1. Displaying Color Images

   Image data can either be on host or device (GPU). Multiple image formats are supported

    - R 8 bit unsigned
    - R 16 bit unsigned
    - R 16 bit float
    - R 32 bit unsigned
    - R 32 bit float
    - RGB 8 bit unsigned
    - BGR 8 bit unsigned

- RGBA 8 bit unsigned
- BGRA 8 bit unsigned
- RGBA 16 bit unsigned
- RGBA 16 bit float
- RGBA 32 bit float

When the `type` parameter is set to `color_lut` the final color is looked up using the values from the `color_lut` parameter. For color lookups these image formats are supported

- R 8 bit unsigned
- R 16 bit unsigned
- R 32 bit unsigned

2. Drawing Geometry

In all cases, `x` and `y` are normalized coordinates in the range `[0, 1]`. The `x` and `y` correspond to the horizontal and vertical axes of the display, respectively. The origin `(0, 0)` is at the top left of the display. Geometric primitives outside of the visible area are clipped. Coordinate arrays are expected to have the shape `(N, C)` where `N` is the coordinate count and `C` is the component count for each coordinate.

- Points are defined by a `(x, y)` coordinate pair.
- Lines are defined by a set of two `(x, y)` coordinate pairs.
- Lines strips are defined by a sequence of `(x, y)` coordinate pairs. The first two coordinates define the first line, each additional coordinate adds a line connecting to the previous coordinate.
- Triangles are defined by a set of three `(x, y)` coordinate pairs.

- Crosses are defined by `(x, y, size)` tuples. `size` specifies the size of the cross in the `x` direction and is optional, if omitted it's set to `0.05`. The size in the `y` direction is calculated using the aspect ratio of the window to make the crosses square.
- Rectangles (bounding boxes) are defined by a pair of 2-tuples defining the upper-left and lower-right coordinates of a box: `(x1, y1), (x2, y2)`.
- Ovals are defined by `(x, y, size_x, size_y)` tuples. `size_x` and `size_y` are optional, if omitted they are set to `0.05`.
- Texts are defined by `(x, y, size)` tuples. `size` specifies the size of the text in `y` direction and is optional, if omitted it's set to `0.05`. The size in the `x` direction is calculated using the aspect ratio of the window. The index of each coordinate references a text string from the `text` parameter and the index is clamped to the size of the text array. For example, if there is one item set for the `text` parameter, e.g. `text=["my_text"]` and three coordinates, then `my_text` is rendered three times. If `text=["first text", "second text"]` and three coordinates are specified, then `first text` is rendered at the first coordinate, `second text` at the second coordinate and then `second text` again at the third coordinate. The `text` string array is fixed and can't be changed after initialization. To hide text which should not be displayed, specify coordinates greater than `(1.0, 1.0)` for the text item, the text is then clipped away.
- 3D Points are defined by a `(x, y, z)` coordinate tuple.
- 3D Lines are defined by a set of two `(x, y, z)` coordinate tuples.
- 3D Lines strips are defined by a sequence of `(x, y, z)` coordinate tuples. The first two coordinates define the first line, each additional coordinate adds a line connecting to the previous coordinate.
- 3D Triangles are defined by a set of three `(x, y, z)` coordinate tuples.

3. Displaying Depth Maps

When `type` is `depth_map` the provided data is interpreted as a rectangular array of depth values. Additionally a 2d array with a color value for each point in the grid can be specified by setting `type` to `depth_map_color`.

The type of geometry drawn can be selected by setting `depth_map_render_mode`.

Depth maps are rendered in 3D and support camera movement. The camera is controlled 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)

4. Output

   By default a window is opened to display the rendering, but the extension can also be run in headless mode with the `headless` parameter.

   Using a display in exclusive mode is also supported with the `use_exclusive_display` parameter. This reduces the latency by avoiding the desktop compositor.

   The rendered framebuffer can be output to `render_buffer_output`.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.

fragment	The fragment (holoscan.core.Fragment) that the operator belongs to.
id	The identifier of the component.
name	The name of the operator.
operator_type	The operator type.
resources	Resources associated with the operator.
spec	The operator spec (holoscan.core.OperatorSpec) associated with the operator.

Methods

InputSpec	InputSpec for the HolovizOp operator.
add_arg (*args, **kwargs)	Overloaded function.
compute (self, arg0, arg1, arg2)	Operator compute method.
initialize (self)	Initialize the operator.
setup (self, s	Define the operator specification.

pec)	
start (self)	Operator start method.
stop (self)	Operator stop method.

DepthMapRenderMode	
InputType	
OperatorType	

class DepthMapRenderMode

Bases: pybind11_builtins.pybind11_object

Members:

POINTS

LINES

TRIANGLES

Attributes

name	

value	

LINES = *<DepthMapRenderMode.LINES: 1>*

POINTS = *<DepthMapRenderMode.POINTS: 0>*

TRIANGLES = *<DepthMapRenderMode.TRIANGLES: 2>*

__init__(*self: holoscan.operators.holoviz._holoviz.HolovizOp.DepthMapRenderMode*, *value: int*)
None

property name

property value

class InputSpec

Bases: pybind11_builtins.pybind11_object

InputSpec for the HolovizOp operator.

Parameters

tensor_name

The tensor name for this input.

type

The type of data that this tensor represents.

Attributes

type	(holoscan.operators.HolovizOp.InputType) The type of data that this tensor represents.
opacity	(float) The opacity of the object. Must be in range [0.0, 1.0] where 1.0 is fully opaque.
priority	(int) Layer priority, determines the render order. Layers with higher priority values are rendered on top of layers with lower priority.
color	(4-tuple of float) RGBA values in range [0.0, 1.0] for rendered geometry.
line_width	(float) Line width for geometry made of lines.
point_size	(float) Point size for geometry made of points.
text	(sequence of str) Sequence of strings used when type is *HolovizOp.InputType.TEXT*.

depth_map_render_mode	(holoscan.operators.HolovizOp.DepthMapRenderMode) The depth map render mode. Used only if *type* is *HolovizOp.InputType.DEPTH_MAP* or *HolovizOp.InputType.DEPTH_MAP_COLOR*.
views	(list of HolovizOp.InputSpec.View) Sequence of layer views. By default a layer will fill the whole window. When using a view, the layer can be placed freely within the window. When multiple views are specified, the layer is drawn multiple times using the specified layer views.

Methods

View	View for the InputSpec of a HolovizOp operator.
description (self)	Returns

class View

Bases: pybind11_builtins.pybind11_object

View for the InputSpec of a HolovizOp operator.

Notes

Layers can also be placed in 3D space by specifying a 3D transformation *matrix*. Note that for geometry layers there is a default matrix which allows coordinates in the range of [0 ... 1] instead of the Vulkan [-1 ... 1] range. When specifying a matrix for a geometry layer, this default matrix is overwritten.

When multiple views are specified, the layer is drawn multiple times using the specified layer views.

It's possible to specify a negative term for height, which flips the image. When using a negative height, one should also adjust the y value to point

to the lower left corner of the viewport instead of the upper left corner.

Attributes

offset_x, offset_y	(float) Offset of top-left corner of the view. (0, 0) is the upper left and (1, 1) is the lower right.
width	(float) Normalized width (range [0.0, 1.0]).
height	(float) Normalized height (range [0.0, 1.0]).
matrix	(sequence of float) 16-elements representing a 4x4 transformation matrix.

__init__(*self: holoscan.operators.holoviz._holoviz.HolovizOp.InputSpec.View*)
None

View for the InputSpec of a HolovizOp operator.

Notes

Layers can also be placed in 3D space by specifying a 3D transformation *matrix*. Note that for geometry layers there is a default matrix which allows coordinates in the range of [0 ... 1] instead of the Vulkan [-1 ... 1] range. When specifying a matrix for a geometry layer, this default matrix is overwritten.

When multiple views are specified, the layer is drawn multiple times using the specified layer views.

It's possible to specify a negative term for height, which flips the image. When using a negative height, one should also adjust the y value to point to the lower left corner of the viewport instead of the upper left corner.

Attributes

offset_x, offset_y	(float) Offset of top-left corner of the view. (0, 0) is the upper left and (1, 1) is the lower right.
width	(float) Normalized width (range [0.0, 1.0]).
height	(float) Normalized height (range [0.0, 1.0]).

matrix	(sequence of float) 16-elements representing a 4x4 transformation matrix.

property height

property matrix

property offset_x

property offset_y

property width

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: holoscan.operators.holoviz._holoviz.HolovizOp.InputSpec, arg0: str, arg1: holoscan.operators.holoviz._holoviz.HolovizOp.InputType) -> None

InputSpec for the HolovizOp operator.

Parameters

tensor_name

The tensor name for this input.

type

The type of data that this tensor represents.

Attributes

type	(holoscan.operators.HolovizOp.InputType) The type of data that this tensor represents.
opacity	(float) The opacity of the object. Must be in range [0.0, 1.0] where 1.0 is fully opaque.
priority	(int) Layer priority, determines the render order. Layers with higher priority values are rendered on

	top of layers with lower priority.
color	(4-tuple of float) RGBA values in range [0.0, 1.0] for rendered geometry.
line_width	(float) Line width for geometry made of lines.
point_size	(float) Point size for geometry made of points.
text	(sequence of str) Sequence of strings used when type is *HolovizOp.InputType.TEXT*.
depth_map_render_mode	(holoscan.operators.HolovizOp.DepthMapRenderMode) The depth map render mode. Used only if *type* is *HolovizOp.InputType.DEPTH_MAP* or *HolovizOp.InputType.DEPTH_MAP_COLOR*.
views	(list of HolovizOp.InputSpec.View) Sequence of layer views. By default a layer will fill the whole window. When using a view, the layer can be placed freely within the window. When multiple views are specified, the layer is drawn multiple times using the specified layer views.
2. __init__(self: holoscan.operators.holoviz._holoviz.HolovizOp.InputSpec, arg0: str, arg1: str) -> None	

property color

property depth_map_render_mode

description(*self: holoscan.operators.holoviz._holoviz.HolovizOp.InputSpec*) str

Returns

description

YAML string representation of the InputSpec class.

property line_width

property opacity

property point_size

property priority

property text

property type

property views

class InputType

Bases: pybind11_builtins.pybind11_object

Members:

UNKNOWN

COLOR

COLOR_LUT

POINTS

LINES

LINE_STRIP

TRIANGLES

CROSSES

RECTANGLES

OVALS

TEXT

DEPTH_MAP

DEPTH_MAP_COLOR

POINTS_3D

LINES_3D

LINE_STRIP_3D

TRIANGLES_3D

Attributes

name	

value	

COLOR = *<InputType.COLOR: 1>*

COLOR_LUT = *<InputType.COLOR_LUT: 2>*

CROSSES = *<InputType.CROSSES: 7>*

DEPTH_MAP = *<InputType.DEPTH_MAP: 11>*

DEPTH_MAP_COLOR = *<InputType.DEPTH_MAP_COLOR: 12>*

LINES = *<InputType.LINES: 4>*

LINES_3D = *<InputType.LINES_3D: 14>*

LINE_STRIP = *<InputType.LINE_STRIP: 5>*

LINE_STRIP_3D = *<InputType.LINE_STRIP_3D: 15>*

OVALS = *<InputType.OVALS: 9>*

POINTS = *<InputType.POINTS: 3>*

POINTS_3D = *<InputType.POINTS_3D: 13>*

RECTANGLES = *<InputType.RECTANGLES: 8>*

TEXT = *<InputType.TEXT: 10>*

TRIANGLES = *<InputType.TRIANGLES: 6>*

TRIANGLES_3D = *<InputType.TRIANGLES_3D: 16>*

UNKNOWN = *<InputType.UNKNOWN: 0>*

__init__(*self: holoscan.operators.holoviz._holoviz.HolovizOp.InputType*, *value: int*) None

property name

property value

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = <*OperatorType.VIRTUAL: 2*>

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.operators.holoviz._holoviz.HolovizOp*, *fragment: holoscan.core._core.Fragment*, *allocator: holoscan.resources._resources.Allocator*, *receivers: List[holoscan.core._core.IOSpec] = []*, *tensors: List[holoscan::ops::HolovizOp::InputSpec] = []*, *color_lut: List[List[float]] = []*, *window_title: str = 'Holoviz'*, *display_name: str = 'DP-0'*, *width: int = 1920*, *height: int = 1080*, *framerate: int = 60*, *use_exclusive_display: bool = False*, *fullscreen: bool = False*, *headless: bool = False*, *enable_render_buffer_input: bool = False*, *enable_render_buffer_output: bool = False*, *enable_camera_pose_output: bool = False*, *font_path: str = ''*, *cuda_stream_pool: holoscan.resources._resources.CudaStreamPool = None*, *name: str = 'holoviz_op'*)
None

Holoviz visualization operator using Holoviz module.

This is a Vulkan-based visualizer.

==Named Inputs==

receiversmulti-receiver accepting nvidia::gxf::Tensor and/or nvidia::gxf::VideoBuffer

Any number of upstream ports may be connected to this `receivers` port. This port can accept either VideoBuffers or Tensors. These inputs can be in either host or device memory. Each tensor or video buffer will result in a layer. The operator autodetects the layer type for certain input types (e.g. a video buffer will result in an image layer). For other input types or more complex use cases, input specifications can be provided either at initialization time as a parameter or dynamically at run time (via `input_specs`). On each call to `compute`, tensors corresponding to all names specified in the `tensors` parameter must be found or an exception will be raised. Any extra, named tensors not present in the `tensors` parameter specification (or optional, dynamic `input_specs` input) will be ignored.

input_specslist[holoscan.operators.HolovizOp.InputSpec] (optional)

A list of `InputSpec` objects. This port can be used to dynamically update the overlay specification at run time. No inputs are required on this port in order for the operator to `compute`.

render_buffer_inputnvidia::gxf::VideoBuffer (optional)

An empty render buffer can optionally be provided. The video buffer must have format GXF_VIDEO_FORMAT_RGBA and be in device memory. This input port only exists if `enable_render_buffer_input` was set to `True`, in which case `compute` will only be called when a message arrives on this input.

==Named Outputs==

render_buffer_outputnvidia::gxf::VideoBuffer (optional)

Output for a filled render buffer. If an input render buffer is specified, it is using that one, else it allocates a new buffer. The video buffer will have format GXF_VIDEO_FORMAT_RGBA and will be in device memory. This output is useful for offline rendering or headless mode. This output port only exists if `enable_render_buffer_output` was set to `True`.

camera_pose_outputstd::array<float, 16> (optional)

The camera pose. The parameters returned represent the values of a 4x4 row major projection matrix. This output port only exists if `enable_camera_pose_output` was set to `True`.

Parameters

fragment

The fragment that the operator belongs to.

allocator

Allocator used to allocate render buffer output. If `None`, will default to `holoscan.core.UnboundedAllocator`.

receivers

List of input receivers.

tensors

List of input tensors. Each tensor is defined by a dictionary where the `"name"` key must correspond to a tensor sent to the operator's input. See the notes section below for further details on how the tensor dictionary is defined.

color_lut

Color lookup table for tensors of type `color_lut`. Should be shape `(n_colors, 4)`.

window_title

Title on window canvas. Default value is `"Holoviz"`.

display_name

In exclusive mode, name of display to use as shown with xrandr. Default value is `"DP-0"`.

width

Window width or display resolution width if in exclusive or fullscreen mode. Default value is `1920`.

height

Window height or display resolution width if in exclusive or fullscreen mode. Default value is `1080`.

framerate

Display framerate in Hz if in exclusive mode. Default value is `60.0`.

use_exclusive_display

Enable exclusive display. Default value is `False`.

fullscreen

Enable fullscreen window. Default value is `False`.

headless

Enable headless mode. No window is opened, the render buffer is output to port `render_buffer_output`. Default value is `False`.

enable_render_buffer_input

If `True`, an additional input port, named `"render_buffer_input"` is added to the operator. Default value is `False`.

enable_render_buffer_output

If `True`, an additional output port, named `"render_buffer_output"` is added to the operator. Default value is `False`.

enable_camera_pose_output

If `True`, an additional output port, named `"camera_pose_output"` is added to the operator. Default value is `False`.

font_path

File path for the font used for rendering text. Default value is `""`.

cuda_stream_pool

`holoscan.resources.CudaStreamPool` instance to allocate CUDA streams. Default value is `None`.

name

The name of the operator. Default value is `"holoviz_op"`.

Notes

The `tensors` argument is used to specify the tensors to display. Each tensor is defined using a dictionary, that must, at minimum include a 'name' key that corresponds to a tensor found on the operator's input. A 'type' key should also be provided to indicate the type of entry to display. The 'type' key will be one of {`"color"`, `"color_lut"`, `"crosses"`, `"lines"`, `"lines_3d"`, `"line_strip"`, `"line_strip_3d"`, `"ovals"`, `"points"`, `"points_3d"`, `"rectangles"`, `"text"`, `"triangles"`, `"triangles_3d"`, `"depth_map"`, `"depth_map_color"`, `"unknown"`}. The default type is `"unknown"` which will attempt to guess the corresponding type based on the tensor dimensions. Concrete examples are given below.

To show a single 2D RGB or RGBA image, use a list containing a single tensor of type `"color"`.

```
tensors = [dict(name="video", type="color", opacity=1.0, priority=0)]
```

Here, the optional key `opacity` is used to scale the opacity of the tensor. The `priority` key is used to specify the render priority for layers. Layers with a higher priority will be rendered on top of those with a lower priority.

If we also had a `"boxes"` tensor representing rectangular bounding boxes, we could display them on top of the image like this.

```
tensors = [ dict(name="video", type="color", priority=0),
dict(name="boxes", type="rectangles", color=[1.0, 0.0, 0.0], line_width=2,
priority=1), ]
```

where the `color` and `line_width` keys specify the color and line width of the bounding box.

The details of the dictionary is as follows:

- **name**: name of the tensor containing the input data to display
  - type: `str`
- **type**: input type (default `"unknown"`)

- type: `str`
- possible values:
  - **unknown**: unknown type, the operator tries to guess the type by inspecting the tensor.
  - **color**: RGB or RGBA color 2d image.
  - **color_lut**: single channel 2d image, color is looked up.
  - **points**: point primitives, one coordinate (x, y) per primitive.
  - **lines**: line primitives, two coordinates (x0, y0) and (x1, y1) per primitive.
  - **line_strip**: line strip primitive, a line primitive i is defined by each coordinate (xi, yi) and the following (xi+1, yi+1).
  - **triangles**: triangle primitive, three coordinates (x0, y0), (x1, y1) and (x2, y2) per primitive.
  - **crosses**: cross primitive, a cross is defined by the center coordinate and the size (xi, yi, si).
  - **rectangles**: axis aligned rectangle primitive, each rectangle is defined by two coordinates (xi, yi) and (xi+1, yi+1).
  - **ovals**: oval primitive, an oval primitive is defined by the center coordinate and the axis sizes (xi, yi, sxi, syi).
  - **text**: text is defined by the top left coordinate and the size (x, y, s) per string, text strings are defined by InputSpec member **text**.
  - **depth_map**: single channel 2d array where each element represents a depth value. The data is rendered as a 3d object using points, lines or triangles. The color for the elements can be specified through `depth_map_color`. Supported format: 8-bit unsigned normalized format that has a single 8-bit depth component.

        - **depth_map_color**: RGBA 2d image, same size as the depth map. One color value for each element of the depth map grid. Supported format: 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.

- **opacity**: layer opacity, 1.0 is fully opaque, 0.0 is fully transparent (default: `1.0`)
    - type: `float`
- **priority**: layer priority, determines the render order, layers with higher priority

  values are rendered on top of layers with lower priority values (default: `0`)
    - type: `int`
- **color**: RGBA color of rendered geometry (default: `[1.f, 1.f, 1.f, 1.f]`)
    - type: `List[float]`
- **line_width**: line width for geometry made of lines (default: `1.0`)
    - type: `float`
- **point_size**: point size for geometry made of points (default: `1.0`)
    - type: `float`
- **text**: array of text strings, used when `type` is text (default: `[]`)
    - type: `List[str]`
- **depth_map_render_mode**: depth map render mode (default: `points`)
    - type: `str`

- possible values:
  - **points**: render as points
  - **lines**: render as lines
  - **triangles**: render as triangles

1. Displaying Color Images

   Image data can either be on host or device (GPU). Multiple image formats are supported

   - R 8 bit unsigned
   - R 16 bit unsigned
   - R 16 bit float
   - R 32 bit unsigned
   - R 32 bit float
   - RGB 8 bit unsigned
   - BGR 8 bit unsigned
   - RGBA 8 bit unsigned
   - BGRA 8 bit unsigned
   - RGBA 16 bit unsigned
   - RGBA 16 bit float
   - RGBA 32 bit float

   When the `type` parameter is set to `color_lut` the final color is looked up using the values from the `color_lut` parameter. For color lookups these image formats are supported

   - R 8 bit unsigned

- R 16 bit unsigned
- R 32 bit unsigned

2. Drawing Geometry

   In all cases, `x` and `y` are normalized coordinates in the range `[0, 1]`. The `x` and `y` correspond to the horizontal and vertical axes of the display, respectively. The origin `(0, 0)` is at the top left of the display. Geometric primitives outside of the visible area are clipped. Coordinate arrays are expected to have the shape `(N, C)` where `N` is the coordinate count and `C` is the component count for each coordinate.

   - Points are defined by a `(x, y)` coordinate pair.
   - Lines are defined by a set of two `(x, y)` coordinate pairs.
   - Lines strips are defined by a sequence of `(x, y)` coordinate pairs. The first two coordinates define the first line, each additional coordinate adds a line connecting to the previous coordinate.
   - Triangles are defined by a set of three `(x, y)` coordinate pairs.
   - Crosses are defined by `(x, y, size)` tuples. `size` specifies the size of the cross in the `x` direction and is optional, if omitted it's set to `0.05`. The size in the `y` direction is calculated using the aspect ratio of the window to make the crosses square.
   - Rectangles (bounding boxes) are defined by a pair of 2-tuples defining the upper-left and lower-right coordinates of a box: `(x1, y1), (x2, y2)`.
   - Ovals are defined by `(x, y, size_x, size_y)` tuples. `size_x` and `size_y` are optional, if omitted they are set to `0.05`.
   - Texts are defined by `(x, y, size)` tuples. `size` specifies the size of the text in `y` direction and is optional, if omitted it's set to `0.05`. The size in the `x` direction is calculated using the aspect ratio of

the window. The index of each coordinate references a text string from the `text` parameter and the index is clamped to the size of the text array. For example, if there is one item set for the `text` parameter, e.g. `text=["my_text"]` and three coordinates, then `my_text` is rendered three times. If `text=["first text", "second text"]` and three coordinates are specified, then `first text` is rendered at the first coordinate, `second text` at the second coordinate and then `second text` again at the third coordinate. The `text` string array is fixed and can't be changed after initialization. To hide text which should not be displayed, specify coordinates greater than `(1.0, 1.0)` for the text item, the text is then clipped away.

- 3D Points are defined by a `(x, y, z)` coordinate tuple.
- 3D Lines are defined by a set of two `(x, y, z)` coordinate tuples.
- 3D Lines strips are defined by a sequence of `(x, y, z)` coordinate tuples. The first two coordinates define the first line, each additional coordinate adds a line connecting to the previous coordinate.
- 3D Triangles are defined by a set of three `(x, y, z)` coordinate tuples.

3. Displaying Depth Maps

   When `type` is `depth_map` the provided data is interpreted as a rectangular array of depth values. Additionally a 2d array with a color value for each point in the grid can be specified by setting `type` to `depth_map_color`.

   The type of geometry drawn can be selected by setting `depth_map_render_mode`.

   Depth maps are rendered in 3D and support camera movement. The camera is controlled 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)

4. Output

   By default a window is opened to display the rendering, but the extension can also be run in headless mode with the `headless` parameter.

   Using a display in exclusive mode is also supported with the `use_exclusive_display` parameter. This reduces the latency by avoiding the desktop compositor.

   The rendered framebuffer can be output to `render_buffer_output`.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.operators.holoviz._holoviz.HolovizOp*) None

Initialize the operator.

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

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.operators.holoviz._holoviz.HolovizOp*, *spec: holoscan.core._core.OperatorSpec*) None

Define the operator specification.

Parameters

spec

The operator specification.

property spec

The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.operators.InferenceOp

Bases: `holoscan.core._core.Operator`

Inference operator.

==Named Inputs==

receiversmulti-receiver accepting nvidia::gxf::Tensor(s)

Any number of upstream ports may be connected to this `receivers` port. The operator will search across all messages for tensors matching those specified in `in_tensor_names`. These are the set of input tensors used by the models in `inference_map`.

==Named Outputs==

transmitternvidia::gxf::Tensor(s)

A message containing tensors corresponding to the inference results from all models will be emitted. The names of the tensors transmitted correspond to those in `out_tensor_names`.

For more details on `InferenceOp` parameters, see [Customizing the Inference Operator](https://docs.nvidia.com/holoscan/sdk-user-guide/examples/byom.html#customizing-the-inference-operator) or refer to [Inference](https://docs.nvidia.com/holoscan/sdk-user-guide/inference.html).

Parameters

fragment

The fragment that the operator belongs to.

backend

Backend to use for inference. Set `"trt"` for TensorRT, `"torch"` for LibTorch and `"onnxrt"` for the ONNX runtime.

allocator

Memory allocator to use for the output.

inference_map

Tensor to model map.

model_path_map

Path to the ONNX model to be loaded.

pre_processor_map

Pre processed data to model map.

device_map

Mapping of model to GPU ID for inference.

backend_map

Mapping of model to backend type for inference. Backend options: `"trt"` or `"torch"`

in_tensor_names

Input tensors.

out_tensor_names

Output tensors.

infer_on_cpu

Whether to run the computation on the CPU instead of GPU. Default value is `False`.

parallel_inference

Whether to enable parallel execution. Default value is `True`.

input_on_cuda

Whether the input buffer is on the GPU. Default value is `True`.

output_on_cuda

Whether the output buffer is on the GPU. Default value is `True`.

transmit_on_cuda

Whether to transmit the message on the GPU. Default value is `True`.

enable_fp16

Use 16-bit floating point computations. Default value is `False`.

is_engine_path

Whether the input model path mapping is for trt engine files. Default value is `False`.

cuda_stream_pool

`holoscan.resources.CudaStreamPool` instance to allocate CUDA streams. Default value is `None`.

name

The name of the operator. Default value is `"inference"`.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

`add_arg` (*args, **kwargs)	Overloaded function.

`compute`(self, arg0, arg1, arg2)	Operator compute method.
`initialize`(self)	Initialize the operator.
`setup`(self, spec)	Define the operator specification.
`start`(self)	Operator start method.
`stop`(self)	Operator stop method.

DataMap	
DataVecMap	
OperatorType	

class DataMap

Bases: `pybind11_builtins.pybind11_object`

Methods

`get_map`(self)	
`insert`(self)	

__init__(*self:* *holoscan.operators.inference._inference.InferenceOp.DataMap*)
None

get_map(*self:* *holoscan.operators.inference._inference.InferenceOp.DataMap*)
Dict[str, str]

insert(*self:* *holoscan.operators.inference._inference.InferenceOp.DataMap*)
Dict[str, str]

class DataVecMap

Bases: pybind11_builtins.pybind11_object

Methods

get_map (self)	
insert (self)	

__init__(*self:* *holoscan.operators.inference._inference.InferenceOp.DataVecMap*)
None

get_map(*self:* *holoscan.operators.inference._inference.InferenceOp.DataVecMap*)
Dict[str, List[str]]

insert(*self:* *holoscan.operators.inference._inference.InferenceOp.DataVecMap*)
Dict[str, List[str]]

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.

- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.operators.inference._inference.InferenceOp*, *fragment: holoscan.core._core.Fragment*, *backend: str*, *allocator: holoscan.resources._resources.Allocator*, *inference_map: dict*, *model_path_map: dict*, *pre_processor_map: dict*, *device_map: dict = {}*, *backend_map: dict = {}*, *in_tensor_names: List[str] = []*, *out_tensor_names: List[str] = []*, *infer_on_cpu: bool = False*, *parallel_inference: bool = True*, *input_on_cuda: bool = True*, *output_on_cuda: bool = True*, *transmit_on_cuda: bool = True*, *enable_fp16: bool = False*, *is_engine_path: bool = False*, *cuda_stream_pool: holoscan.resources._resources.CudaStreamPool = None*, *name: str = 'inference'*) None

Inference operator.

==Named Inputs==

receiversmulti-receiver accepting nvidia::gxf::Tensor(s)

Any number of upstream ports may be connected to this `receivers` port. The operator will search across all messages for tensors matching those specified in `in_tensor_names`. These are the set of input tensors used by the models in `inference_map`.

==Named Outputs==

transmitternvidia::gxf::Tensor(s)

A message containing tensors corresponding to the inference results from all models will be emitted. The names of the tensors transmitted correspond to those in `out_tensor_names`.

For more details on `InferenceOp` parameters, see [Customizing the Inference Operator](https://docs.nvidia.com/holoscan/sdk-user-guide/examples/byom.html#customizing-the-inference-operator) or refer to [Inference](https://docs.nvidia.com/holoscan/sdk-user-guide/inference.html).

Parameters

fragment

The fragment that the operator belongs to.

backend

Backend to use for inference. Set `"trt"` for TensorRT, `"torch"` for LibTorch and `"onnxrt"` for the ONNX runtime.

allocator

Memory allocator to use for the output.

inference_map

Tensor to model map.

model_path_map

Path to the ONNX model to be loaded.

pre_processor_map

Pre processed data to model map.

device_map

Mapping of model to GPU ID for inference.

backend_map

Mapping of model to backend type for inference. Backend options: `"trt"` or `"torch"`

in_tensor_names

Input tensors.

out_tensor_names

Output tensors.

infer_on_cpu

Whether to run the computation on the CPU instead of GPU. Default value is `False`.

parallel_inference

Whether to enable parallel execution. Default value is `True`.

input_on_cuda

Whether the input buffer is on the GPU. Default value is `True`.

output_on_cuda

Whether the output buffer is on the GPU. Default value is `True`.

transmit_on_cuda

Whether to transmit the message on the GPU. Default value is `True`.

enable_fp16

Use 16-bit floating point computations. Default value is `False`.

is_engine_path

Whether the input model path mapping is for trt engine files. Default value is `False`.

cuda_stream_pool

`holoscan.resources.CudaStreamPool` instance to allocate CUDA streams. Default value is `None`.

name

The name of the operator. Default value is `"inference"`.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.operators.inference._inference.InferenceOp*) None

Initialize the operator.

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

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.operators.inference._inference.InferenceOp*, *spec: holoscan.core._core.OperatorSpec*) None

Define the operator specification.

Parameters

spec

The operator specification.

property spec

The operator spec (holoscan.core.OperatorSpec) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.operators.InferenceProcessorOp

Bases: holoscan.core._core.Operator

Holoinfer Processing operator.

==Named Inputs==

receiversmulti-receiver accepting nvidia::gxf::Tensor(s)

Any number of upstream ports may be connected to this receivers port. The operator will search across all messages for tensors matching those specified in in_tensor_names. These are the set of input tensors used by the processing operations specified in process_map.

==Named Outputs==

transmitternvidia::gxf::Tensor(s)

A message containing tensors corresponding to the processed results from operations will be emitted. The names of the tensors transmitted correspond to those in `out_tensor_names`.

Parameters

fragment

The fragment that the operator belongs to.

allocator

Memory allocator to use for the output.

process_operations

Operations in sequence on tensors.

processed_map

Input-output tensor mapping.

in_tensor_names

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

out_tensor_names

Names of output tensors in the order to be fed into the operator.

input_on_cuda

Whether the input buffer is on the GPU. Default value is `False`.

output_on_cuda

Whether the output buffer is on the GPU. Default value is `False`.

transmit_on_cuda

Whether to transmit the message on the GPU. Default value is `False`.

cuda_stream_pool

`holoscan.resources.CudaStreamPool` instance to allocate CUDA streams. Default value is `None`.

config_path

File path to the config file. Default value is `""`.

disable_transmitter

If `True`, disable the transmitter output port of the operator. Default value is `False`.

name

The name of the operator. Default value is `"postprocessor"`.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.

`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`compute` (self, arg0, arg1, arg2)	Operator compute method.
`initialize` (self)	Initialize the operator.
`setup` (self, spec)	Define the operator specification.
`start` (self)	Operator start method.
`stop` (self)	Operator stop method.

DataMap	
DataVecMap	
OperatorType	

class DataMap

Bases: pybind11_builtins.pybind11_object

Methods

get_map (self)	
insert (self)	

__init__(*self:* *holoscan.operators.inference_processor._inference_processor.InferenceProcessorOp*
None

get_map(*self:* *holoscan.operators.inference_processor._inference_processor.InferenceProcessorOp*
Dict[str, str]

insert(*self:* *holoscan.operators.inference_processor._inference_processor.InferenceProcessorOp*
Dict[str, str]

class DataVecMap

Bases: pybind11_builtins.pybind11_object

Methods

get_map (self)	
insert (self)	

__init__(*self:*
holoscan.operators.inference_processor._inference_processor.InferenceProcessorOp
None

get_map(*self:*
holoscan.operators.inference_processor._inference_processor.InferenceProcessorOp
Dict[str, List[str]]

insert(*self:*
holoscan.operators.inference_processor._inference_processor.InferenceProcessorOp
Dict[str, List[str]]

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.operators.inference_processor._inference_processor.InferenceProcessorOp*, *fragment: holoscan.core._core.Fragment*, *allocator: holoscan.resources._resources.Allocator*, *process_operations: dict = {}*, *processed_map: dict = {}*, *in_tensor_names: List[str] = []*, *out_tensor_names: List[str] = []*, *input_on_cuda: bool = False*, *output_on_cuda: bool = False*, *transmit_on_cuda: bool = False*, *disable_transmitter: bool = False*, *cuda_stream_pool: holoscan.resources._resources.CudaStreamPool = None*, *config_path: str = ''*, *name: str = 'postprocessor'*) None

Holoinfer Processing operator.

==Named Inputs==

receiversmulti-receiver accepting nvidia::gxf::Tensor(s)

Any number of upstream ports may be connected to this `receivers` port. The operator will search across all messages for tensors matching those specified in `in_tensor_names`. These are the set of input tensors used by the processing operations specified in `process_map`.

==Named Outputs==

transmitternvidia::gxf::Tensor(s)

A message containing tensors corresponding to the processed results from operations will be emitted. The names of the tensors transmitted correspond to those in `out_tensor_names`.

Parameters

fragment

The fragment that the operator belongs to.

allocator

Memory allocator to use for the output.

process_operations

Operations in sequence on tensors.

processed_map

Input-output tensor mapping.

in_tensor_names

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

out_tensor_names

Names of output tensors in the order to be fed into the operator.

input_on_cuda

Whether the input buffer is on the GPU. Default value is `False`.

output_on_cuda

Whether the output buffer is on the GPU. Default value is `False`.

transmit_on_cuda

Whether to transmit the message on the GPU. Default value is `False`.

cuda_stream_pool

`holoscan.resources.CudaStreamPool` instance to allocate CUDA streams. Default value is `None`.

config_path

File path to the config file. Default value is `""`.

disable_transmitter

If `True`, disable the transmitter output port of the operator. Default value is `False`.

name

The name of the operator. Default value is `"postprocessor"`.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.operators.inference_processor._inference_processor.InferenceProcessorOp*) None

Initialize the operator.

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

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self:* *holoscan.operators.inference_processor._inference_processor.InferenceProcessorOp*, *spec: holoscan.core._core.OperatorSpec*) None

Define the operator specification.

Parameters

spec

The operator specification.

property spec

The operator spec (holoscan.core.OperatorSpec) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.operators.NTV2Channel

Bases: pybind11_builtins.pybind11_object

Members:

NTV2_CHANNEL1

NTV2_CHANNEL2

NTV2_CHANNEL3

NTV2_CHANNEL4

NTV2_CHANNEL5

NTV2_CHANNEL6

NTV2_CHANNEL7

NTV2_CHANNEL8

NTV2_MAX_NUM_CHANNELS

NTV2_CHANNEL_INVALID

Attributes

name	

value	

NTV2_CHANNEL1 = *<NTV2Channel.NTV2_CHANNEL1: 0>*

NTV2_CHANNEL2 = *<NTV2Channel.NTV2_CHANNEL2: 1>*

NTV2_CHANNEL3 = *<NTV2Channel.NTV2_CHANNEL3: 2>*

NTV2_CHANNEL4 = *<NTV2Channel.NTV2_CHANNEL4: 3>*

NTV2_CHANNEL5 = *<NTV2Channel.NTV2_CHANNEL5: 4>*

NTV2_CHANNEL6 = *<NTV2Channel.NTV2_CHANNEL6: 5>*

NTV2_CHANNEL7 = *<NTV2Channel.NTV2_CHANNEL7: 6>*

NTV2_CHANNEL8 = *<NTV2Channel.NTV2_CHANNEL8: 7>*

NTV2_CHANNEL_INVALID = *<NTV2Channel.NTV2_MAX_NUM_CHANNELS: 8>*

NTV2_MAX_NUM_CHANNELS = *<NTV2Channel.NTV2_MAX_NUM_CHANNELS: 8>*

__init__(*self: holoscan.operators.aja_source._aja_source.NTV2Channel*, *value: int*) None

property name

property value

class holoscan.operators.PingRxOp(*fragment*, **args*, ***kwargs*)

Bases: holoscan.core.Operator

Simple receiver operator.

This is an example of a native operator with one input port. On each tick, it receives an integer from the “in” port.

==Named Inputs==

inany

A received value.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`compute` (op_input, op_out	Default implementation of compute

put, context)	
`initialize`()	Default implementation of initialize
`setup`(spec)	Default implementation of setup method.
`start`()	Default implementation of start
`stop`()	Default implementation of stop

OperatorType	

class OperatorType

Bases: `pybind11_builtins.pybind11_object`

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.core._core.Operator*, *arg0: object*, *arg1: holoscan::Fragment*, **args*, ***kwargs*) None

Operator class.

Can be initialized with any number of Python positional and keyword arguments.

If a *name* keyword argument is provided, it must be a *str* and will be used to set the name of the Operator.

Condition classes will be added to `self.conditions`, *Resource* classes will be added to `self.resources`, and any other arguments will be cast from a Python argument type to a C++ *Arg* and stored in `self.args`. (For details on how the casting is done, see the *py_object_to_arg* utility). When a Condition or Resource is provided via a kwarg, it's name will be automatically be updated to the name of the kwarg.

Parameters

fragment

The *holoscan.core.Fragment* (or *holoscan.core.Application*) to which this Operator will belong.

***args**

Positional arguments.

****kwargs**

Keyword arguments.

Raises

RuntimeError

If *name* kwarg is provided, but is not of *str* type. If multiple arguments of type *Fragment* are provided. If any other arguments cannot be converted to *Arg* type via *py_object_to_arg*.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*op_input*, *op_output*, *context*)

Default implementation of compute

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1` , and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize()

Default implementation of initialize

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*spec: holoscan.core._core.PyOperatorSpec*)

Default implementation of setup method.

property spec

The operator spec (holoscan.core.OperatorSpec) associated with the operator.

start()

Default implementation of start

stop()

Default implementation of stop

class holoscan.operators.PingTxOp(*fragment*, **args*, ***kwargs*)

Bases: holoscan.core.Operator

Simple transmitter operator.

On each tick, it transmits an integer to the “out” port.

==Named Outputs==

outint

An index value that increments by one on each call to *compute*. The starting value is 1.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

add_arg(*args, **kwargs)	Overloaded function.
compute(op_input, op_output, context)	Default implementation of compute
initialize()	Default implementation of initialize
setup(spec)	Default implementation of setup method.
start()	Default implementation of start
stop()	Default implementation of stop

OperatorType	

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = <*OperatorType.GXF: 1*>

NATIVE = <*OperatorType.NATIVE: 0*>

VIRTUAL = <*OperatorType.VIRTUAL: 2*>

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.core._core.Operator*, *arg0: object*, *arg1: holoscan::Fragment*, **args*, ***kwargs*) None

Operator class.

Can be initialized with any number of Python positional and keyword arguments.

If a *name* keyword argument is provided, it must be a *str* and will be used to set the name of the Operator.

Condition classes will be added to `self.conditions`, *Resource* classes will be added to `self.resources`, and any other arguments will be cast from a Python

argument type to a C++ *Arg* and stored in `self.args`. (For details on how the casting is done, see the *py_object_to_arg* utility). When a Condition or Resource is provided via a kwarg, it's name will be automatically be updated to the name of the kwarg.

Parameters

fragment

The *holoscan.core.Fragment* (or *holoscan.core.Application*) to which this Operator will belong.

***args**

Positional arguments.

****kwargs**

Keyword arguments.

Raises

RuntimeError

If *name* kwarg is provided, but is not of *str* type. If multiple arguments of type *Fragment* are provided. If any other arguments cannot be converted to *Arg* type via *py_object_to_arg*.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*op_input*, *op_output*, *context*)

Default implementation of compute

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize()

Default implementation of initialize

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*spec: holoscan.core._core.PyOperatorSpec*)

Default implementation of setup method.

property spec

The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

start()

Default implementation of start

stop()

Default implementation of stop

class holoscan.operators.SegmentationPostprocessorOp

Bases: holoscan.core._core.Operator

Operator carrying out post-processing operations on segmentation outputs.

==Named Inputs==

in_tensornvidia::gxf::Tensor

Expects a message containing a 32-bit floating point tensor with name `in_tensor_name`. The expected data layout of this tensor is HWC, NCHW or NHWC format as specified via `data_format`.

==Named Outputs==

out_tensornvidia::gxf::Tensor

Emits a message containing a tensor named “out_tensor” that contains the segmentation labels. This tensor will have unsigned 8-bit integer data type and shape (H, W, 1).

Parameters

fragment

The fragment that the operator belongs to.

allocator

Memory allocator to use for the output.

in_tensor_name

Name of the input tensor. Default value is `""`.

network_output_type

Network output type (e.g. ‘softmax’). Default value is `"softmax"`.

data_format

Data format of network output. Default value is `"hwc"`.

cuda_stream_pool

`holoscan.resources.CudaStreamPool` instance to allocate CUDA streams. Default value is `None`.

name

The name of the operator. Default value is `"segmentation_postprocessor"`.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.

resources	Resources associated with the operator.
spec	The operator spec (holoscan.core.OperatorSpec) associated with the operator.

Methods

add_arg (*args, **kwargs)	Overloaded function.
compute (self, arg0, arg1, arg2)	Operator compute method.
initialize (self)	Operator initialization method.
setup (self, spec)	Define the operator specification.
start (self)	Operator start method.
stop (self)	Operator stop method.

OperatorType	

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.operators.segmentation_postprocessor._segmentation_postprocessor.Segmenta fragment: holoscan.core._core.Fragment*, *allocator: holoscan.resources._resources.Allocator*, *in_tensor_name: str = ''*, *network_output_type:*

str = 'softmax', *data_format: str = 'hwc'*, *cuda_stream_pool: holoscan.resources._resources.CudaStreamPool = None*, *name: str = 'segmentation_postprocessor'*) None

Operator carrying out post-processing operations on segmentation outputs.

==Named Inputs==

in_tensornvidia::gxf::Tensor

Expects a message containing a 32-bit floating point tensor with name `in_tensor_name`. The expected data layout of this tensor is HWC, NCHW or NHWC format as specified via `data_format`.

==Named Outputs==

out_tensornvidia::gxf::Tensor

Emits a message containing a tensor named “out_tensor” that contains the segmentation labels. This tensor will have unsigned 8-bit integer data type and shape (H, W, 1).

Parameters

fragment

The fragment that the operator belongs to.

allocator

Memory allocator to use for the output.

in_tensor_name

Name of the input tensor. Default value is `""`.

network_output_type

Network output type (e.g. ‘softmax’). Default value is `"softmax"`.

data_format

Data format of network output. Default value is `"hwc"`.

cuda_stream_pool

`holoscan.resources.CudaStreamPool` instance to allocate CUDA streams. Default value is `None`.

name

The name of the operator. Default value is `"segmentation_postprocessor"`.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1` , and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.core._core.Operator*) None

Operator initialization method.

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.operators.segmentation_postprocessor._segmentation_postprocessor.Segmenta spec: holoscan.core._core.OperatorSpec*) None

Define the operator specification.

Parameters

spec

The operator specification.

property spec

The operator spec (holoscan.core.OperatorSpec) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.operators.V4L2VideoCaptureOp

Bases: `holoscan.core._core.Operator`

Operator to get a video stream from a V4L2 source.

https://www.kernel.org/doc/html/v4.9/media/uapi/v4l/v4l2.html

Inputs a video stream from a V4L2 node, including USB cameras and HDMI IN.

- Input stream is on host. If no pixel format is specified in the yaml configuration file, the pixel format will be automatically selected. However, only `AB24` and `YUYV` are then supported. If a pixel format is specified in the yaml file, then this format will be used. However, note that the operator then expects that this format can be encoded as RGBA32. If not, the behavior is undefined.
- Output stream is on host. Always RGBA32 at this time.

Use `holoscan.operators.FormatConverterOp` to move data from the host to a GPU device.

==Named Outputs==

signalnvidia::gxf::VideoBuffer

A message containing a video buffer on the host with format GXF_VIDEO_FORMAT_RGBA.

Parameters

fragment

The fragment that the operator belongs to.

allocator

Memory allocator to use for the output.

device

The device to target (e.g. “/dev/video0” for device 0). Default value is `"/dev/video0"`.

width

Width of the video stream. Default value is `0`.

height

Height of the video stream. Default value is `0`.

num_buffers

Number of V4L2 buffers to use. Default value is `4`.

pixel_format

Video stream pixel format (little endian four character code (fourcc)). Default value is `"auto"`.

name

The name of the operator. Default value is `"v4l2_video_capture"`.

exposure_time

Exposure time of the camera sensor in multiples of 100 µs (e.g. setting exposure_time to 100 is 10 ms). Default: auto exposure, or camera sensor default. Use *v4l2-ctl -d /dev/<your_device> -L* for a range of values supported by your device. When not set by the user, V4L2_CID_EXPOSURE_AUTO is set to V4L2_EXPOSURE_AUTO, or to V4L2_EXPOSURE_APERTURE_PRIORITY if the former is not supported. When set by the user, V4L2_CID_EXPOSURE_AUTO is set to V4L2_EXPOSURE_SHUTTER_PRIORITY, or to V4L2_EXPOSURE_MANUAL if

the former is not supported. The provided value is then used to set V4L2_CID_EXPOSURE_ABSOLUTE.
gain

Gain of the camera sensor. Default: auto gain, or camera sensor default. Use *v4l2-ctl -d /dev/<your_device> -L* for a range of values supported by your device. When not set by the user, V4L2_CID_AUTOGAIN is set to true (if supported). When set by the user, V4L2_CID_AUTOGAIN is set to false (if supported). The provided value is then used to set V4L2_CID_GAIN.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

add_arg(*args, **kwargs)	Overloaded function.
compute(self, arg0, arg1, arg2)	Operator compute method.
initialize(self)	Initialize the operator.
setup(self, spec)	Define the operator specification.
start(self)	Operator start method.
stop(self)	Operator stop method.

OperatorType	

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.operators.v4l2_video_capture._v4l2_video_capture.V4L2VideoCaptureOp*, *fragment: holoscan.core._core.Fragment*, *allocator: holoscan.resources._resources.Allocator*, *device: str = '0'*, *width: int = 0*, *height: int = 0*, *num_buffers: int = 4*, *pixel_format: str = 'auto'*, *name: str = 'v4l2_video_capture'*, *exposure_time: Optional[int] = None*, *gain: Optional[int] = None*) None

Operator to get a video stream from a V4L2 source.

https://www.kernel.org/doc/html/v4.9/media/uapi/v4l/v4l2.html

Inputs a video stream from a V4L2 node, including USB cameras and HDMI IN.

- Input stream is on host. If no pixel format is specified in the yaml configuration file, the pixel format will be automatically selected. However, only `AB24` and `YUYV` are then supported. If a pixel format is specified in the yaml file, then this format will be used. However, note that the operator then expects that this format can be encoded as RGBA32. If not, the behavior is undefined.
- Output stream is on host. Always RGBA32 at this time.

Use `holoscan.operators.FormatConverterOp` to move data from the host to a GPU device.

==Named Outputs==

signalnvidia::gxf::VideoBuffer

A message containing a video buffer on the host with format GXF_VIDEO_FORMAT_RGBA.

Parameters

fragment

The fragment that the operator belongs to.

allocator

Memory allocator to use for the output.

device

The device to target (e.g. “/dev/video0” for device 0). Default value is `"/dev/video0"`.

width

Width of the video stream. Default value is `0`.

height

Height of the video stream. Default value is `0`.

num_buffers

Number of V4L2 buffers to use. Default value is `4`.

pixel_format

Video stream pixel format (little endian four character code (fourcc)). Default value is `"auto"`.

name

The name of the operator. Default value is `"v4l2_video_capture"`.

exposure_time

Exposure time of the camera sensor in multiples of 100 µs (e.g. setting exposure_time to 100 is 10 ms). Default: auto exposure, or camera sensor default. Use *v4l2-ctl -d /dev/<your_device> -L* for a range of values supported by your device. When not set by the user, V4L2_CID_EXPOSURE_AUTO is set to V4L2_EXPOSURE_AUTO, or to V4L2_EXPOSURE_APERTURE_PRIORITY if the former is not supported. When set by the user, V4L2_CID_EXPOSURE_AUTO is set to V4L2_EXPOSURE_SHUTTER_PRIORITY, or to V4L2_EXPOSURE_MANUAL if the former is not supported. The provided value is then used to set V4L2_CID_EXPOSURE_ABSOLUTE.

gain

Gain of the camera sensor. Default: auto gain, or camera sensor default. Use *v4l2-ctl -d /dev/<your_device> -L* for a range of values supported by your device. When not set by the user, V4L2_CID_AUTOGAIN is set to true (if supported). When set by the user, V4L2_CID_AUTOGAIN is set to false (if supported). The provided value is then used to set V4L2_CID_GAIN.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1` , and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.operators.v4l2_video_capture._v4l2_video_capture.V4L2VideoCaptureOp*)
None

Initialize the operator.

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

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.operators.v4l2_video_capture._v4l2_video_capture.V4L2VideoCaptureOp*, *spec: holoscan.core._core.OperatorSpec*) None

Define the operator specification.

Parameters

spec : holoscan.core.OperatorSpec

The operator specification.

property spec

The operator spec (holoscan.core.OperatorSpec) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.operators.VideoStreamRecorderOp

Bases: holoscan.core._core.Operator

Operator class to record a video stream to a file.

==Named Inputs==

inputnvidia::gxf::Tensor

A message containing a video frame to serialize to disk. The input tensor can be on either the CPU or GPU. This data location will be recorded as part of the metadata serialized to disk and if the data is later read back in via *VideoStreamReplayerOp*, the tensor output of that operator will be on the same device (CPU or GPU).

Parameters

fragment

The fragment that the operator belongs to.

directory

Directory path for storing files.

basename

User specified file name without extension.

flush_on_tick

Flushes output buffer on every tick when `True`. Default value is `False`.

name

The name of the operator. Default value is `"video_stream_recorder"`.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.

`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.
`resources`	Resources associated with the operator.
`spec`	The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`compute` (self, arg0, arg1, arg2)	Operator compute method.
`initialize` (self)	Initialize the operator.
`setup` (self, spec)	Define the operator specification.
`start` (self)	Operator start method.

stop (self)	Operator stop method.

OperatorType	

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.operators.video_stream_recorder._video_stream_recorder.VideoStreamRecorder*, *fragment: holoscan.core._core.Fragment*, *directory: str*, *basename: str*, *flush_on_tick: bool = False*, *name: str = 'recorder'*) None

Operator class to record a video stream to a file.

==Named Inputs==

inputnvidia::gxf::Tensor

A message containing a video frame to serialize to disk. The input tensor can be on either the CPU or GPU. This data location will be recorded as part of the metadata serialized to disk and if the data is later read back in via *VideoStreamReplayerOp*, the tensor output of that operator will be on the same device (CPU or GPU).

Parameters

fragment

The fragment that the operator belongs to.

directory

Directory path for storing files.

basename

User specified file name without extension.

flush_on_tick

Flushes output buffer on every tick when `True`. Default value is `False`.

name

The name of the operator. Default value is `"video_stream_recorder"`.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.operators.video_stream_recorder._video_stream_recorder.VideoStreamRecorder*) None

Initialize the operator.

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

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.operators.video_stream_recorder._video_stream_recorder.VideoStreamRecorder*, *spec: holoscan.core._core.OperatorSpec*) None

Define the operator specification.

Parameters

spec

The operator specification.

property spec

The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

class holoscan.operators.VideoStreamReplayerOp

Bases: holoscan.core._core.Operator

Operator class to replay a video stream from a file.

==Named Outputs==

outputnvidia::gxf::Tensor

A message containing a video frame deserialized from disk. Depending on the metadata in the file being read, this tensor could be on either CPU or GPU. For the data used in examples distributed with the SDK, the tensor will be an unnamed GPU tensor (name == “”).

Parameters

fragment

The fragment that the operator belongs to.

directory

Directory path for reading files from.

basename

User specified file name without extension.

batch_size

Number of entities to read and publish for one tick. Default value is `1`.

ignore_corrupted_entities

If an entity could not be deserialized, it is ignored by default; otherwise a failure is generated. Default value is `True`.

frame_rate

Frame rate to replay. If zero value is specified, it follows timings in timestamps. Default value is `0.0`.

realtime

Playback video in realtime, based on frame_rate or timestamps. Default value is `True`.

repeat

Repeat video stream in a loop. Default value is `False`.

count

Number of frame counts to playback. If zero value is specified, it is ignored. If the count is less than the number of frames in the video, it would finish early. Default value is `0`.

name

The name of the operator. Default value is `"video_stream_replayer"`.

Attributes

`args`	The list of arguments associated with the component.
`conditions`	Conditions associated with the operator.
`description`	YAML formatted string describing the operator.
`fragment`	The fragment (`holoscan.core.Fragment`) that the operator belongs to.
`id`	The identifier of the component.
`name`	The name of the operator.
`operator_type`	The operator type.

resources	Resources associated with the operator.
spec	The operator spec (holoscan.core.OperatorSpec) associated with the operator.

Methods

add_arg (*args, **kwargs)	Overloaded function.
compute (self, arg0, arg1, arg2)	Operator compute method.
initialize (self)	Initialize the operator.
setup (self, spec)	Define the operator specification.
start (self)	Operator start method.
stop (self)	Operator stop method.

OperatorType	

class OperatorType

Bases: pybind11_builtins.pybind11_object

Enum class for operator types used by the executor.

- NATIVE: Native operator.
- GXF: GXF operator.
- VIRTUAL: Virtual operator. (for internal use, not intended for use by application authors)

Members:

NATIVE

GXF

VIRTUAL

Attributes

name	

value	

GXF = *<OperatorType.GXF: 1>*

NATIVE = *<OperatorType.NATIVE: 0>*

VIRTUAL = *<OperatorType.VIRTUAL: 2>*

__init__(*self: holoscan.core._core.Operator.OperatorType*, *value: int*) None

property name

property value

__init__(*self: holoscan.operators.video_stream_replayer._video_stream_replayer.VideoStreamReplayerO fragment: holoscan.core._core.Fragment*, *directory: str*, *basename: str*, *batch_size: int* =

1, *ignore_corrupted_entities: bool = True*, *frame_rate: float = 1.0*, *realtime: bool = True*, *repeat: bool = False*, *count: int = 0*, *name: str = 'format_converter'*) None

Operator class to replay a video stream from a file.

==Named Outputs==

outputnvidia::gxf::Tensor

A message containing a video frame deserialized from disk. Depending on the metadata in the file being read, this tensor could be on either CPU or GPU. For the data used in examples distributed with the SDK, the tensor will be an unnamed GPU tensor (name == “”).

Parameters

fragment

The fragment that the operator belongs to.

directory

Directory path for reading files from.

basename

User specified file name without extension.

batch_size

Number of entities to read and publish for one tick. Default value is `1`.

ignore_corrupted_entities

If an entity could not be deserialized, it is ignored by default; otherwise a failure is generated. Default value is `True`.

frame_rate

Frame rate to replay. If zero value is specified, it follows timings in timestamps. Default value is `0.0`.

realtime

Playback video in realtime, based on frame_rate or timestamps. Default value is `True`.

repeat

Repeat video stream in a loop. Default value is `False`.

count

Number of frame counts to playback. If zero value is specified, it is ignored. If the count is less than the number of frames in the video, it would finish early. Default value is `0`.

name

The name of the operator. Default value is `"video_stream_replayer"`.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

3. add_arg(self: holoscan.core._core.Operator, **kwargs) -> None

Add arguments to the component via Python kwargs.

4. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Condition) -> None

5. add_arg(self: holoscan.core._core.Operator, arg: holoscan.core._core.Resource) -> None

Add a condition or resource to the Operator.

This can be used to add a condition or resource to an operator after it has already been constructed.

Parameters

arg

The condition or resource to add.

property args

The list of arguments associated with the component.

Returns

arglist

compute(*self: holoscan.core._core.Operator*, *arg0: holoscan.core._core.InputContext*, *arg1: holoscan.core._core.OutputContext*, *arg2: holoscan.core._core.ExecutionContext*) None

Operator compute method. This method defines the primary computation to be executed by the operator.

property conditions

Conditions associated with the operator.

property description

YAML formatted string describing the operator.

property fragment

The fragment (`holoscan.core.Fragment`) that the operator belongs to.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.operators.video_stream_replayer._video_stream_replayer.VideoStreamReplayerO*
None

Initialize the operator.

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

property name

The name of the operator.

property operator_type

The operator type.

holoscan.core.Operator.OperatorType enum representing the type of the operator. The two types currently implemented are native and GXF.

property resources

Resources associated with the operator.

setup(*self: holoscan.operators.video_stream_replayer._video_stream_replayer.VideoStreamReplayerO*
spec: holoscan.core._core.OperatorSpec) None

Define the operator specification.

Parameters

spec

The operator specification.

property spec

The operator spec (`holoscan.core.OperatorSpec`) associated with the operator.

start(*self: holoscan.core._core.Operator*) None

Operator start method.

stop(*self: holoscan.core._core.Operator*) None

Operator stop method.

holoscan.resources

This module provides a Python API to underlying C++ API Resources.

`holoscan.resources.Allocator`	Base allocator class.
`holoscan.resources.BlockMemoryPool`	Block memory pool resource.

holoscan.resources.Clock	Base clock class.
holoscan.resources.CudaStreamPool	CUDA stream pool.
holoscan.resources.DoubleBufferReceiver	Receiver using a double-buffered queue.
holoscan.resources.DoubleBufferTransmitter	Transmitter using a double-buffered queue.
holoscan.resources.ManualClock	Manual clock class.

holoscan.resources.MemoryStorageType	Members:
holoscan.resources.RealtimeClock	Real-time clock class.
holoscan.resources.Receiver	Base GXF receiver class.
holoscan.resources.SerializationBuffer	Serialization Buffer.
holoscan.resources.StdComponentSerializer	Serializer for GXF Timestamp and Tensor components.

holoscan.resources.StdEntitySerializer	Default serializer for GXF entities.
holoscan.resources.Transmitter	Base GXF transmitter class.
holoscan.resources.UnboundedAllocator	Unbounded allocator.
holoscan.resources.UcxComponentSerializer	UCX component serializer.
holoscan.resources.UcxEntitySerializer	UCX entity serializer.

holoscan.resources.UcxHoloscanComponentSerializer	UCX Holoscan component serializer.
holoscan.resources.UcxReceiver	UCX network receiver using a double-buffered queue.
holoscan.resources.UcxSerializationBuffer	UCX serialization buffer.
holoscan.resources.UcxTransmitter	UCX network transmitter using a double-buffered queue.

class holoscan.resources.Allocator

Bases: holoscan.gxf._gxf.GXFResource

Base allocator class.

Attributes

args	The list of arguments associated with the component.
block_size	Get the block size of the allocator.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
allocate(self, size, type)	Allocate the requested amount of memory.
free(self, pointer)	Free the allocated memory
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the component.
is_available(self, size)	Boolean representing whether the resource is available.
setup(self, arg0)	setup method for the resource.

__init__(*self: holoscan.resources._resources.Allocator*) None

Base allocator class.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

allocate(*self: holoscan.resources._resources.Allocator*, *size: int*, *type: holoscan.resources._resources.MemoryStorageType*) int

Allocate the requested amount of memory.

Parameters

size

The amount of memory to allocate

type

Enum representing the type of memory to allocate.

Returns

Opaque PyCapsule object representing a std::byte* pointer to the allocated memory.

property args

The list of arguments associated with the component.

Returns

arglist

property block_size

Get the block size of the allocator.

Returns

int

The block size of the allocator. Returns 1 for byte-based allocators.

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

free(*self: holoscan.resources._resources.Allocator*, *pointer: int*) None

Free the allocated memory

Parameters

pointer

Opaque PyCapsule object representing a std::byte* pointer to the allocated memory.

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

is_available(*self: holoscan.resources._resources.Allocator*, *size: int*) bool

Boolean representing whether the resource is available.

Returns

bool

Availability of the resource.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.core._core.Resource*, *arg0: holoscan.core._core.ComponentSpec*) None

setup method for the resource.

property spec

class holoscan.resources.BlockMemoryPool

Bases: holoscan.resources._resources.Allocator

Block memory pool resource.

Provides a maximum number of equally sized blocks of memory.

Attributes

args	The list of arguments associated with the component.
block_size	Get the block size of the allocator.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.

gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
allocate (self, size, type)	Allocate the requested amount of memory.
free (self, pointer)	Free the allocated memory
gxf_initialize (self)	Initialize the component.

initialize (self)	Initialize the component.
is_available (self, size)	Boolean representing whether the resource is available.
setup (self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.BlockMemoryPool*, *fragment: holoscan.core._core.Fragment*, *storage_type: int*, *block_size: int*, *num_blocks: int*, *dev_id: int = 0*, *name: str = 'block_memory_pool'*) None

Block memory pool resource.

Provides a maximum number of equally sized blocks of memory.

Parameters

fragment

The fragment to assign the resource to.

storage_type

The storage type (0=Host, 1=Device, 2=System).

block_size

The size of each block in the memory pool (in bytes).

num_blocks

The number of blocks in the memory pool.

dev_id

CUDA device ID. Specifies the device on which to create the memory pool.

name

The name of the memory pool.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

allocate(*self: holoscan.resources._resources.Allocator*, *size: int*, *type: holoscan.resources._resources.MemoryStorageType*) int

Allocate the requested amount of memory.

Parameters

size

The amount of memory to allocate

type

Enum representing the type of memory to allocate.

Returns

Opaque PyCapsule object representing a std::byte* pointer to the allocated memory.

property args

The list of arguments associated with the component.

Returns

arglist

property block_size

Get the block size of the allocator.

Returns

int

The block size of the allocator. Returns 1 for byte-based allocators.

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

free(*self: holoscan.resources._resources.Allocator*, *pointer: int*) None

Free the allocated memory

Parameters

pointer

Opaque PyCapsule object representing a std::byte* pointer to the allocated memory.

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

is_available(*self: holoscan.resources._resources.Allocator*, *size: int*) bool

Boolean representing whether the resource is available.

Returns

bool

Availability of the resource.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.BlockMemoryPool*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.Clock

Bases: holoscan.gxf._gxf.GXFResource

Base clock class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.

fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
gxf_initialize (self)	Initialize the component.
initialize (self)	Initialize the component.

`setup`(self, arg0)	setup method for the resource.

__init__(**args*, ***kwargs*)

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self:* *holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.core._core.Resource*, *arg0: holoscan.core._core.ComponentSpec*)
None

setup method for the resource.

property spec

class holoscan.resources.CudaStreamPool

Bases: holoscan.resources._resources.Allocator

CUDA stream pool.

Attributes

args	The list of arguments associated with the component.
block_size	Get the block size of the allocator.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.

`gxf_typename`	The GXF type name of the resource.
`id`	The identifier of the component.
`name`	The name of the resource.

spec	

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`allocate` (self, size, type)	Allocate the requested amount of memory.
`free` (self, pointer)	Free the allocated memory
`gxf_initialize` (self)	Initialize the component.
`initialize` (self)	Initialize the component.
`is_available` (self, si	Boolean representing whether the resource is available.

ze)	
`setup` (self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.CudaStreamPool*, *fragment: holoscan.core._core.Fragment*, *dev_id: int = 0*, *stream_flags: int = 0*, *stream_priority: int = 0*, *reserved_size: int = 1*, *max_size: int = 0*, *name: str = 'cuda_stream_pool'*) None

CUDA stream pool.

Parameters

fragment

The fragment to assign the resource to.

dev_id

CUDA device ID. Specifies the device on which to create the stream pool.

stream_flags

Flags for CUDA streams in the pool. This will be passed to CUDA's cudaStreamCreateWithPriority [Rb9bddbe55e1a-1] when creating the streams. The default value of 0 corresponds to `cudaStreamDefault`. A value of 1 corresponds to `cudaStreamNonBlocking`, indicating that the stream can run concurrently with work in stream 0 (default stream) and should not perform any implicit synchronization with it.

stream_priority

Priority value for CUDA streams in the pool. This is an integer value passed to cudaSreamCreateWithPriority [Rb9bddbe55e1a-1]. Lower numbers represent higher priorities.

reserved_size

The number of CUDA streams to initially reserve in the pool (prior to first request).

max_size

The maximum number of streams that can be allocated, unlimited by default.

name

The name of the stream pool.

References

[1]

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

allocate(*self: holoscan.resources._resources.Allocator*, *size: int*, *type: holoscan.resources._resources.MemoryStorageType*) int

Allocate the requested amount of memory.

Parameters

size

The amount of memory to allocate

type

Enum representing the type of memory to allocate.

Returns

Opaque PyCapsule object representing a std::byte* pointer to the allocated memory.

property args

The list of arguments associated with the component.

Returns

arglist

property block_size

Get the block size of the allocator.

Returns

int

The block size of the allocator. Returns 1 for byte-based allocators.

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

free(*self: holoscan.resources._resources.Allocator*, *pointer: int*) None

Free the allocated memory

Parameters

pointer

Opaque PyCapsule object representing a std::byte* pointer to the allocated memory.

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self:* *holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

is_available(*self: holoscan.resources._resources.Allocator*, *size: int*) bool

Boolean representing whether the resource is available.

Returns

bool

Availability of the resource.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.CudaStreamPool*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.DoubleBufferReceiver

Bases: holoscan.resources._resources.Receiver

Receiver using a double-buffered queue.

New messages are first pushed to a back stage.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the component.
setup(self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.DoubleBufferReceiver*, *fragment: holoscan.core._core.Fragment*, *capacity: int = 1*, *policy: int = 2*, *name: str = 'double_buffer_receiver'*) None

Receiver using a double-buffered queue.

New messages are first pushed to a back stage.

Parameters

fragment

The fragment to assign the resource to.

capacity

The capacity of the receiver.

policy

The policy to use (0=pop, 1=reject, 2=fault).

name

The name of the receiver.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.DoubleBufferReceiver*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.DoubleBufferTransmitter

Bases: holoscan.resources._resources.Transmitter

Transmitter using a double-buffered queue.

Messages are pushed to a back stage after they are published.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.

gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
gxf_initialize (self)	Initialize the component.
initialize (self)	Initialize the component.
setup (self, s	Define the component specification.

pec)	

__init__(*self: holoscan.resources._resources.DoubleBufferTransmitter*, *fragment: holoscan.core._core.Fragment*, *capacity: int = 1*, *policy: int = 2*, *name: str = 'double_buffer_transmitter'*) None

Transmitter using a double-buffered queue.

Messages are pushed to a back stage after they are published.

Parameters

fragment

The fragment to assign the resource to.

capacity

The capacity of the transmitter.

policy

The policy to use (0=pop, 1=reject, 2=fault).

name

The name of the transmitter.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self:* *holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.DoubleBufferTransmitter*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.ManualClock

Bases: holoscan.resources._resources.Clock

Manual clock class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the component.
setup(self, spec)	Define the component specification.
sleep_for(self, arg0)	Set the GXF scheduler to sleep for a specified duration.
sleep_until(self, target_time_ns)	Set the GXF scheduler to sleep until a specified timestamp.
time(self)	The current time of the clock (in seconds).
timestamp(self)	The current timestamp of the clock (in nanoseconds).

__init__(*self: holoscan.resources._resources.ManualClock*, *fragment: holoscan.core._core.Fragment*, *initial_timestamp: int = 0*, *name: str = 'realtime_clock'*)
None

Manual clock.

Parameters

fragment

The fragment to assign the resource to.

initial_timestamp

The initial timestamp on the clock (in nanoseconds).

name

The name of the clock.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.ManualClock*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

sleep_for(*self: holoscan.resources._resources.ManualClock*, *arg0: object*) None

Set the GXF scheduler to sleep for a specified duration.

Parameters

duration_ns

The duration to sleep (in nanoseconds).

sleep_until(*self: holoscan.resources._resources.ManualClock*, *target_time_ns: int*) None

Set the GXF scheduler to sleep until a specified timestamp.

Parameters

target_time_ns

The target timestamp (in nanoseconds).

property spec

time(*self: holoscan.resources._resources.ManualClock*) float

The current time of the clock (in seconds).

Parameters

time

The current time of the clock (in seconds).

timestamp(*self: holoscan.resources._resources.ManualClock*) int

The current timestamp of the clock (in nanoseconds).

Parameters

timestamp

The current timestamp of the clock (in nanoseconds).

class holoscan.resources.MemoryStorageType

Bases: pybind11_builtins.pybind11_object

Members:

HOST

DEVICE

SYSTEM

Attributes

name	

value	

DEVICE = *<MemoryStorageType.DEVICE: 1>*

HOST = *<MemoryStorageType.HOST: 0>*

SYSTEM = *<MemoryStorageType.SYSTEM: 2>*

__init__(*self: holoscan.resources._resources.MemoryStorageType*, *value: int*) None

property name

property value

class holoscan.resources.RealtimeClock

Bases: holoscan.resources._resources.Clock

Real-time clock class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.

gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
gxf_initialize (self)	Initialize the component.
initialize (self)	Initialize the component.
set_time_scale (self, time_scale)	Adjust the time scaling used by the clock.

setup(self, spec)	Define the component specification.
sleep_for(self, arg0)	Set the GXF scheduler to sleep for a specified duration.
sleep_until(self, target_time_ns)	Set the GXF scheduler to sleep until a specified timestamp.
time(self)	The current time of the clock (in seconds).
timestamp(self)	The current timestamp of the clock (in nanoseconds).

__init__(*self: holoscan.resources._resources.RealtimeClock*, *fragment: holoscan.core._core.Fragment*, *initial_time_offset: float = 0.0*, *initial_time_scale: float = 1.0*, *use_time_since_epoch: bool = False*, *name: str = 'realtime_clock'*) None

Realtime clock.

Parameters

fragment

The fragment to assign the resource to.

initial_timestamp

The initial time offset used until time scale is changed manually.

initial_time_scale

The initial time scale used until time scale is changed manually.

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()`.

name

The name of the clock.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

set_time_scale(*self: holoscan.resources._resources.RealtimeClock*, *time_scale: float*) None

Adjust the time scaling used by the clock.

Parameters

time_scale

Durations (e.g. for periodic condition or sleep_for) are reduced by this scale value. A scale of 1.0 represents real-time while a scale of 2.0 would represent a clock where time elapses twice as fast.

setup(*self: holoscan.resources._resources.RealtimeClock*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

sleep_for(*self: holoscan.resources._resources.RealtimeClock*, *arg0: object*) None

Set the GXF scheduler to sleep for a specified duration.

Parameters

duration_ns

The duration to sleep (in nanoseconds).

sleep_until(*self: holoscan.resources._resources.RealtimeClock*, *target_time_ns: int*) None

Set the GXF scheduler to sleep until a specified timestamp.

Parameters

target_time_ns

The target timestamp (in nanoseconds).

property spec

time(*self: holoscan.resources._resources.RealtimeClock*) float

The current time of the clock (in seconds).

Parameters

time

The current time of the clock (in seconds).

timestamp(*self: holoscan.resources._resources.RealtimeClock*) int

The current timestamp of the clock (in nanoseconds).

Parameters

timestamp

The current timestamp of the clock (in nanoseconds).

class holoscan.resources.Receiver

Bases: holoscan.gxf._gxf.GXFResource

Base GXF receiver class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.

`gxf_initialize`(self)	Initialize the component.
`initialize`(self)	Initialize the component.
`setup`(self, arg0)	setup method for the resource.

__init__(*self: holoscan.resources._resources.Receiver*) None

Base GXF receiver class.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self:* *holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self:* *holoscan.core._core.Resource*, *arg0:* *holoscan.core._core.ComponentSpec*) None

setup method for the resource.

property spec

class holoscan.resources.SerializationBuffer

Bases: `holoscan.gxf._gxf.GXFResource`

Serialization Buffer.

Attributes

`args`	The list of arguments associated with the component.
`description`	YAML formatted string describing the resource.

fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
gxf_initialize (self)	Initialize the component.

initialize (self)	Initialize the component.
setup (self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.SerializationBuffer*, *fragment: holoscan.core._core.Fragment*, *allocator: holoscan.resources._resources.Allocator = None*, *buffer_size: int = 4096*, *name: str = 'serialization_buffer'*) None

Serialization Buffer.

Parameters

fragment

The fragment to assign the resource to.

allocator

The memory allocator for tensor components.

buffer_size

The size of the buffer in bytes.

name

The name of the serialization buffer

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self:* *holoscan.resources._resources.SerializationBuffer*, *spec:* *holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.StdComponentSerializer

Bases: holoscan.gxf._gxf.GXFResource

Serializer for GXF Timestamp and Tensor components.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.

name	The name of the resource.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the resource
setup(self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.StdComponentSerializer*, *fragment: holoscan.core._core.Fragment*, *name: str = 'standard_component_serializer'*) None

Serializer for GXF Timestamp and Tensor components.

Parameters

fragment

The fragment to assign the resource to.

name

The name of the serializer.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.resources._resources.StdComponentSerializer*) None

Initialize the resource

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

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.StdComponentSerializer*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.StdEntitySerializer

Bases: holoscan.gxf._gxf.GXFResource

Default serializer for GXF entities.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.

gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the resource
setup(self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.StdEntitySerializer*, *fragment: holoscan.core._core.Fragment*, *name: str = 'std_entity_serializer'*) None

Default serializer for GXF entities.

Parameters

fragment

The fragment to assign the resource to.

name

The name of the serializer.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.resources._resources.StdEntitySerializer*) None

Initialize the resource

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

property name

The name of the resource.

Returns

name

setup(*self:* *holoscan.resources._resources.StdEntitySerializer*, *spec:* *holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.Transmitter

Bases: holoscan.gxf._gxf.GXFResource

Base GXF transmitter class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.

`gxf_initialize`(self)	Initialize the component.
`initialize`(self)	Initialize the component.
`setup`(self, arg0)	setup method for the resource.

__init__(*self: holoscan.resources._resources.Transmitter*) None

Base GXF transmitter class.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self:* *holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self:* *holoscan.core._core.Resource*, *arg0:* *holoscan.core._core.ComponentSpec*) None

setup method for the resource.

property spec

class holoscan.resources.UcxComponentSerializer

Bases: `holoscan.gxf._gxf.GXFResource`

UCX component serializer.

Attributes

`args`	The list of arguments associated with the component.
`description`	YAML formatted string describing the resource.

fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
gxf_initialize (self)	Initialize the component.

initialize (self)	Initialize the component.
setup (self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.UcxComponentSerializer*, *fragment: holoscan.core._core.Fragment*, *allocator: holoscan.resources._resources.Allocator* = *None*, *name: str = 'ucx_component_serializer'*) None

UCX component serializer.

Parameters

fragment

The fragment to assign the resource to.

allocator

The memory allocator for tensor components.

name

The name of the component serializer.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.UcxComponentSerializer*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.UcxEntitySerializer

Bases: holoscan.gxf._gxf.GXFResource

UCX entity serializer.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the component.
setup(self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.UcxEntitySerializer*, *fragment: holoscan.core._core.Fragment*, *verbose_warning: bool = False*, *name: str = 'ucx_entity_serializer'*) None

UCX entity serializer.

Parameters

fragment

The fragment to assign the resource to.

component_serializer

The component serializers used by the entity serializer.

verbose_warning

Whether to use verbose warnings during serialization.

name

The name of the entity serializer.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.UcxEntitySerializer*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.UcxHoloscanComponentSerializer

Bases: holoscan.gxf._gxf.GXFResource

UCX Holoscan component serializer.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.

gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the component.
setup(self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.UcxHoloscanComponentSerializer*, *fragment: holoscan.core._core.Fragment*, *allocator: holoscan.resources._resources.Allocator* = *None*, *name: str = 'ucx_component_serializer'*) None

UCX Holoscan component serializer.

Parameters

fragment

The fragment to assign the resource to.

allocator

The memory allocator for tensor components.

name

The name of the component serializer.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self:* *holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self:* *holoscan.resources._resources.UcxHoloscanComponentSerializer*, *spec:* *holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.UcxReceiver

Bases: holoscan.resources._resources.Receiver

UCX network receiver using a double-buffered queue.

New messages are first pushed to a back stage.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args,	Overloaded function.

**kwargs)	
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the component.
setup(self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.UcxReceiver*, *fragment: holoscan.core._core.Fragment*, *buffer: holoscan::UcxSerializationBuffer = None*, *capacity: int = 1*, *policy: int = 2*, *address: str = '0.0.0.0'*, *port: int = 13337*, *name: str = 'ucx_receiver'*)

None

UCX network receiver using a double-buffered queue.

New messages are first pushed to a back stage.

Parameters

fragment

The fragment to assign the resource to.

buffer

The serialization buffer used by the transmitter.

capacity

The capacity of the receiver.

policy

The policy to use (0=pop, 1=reject, 2=fault).

address

The IP address used by the transmitter.

port

The network port used by the transmitter.

name

The name of the receiver.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.UcxReceiver*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.UcxSerializationBuffer

Bases: holoscan.gxf._gxf.GXFResource

UCX serialization buffer.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.

gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg (*args, **kwargs)	Overloaded function.
gxf_initialize (self)	Initialize the component.
initialize (self)	Initialize the component.

setup(self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.UcxSerializationBuffer*, *fragment: holoscan.core._core.Fragment*, *allocator: holoscan.resources._resources.Allocator* = *None*, *buffer_size: int = 4096*, *name: str = 'serialization_buffer'*) None

UCX serialization buffer.

Parameters

fragment

The fragment to assign the resource to.

allocator

The memory allocator for tensor components.

buffer_size

The size of the buffer in bytes.

name

The name of the serialization buffer

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.UcxSerializationBuffer*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.UcxTransmitter

Bases: holoscan.resources._resources.Transmitter

UCX network transmitter using a double-buffered queue.

Messages are pushed to a back stage after they are published.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the resource.
id	The identifier of the component.
name	The name of the resource.

spec	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the component.
setup(self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.UcxTransmitter*, *fragment: holoscan.core._core.Fragment*, *buffer: holoscan::UcxSerializationBuffer = None*, *capacity: int = 1*, *policy: int = 2*, *receiver_address: str = '0.0.0.0'*, *local_address: str = '0.0.0.0'*, *port: int = 13337*, *local_port: int = 0*, *maximum_connection_retries: int = 10*, *name: str = 'ucx_transmitter'*) None

UCX network transmitter using a double-buffered queue.

Messages are pushed to a back stage after they are published.

Parameters

fragment

The fragment to assign the resource to.

buffer

The serialization buffer used by the transmitter.

capacity

The capacity of the transmitter.

policy

The policy to use (0=pop, 1=reject, 2=fault).

receiver_address

The IP address used by the transmitter.

local_address

The local IP address to use for connection.

port

The network port used by the transmitter.

local_port

The local network port to use for connection.

maximum_connection_retries

The maximum number of times the transmitter will retry making a connection.

name

The name of the transmitter.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

property name

The name of the resource.

Returns

name

setup(*self:* *holoscan.resources._resources.UcxTransmitter*, *spec:* *holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

class holoscan.resources.UnboundedAllocator

Bases: holoscan.resources._resources.Allocator

Unbounded allocator.

This allocator uses dynamic memory allocation without an upper bound.

Attributes

args	The list of arguments associated with the component.
block_size	Get the block size of the allocator.
description	YAML formatted string describing the resource.
fragment	Fragment that the resource belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.

`gxf_typename`	The GXF type name of the resource.
`id`	The identifier of the component.
`name`	The name of the resource.

spec	

Methods

`add_arg` (*args, **kwargs)	Overloaded function.
`allocate` (self, size, type)	Allocate the requested amount of memory.
`free` (self, pointer)	Free the allocated memory
`gxf_initialize` (self)	Initialize the component.
`initialize` (self)	Initialize the component.
`is_available` (self, si	Boolean representing whether the resource is available.

ze)	
setup (self, spec)	Define the component specification.

__init__(*self: holoscan.resources._resources.UnboundedAllocator*, *fragment: holoscan.core._core.Fragment*, *name: str = 'unbounded_allocator'*) None

Unbounded allocator.

This allocator uses dynamic memory allocation without an upper bound.

Parameters

fragment

The fragment to assign the resource to.

name

The name of the serializer.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

allocate(*self: holoscan.resources._resources.Allocator*, *size: int*, *type: holoscan.resources._resources.MemoryStorageType*) int

Allocate the requested amount of memory.

Parameters

size

The amount of memory to allocate

type

Enum representing the type of memory to allocate.

Returns

Opaque PyCapsule object representing a std::byte* pointer to the allocated memory.

property args

The list of arguments associated with the component.

Returns

arglist

property block_size

Get the block size of the allocator.

Returns

int

The block size of the allocator. Returns 1 for byte-based allocators.

property description

YAML formatted string describing the resource.

property fragment

Fragment that the resource belongs to.

Returns

name

free(*self: holoscan.resources._resources.Allocator*, *pointer: int*) None

Free the allocated memory

Parameters

pointer

Opaque PyCapsule object representing a std::byte* pointer to the allocated memory.

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the resource.

Returns

str

The GXF type name of the resource

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFResource*) None

Initialize the component.

is_available(*self: holoscan.resources._resources.Allocator*, *size: int*) bool

Boolean representing whether the resource is available.

Returns

bool

Availability of the resource.

property name

The name of the resource.

Returns

name

setup(*self: holoscan.resources._resources.UnboundedAllocator*, *spec: holoscan.core._core.ComponentSpec*) None

Define the component specification.

Parameters

spec

Component specification associated with the resource.

property spec

holoscan.schedulers

This module provides a Python API to underlying C++ API Schedulers.

holoscan.schedulers.EventBasedScheduler	Event-based multi-thread scheduler class.
holoscan.schedulers.GreedyScheduler	GreedyScheduler scheduler class.
holoscan.schedulers.MultiThreadScheduler	Multi-thread scheduler class.

class holoscan.schedulers.EventBasedScheduler

Bases: holoscan.gxf._gxf.GXFScheduler, holoscan.core._core.Component, holoscan.gxf._gxf.GXFComponent

Event-based multi-thread scheduler class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the component.
fragment	Fragment that the scheduler belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the scheduler.
id	The identifier of the component.
name	The name of the scheduler.

clock	
max_duration_ms	
spec	
stop_on_deadlock	
stop_on_deadlock_timeout	

worker_thread_number	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the scheduler.
setup(self, arg0)	setup method for the scheduler.

__init__(*self: holoscan.schedulers._schedulers.EventBasedScheduler*, *fragment: holoscan.core._core.Fragment*, *, *clock: holoscan.resources._resources.Clock = None*, *worker_thread_number: int = 1*, *stop_on_deadlock: bool = True*, *max_duration_ms: int = -1*, *stop_on_deadlock_timeout: int = 0*, *name: str = 'multithread_scheduler'*) None

Event-based multi-thread scheduler

Parameters

fragment

The fragment the condition will be associated with

clock

The clock used by the scheduler to define the flow of time. If None, a default-constructed *holoscan.resources.RealtimeClock* will be used.

worker_thread_number

The number of worker threads.

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.

max_duration_ms

The maximum duration for which the scheduler will execute (in ms). If not specified (or if a negative value is provided), the scheduler will run until all work is done. If periodic terms are present, this means the application will run indefinitely.

stop_on_deadlock_timeout

The scheduler will wait this amount of time before determining that it is in deadlock and should stop. It will reset if a job comes in during the wait. A negative value means not stop on deadlock. This parameter only applies when *stop_on_deadlock=true*",

name

The name of the scheduler.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property clock

property description

YAML formatted string describing the component.

property fragment

Fragment that the scheduler belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the scheduler.

Returns

str

The GXF type name of the scheduler

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.gxf._gxf.GXFScheduler*) None

Initialize the scheduler.

property max_duration_ms

property name

The name of the scheduler.

Returns

name

setup(*self:* *holoscan.core._core.Scheduler*, *arg0:* *holoscan.core._core.ComponentSpec*) None

setup method for the scheduler.

property spec

property stop_on_deadlock

property stop_on_deadlock_timeout

property worker_thread_number

class holoscan.schedulers.GreedyScheduler

Bases: holoscan.gxf._gxf.GXFScheduler, holoscan.core._core.Component, holoscan.gxf._gxf.GXFComponent

GreedyScheduler scheduler class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the component.
fragment	Fragment that the scheduler belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.
gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the scheduler.
id	The identifier of the component.

name	The name of the scheduler.

check_recession_period_ms	
clock	
max_duration_ms	
spec	
stop_on_deadlock	
stop_on_deadlock_timeout	

Methods

add_arg(*args, **kwargs)	Overloaded function.
gxf_initialize(self)	Initialize the component.
initialize(self)	Initialize the scheduler.
setup(self, arg0)	setup method for the scheduler.

__init__(*self: holoscan.schedulers._schedulers.GreedyScheduler*, *fragment: holoscan.core._core.Fragment*, *, *clock: holoscan.resources._resources.Clock = None*, *stop_on_deadlock: bool = True*, *max_duration_ms: int = - 1*, *check_recession_period_ms: float = 0.0*, *stop_on_deadlock_timeout: int = 0*, *name: str = 'greedy_scheduler'*) None

Greedy scheduler

Parameters

fragment

The fragment the condition will be associated with

clock

The clock used by the scheduler to define the flow of time. If None, a default-constructed *holoscan.resources.RealtimeClock* will be used.

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.

max_duration_ms

The maximum duration for which the scheduler will execute (in ms). If not specified (or if a negative value is provided), the scheduler will run until all work is done. If periodic terms are present, this means the application will run indefinitely.

check_recession_period_ms

The maximum duration for which the scheduler would wait (in ms) when all operators are not ready to run in the current iteration.

stop_on_deadlock_timeout

The scheduler will wait this amount of time before determining that it is in deadlock and should stop. It will reset if a job comes in during the wait. A negative value means not stop on deadlock. This parameter only applies when *stop_on_deadlock=true*",

name

The name of the scheduler.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property check_recession_period_ms

property clock

property description

YAML formatted string describing the component.

property fragment

Fragment that the scheduler belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self: holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the scheduler.

Returns

str

The GXF type name of the scheduler

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self: holoscan.gxf._gxf.GXFScheduler*) None

Initialize the scheduler.

property max_duration_ms

property name

The name of the scheduler.

Returns

name

setup(*self: holoscan.core._core.Scheduler*, *arg0: holoscan.core._core.ComponentSpec*) None

setup method for the scheduler.

property spec

property stop_on_deadlock

property stop_on_deadlock_timeout

class holoscan.schedulers.MultiThreadScheduler

Bases: holoscan.gxf._gxf.GXFScheduler, holoscan.core._core.Component, holoscan.gxf._gxf.GXFComponent

Multi-thread scheduler class.

Attributes

args	The list of arguments associated with the component.
description	YAML formatted string describing the component.
fragment	Fragment that the scheduler belongs to.
gxf_cid	The GXF component ID.
gxf_cname	The name of the component.

gxf_context	The GXF context of the component.
gxf_eid	The GXF entity ID.
gxf_typename	The GXF type name of the scheduler.
id	The identifier of the component.
name	The name of the scheduler.

check_recession_period_ms	
clock	
max_duration_ms	
spec	
stop_on_deadlock	
stop_on_deadlock_timeout	
worker_thread_number	

Methods

add_arg (*args, **kwargs)	Overloaded function.
gxf_initialize (self)	Initialize the component.

initialize (self)	Initialize the scheduler.
setup (self, arg0)	setup method for the scheduler.

__init__(*self: holoscan.schedulers._schedulers.MultiThreadScheduler*, *fragment: holoscan.core._core.Fragment*, ***, *clock: holoscan.resources._resources.Clock = None*, *worker_thread_number: int = 1*, *stop_on_deadlock: bool = True*, *check_recession_period_ms: float = 5.0*, *max_duration_ms: int = - 1*, *stop_on_deadlock_timeout: int = 0*, *name: str = 'multithread_scheduler'*) None

Multi-thread scheduler

Parameters

fragment

The fragment the condition will be associated with

clock

The clock used by the scheduler to define the flow of time. If None, a default-constructed *holoscan.resources.RealtimeClock* will be used.

worker_thread_number

The number of worker threads.

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.

check_recession_period_ms

The maximum duration for which the scheduler would wait (in ms) when an operator is not ready to run yet.

max_duration_ms

The maximum duration for which the scheduler will execute (in ms). If not specified (or if a negative value is provided), the scheduler will run until all work is done. If periodic terms are present, this means the application will run indefinitely.

stop_on_deadlock_timeout

The scheduler will wait this amount of time before determining that it is in deadlock and should stop. It will reset if a job comes in during the wait. A negative value means not stop on deadlock. This parameter only applies when *stop_on_deadlock=true*",

name

The name of the scheduler.

add_arg(**args*, ***kwargs*)

Overloaded function.

1. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.Arg) -> None

Add an argument to the component.

2. add_arg(self: holoscan.core._core.ComponentBase, arg: holoscan.core._core.ArgList) -> None

Add a list of arguments to the component.

property args

The list of arguments associated with the component.

Returns

arglist

property check_recession_period_ms

property clock

property description

YAML formatted string describing the component.

property fragment

Fragment that the scheduler belongs to.

Returns

name

property gxf_cid

The GXF component ID.

property gxf_cname

The name of the component.

property gxf_context

The GXF context of the component.

property gxf_eid

The GXF entity ID.

gxf_initialize(*self:* *holoscan.gxf._gxf.GXFComponent*) None

Initialize the component.

property gxf_typename

The GXF type name of the scheduler.

Returns

str

The GXF type name of the scheduler

property id

The identifier of the component.

The identifier is initially set to `-1`, and will become a valid value when the component is initialized.

With the default executor (*holoscan.gxf.GXFExecutor*), the identifier is set to the GXF component ID.

Returns

id

initialize(*self:* *holoscan.gxf._gxf.GXFScheduler*) None

Initialize the scheduler.

property max_duration_ms

property name

The name of the scheduler.

Returns

name

setup(*self:* *holoscan.core._core.Scheduler*, *arg0:* *holoscan.core._core.ComponentSpec*) None

setup method for the scheduler.

property spec

property stop_on_deadlock

property stop_on_deadlock_timeout

property worker_thread_number

Holoscan Application Package Specification (HAP)

Introduction

The Holoscan Application Package specification extends the MONAI Deploy Application Package specification to provide the streaming capabilities, multi-fragment and other features of the Holoscan SDK.

Overview

This document includes the specification of the Holoscan Application Package (HAP). A HAP is a containerized application or service which is self-descriptive, as defined by this document.

Goal

This document aims to define the structure and purpose of a HAP, including which parts are optional and which are required so that developers can easily create conformant HAPs.

Assumptions, Constraints, Dependencies

The following assumptions relate to HAP execution, inspection and general usage:

- Containerized applications will be based on Linux x64 (AMD64) and/or ARM64 (aarch64).
- Containerized applications' host environment will be based on Linux x64 (AMD64) and/or ARM64 (aarch64) with container support.
- Developers expect the local execution of their applications to behave identically to the execution of the containerized version.

- Developers expect the local execution of their containerized applications to behave identically to the execution in deployment.
- Developers and operations engineers want the application packages to be self-describing.
- Applications may be created using tool other than that provided in the Holoscan SDK or the MONAI Deploy App SDK.
- Holoscan Application Package may be created using a tool other than that provided in the Holoscan SDK or the MONAI Deploy App SDK.
- Pre-existing, containerized applications must be “converted” into Holoscan Application Packages.
- A Holoscan Application Package may contain a classical application (non-fragment based), a single-fragment application, or a multi-fragment application. (Please see the definition of fragment in Definitions, Acronyms, Abbreviations)
- The scalability of a multi-fragment application based on Holoscan SDK is outside the scope of this document.
- Application packages are expected to be deployed in one of the supported environments. For additional information, see Holoscan Operating Environments.

Definitions, Acronyms, Abbreviations

Term	Definition
ARM64	Or, AARCH64. See Wikipedia for details.
Container	See What’s a container?
Fragment	A fragment is a building block of the Application. It is a directed graph of operators. For details, please refer to the MONAI Deploy App SDK or Holoscan App SDK.
Gigibytes (GiB)	A gibibyte (GiB) is a unit of measurement used in computer data storage that equals to 1,073,741,824 bytes.
HAP	Holoscan Application Package. A containerized application or service which is self-descriptive.

Hosting Service	A service that hosts and orchestrates HAP containers.
MAP	MONAI Application Package. A containerized application or service which is self-descriptive.
Mebibytes (MiB)	A mebibyte (MiB) is a unit of measurement used in computer data storage that equals to 1,048,576 bytes.
MONAI	Medical Open Network for Artificial Intelligence.
SDK	Software Development Kit.
Semantic Version	See Semantic Versioning 2.0.
x64	Or, x86-64 or AMD64. See Wikipedia for details.

Requirements

The following requirements MUST be met by the HAP specification to be considered complete and approved. All requirements marked as `MUST` or `SHALL` MUST be implemented in order to be supported by a HAP-ready hosting service.

Single Artifact

- A HAP SHALL comprise a single container, meeting the minimum requirements set forth by this document.
- A HAP SHALL be a containerized application to maximize the portability of its application.

Self-Describing

- A HAP MUST be self-describing and provide a mechanism for extracting its description.
    - A HAP SHALL provide a method to print the metadata files to the console.
    - A HAP SHALL provide a method to copy the metadata files to a user-specified directory.
- The method of description SHALL be in a machine-readable and writable format.

- The method of description SHALL be in a human-readable format.
- The method of description SHOULD be a human writable format.
- The method of description SHALL be declarative and immutable.
- The method of description SHALL provide the following information about the HAP:
    - Execution requirements such as dependencies and restrictions.
    - Resource requirements include CPU cores, system memory, shared memory, GPU, and GPU memory.

Runtime Characteristics of the HAP

- A HAP SHALL start the packaged Application when it is executed by the users when arguments are specified.
- A HAP SHALL describe the packaged Application as a long-running service or an application so an external agent can manage its lifecycle.

IO Specification

- A HAP SHALL provide information about its expected inputs such that an external agent can determine if the HAP can receive a workload.
- A HAP SHALL provide sufficient information about its outputs so that an external agent knows how to handle the results.

Local Execution

A HAP MUST be in a format that supports local execution in a development environment.

[Note] See Holoscan Operating Environments for additional information about supported environments.

Compatible with Kubernetes

- A HAP SHALL support deployment using Kubernetes.

OCI Compliance

The containerized portion of a HAP SHALL comply with Open Container Initiative format standards.

Image Annotations

All annotations for the containerized portion of a HAP MUST adhere to the specifications laid out by The OpenContainers Annotations Spec

- `org.opencontainers.image.title`: A HAP container image SHALL provide a human-readable title (string).
- `org.opencontainers.image.version`: A HAP container image SHALL provide a version of the packaged application using the semantic versioning format. This value is the same as the value defined in `/etc/holoscan/app.json#version` in the Table of Application Manifest Fields.
- All other OpenContainers predefined keys SHOULD be provided when available.

Hosting Environment

The HAP Hosting Environment executes the HAP and provides the application with a customized set of environment variables and command line options as part of the invocation.

- The Hosting Service MUST, by default, execute the application as defined by `/etc/holoscan/app.json#command` and then exit when the application or the service completes.
- The Hosting Service MUST provide any environment variables specified by `/etc/holoscan/app.json#environment`.
- The Hosting Service SHOULD monitor the Application process and record its CPU, system memory, and GPU utilization metrics.
- The Hosting Service SHOULD monitor the Application process and enforce any timeout value specified in `/etc/holoscan/app.json#timeout`.

Table of Environment Variables

A HAP SHALL contain the following environment variables and their default values, if not specified by the user, in the Application Manifest `/etc/holoscan/app.json#environment`.

Variable	Default	Format	Description
HOLOSCAN_INPUT_PATH	/var/holoscan/input/	Folder Path	Path to the input folder for the Application.
HOLOSCAN_OUTPUT_PATH	/var/holoscan/output/	Folder Path	Path to the output folder for the Application.
HOLOSCAN_WORKDIR	/var/holoscan/	Folder Path	Path to the Application's working directory.
HOLOSCAN_MODEL_PATH	/opt/holoscan/models/	Folder Path	Path to the Application's models directory.
HOLOSCAN_CONFIG_PATH	/var/holoscan/app.yaml	File Path	Path to the Application's configuration file.
HOLOSCAN_APP_MANIFEST_PATH	/etc/holoscan/app.config	File Path	Path to the Application's configuration file.
HOLOSCAN_PKG_MANIFEST_PATH	/etc/holoscan/pkg.config	File Path	Path to the Application's configuration file.
HOLOSCAN_DOCS	/opt/holoscan/docs	Folder Path	Path to the folder containing application documentation and licenses.
HOLOSCAN_LOGS	/var/holoscan/logs	Folder Path	Path to the Application's logs.

Architecture & Design

Description

The Holoscan Application Package (HAP) is a functional package designed to act on datasets of a prescribed format. A HAP is a container image that adheres to the specification provided in this document.

Application

The primary component of a HAP is the application. The application is provided by an application developer and incorporated into the HAP using the Holoscan Application Packager.

All application code and binaries SHALL be in the `/opt/holoscan/app/` folder, except for any dependencies installed by the Holoscan Application Packager during the creation of the HAP.

All AI models (PyTorch, TensorFlow, TensorRT, etc.) SHOULD be in separate sub-folders of the `/opt/holoscan/models/` folder. In specific use cases where the app package developer is prevented from enclosing the model files in the package/container due to intellectual property concerns, the models can be supplied from the host system when the app package is run, e.g., via the volume mount mappings and the use of container env variables.

Manifests

A HAP SHALL contain two manifests: the Application Manifest and the Package Manifest. The Package Manifest shall be stored in `/etc/holoscan/pkg.json`, and the Application Manifest shall be stored in `/etc/holoscan/app.json`. Once a HAP is created, its manifests are expected to be immutable.

Application Manifest

Table of Application Manifest Fields

Name	Required	Default	Type	Format	Description
`apiVersion`	No	0.0.0	string	semantic version	Version of the manifest file schema.
`command`	**Yes**	N/A	string	shell command	Shell command used to run the Application.
`environment`	**No**	N/A	object	object w/ name-value pairs	An object of name-value pairs that will be passed to the application during execution.
`input`	**Yes**	N/A	object	object	Data structure which provides information about Application

					inputs.
`input.formats`	**Yes**	N/A	array	array of objects	List of input data formats accepted by the Application.
`input.path`	No	input/	string	relative file-system path	Folder path relative to the working directory from which the application will read inputs.
`readiness`	No	N/A	object	object	An object of name-value pairs that defines the readiness probe.
`readiness.type`	**Yes**	N/A	string	string	Type of the probe: `tcp`, `grpc`, `http-get` or `command`.
`readiness.command`	**Yes** (when type is `command`)	N/A	array	shell command	Shell command and arguments in string array form.
`readiness.port`	**Yes** (when type is `tcp`, `grpc`, or `http-get`)	N/A	integer	number	The port number of readiness probe.
`readiness.path`	**Yes** (when type is `http-get`)	N/A	string	string	HTTP path and query to access the readiness probe.
`readiness.initialDelaySeconds`	No	1	integer	number	Number of seconds after the container has started before the readiness probe is initialized and performed.
`readiness.periodSeconds`	No	10	integer	number	Number of seconds between performing the readiness probe.
`readiness.timeoutSeconds`	No	1	integer	number	Number of seconds after which the probe times out.

`readiness.failureThreshold`	No	3	integer	number	Number of retries to be performed before considering the application is unhealthy.
`liveness`	No	N/A	object	object	An object of name-value pairs that defines the liveness probe. Recommended for service applications.
`liveness.type`	**Yes**	N/A	string	string	Type of the probe: `tcp`, `grpc`, `http-get` or `command`.
`liveness.command`	**Yes** (when type is `command`)	N/A	array	shell command	Shell command and arguments in string array form.
`liveness.port`	**Yes** (when type is `tcp`, `grpc`, or `http-get`)	N/A	integer	number	The port number of the liveness probe.
`liveness.path`	**Yes** (when type is `http-get`)	N/A	string	string	HTTP path and query to access the liveness probe.
`liveness.initialDelaySeconds`	No	1	integer	number	Number of seconds after the container has started before the liveness probe is initialized and performed.
`liveness.periodSeconds`	No	10	integer	number	Number of seconds between performing the liveness probe.
`liveness.timeoutSeconds`	No	1	integer	number	Number of seconds after which the probe times out.

`liveness.failureThreshold`	No	3	integer	number	Number of retries to be performed before considering the application is unhealthy.
`output`	**Yes**	N/A	object	object	Data structure which provides information about Application output.
`output.format`	**Yes**	N/A	object	object	Details about the format of the outputs produced by the application.
`output.path`	No	output/	string	relative file-system path	Folder path relative to the working directory to which the application will write outputs.
`sdk`	No	N/A	string	string	SDK used for the Application.
`sdkVersion`	No	0.0.0	string	semantic version	Version of the SDK used the Application.
`timeout`	No	0	integer	number	The maximum number of seconds the application should execute before being timed out and terminated. Recommended for a single batch/execution type of applications.
`version`	No	0.0.0	string	semantic version	Version of the Application.
`workingDirectory`	No	/var/holoscan/	string	absolute file-system path	Folder, or directory, in which the application will be executed.

The Application Manifest file provides information about the HAP's Application.

- The Application Manifest MUST define the type of the containerized application (`/etc/holoscan/app.json#type`).

    - Type SHALL have the value of either `service` or `application.`

- The Application Manifest MUST define the command used to run the Application (`/etc/holoscan/app.json#command`).

- The Application Manifest SHOULD define the version of the manifest file schema (`/etc/holoscan/app.json#apiVersion`).

    - The Manifest schema version SHALL be provided as a semantic version string.

    - When not provided, the default value `0.0.0` SHALL be assumed.

- The Application Manifest SHOULD define the SDK used to create the Application (`/etc/holoscan/app.json#sdk`).

- The Application Manifest SHOULD define the version of the SDK used to create the Application (`/etc/holoscan/app.json#sdkVersion`).

    - SDK version SHALL be provided as a semantic version string.

    - When not provided, the default value `0.0.0` SHALL be assumed.

- The Application Manifest SHOULD define the version of the application itself (`/etc/holoscan/app.json#version`).

    - The Application version SHALL be provided as a semantic version string.

    - When not provided, the default value `0.0.0` SHALL be assumed.

- The Application Manifest SHOULD define the application's working directory (`/etc/holoscan/app.json#workingDirectory`).

    - The Application will execute with its current directory set to this value.

    - The value provided must be an absolute path (the first character is `/`).

    - The default path `/var/holoscan/` SHALL be assumed when not provided.

- The Application Manifest SHOULD define the data input path, relative to the working directory, used by the Application (`/etc/holoscan/app.json#input.path`).

    - The input path SHOULD be a relative to the working directory or an absolute file-system path to a directory.
        - When the value is a relative file-system path (the first character is not `/`), it is relative to the application's working directory.
        - When the value is an absolute file-system path (the first character is `/`), the file-system path is used as-is.
    - When not provided, the default value `input/` SHALL be assumed.
- The Application Manifest SHOULD define input data formats supported by the Application (`/etc/holoscan/app.json#input.formats`).
    - Possible values include, but are not limited to, `none`, `network`, `file`.
- The Application Manifest SHOULD define the output path relative to the working directory used by the Application (`/etc/holoscan/app.json#output.path`).
    - The output path SHOULD be relative to the working directory or an absolute file-system path to a directory.
        - When the value is a relative file-system path (the first character is not `/`), it is relative to the application's working directory.
        - When the value is an absolute file-system path (the first character is `/`), the file-system path is used as-is.
    - When not provided, the default value `output/` SHALL be assumed.
- The Application Manifest SHOULD define the output data format produced by the Application (`/etc/holoscan/app.json#output.format`).
    - Possible values include, but are not limited to, `none`, `screen`, `file`, `network`.
- The Application Manifest SHOULD configure a check to determine whether or not the application is "ready."
    - The Application Manifest SHALL define the probe type to be performed (`/etc/holoscan/app.json#readiness.type`).

        - Possible values include `tcp`, `grpc`, `http-get`, and `command`.
    - The Application Manifest SHALL define the probe commands to execute when the type is `command` (`/etc/holoscan/app.json#readiness.command`).
        - The data structure is expected to be an array of strings.
    - The Application Manifest SHALL define the port to perform the readiness probe when the type is `grpc`, `tcp`, or `http-get`. (`/etc/holoscan/app.json#readiness.port`)
        - The value provided must be a valid port number ranging from 1 through 65535. (Please note that port numbers below 1024 are root-only privileged ports.)
    - The Application Manifest SHALL define the path to perform the readiness probe when the type is `http-get` (`/etc/holoscan/app.json#readiness.path`).
        - The value provided must be an absolute path (the first character is `/`).
    - The Application Manifest SHALL define the number of seconds after the container has started before the readiness probe is initiated. (`/etc/holoscan/app.json#readiness.initialDelaySeconds`).
        - The default value `0` SHALL be assumed when not provided.
    - The Application Manifest SHALL define how often to perform the readiness probe (`/etc/holoscan/app.json#readiness.periodSeconds`).
        - When not provided, the default value `10` SHALL be assumed.
    - The Application Manifest SHALL define the number of seconds after which the probe times out (`/etc/holoscan/app.json#readiness.timeoutSeconds`)
        - When not provided, the default value `1` SHALL be assumed.
    - The Application Manifest SHALL define the number of times to perform the probe before considering the service is not ready (`/etc/holoscan/app.json#readiness.failureThreshold`)

        - The default value `3` SHALL be assumed when not provided.

- The Application Manifest SHOULD configure a check to determine whether or not the application is “live” or not.
    - The Application Manifest SHALL define the type of probe to be performed (`/etc/holoscan/app.json#liveness.type`).
        - Possible values include `tcp` , `grpc` , `http-get` , and `command` .
    - The Application Manifest SHALL define the probe commands to execute when the type is `command` (`/etc/holoscan/app.json#liveness.command`).
        - The data structure is expected to be an array of strings.
    - The Application Manifest SHALL define the port to perform the liveness probe when the type is `grpc` , `tcp` , or `http-get` . (`/etc/holoscan/app.json#liveness.port`)
        - The value provided must be a valid port number ranging from 1 through 65535. (Please note that port numbers below 1024 are root-only privileged ports.)
    - The Application Manifest SHALL define the path to perform the liveness probe when the type is `http-get` (`/etc/holoscan/app.json#liveness.path`).
        - The value provided must be an absolute path (the first character is `/`).
    - The Application Manifest SHALL define the number of seconds after the container has started before the liveness probe is initiated. (`/etc/holoscan/app.json#liveness.initialDelaySeconds`).
        - The default value `0` SHALL be assumed when not provided.
    - The Application Manifest SHALL define how often to perform the liveness probe (`/etc/holoscan/app.json#liveness.periodSeconds`).
        - When not provided, the default value `10` SHALL be assumed.

    - The Application Manifest SHALL define the number of seconds after which the probe times out (`/etc/holoscan/app.json#liveness.timeoutSeconds`)
        - The default value `1` SHALL be assumed when not provided.
    - The Application Manifest SHALL define the number of times to perform the probe before considering the service is not alive (`/etc/holoscan/app.json#liveness.failureThreshold`)
        - When not provided, the default value `3` SHALL be assumed.
- The Application Manifest SHOULD define any timeout applied to the Application (`/etc/holoscan/app.json#timeout`).
    - When the value is `0` , timeout SHALL be disabled.
    - When not provided, the default value `0` SHALL be assumed.
- The Application Manifest MUST enable the specification of environment variables for the Application (`/etc/holoscan/app.json#environment`)
    - The data structure is expected to be in `"name": "value"` members of the object.
    - The field's name will be the name of the environment variable verbatim and must conform to all requirements for environment variables and JSON field names.
    - The field's value will be the value of the environment variable and must conform to all requirements for environment variables.

Package Manifest

Table of Package Manifest Fields

Name	Required	Default	Type	Format	Description
`apiVersion`	No	`0.0.0`	string	semantic version	Version of the manifest file schema.

`applicationRoot`	**Yes**	`/opt/holoscan/app/`	string	absolute file-system path	Absolute file-system path to the folder which contains the Application
`modelRoot`	No	`/opt/holoscan/models/`	string	absolute file-system path	Absolute file-system path to the folder which contains the model(s).
`models`	No	N/A	array	array of objects	Array of objects which describe models in the package.
`models[*].name`	**Yes**	N/A	string	string	Name of the model.
`models[*].path`	No	N/A	string	Relative file-system path	File-system path to the folder which contains the model that is relative to the value defined in `modelRoot`.
`resources`	No	N/A	object	object	Object describing resource requirements for the Application.
`resources.cpu`	No	`1`	decimal (2)	number	Number of CPU cores required by the Application or the Fragment.
`resources.cpuLimit`	No	N/A	decimal (2)	number	The CPU core limit for the Application or the Fragment. (1)
`resources.gpu`	No	`0`	decimal (2)	number	Number of GPU devices required by the Application or the Fragment.
`resources.gpuLimit`	No	N/A	decimal (2)	number	The GPU device limit for the Application or the Fragment. (1)
`resources.memory`	No	`1Gi`	string	memory size	The memory required by the Application or the Fragment.

`resources.memoryLimit`	No	N/A	string	memory size	The memory limit for the Application or the Fragment. (1)
`resources.gpuMemory`	No	N/A	string	memory size	The GPU memory required by the Application or the Fragment.
`resources.gpuMemoryLimit`	No	N/A	string	memory size	The GPU memory limit for the Application or the Fragment. (1)
`resources.sharedMemory`	No	`64Mi`	string	memory size	The shared memory required by the Application or the Fragment.
`resources.fragments`	No	N/A	object	objects	Nested objects which describe resources for a Multi-Fragment Application.
`resources.fragments.&lt;fragment-name&gt;`	**Yes**	N/A	string	string	Name of the fragment.
`resources.fragments.&lt;fragment-name&gt;.cpu`	No	`1`	decimal (2)	number	Number of CPU cores required by the Fragment.
`resources.fragments.&lt;fragment-name&gt;.cpuLimit`	No	N/A	decimal (2)	number	The CPU core limit for the Fragment. (1)
`resources.fragments.&lt;fragment-name&gt;.gpu`	No	`0`	decimal (2)	number	Number of GPU devices required by the Fragment.

resources.fragments.<fragment-name>.gpuLimit	No	N/A	decimal (2)	number	The GPU device limit for the Fragment. (1)
resources.fragments.<fragment-name>.memory	No	1Gi	string	memory size	The memory required by the Fragment.
resources.fragments.<fragment-name>.memoryLimit	No	N/A	string	memory size	The memory limit for the Fragment. (1)
resources.fragments.<fragment-name>.gpuMemory	No	N/A	string	memory size	The GPU memory required by the Fragment.
resources.fragments.<fragment-name>.gpuMemoryLimit	No	N/A	string	memory size	The GPU memory limit for the Fragment. (1)
resources.fragments.<fragment-name>.sharedMemory	No	64Mi	string	memory size	The shared memory required by the Fragment.
version	No	0.0.0	string	semantic version	Version of the package.

[Notes] (1) Use of resource limits depend on the orchestration service or the hosting environement's configuration and implementation. (2) Consider rounding up to a whole number as decimal values may not be supported by all orchestration/hosting services.

The Package Manifest file provides information about the HAP's package layout. It is not intended as a mechanism for controlling how the HAP is used or how the HAP's Application is executed.

- The Package Manifest MUST be UTF-8 encoded and use the JavaScript Object Notation (JSON) format.
- The Package Manifest SHOULD support either CRLF or LF style line endings.
- The Package Manifest SHOULD specify the folder which contains the application (`/etc/holoscan/pkg.json#applicationRoot`).
    - When not provided, the default path `/opt/holoscan/app/` will be assumed.
- The Package Manifest SHOULD provide the version of the package file manifest schema (`/etc/holoscan/pkg.json#apiVersion`).
    - The Manifest schema version SHALL be provided as a semantic version string.
- The Package Manifest SHOULD provide the package version of itself (`/etc/holoscan/pkg.json#version`).
    - The Package version SHALL be provided as a semantic version string.
- The Package Manifest SHOULD provide the directory path to the user-provided models. (`/etc/holoscan/pkg.json#modelRoot`).
    - The value provided must be an absolute path (the first character is `/`).
    - When not provided, the default path `/opt/holoscan/models/` SHALL be assumed.
- The Package Manifest SHOULD list the models used by the application (`/etc/holoscan/pkg.json#models`).
    - Models SHALL be defined by name (`/etc/holoscan/pkg.json#models[*].name`).

        - Model names SHALL NOT contain any Unicode whitespace or control characters.
        - Model names SHALL NOT exceed 128 bytes in length.
    - Models SHOULD provide a file-system path if they're included in the HAP itself (`/etc/holoscan/pkg.json#models[*].path`).
        - When the value is a relative file-system path (the first character is not `/`), it is relative to the model root directory defined in `/etc/holoscan/pkg.json#modelRoot`.
        - When the value is an absolute file-system path (the first character is `/`), the file-system path is used as-is.
        - When no value is provided, the name is assumed as the name of the directory relative to the model root directory defined in `/etc/holoscan/pkg.json#modelRoot`.
- The Package Manifest SHOULD specify the resources required to execute the Application and the fragments for a Multi-Fragment Application.

  This information is used to provision resources when running the containerized application using a compatible application deployment service.
- A classic Application or a single Fragment Application SHALL define its resources in the `/etc/holoscan/pkg.json#resource` object.
    - The `/etc/holoscan/pkg.json#resource` object is for the whole application. It CAN also be used as a catchall for all fragments in a multi-fragment application where applicable.
    - CPU requirements SHALL be denoted using the decimal count of CPU cores (`/etc/holoscan/pkg.json#resources.cpu`).
    - Optional CPU limits SHALL be denoted using the decimal count of CPU cores (`/etc/holoscan/pkg.json#resources.cpuLimit`)
    - GPU requirements SHALL be denoted using the decimal count of GPUs (`/etc/holoscan/pkg.json#resources.gpu`).

- Optional GPU limits SHALL be denoted using the decimal count of GPUs (`/etc/holoscan/pkg.json#resources.gpuLimit`)
- Memory requirements SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.memory`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi` , `2048Mi`
- Optional memory limits SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.memoryLimit`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi` , `2048Mi`
- GPU memory requirements SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.gpuMemory`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi` , `2048Mi`
- Optional GPU memory limits SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.gpuMemoryLimit`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi` , `2048Mi`
- Shared memory requirements SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.sharedMemory`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi` , `2048Mi`

    - Optional shared memory limits SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.sharedMemoryLimit`).
        - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
            - Example: `1.5Gi` , `2048Mi`
    - Integer values MUST be positive and not contain any position separators.
        - Example legal values: `1` , `42` , `2048`
        - Example illegal values: `-1` , `1.5` , `2,048`
    - Decimal values MUST be positive, rounded to the nearest tenth, use the dot (`.`) character to separate whole and fractional values, and not contain any positional separators.
        - Example legal values: `1` , `1.0` , `0.5` , `2.5` , `1024`
        - Example illegal values: `1,024` , `-1.0` , `3.14`
    - When not provided, the default values of `cpu=1` , `gpu=0` , `memory="1Gi"` , and `sharedMemory="64Mi"` will be assumed.
- A Multi-Fragment Application SHOULD define its resources in the `/etc/holoscan/pkg.json#resource.fragments.&lt;fragment-name&gt;` object.
    - When a matching `fragment-name` cannot be found, the `/etc/holoscan/pkg.json#resource` definition is used.
    - Fragment names (`fragment-name`) SHALL NOT contain any Unicode whitespace or control characters.
    - Fragment names (`fragment-name`) SHALL NOT exceed 128 bytes in length.
    - CPU requirements for fragments SHALL be denoted using the decimal count of CPU cores (`/etc/holoscan/pkg.json#resources.fragments.&lt;fragment-name&gt;.cpu`).

- Optional CPU limits for fragments SHALL be denoted using the decimal count of CPU cores (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.cpuLimit`).
- GPU requirements for fragments SHALL be denoted using the decimal count of GPUs (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.gpu`).
- Optional GPU limits for fragments SHALL be denoted using the decimal count of GPUs (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.gpuLimit`).
- Memory requirements for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.memory`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi`, `2048Mi`
- Optional memory limits for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.memoryLimit`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi`, `2048Mi`
- GPU memory requirements for fragments SHALL be denoted using decimal values followed by units (

`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.gpuMemory`
).

- Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
  - Example: `1.5Gi` , `2048Mi`

- Optional GPU memory limits for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.gpuMemoryLimit`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi` , `2048Mi`
- Shared memory requirements for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.sharedMemory`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi` , `2048Mi`
- Optional shared memory limits for fragments SHALL be denoted using decimal values followed by units (`/etc/holoscan/pkg.json#resources.fragments.<fragment-name>.sharedMemoryLimit`).
  - Supported units SHALL be mebibytes (`MiB`) and gibibytes (`GiB`).
    - Example: `1.5Gi` , `2048Mi`
- Integer values MUST be positive and not contain any position separators.

        - Example legal values: `1`, `42`, `2048`
        - Example illegal values: `-1`, `1.5`, `2,048`
    - Decimal values MUST be positive, rounded to the nearest tenth, use the dot (`.`) character to separate whole and fractional values, and not contain any positional separators.
        - Example legal values: `1`, `1.0`, `0.5`, `2.5`, `1024`
        - Example illegal values: `1,024`, `-1.0`, `3.14`
    - When not provided, the default values of `cpu=1`, `gpu=0`, `memory="1Gi"`, and `sharedMemory="64Mi"` will be assumed.

Supplemental Application Files

- A HAP SHOULD package supplemental application files provided by the user.
    - Supplemental files SHOULD be in sub-folders of the `/opt/holoscan/docs/` folder.
    - Supplemental files include, but are not limited to, the following:
        - README.md
        - License.txt
        - Changelog.txt
        - EULA
        - Documentation
        - Third-party licenses

Container Behavior and Interaction

A HAP is a single container supporting the following defined behaviors when started.

Default Behavior

When a HAP is started from the CLI or other means without any parameters, the HAP shall execute the contained application. The HAP internally may use `Entrypoint`, `CMD`, or a combination of both.

Manifest Export

A HAP SHOULD provide at least one method to access the *embedded application*, *models*, *licenses*, *README*, or *manifest files*, namely, `app.json` and `package.json`.

- The Method SHOULD provide a container command, `show`, to print one or more manifest files to the console.
- The Method SHOULD provide a container command, `export`, to copy one or more manifest files to a mounted volume path, as described below
    - `/var/run/holoscan/export/app/`: when detected, the Method copies the contents of `/opt/holoscan/app/` to the folder.
    - `/var/run/holoscan/export/config/`: when detected, the Method copies `/var/holoscan/app.yaml`, `/etc/holoscan/app.json` and `/etc/holoscan/pkg.json` to the folder.
    - `/var/run/holoscan/export/models/`: when detected, the Method copies the contents of `/opt/holoscan/models/` to the folder.
    - `/var/run/holoscan/export/docs/`: when detected, the Method copies the contents of `/opt/holoscan/docs/` to the folder.
    - `/var/run/holoscan/export/`: when detected without any of the above being detected, the Method SHALL copy all of the above.

Since a HAP is an OCI compliant container, a user can also run a HAP and log in to an interactive shell, using a method supported by the container engine and its command line interface, e.g. Docker supports this by setting the entrypoint option. The files in the HAP can then be opened or copied to the mapped volumes with shell commands or scripts. A specific implementation of a HAP may choose to streamline such a process with scripts and applicable user documentation.

Table of Important Paths

Path	Purpose
`/etc/holoscan/`	HAP manifests and immutable configuration files.
`/etc/holoscan/app.json`	Application Manifest file.
`/etc/holoscan/pkg.json`	Package Manifest file.
`/opt/holoscan/app/`	Application code, scripts, and other files.
`/opt/holoscan/models/`	AI models. Each model should be in a separate sub-folder.
`/opt/holoscan/docs/`	Documentation, licenses, EULA, changelog, etc...
`/var/holoscan/`	Default working directory.
`/var/holoscan/input/`	Default input directory.
`/var/holoscan/output/`	Default output directory.
`/var/run/holoscan/export/`	Special case folder, causes the Script to export contents related to the app. (see: Manifest Export)
`/var/run/holoscan/export/app/`	Special case folder, causes the Script to export the contents of `/opt/holoscan/app/` to the folder.
`/var/run/holoscan/export/config/`	Special case folder, causes the Script to export `/etc/holoscan/app.json` and `/etc/holoscan/pkg.json` to the folder.
`/var/run/holoscan/export/models/`	Special case folder, causes the Script to export the contents of `/opt/holoscan/models/` to the folder.

Operating Environments

Holoscan SDK supports the following operating environments.

Operating Environment Name	Characteristics
AGX Devkit	Clara AGX devkit with RTX 6000 dGPU only
IGX Orin Devkit	Clara Holoscan devkit with A6000 dGPU only
IGX Orin Devkit - integrated GPU only	IGX Orin Devkit, iGPU only
IGX Orin Devkit with discrete GPU	IGX Orin Devkit, with RTX A6000 dGPU
Jetson AGX Orin Devkit	Jetson Orin Devkit, iGPU only
Jetson Orin Nano Devkit	Jetson Orin Nano Devkit, iGPU only
X86_64	dGPU only on Ubuntu 18.04 and 20.04

Holoscan CLI

`holoscan` - a command-line interface for packaging and running your Holoscan applications into HAP-compliant containers.

Synopsis

`holoscan` [--help|-h] [--log-level|-l {DEBUG,INFO,WARN,ERROR,CRITICAL}] {package,run,version,nics}

Positional Arguments

`package`

Package a Holoscan application

`run`

Run a packaged Holoscan application

`version`

Print version information for the Holoscan SDK

`nics`

Print all available network interface cards and its assigned IP address

CLI-Wide Flags

`[--help|-h]`

Display detailed help.

[--log-level|-l {DEBUG,INFO,WARN,ERROR,CRITICAL}]

Override the default logging verbosity. Defaults to `INFO`.

Application Runner Configuration

The Holoscan runner requires a YAML configuration file to define some properties necessary to deploy an application.

> **Note**
>
> That file is the same configuration file commonly used to configure other aspects of an application, documented here.

Configuration

The configuration file can be defined in two ways:

- At package time, with the `--config` flag of the `holoscan package` command (Required/Default)
- At runtime, with the `--config` flag of the `holoscan run` command (Optional/Override)

Properties

The `holoscan run` command parses two specific YAML nodes from the configuration file:

- A required `application` parameter group to generate a HAP-compliant` container image for the application, including:
  - the `title` (name) and `version` of the application.

    - optionally, `inputFormats` and `outputFormats` if the application expects any inputs or outputs respectively.

- An optional `resources` parameter group that defines the system resources required to run the application, such as the number of CPUs, GPUs and amount of memory required. If the application contains multiple fragments for distributed workloads, resource definitions can be assigned to each fragment.

Example

Below is an example configuration file with the `application` and optional `resources` parameter groups, for an application with two-fragments (`first-fragment` and `second-fragment`):

```
application: title: My Application Title version: 1.0.1 inputFormats: ["files"] # optional outputFormats: ["screen"] # optional resources: # optional # non-distributed app cpu: 1 # optional cpuLimit: 5 # optional gpu: 1 # optional gpuLimit: 5 # optional memory: 1Mi # optional memoryLimit: 2Gi # optional gpuMemory: 1Gi # optional gpuMemoryLimit: 1.5Gi # optional sharedMemory: 1Gi # optional # distributed app fragments: # optional first-fragment: # optional cpu: 1 # optional cpuLimit: 5 # optional gpu: 1 # optional gpuLimit: 5 # optional memory: 100Mi # optional memoryLimit: 1Gi # optional gpuMemory: 1Gi # optional gpuMemoryLimit: 10Gi # optional sharedMemory: 1Gi # optional second-fragment: # optional cpu: 1 # optional cpuLimit: 2 # optional gpu: 1 # optional gpuLimit: 2 # optional memory: 1Gi # optional memoryLimit: 2Gi # optional gpuMemory: 1Gi # optional gpuMemoryLimit: 5Gi # optional sharedMemory: 10Mi # optional
```

For details, please refer to the HAP specification.

GXF Core concepts

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

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

In comparison, the core concepts of the Holoscan SDK can be found here.

Holoscan and GXF

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:

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

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).

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).

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.

Holoscan SDK v1.0

1. The remaining GXF-based DemosiacOp operator was switched to a native implementation. Now all operators provided by the SDK are native operators.

Current limitations

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

- Job Statistics

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

- Graph Composer
- Behavior Trees
- Epoch Scheduler
- Target Time Scheduling Term
- Multi-Message Available Scheduling Term
- Expiring Message Available Scheduling Term

GXF by Example

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

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 23 *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 out: out_tensor out_tensor_name: source_video out_dtype: "float32" scale_min: 0.0 scale_max: 255.0 pool: pool --- # other entities declared --- components: - name: input_connection type: nvidia::gxf::Connection parameters: source: upstream_entity/output target: format_converter/in_tensor --- components: - name: output_connection type: nvidia::gxf::Connection parameters: source:
```

```
format_converter/out_tensor target: downstream_entity/input --- name: scheduler components: - type: nvidia::gxf::GreedyScheduler
```

Above:

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

The YAML snippet above can be visually represented as follows.

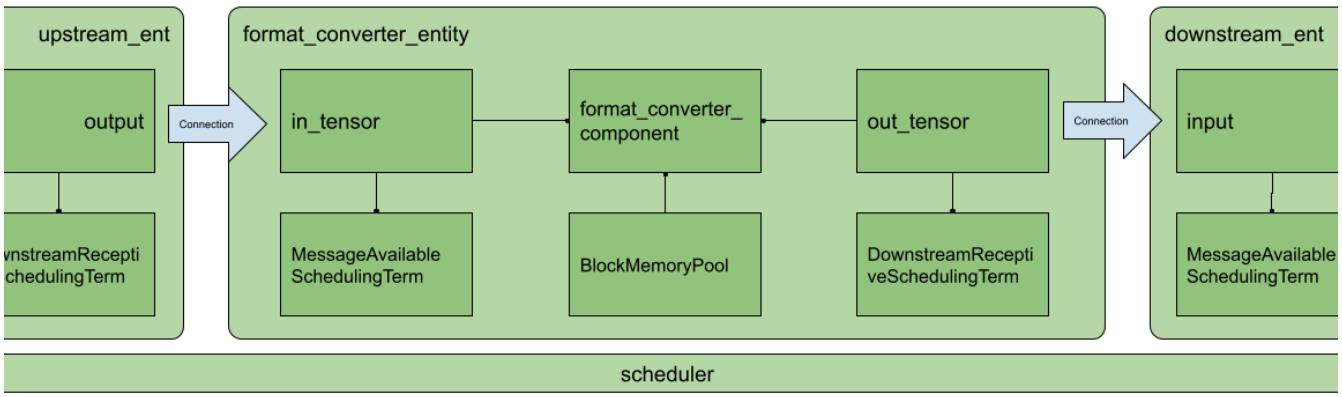


Fig. 21 *Arrangement of components and entities in a Holoscan application*

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

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

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

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

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

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 24 *MessageAvailableSchedulingTerm*

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

Similarly, `DownStreamReceptiveSchedulingTerm` checks whether the `out_tensor` transmitter queue has at least one outgoing message in it. If there 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 25 *DownstreamReceptiveSchedulingTerm*

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

If we were to draw the entity in Fig. 21 in greater detail it would look something like the following.

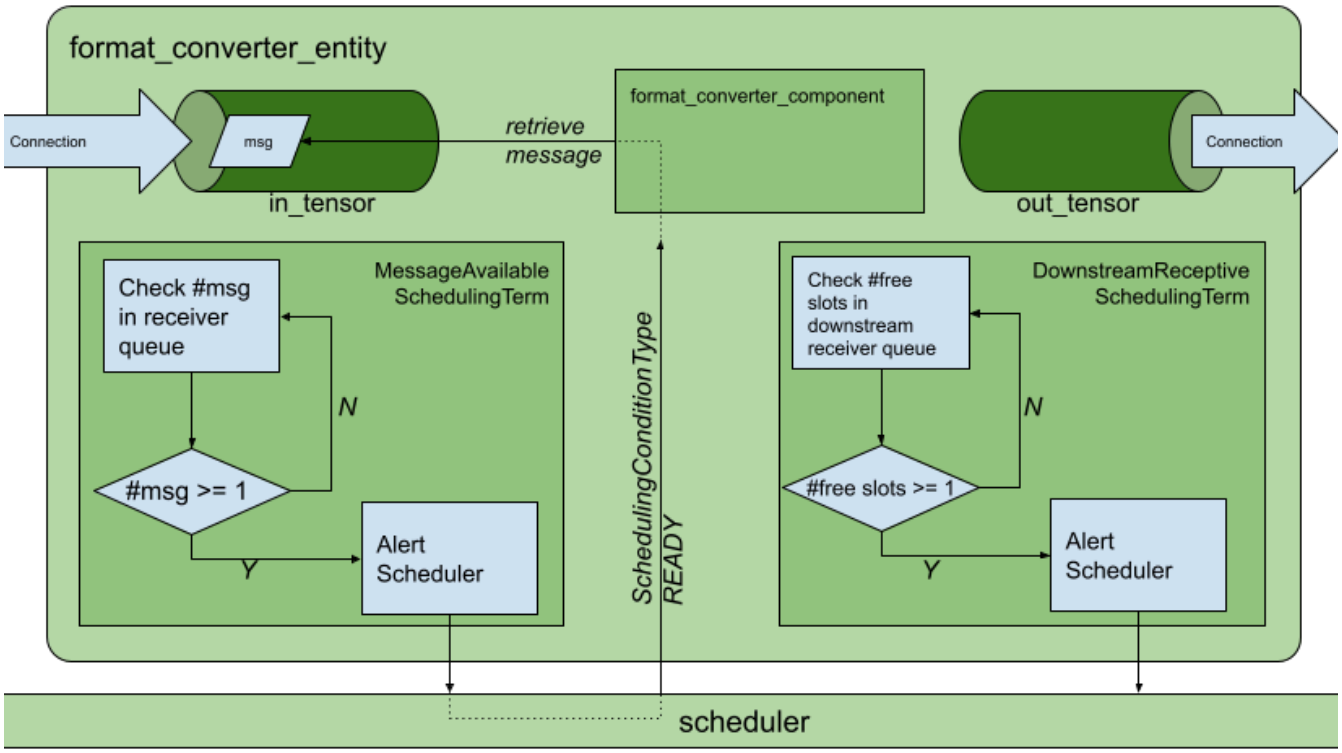


Fig. 22 *Receive and transmit* `Queues` *and* `SchedulingTerm` *s in entities.*

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.

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.

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. 23.

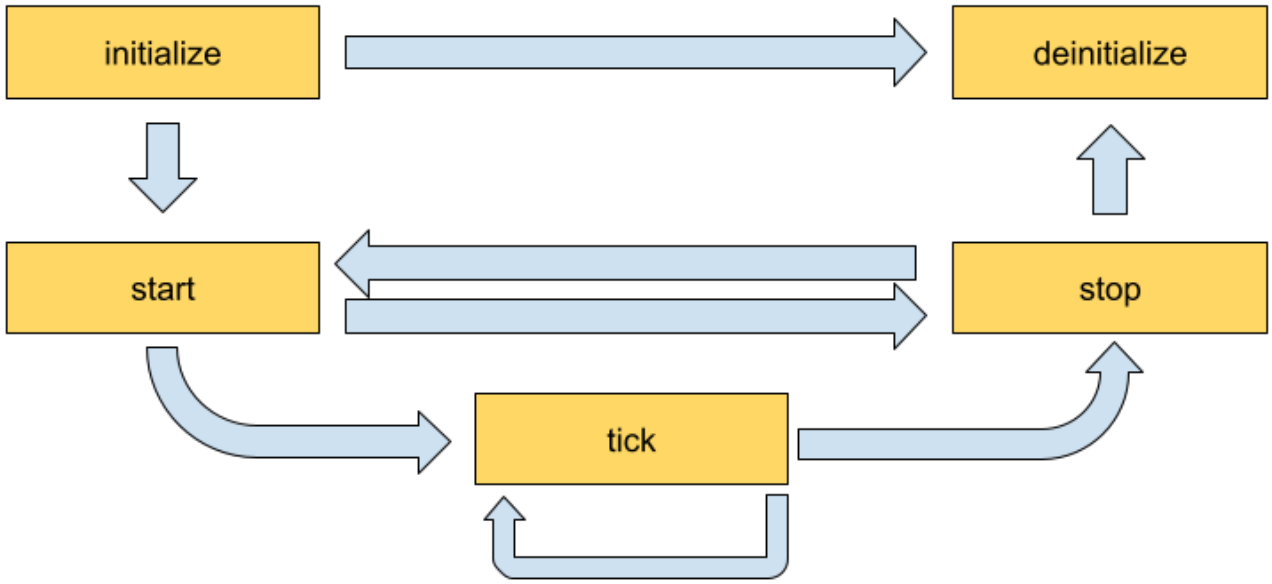


Fig. 23 *Sequence of method calls in the lifecycle of a Holoscan extension*

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 26 *gxf_extensions/my_recorder/my_recorder.hpp*

```
#include <string> #include "gxf/core/handle.hpp" #include "gxf/std/codelet.hpp" #include "gxf/std/receiver.hpp" #include "gxf/std/transmitter.hpp" #include "gxf/serialization/file_stream.hpp" #include "gxf/serialization/entity_serializer.hpp" class MyRecorder : public nvidia::gxf::Codelet { public: gxf_result_t
```

```
registerInterface(nvidia::gxf::Registrar* registrar) override; gxf_result_t initialize() override; gxf_result_t deinitialize() override; gxf_result_t start() override; gxf_result_t tick() override; gxf_result_t stop() override; private: nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::Receiver>> receiver_; nvidia::gxf::Parameter<nvidia::gxf::Handle<nvidia::gxf::EntitySerializer>> my_serializer_; nvidia::gxf::Parameter<std::string> directory_; nvidia::gxf::Parameter<std::string> basename_; nvidia::gxf::Parameter<bool> flush_on_tick_; // File stream for data index nvidia::gxf::FileStream index_file_stream_; // File stream for binary data nvidia::gxf::FileStream binary_file_stream_; // Offset into binary file size_t binary_file_offset_; };
```

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 27 *registerInterface in gxf_extensions/my_recorder/my_recorder.cpp*

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

For pure GXF applications, our component's parameters can be specified in the following format in the YAML file:

Listing 28 *Example parameters for MyRecorder component*

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

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

Implement the Lifecycle Methods

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

Listing 29 *initialize in gxf_extensions/my_recorder/my_recorder.cpp*

```
gxf_result_t MyRecorder::initialize() { // Create path by appending receiver name to directory path if basename is not provided std::string path = directory_.get() + '/'; if (const auto& basename = basename_.try_get()) { path += basename.value(); } else { path += receiver_->name(); } // Initialize index file stream as write-only index_file_stream_ = nvidia::gxf::FileStream("", path + nvidia::gxf::FileStream::kIndexFileExtension); // Initialize binary file stream as write-only binary_file_stream_ = nvidia::gxf::FileStream("", path + nvidia::gxf::FileStream::kBinaryFileExtension); // Open index file stream nvidia::gxf::Expected<void> result = index_file_stream_.open(); if (!result) { return nvidia::gxf::ToResultCode(result); } // Open binary file stream result = binary_file_stream_.open(); if (!result) { return nvidia::gxf::ToResultCode(result); } binary_file_offset_ = 0; return GXF_SUCCESS; }
```

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

Listing 30 *deinitialize in gxf_extensions/my_recorder/my_recorder.cpp*

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

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 31 *start/stop in gxf_extensions/my_recorder/my_recorder.cpp*

```
gxf_result_t MyRecorder::start() { return GXF_SUCCESS; } gxf_result_t
MyRecorder::stop() { return GXF_SUCCESS; }
```

Tip

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

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

Listing 32 *tick in gxf_extensions/my_recorder/my_recorder.cpp*

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

```
nvidia::gxf::Expected<size_t> size = my_serializer_->serializeEntity(entity.value(), &binary_file_stream_); if (!size) { return nvidia::gxf::ToResultCode(size); } // Create entity index nvidia::gxf::EntityIndex index; index.log_time = std::chrono::system_clock::now().time_since_epoch().count(); index.data_size = size.value(); index.data_offset = binary_file_offset_; // Write entity index to index file nvidia::gxf::Expected<size_t> result = index_file_stream_.writeTrivialType(&index); if (!result) { return nvidia::gxf::ToResultCode(result); } binary_file_offset_ += size.value(); if (flush_on_tick_) { // Flush binary file output stream nvidia::gxf::Expected<void> result = binary_file_stream_.flush(); if (!result) { return nvidia::gxf::ToResultCode(result); } // Flush index file output stream result = index_file_stream_.flush(); if (!result) { return nvidia::gxf::ToResultCode(result); } } return GXF_SUCCESS; }
```

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 33 *gxf_extensions/my_recorder/my_recorder_ext.cpp*

```
#include "gxf/std/extension_factory_helper.hpp" #include "my_recorder.hpp" GXF_EXT_FACTORY_BEGIN() GXF_EXT_FACTORY_SET_INFO(0xb891cef3ce754825, 0x9dd3dcac9bbd8483, "MyRecorderExtension", "My example recorder extension", "NVIDIA", "0.1.0", "LICENSE"); GXF_EXT_FACTORY_ADD(0x2464fabf91b34ccf, 0xb554977fa22096bd, MyRecorder, nvidia::gxf::Codelet, "My example recorder codelet."); 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 34 *gxf_extensions/my_recorder/CMakeLists.txt*

```
# Create library add_library(my_recorder_lib SHARED my_recorder.cpp my_recorder.hpp ) target_link_libraries(my_recorder_lib PUBLIC GXF::std GXF::serialization yaml-cpp ) # Create extension add_library(my_recorder SHARED my_recorder_ext.cpp ) target_link_libraries(my_recorder PUBLIC my_recorder_lib ) # Install GXF extension as a component 'holoscan-gxf_extensions' install_gxf_extension(my_recorder) # this will also install my_recorder_lib # 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.

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 35 *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/racerx" basename: "racerx"
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 --- name: recorder components: - name: input type:
```

```
nvidia::gxf::DoubleBufferReceiver - name: allocator type: nvidia::gxf::UnboundedAllocator - name: component_serializer type: nvidia::gxf::StdComponentSerializer parameters: allocator: allocator - name: entity_serializer type: nvidia::holoscan::stream_playback::VideoStreamSerializer # inheriting from nvidia::gxf::EntitySerializer parameters: component_serializers: [component_serializer] - type: MyRecorder parameters: receiver: input serializer: entity_serializer out_directory: "/tmp" basename: "tensor_out" - type: nvidia::gxf::MessageAvailableSchedulingTerm parameters: receiver: input min_size: 1 --- components: - name: input_connection type: nvidia::gxf::Connection parameters: source: replayer/output target: recorder/input --- name: scheduler components: - name: clock type: nvidia::gxf::RealtimeClock - name: greedy_scheduler type: nvidia::gxf::GreedyScheduler parameters: clock: clock
```

Above:

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

Note the use of the `component_serializer` in our newly built recorder. This component is declared separately in the 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 36 *apps/my_recorder_app_gxf/CMakeLists.txt*

```
create_gxe_application( NAME my_recorder_gxf YAML my_recorder_gxf.yaml
EXTENSIONS GXF::std GXF::cuda GXF::multimedia GXF::serialization my_recorder
stream_playback ) # Download the associated dataset if needed
if(HOLOSCAN_DOWNLOAD_DATASETS) add_dependencies(my_recorder_gxf
racerx_data) 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.

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 (`racerx.gxf_index` and `racerx.gxf_entities` under `data/racerx`) exactly.

```
@LINUX:/workspace/holoscan-sdk/build$ ls -al /tmp/ total 821384 drwxrwxrwt 1 root root 4096 Aug 24 04:37 . drwxr-xr-x 1 root root 4096 Aug 24 04:36 .. drwxrwxrwt 2 root root 4096 Aug 11 21:42 .X11-unix -rw-r--r-- 1 1000 1000 729309 Aug 24 04:47 gxf_log -rw-r--r-- 1 1000 1000 840054484 Aug 24 04:47 tensor_out.gxf_entities -rw-r--r-- 1 1000 1000 16392 Aug 24 04:47 tensor_out.gxf_index @LINUX:/workspace/holoscan-sdk/build$ ls -al ../data/racerx total 839116 drwxr-xr-x 2 1000 1000 4096 Aug 24 02:08 . drwxr-xr-x 4 1000 1000 4096 Aug 24 02:07 .. -rw-r--r-- 1 1000 1000 19164125 Jun 17 16:31 racerx-medium.mp4 -rw-r--r-- 1 1000 1000 840054484 Jun 17 16:31 racerx.gxf_entities -rw-r--r-- 1 1000 1000 16392 Jun 17 16:31 racerx.gxf_index
```

Using Holoscan Operators in GXF Applications

For users who are familiar with the GXF development ecosystem (used in Holoscan SDK 0.2), we provide an export feature to leverage native Holoscan operators as GXF codelets to execute in GXF applications and GraphComposer.

We demonstrate how to wrap a native C++ holoscan operator as a GXF codelet in the `wrap_operator_as_gxf_extension`, as described below.

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.

2. Creating the GXF extension that wraps the operator

To wrap the native operator as a GXF codelet in a GXF extension, we provide the CMake `wrap_operator_as_gxf_extension` function in the SDK. An example of how it wraps `PingTxNativeOp` and `PingRxNativeOp` can be found here.

- It leverages the CMake target names of the operators defined in their respective `CMakeLists.txt` (ping_tx_native_op, ping_rx_native_op)
- The function parameters are documented at the top of the WrapOperatorAsGXFExtension.cmake file (ignore implementation below).

> ⚠ **Warning**
>
> - A unique GXF extension is currently needed for each native operator to export (operators cannot be bundled in a single extension at this time).
> - Wrapping other GXF entities than operators (as codelets) is not currently supported.

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

Introduction

- Graph Specification
- Graph Execution Engine

Messaging and Scheduling

- Graph Specification TimeStamping
- The GXF Scheduler
- Behavior Trees

Core C APIs

- GXF Core C APIs

Extensions

- CudaExtension
- MultimediaExtension
- NetworkExtension
- SerializationExtension
- StandardExtension

Graph Specification

Graph Specification is a format to describe high-performance AI applications in a modular and extensible way. It allows writing applications in a standard format and sharing

components across multiple applications without code modification. Graph Specification is based on entity-composition pattern. Every object in graph is represented with entity (aka Node) and components. Developers implement custom components which can be added to entity to achieve the required functionality.

Concepts

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

Graph

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

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

SubGraph

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

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

A subgraph interface can be defined as follows:

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

Node

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

Components

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

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

Edges

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

Extension

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

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

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

The section :doc: *GraphComposer_Dev_Workflow* talks more about this with a focus on developing extensions and components.

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

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

Graph Specification TimeStamping

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 `0`. The message is published using the `publish(Entity message)`. **This will be deprecated in the next release.**

2. `tx.publish(Entity message, int64_t acqtime)`: You can simply call `publish(Entity message, int64_t acqtime)` with the `acqtime`. Timestamp will be added automatically.

The GXF Scheduler

The execution of entities in a graph is governed by the scheduler and the scheduling terms associated with every entity. A scheduler is a component responsible for orchestrating the execution of all the entities defined in a graph. A scheduler typically keeps track of the graph entities and their current execution states and passes them on to a nvidia::gxf::EntityExecutor component when ready for execution. The following diagram depicts the flow for an entity execution.

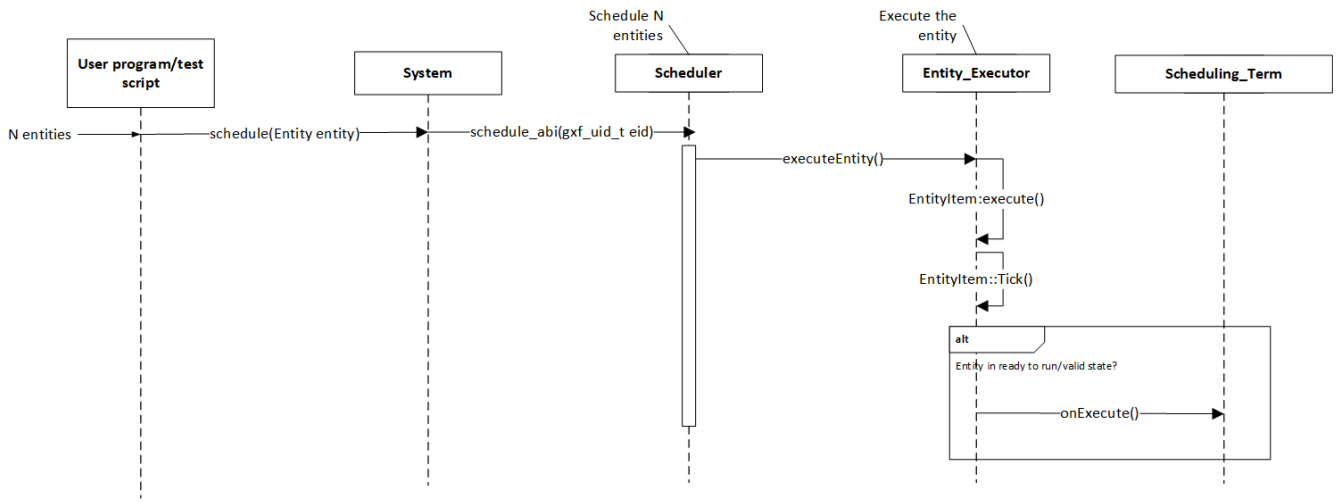


Figure: Entity execution sequence

As shown in the sequence diagram, the schedulers begin executing the graph entities via the nvidia::gxf::System::runAsync_abi() interface and continue this process until it meets the certain ending criteria. A single entity can have multiple codelets. These codelets are executed in the same order in which they were defined in the entity. A failure in execution of any single codelet stops the execution of all the entities. Entities are naturally unscheduled from execution when any one of their scheduling term reaches NEVER state.

Scheduling terms are components used to define the execution readiness of an entity. An entity can have multiple scheduling terms associated with it and each scheduling term represents the state of an entity using SchedulingCondition.

The table below shows various states of nvidia::gxf::SchedulingConditionType described using nvidia::gxf::SchedulingCondition.

SchedulingConditionType	Description
NEVER	Entity will never execute again
READY	Entity is ready for execution
WAIT	Entity may execute in the future
WAIT_TIME	Entity will be ready for execution after specified duration
WAIT_EVENT	Entity is waiting on an asynchronous event with unknown time interval

Schedulers define deadlock as a condition when there are no entities which are in READY, WAIT_TIME or WAIT_EVENT state which guarantee execution at a future point in time. This implies all the entities are in WAIT state for which the scheduler does not know if they ever will reach the READY state in the future. The scheduler can be configured to stop when it reaches such a state using the stop_on_deadlock parameter, else the entities are polled to check if any of them have reached READY state. max_duration configuration parameter can be used to stop execution of all entities regardless of their state after a specified amount of time has elapsed.

There are two types of schedulers currently supported by GXF

1. Greedy Scheduler

2. Multithread Scheduler

Greedy Scheduler

This is a basic single threaded scheduler which tests scheduling term greedily. It is great for simple use cases and predictable execution but may incur a large overhead of scheduling term execution, making it unsuitable for large applications. The scheduler requires a clock to keep track of time. Based on the choice of clock the scheduler will execute differently. If a Realtime clock is used the scheduler will execute in real-time. This means pausing execution - sleeping the thread, until periodic scheduling terms are due again. If a ManualClock is used scheduling will happen “time-compressed”. This means flow of time is altered to execute codelets in immediate succession.

The GreedyScheduler maintains a running count of entities which are in READY, WAIT_TIME and WAIT_EVENT states. The following activity diagram depicts the gist of the decision making for scheduling an entity by the greedy scheduler -

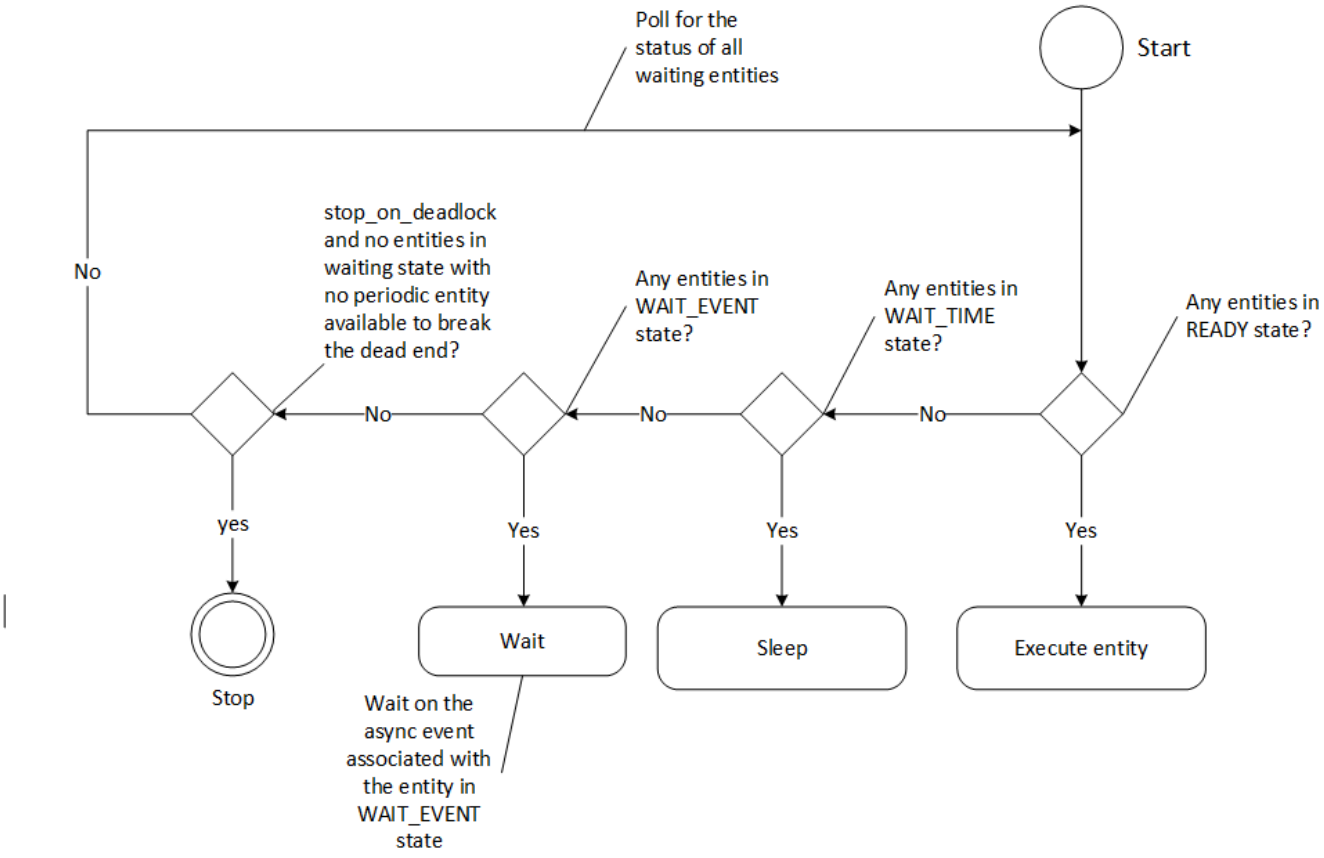


Figure: Greedy Scheduler Activity Diagram

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
```

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.

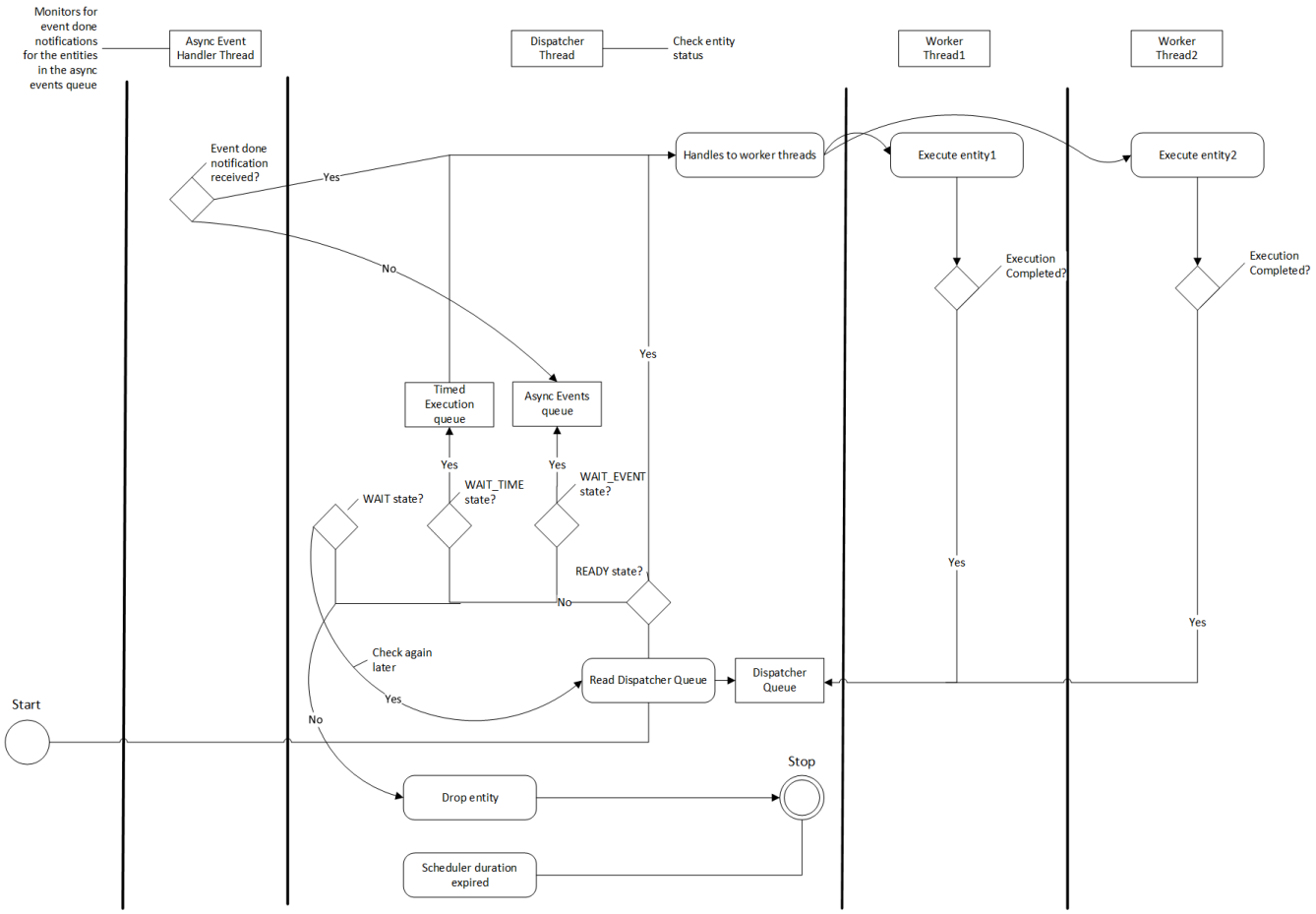


Figure: MultiThread Scheduler Activity Diagram

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
```

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.

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 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 it's regular execution cycle performed by the gxf scheduler. When the scheduling term is in WAIT state, the scheduler regularly polls for the state of the entity. When the scheduling term is in EVENT_WAITING state, schedulers will not check the status of the entity again until they receive an event notification which can be triggered using the GxfEntityEventNotify api. Setting the state of the scheduling term to EVENT_DONE automatically sends this notification to the scheduler. Entities can use the EVENT_NEVER state to indicate the end of its execution cycle.

Example usage -

```
- name: async_scheduling_term type: nvidia::gxf::AsynchronousSchedulingTerm
```

DownsteamReceptiveSchedulingTerm

This scheduling term specifies that an entity shall be executed if the receiver for a given transmitter can accept new messages.

Example usage -

```
- name: downstream_st type: nvidia::gxf::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 it's execution behavior. Scheduling terms are AND combined to describe the current state of an entity. For an entity to be executed by the scheduler, all the scheduling terms must be in READY state and conversely, the entity is unscheduled from execution whenever any one of the scheduling term reaches NEVER state. The priority of various states during AND combine follows the order NEVER, WAIT_EVENT, WAIT, WAIT_TIME, and READY.

Example usage -

```
components: - name: integers type: nvidia::gxf::DoubleBufferTransmitter - name: fibonacci type: nvidia::gxf::DoubleBufferTransmitter - type: nvidia::gxf::CountSchedulingTerm parameters: count: 100 - type: nvidia::gxf::DownstreamReceptiveSchedulingTerm parameters: transmitter: integers min_size: 1
```

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
```

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.

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.

Behavior Tree Codelets

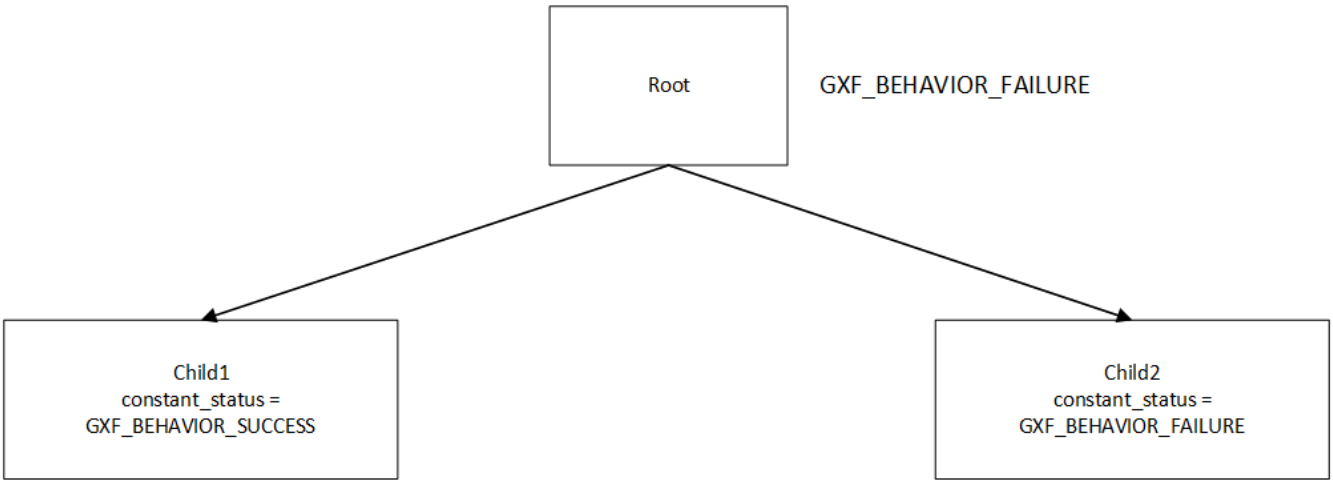
Each behavior tree codelet can have a set of parameters defining how it should behave. Note that in all the examples given below, the naming convention for configuring the `children` parameter for root codelets is `[child_codelet_name\child_codelet_scheduling_term]`.

Constant Behavior

After each tick period, switches its own status to the configured desired constant status.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
constant_status	The desired status to switch to during each tick time.

An example diagram depicting Constant behavior used in conjunction with a Sequence behavior defined for root entity is shown below



Here, the child1 is configured to return a constant status of success (GXF_BEHAVIOR_SUCCESS) and child2 returns failure (GXF_BEHAVIOR_FAILURE), resulting into the root node (configured to exhibit sequence behavior) returning GXF_BEHAVIOR_FAILURE.

The controller for each child can be configured to repeat the execution on failure. A code snippet of configuring the example described is shown below.

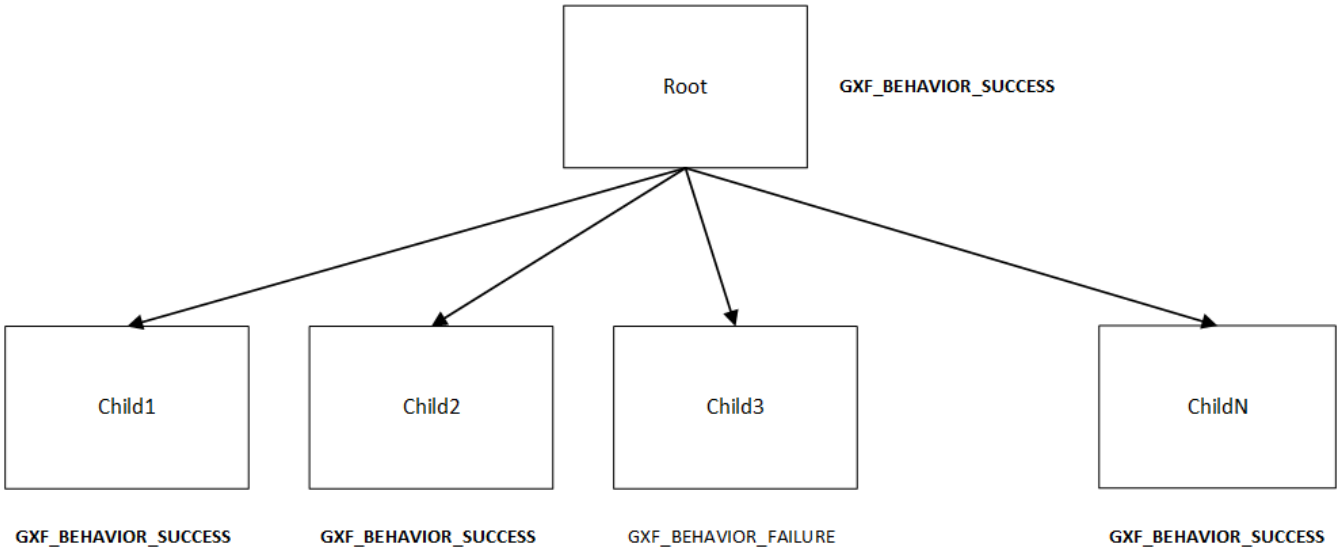
```
name: root components: - name: root_controller type:
nvidia::gxf::EntityCountFailureRepeatController parameters: max_repeat_count: 0 -
name: root_st type: nvidia::gxf::BTSchedulingTerm parameters: is_root: true - name:
root_codelet type: nvidia::gxf::SequenceBehavior parameters: children: [
child1/child1_st, child2/child2_st ] s_term: root_st --- name: child2 components: -
name: child2_controller type: nvidia::gxf::EntityCountFailureRepeatController
parameters: max_repeat_count: 3 return_behavior_running_if_failure_repeat: true -
name: child2_st type: nvidia::gxf::BTSchedulingTerm parameters: is_root: false -
name: child2_codelet type: nvidia::gxf::ConstantBehavior parameters: s_term:
child2_st constant_status: 1
```

Parallel Behavior

Runs its child nodes in parallel. By default, succeeds when all child nodes succeed, and fails when all child nodes fail. This behavior can be customized using the parameters below.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
children	Child entities
success_threshold	Number of successful children required for success. A value of -1 means all children must succeed for this node to succeed.
failure_threshold	Number of failed children required for failure. A value of -1 means all children must fail for this node to fail.

The diagram below shows a graphical representation of a parallel behavior configured with failure_threshold configured as -1. Hence, the root node returns GXF_BEHAVIOR_SUCCESS even if one child returns a failure status.



A code snippet to configure the example described is shown below.

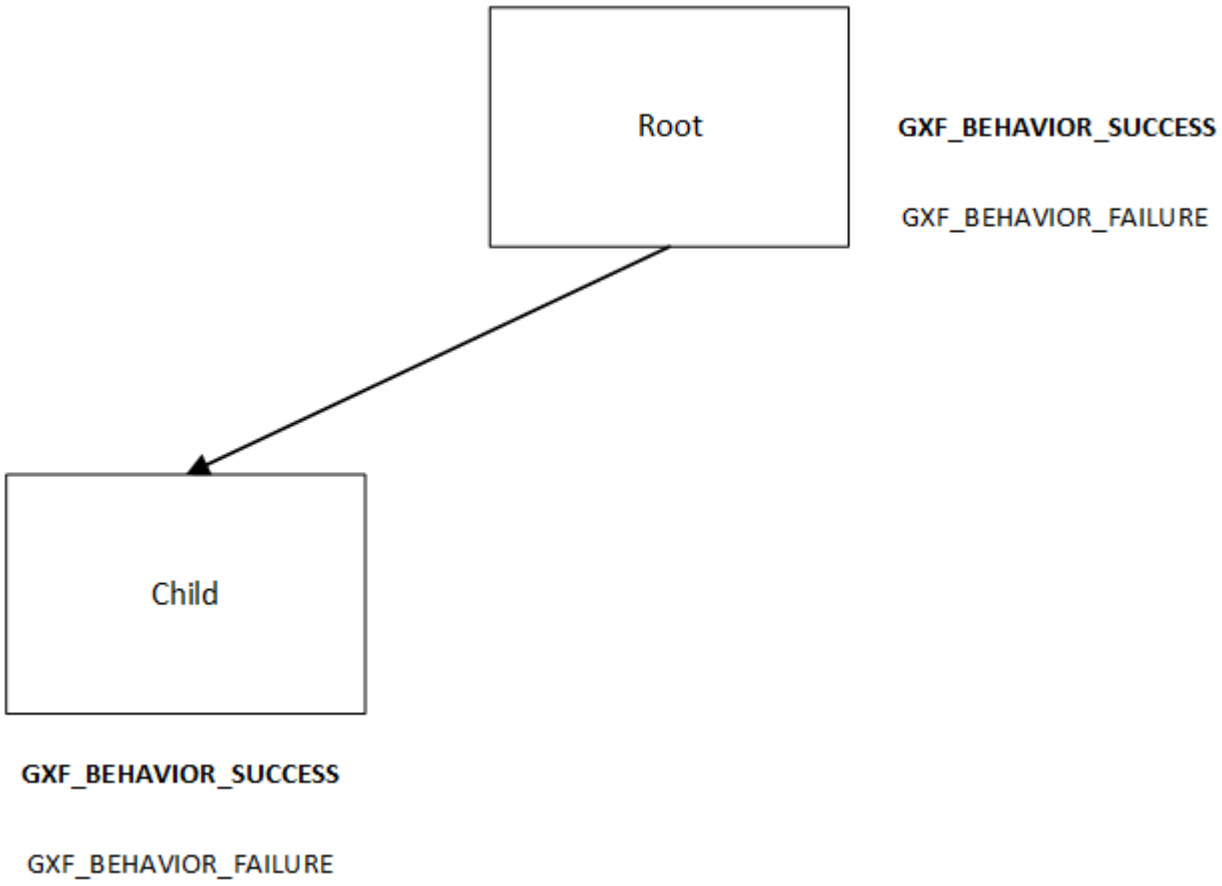
```
name: root components: - name: root_controller type:
nvidia::gxf::EntityCountFailureRepeatController parameters: max_repeat_count: 0 -
name: root_st type: nvidia::gxf::BTSchedulingTerm parameters: is_root: true - name:
root_codelet type: nvidia::gxf::ParallelBehavior parameters: children: [
child1/child1_st, child2/child2_st ] s_term: root_st success_threshold: 1
failure_threshold: -1
```

Repeat Behavior

Repeats its only child entity. By default, won't repeat when the child entity fails. This can be customized using the parameters below.

Parameter	Description
s_term	scheduling term used for scheduling the entity itself
repeat_after_failure	Denotes whether to repeat the child after it has failed.

The diagram below shows a graphical representation of a repeat behavior. The root entity can be configured to repeat the only child to repeat after failure. It succeeds when the child entity succeeds.



A code snippet to configure a repeat behavior is as shown below -

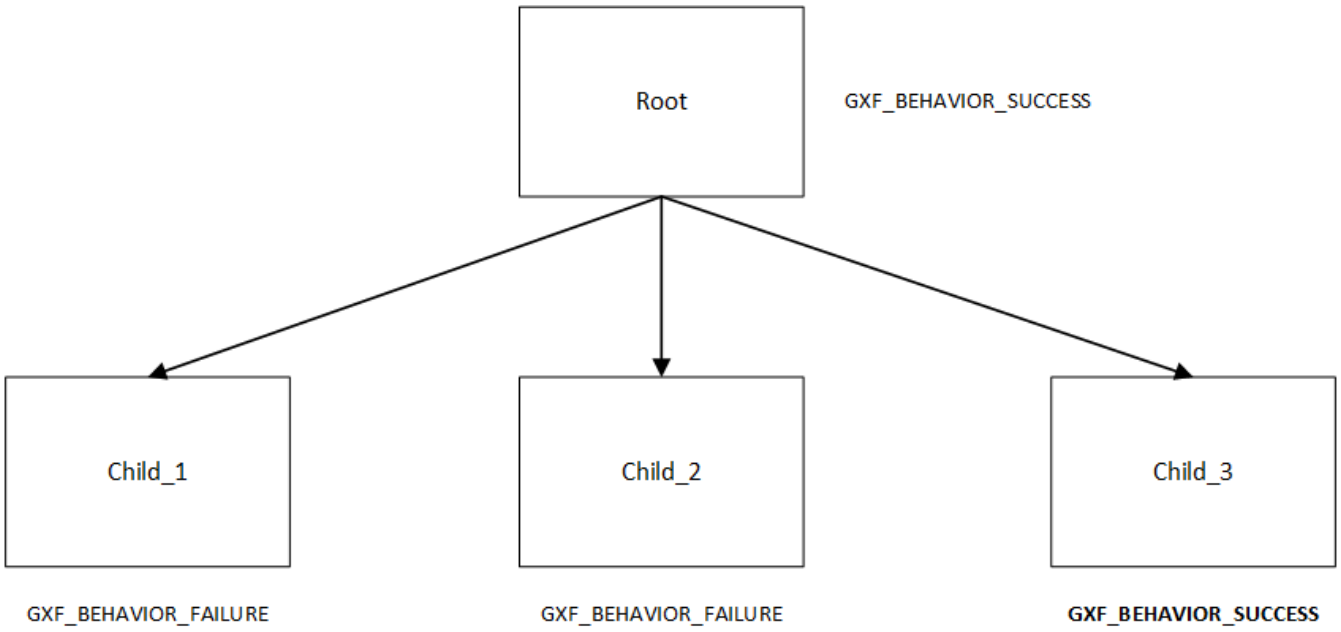
```
name: repeat_knock components: - name: repeat_knock_controller type: nvidia::gxf::EntityCountFailureRepeatController parameters: max_repeat_count: 0 - name: repeat_knock_st type: nvidia::gxf::BTSchedulingTerm parameters: is_root: false - name: repeat_codelet type: nvidia::gxf::RepeatBehavior parameters: s_term: repeat_knock_st children: [ knock_on_door/knock_on_door_st ] repeat_after_failure: true ---
```

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 ]
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 ---
```

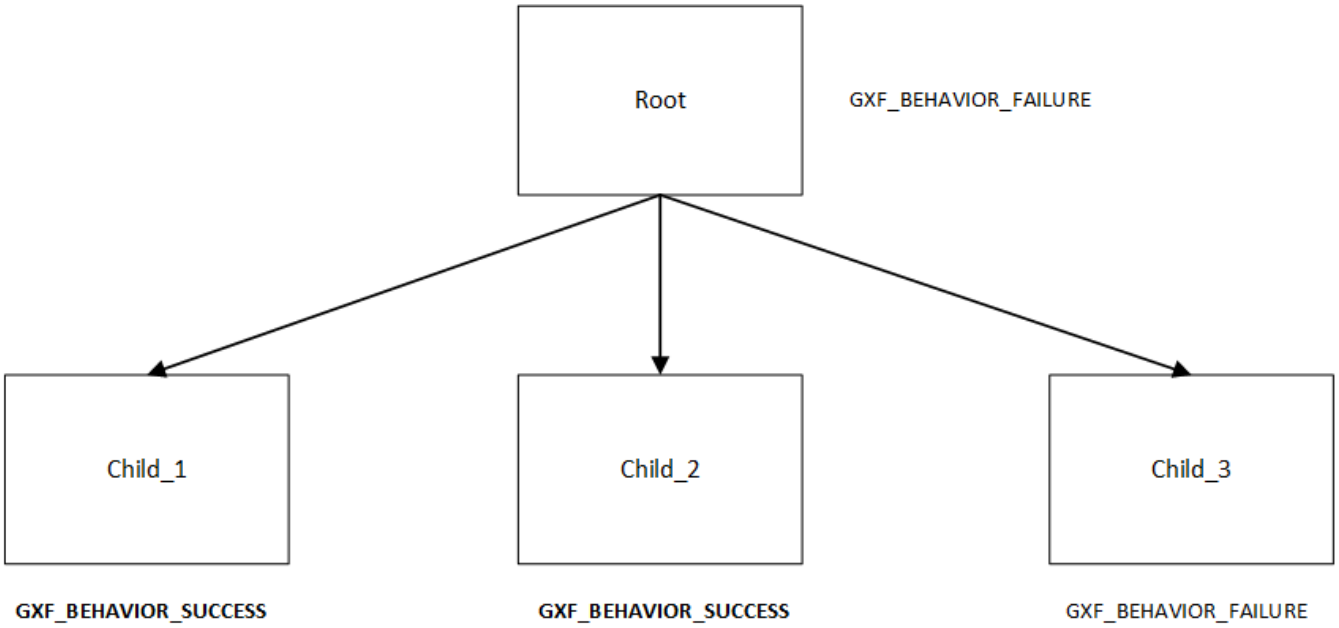
Sequence Behavior

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.

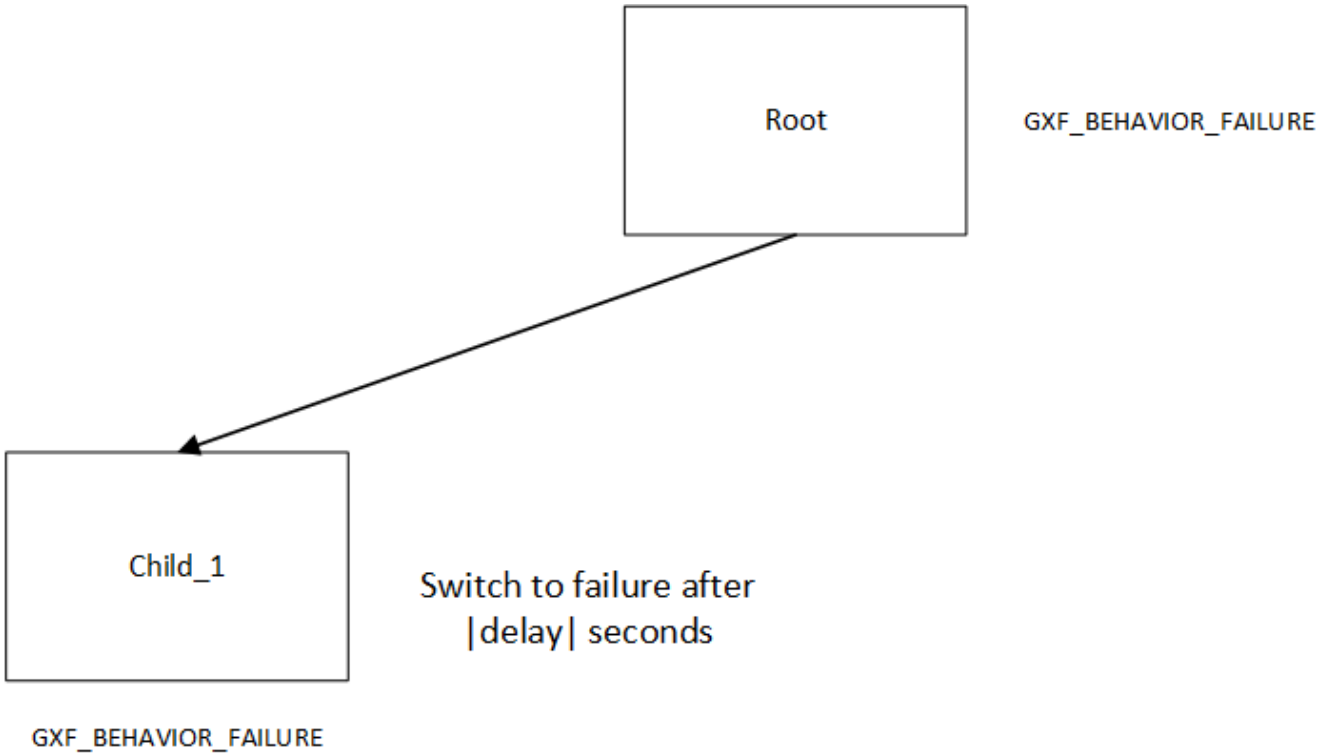
```
name: root components: - name: root_controller type:
nvidia::gxf::EntityCountFailureRepeatController parameters: max_repeat_count: 0 -
name: root_st type: nvidia::gxf::BTSchedulingTerm parameters: is_root: true - name:
root_switch_codelet type: nvidia::gxf::SwitchBehavior parameters: children: [
scene/scene_st, ref/ref_st ] s_term: root_st desired_behavior: 0 --- name: scene
components: - name: scene_controller type:
nvidia::gxf::EntityCountFailureRepeatController parameters: 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 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 ---
```

GXF Core C APIs

Context

Create context

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

Creates a new GXF context

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

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

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

Create a context from a shared context

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

Creates a new runtime context from shared context.

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

parameter: `shared` A valid GXF shared context.

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

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

Destroy context

`gxf_result_t GxfContextDestroy(gxf_context_t context);`

Destroys a GXF context

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

parameter: `context` A valid GXF context.

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

Extensions

Maximum number of extensions in a context can be `1024`.

Load Extensions from a file

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

Loads extension in the given context from file.

parameter: `context` A valid GXF context

parameter: `filename` A valid filename.

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

This function will be deprecated.

Load Extension libraries

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

Loads GXF extension libraries

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

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

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.

Graph Execution

Loads a list of entities from YAML file

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

parameter: `context` A valid GXF context

parameter: `filename` A valid YAML filename.

parameter: `params_override` An optional array of strings used for override parameters in yaml file.

parameter: `num_overrides` Number of optional override parameter strings.

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

Set the root folder for searching YAML files during loading

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

parameter: `context` A valid GXF context

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

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

Loads a list of entities from YAML text

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

parameter: `context` A valid GXF context

parameter: `text` A valid YAML text.

parameter: `params_override` An optional array of strings used for override parameters in yaml file.

parameter: `num_overrides` Number of optional override parameter strings.

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

Activate all system components

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

parameter: `context` A valid GXF context

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

Deactivate all System components

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

parameter: `context` A valid GXF context

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

Starts the execution of the graph asynchronously

`gxf_result_t GxfGraphRunAsync(gxf_context_t context);`

parameter: `context` A valid GXF context

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

Interrupt the execution of the graph

`gxf_result_t GxfGraphInterrupt(gxf_context_t context);`

parameter: `context` A valid GXF context

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

Waits for the graph to complete execution

`gxf_result_t GxfGraphWait(gxf_context_t context);`

parameter: `context` A valid GXF context

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

Runs all System components and waits for their completion

`gxf_result_t GxfGraphRun(gxf_context_t context);`

parameter: `context` A valid GXF context

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

Entities

Create an entity

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

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

This method will be deprecated.

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

Create a new GXF entity.

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

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

Activate an entity

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

Activates a previously created and inactive entity

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

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

  parameter: `context` A valid GXF context

  parameter: `eid` UID of a valid entity

  returns: GXF error code

Deactivate an entity

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

Deactivates a previously activated entity

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.

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.

Primitive Parameters

64-bit floating point

Set

```
gxf_result_t GxfParameterSetFloat64(gxf_context_t context, gxf_uid_t uid, const char* key, double value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` a double value

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

Get

```
gxf_result_t GxfParameterGetFloat64(gxf_context_t context, gxf_uid_t uid, const char* key, double* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to get the double value.

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

64-bit signed integer

Set

```
gxf_result_t GxfParameterSetInt64(gxf_context_t context, gxf_uid_t uid, const char* key, int64_t value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` 64-bit integer value to set.

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

Get

```
gxf_result_t GxfParameterGetInt64(gxf_context_t context, gxf_uid_t uid, const char* key, int64_t* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to get the 64-bit integer value.

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

64-bit unsigned integer

Set

```
gxf_result_t GxfParameterSetUInt64(gxf_context_t context, gxf_uid_t uid, const char* key, uint64_t value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` unsigned 64-bit integer value to set.

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

Get

```
gxf_result_t GxfParameterGetUInt64(gxf_context_t context, gxf_uid_t uid, const char* key, uint64_t* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to get the unsigned 64-bit integer value.

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

32-bit signed integer

Set

```
gxf_result_t GxfParameterSetInt32(gxf_context_t context, gxf_uid_t uid, const char* key, int32_t value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` 32-bit integer value to set.

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

Get

```
gxf_result_t GxfParameterGetInt32(gxf_context_t context, gxf_uid_t uid, const char* key, int32_t* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to get the 32-bit integer value.

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

String parameter

Set

```
gxf_result_t GxfParameterSetStr(gxf_context_t context, gxf_uid_t uid, const char* key, const char* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` A char array containing value to set.

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

Get

```
gxf_result_t GxfParameterGetStr(gxf_context_t context, gxf_uid_t uid, const char* key,
const char** value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to a char* array to get the value.

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

Boolean

Set

```
gxf_result_t GxfParameterSetBool(gxf_context_t context, gxf_uid_t uid, const char* key,
bool value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` A boolean value to set.

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

Get

```
gxf_result_t GxfParameterGetBool(gxf_context_t context, gxf_uid_t uid, const char* key, bool* value);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` pointer to get the boolean value.

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

Handle

Set

```
gxf_result_t GxfParameterSetHandle(gxf_context_t context, gxf_uid_t uid, const char* key, gxf_uid_t cid);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `cid` Unique identifier to set.

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

Get

```
gxf_result_t GxfParameterGetHandle(gxf_context_t context, gxf_uid_t uid, const char* key, gxf_uid_t* cid);
```

parameter: `context` A valid GXF context.

parameter: `uid` A valid component identifier.

parameter: `key` A valid name of a component to set.

parameter: `value` Pointer to a unique identifier to get the value.

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

Vector Parameters

To set or get the vector parameters of a component, users can use the following C-APIs for various data types:

Set 1-D Vector Parameters

Users can call `gxf_result_t GxfParameterSet1D"DataType"Vector(gxf_context_t context, gxf_uid_t uid, const char* key, data_type* value, uint64_t length)`

`value` should point to an array of the data to be set of the corresponding type. The size of the stored array should match the `length` argument passed.

See the table below for all the supported data types and their corresponding function signatures.

parameter: `key` The name of the parameter

parameter: `value` The value to set of the parameter

parameter: `length` The length of the vector parameter

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

Table 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 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 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 4 *Supported Data Types to Get the Value of 2D Vector Parameters*

Function Name	data_type
GxfParameterGet2DFloat64Vector(...)	double
GxfParameterGet2DInt64Vector(...)	int64_t
GxfParameterGet2DUInt64Vector(...)	uint64_t
GxfParameterGet2DInt32Vector(...)	int32_t

Information Queries

Get Meta Data about the GXF Runtime

`gxf_result_t GxfRuntimeInfo(gxf_context_t context, gxf_runtime_info* info);`

parameter: `context` A valid GXF context.

parameter: `info` pointer to gxf_runtime_info object to get the meta data.

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

Get description and list of components in loaded Extension

`gxf_result_t GxfExtensionInfo(gxf_context_t context, gxf_tid_t tid, gxf_extension_info_t* info);`

parameter: `context` A valid GXF context.

parameter: `tid` The unique identifier of the extension.

parameter: `info` pointer to gxf_extension_info_t object to get the meta data.

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

Get description and list of parameters of Component

```
gxf_result_t GxfComponentInfo(gxf_context_t context, gxf_tid_t tid,
gxf_component_info_t* info);
```

Note: Parameters are only available after at least one instance is created for the Component.

parameter: `context` A valid GXF context.

parameter: `tid` The unique identifier of the component.

parameter: `info` pointer to gxf_component_info_t object to get the meta data.

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

Get parameter type description

Gets a string describing the parameter type

```
const char* GxfParameterTypeStr(gxf_parameter_type_t param_type);
```

parameter: `param_type` Type of parameter to get info about.

returns: C-style string description of the parameter type.

Get flag type description

Gets a string describing the flag type

```
const char* GxfParameterFlagTypeStr(gxf_parameter_flags_t_ flag_type);
```

parameter: `flag_type` Type of flag to get info about.

returns: C-style string description of the flag type.

Get parameter description

Gets description of specific parameter. Fails if the component is not instantiated yet.

```
gxf_result_t GxfGetParameterInfo(gxf_context_t context, gxf_tid_t cid, const char* key,
gxf_parameter_info_t* info);
```

parameter: `context` A valid GXF context.

parameter: `cid` The unique identifier of the component.

parameter: `key` The name of the parameter.

parameter: `info` Pointer to a gxf_parameter_info_t object to get the value.

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

Redirect logs to a file

Redirect console logs to the provided file.

```
gxf_result_t GxfGetParameterInfo(gxf_context_t context, FILE* fp);
```

parameter: `context` A valid GXF context.

parameter: `fp` File path for the redirected logs.

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

Miscellaneous

Get string description of error

`const char* GxfResultStr(gxf_result_t result);`

Gets a string describing an GXF error code.

The caller does not get ownership of the return C string and must not delete it.

parameter: `result` A GXF error code

returns: A pointer to a C string with the error code description.

CudaExtension

Extension for CUDA operations.

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

Components

nvidia::gxf::CudaStream

Holds and provides access to native `cudaStream_t`.

`nvidia::gxf::CudaStream` handle must be allocated by `nvidia::gxf::CudaStreamPool`. Its lifecycle is valid until explicitly recycled through `nvidia::gxf::CudaStreamPool.releaseStream()` or implicitly until `nvidia::gxf::CudaStreamPool` is deactivated.

You may call `stream()` to get the native `cudaStream_t` handle, and to submit GPU operations. After the submission, GPU takes over the input tensors/buffers and keeps them in use. To prevent host carelessly releasing these in-use buffers, CUDA Codelet

needs to call `record(event, input_entity, sync_cb)` to extend `input_entity`'s lifecycle until GPU completely consumes it. Alternatively, you may call `record(event, event_destroy_cb)` for native `cudaEvent_t` operations and free in-use resource via `event_destroy_cb`.

It is required to have a `nvidia::gxf::CudaStreamSync` in the graph pipeline after all the CUDA operations. See more details in `nvidia::gxf::CudaStreamSync`

- Component ID: 5683d692-7884-11eb-9338-c3be62d576be
- Defined in: gxf/cuda/cuda_stream.hpp

nvidia::gxf::CudaStreamId

Holds CUDA stream Id to deduce `nvidia::gxf::CudaStream` handle.

`stream_cid` should be `nvidia::gxf::CudaStream` component id.

- Component ID: 7982aeac-37f1-41be-ade8-6f00b4b5d47c
- Defined in: gxf/cuda/cuda_stream_id.hpp

nvidia::gxf::CudaEvent

Holds and provides access to native `cudaEvent_t` handle.

When a `nvidia::gxf::CudaEvent` is created, you'll need to initialize a native `cudaEvent_t` through `init(flags, dev_id)`, or set third party event through `initWithEvent(event, dev_id, free_fnc)`. The event keeps valid until `deinit` is called explicitly otherwise gets recycled in destructor.

- Component ID: f5388d5c-a709-47e7-86c4-171779bc64f3
- Defined in: gxf/cuda/cuda_event.hpp

nvidia::gxf::CudaStreamPool

`CudaStream` allocation.

You must explicitly call `allocateStream()` to get a valid `nvidia::gxf::CudaStream` handle. This component would hold all the its allocated `nvidia::gxf::CudaStream` entities until `releaseStream(stream)` is called explicitly or the `CudaStreamPool` component is deactivated.

- Component ID: 6733bf8b-ba5e-4fae-b596-af2d1269d0e7
- Base Type: nvidia::gxf::Allocator

Parameters

dev_id

GPU device id.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 0

stream_flags

Flag values to create CUDA streams.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 0

stream_priority

Priority values to create CUDA streams.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 0

reserved_size

User-specified file name without extension.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 1

max_size

Maximum Stream Size.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default Value: 0, no limitation.

nvidia::gxf::CudaStreamSync

Synchronize all CUDA streams which are carried by message entities.

This codelet is required to get connected in the graph pipeline after all CUDA ops codelets. When a message entity is received, it would find all of the `nvidia::gxf::CudaStreamId` in that message, and extract out each `nvidia::gxf::CudaStream`. With each `CudaStream` handle, it synchronizes all previous `nvidia::gxf::CudaStream.record()` events, along with all submitted GPU operations before this point.

> **Note**
>
> `CudaStreamSync` must be set in the graph when `nvidia::gxf::CudaStream.record()` is used, otherwise it may cause memory leak.

- Component ID: 0d1d8142-6648-485d-97d5-277eed00129c
- Base Type: nvidia::gxf::Codelet

Parameters

rx

Receiver to receive all messages carrying `nvidia::gxf::CudaStreamId`.

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

tx

Transmitter to send messages to downstream.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Transmitter

MultimediaExtension

Extension for multimedia related data types, interfaces and components in GXF Core.

- UUID: `6f2d1afc-1057-481a-9da6-a5f61fed178e`
- Version: `2.0.0`
- Author: `NVIDIA`
- License: `LICENSE`

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

NetworkExtension

Extension for communications external to a computation graph.

- UUID: `f50665e5-ade2-f71b-de2a-2380614b1725`

- Version: 1.0.0
- Author: NVIDIA
- License: LICENSE

Interfaces

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`

SerializationExtension

Extension for serializing messages.

- UUID: `bc573c2f-89b3-d4b0-8061-2da8b11fe79a`
- Version: `2.0.0`
- Author: `NVIDIA`
- License: `LICENSE`

Interfaces

nvidia::gxf::ComponentSerializer

Interface for serializing components.

- Component ID: `8c76a828-2177-1484-f841-d39c3fa47613`
- Base Type: `nvidia::gxf::Component`
- Defined in: `gxf/serialization/component_serializer.hpp`

Components

nvidia::gxf::EntityRecorder

Serializes incoming messages and writes them to a file.

- Component ID: `9d5955c7-8fda-22c7-f18f-ea5e2d195be9`
- Base Type: `nvidia::gxf::Codelet`

Parameters

receiver

Receiver channel to log.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_HANDLE`
- Handle Type: `nvidia::gxf::Receiver`

serializers

List of component serializers to serialize entities.

- Flags: `GXF_PARAMETER_FLAGS_NONE`
- Type: `GXF_PARAMETER_TYPE_CUSTOM`
- Custom Type: `std::vector&lt;nvidia::gxf::Handle&lt;nvidia::gxf::ComponentSerializer&gt;&gt;`

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`

StandardExtension

Most commonly used interfaces and components in Gxf Core.

- UUID: 8ec2d5d6-b5df-48bf-8dee-0252606fdd7e
- Version: 2.1.0
- Author: NVIDIA

- License: LICENSE

Interfaces

nvidia::gxf::Codelet

Interface for a component which can be executed to run custom code.

- Component ID: 5c6166fa-6eed-41e7-bbf0-bd48cd6e1014
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/codelet.hpp

nvidia::gxf::Clock

Interface for clock components which provide time.

- Component ID: 779e61c2-ae70-441d-a26c-8ca64b39f8e7
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/clock.hpp

nvidia::gxf::System

Component interface for systems which are run as part of the application run cycle.

- Component ID: d1febca1-80df-454e-a3f2-715f2b3c6e69
- Base Type: nvidia::gxf::Component

nvidia::gxf::Queue

Interface for storing entities in a queue.

- Component ID: 792151bf-3138-4603-a912-5ca91828dea8
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/queue.hpp

nvidia::gxf::Router

Interface for classes which are routing messages in and out of entities.

- Component ID: 8b317aad-f55c-4c07-8520-8f66db92a19e
- Defined in: gxf/std/router.hpp

nvidia::gxf::Transmitter

Interface for publishing entities.

- Component ID: c30cc60f-0db2-409d-92b6-b2db92e02cce
- Base Type: nvidia::gxf::Queue
- Defined in: gxf/std/transmitter.hpp

nvidia::gxf::Receiver

Interface for receiving entities.

- Component ID: a47d2f62-245f-40fc-90b7-5dc78ff2437e
- Base Type: nvidia::gxf::Queue
- Defined in: gxf/std/receiver.hpp

nvidia::gxf::Scheduler

A simple poll-based single-threaded scheduler which executes codelets.

- Component ID: f0103b75-d2e1-4d70-9b13-3fe5b40209be
- Base Type: nvidia::gxf::System
- Defined in: nvidia/gxf/system.hpp

nvidia::gxf::SchedulingTerm

Interface for terms used by a scheduler to determine if codelets in an entity are ready to step.

- Component ID: 184d8e4e-086c-475a-903a-69d723f95d19
- Base Type: nvidia::gxf::Component
- Defined in: gxf/std/scheduling_term.hpp

nvidia::gxf::Allocator

Provides allocation and deallocation of memory.

- Component ID: 3cdd82d0-2326-4867-8de2-d565dbe28e03
- Base Type: nvidia::gxf::Component
- Defined in: nvidia/gxf/allocator.hpp

nvidia::gxf::Monitor

Monitors entities during execution.

- Component ID: 9ccf9421-b35b-8c79-e1f0-97dc23bd38ea
- Base Type: nvidia::gxf::Component
- Defined in: nvidia/gxf/monitor.hpp

Components

nvidia::gxf::RealtimeClock

A real-time clock which runs based off a system steady clock.

- Component ID: 7b170b7b-cf1a-4f3f-997c-bfea25342381
- Base Type: nvidia::gxf::Clock

Parameters

initial_time_offset

The initial time offset used until time scale is changed manually.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_FLOAT64

initial_time_scale

The initial time scale used until time scale is changed manually.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_FLOAT64

use_time_since_epoch

If true, clock time is time since `epoch` + `initial_time_offset` at `initialize()`.Otherwise clock time is `initial_time_offset` at `initialize()`.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

nvidia::gxf::ManualClock

A manual clock which is instrumented manually.

- Component ID: 52fa1f97-eba8-472a-a8ca-4cff1a2c440f
- Base Type: nvidia::gxf::Clock

Parameters

initial_timestamp

The initial timestamp on the clock (in nanoseconds).

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

nvidia::gxf::SystemGroup

A group of systems.

- Component ID: 3d23d470-0aed-41c6-ac92-685c1b5469a0
- Base Type: nvidia::gxf::System

nvidia::gxf::MessageRouter

A router which sends transmitted messages to receivers.

- Component ID: 84fd5d56-fda6-4937-0b3c-c283252553d8
- Base Type: nvidia::gxf::Router

nvidia::gxf::RouterGroup

A group of routers.

- Component ID: ca64ee14-2280-4099-9f10-d4b501e09117
- Base Type: nvidia::gxf::Router

nvidia::gxf::DoubleBufferTransmitter

A transmitter which uses a double-buffered queue where messages are pushed to a backstage after they are published.

- Component ID: 0c3c0ec7-77f1-4389-aef1-6bae85bddc13
- Base Type: nvidia::gxf::Transmitter

Parameters

capacity

- Flags: GXF_PARAMETER_FLAGS_NONE

- Type: GXF_PARAMETER_TYPE_UINT64
- Default: 1

policy

0: pop, 1: reject, 2: fault.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64
- Default: 2

nvidia::gxf::DoubleBufferReceiver

A receiver which uses a double-buffered queue where new messages are first pushed to a backstage.

- Component ID: ee45883d-bf84-4f99-8419-7c5e9deac6a5
- Base Type: nvidia::gxf::Receiver

Parameters

capacity

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64
- Default: 1

policy

0: pop, 1: reject, 2: fault

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64
- Default: 2

nvidia::gxf::Connection

A component which establishes a connection between two other components.

- Component ID: cc71afae-5ede-47e9-b267-60a5c750a89a
- Base Type: nvidia::gxf::Component

Parameters

source

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

target

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

nvidia::gxf::PeriodicSchedulingTerm

A component which specifies that an entity shall be executed periodically.

- Component ID: d392c98a-9b08-49b4-a422-d5fe6cd72e3e
- Base Type: nvidia::gxf::SchedulingTerm

Parameters

recess_period

The recess period indicates the minimum amount of time which has to pass before the entity is permitted to execute again. The period is specified as a string containing of a number and an (optional) unit. If no unit is given the value is assumed to be in nanoseconds. Supported units are: Hz, s, ms. Example: 10ms, 10000000, 0.2s, 50Hz.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

nvidia::gxf::CountSchedulingTerm

A component which specifies that an entity shall be executed exactly a given number of times.

- Component ID: f89da2e4-fddf-4aa2-9a80-1119ba3fde05
- Base Type: nvidia::gxf::SchedulingTerm

Parameters

count

The total number of time this term will permit execution.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

nvidia::gxf::TargetTimeSchedulingTerm

A component where the next execution time of the entity needs to be specified after every tick.

- Component ID: e4aaf5c3-2b10-4c9a-c463-ebf6084149bf

- Base Type: nvidia::gxf::SchedulingTerm

Parameters

clock

The clock used to define target time.

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

nvidia::gxf::DownstreamReceptiveSchedulingTerm

A component which specifies that an entity shall be executed if receivers for a certain transmitter can accept new messages.

- Component ID: 9de75119-8d0f-4819-9a71-2aeaefd23f71
- Base Type: nvidia::gxf::SchedulingTerm

Parameters

min_size

The term permits execution if the receiver connected to the transmitter has at least the specified number of free slots in its back buffer.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

transmitter

The term permits execution if this transmitter can publish a message, i.e. if the receiver which is connected to this transmitter can receive messages.

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

nvidia::gxf::MessageAvailableSchedulingTerm

A scheduling term which specifies that an entity can be executed when the total number of messages over a set of input channels is at least a given number of messages.

- Component ID: fe799e65-f78b-48eb-beb6-e73083a12d5b
- Base Type: nvidia::gxf::SchedulingTerm

Parameters

front_stage_max_size

If set the scheduling term will only allow execution if the number of messages in the front stage does not exceed this count. It can for example be used in combination with codelets which do not clear the front stage in every tick.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_UINT64

min_size

The scheduling term permits execution if the given receiver has at least the given number of messages available.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

receiver

The scheduling term permits execution if this channel has at least a given number of messages available.

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

nvidia::gxf::MultiMessageAvailableSchedulingTerm

A component which specifies that an entity shall be executed when a queue has at least a certain number of elements.

- Component ID: f15dbeaa-afd6-47a6-9ffc-7afd7e1b4c52
- Base Type: nvidia::gxf::SchedulingTerm

Parameters

min_size

The scheduling term permits execution if all given receivers together have at least the given number of messages available.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

receivers

The scheduling term permits execution if the given channels have at least a given number of messages available.

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

nvidia::gxf::ExpiringMessageAvailableSchedulingTerm

A component which tries to wait for specified number of messages in queue for at most specified time.

- Component ID: eb22280c-76ff-11eb-b341-cf6b417c95c9
- Base Type: nvidia::gxf::SchedulingTerm

Parameters

clock

Clock to get time from.

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

max_batch_size

The maximum number of messages to be batched together.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

max_delay_ns

The maximum delay from first message to wait before submitting workload anyway.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

receiver

Receiver to watch on.

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

nvidia::gxf::BooleanSchedulingTerm

A component which acts as a boolean AND term that can be used to control the execution of the entity.

- Component ID: e07a0dc4-3908-4df8-8134-7ce38e60fbef
- Base Type: nvidia::gxf::SchedulingTerm

nvidia::gxf::AsynchronousSchedulingTerm

A component which is used to inform of that an entity is dependent upon an async event for its execution.

- Component ID: 56be1662-ff63-4179-9200-3fcd8dc38673
- Base Type: nvidia::gxf::SchedulingTerm

nvidia::gxf::GreedyScheduler

A simple poll-based single-threaded scheduler which executes codelets.

- Component ID: 869d30ca-a443-4619-b988-7a52e657f39b
- Base Type: nvidia::gxf::Scheduler

Parameters

clock

The clock used by the scheduler to define flow of time. Typical choices are a `RealtimeClock` or a `ManualClock`.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

max_duration_ms

The maximum duration for which the scheduler will execute (in ms). If not specified the scheduler will run until all work is done. If periodic terms are present this means the application will run indefinitely.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_INT64

realtime

This parameter is deprecated. Assign a clock directly.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL

- Type: GXF_PARAMETER_TYPE_BOOL

stop_on_deadlock

If enabled the scheduler will stop when all entities are in a waiting state, but no periodic entity exists to break the dead end. Should be disabled when scheduling conditions can be changed by external actors, for example by clearing queues manually.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

nvidia::gxf::MultiThreadScheduler

A multi thread scheduler that executes codelets for maximum throughput.

- Component ID: de5e0646-7fa5-11eb-a5c4-330ebfa81bbf
- Base Type: nvidia::gxf::Scheduler

Parameters

check_recession_perios_ms

The maximum duration for which the scheduler would wait (in ms) when an entity is not ready to run yet.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

clock

The clock used by the scheduler to define flow of time. Typical choices are a `RealtimeClock` or a `ManualClock`.

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

max_duration_ms

The maximum duration for which the scheduler will execute (in ms). If not specified the scheduler will run until all work is done. If periodic terms are present this means the application will run indefinitely.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_INT64

stop_on_deadlock

If enabled the scheduler will stop when all entities are in a waiting state, but no periodic entity exists to break the dead end. Should be disabled when scheduling conditions can be changed by external actors, for example by clearing queues manually.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

worker_thread_number

Number of threads.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64
- Default: 1

nvidia::gxf::BlockMemoryPool

A memory pools which provides a maximum number of equally sized blocks of memory.

- Component ID: 92b627a3-5dd3-4c3c-976c-4700e8a3b96a
- Base Type: nvidia::gxf::Allocator

Parameters

block_size

The size of one block of memory in byte. Allocation requests can only be fulfilled if they fit into one block. If less memory is requested still a full block is issued.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

do_not_use_cuda_malloc_host

If enabled operator new will be used to allocate host memory instead of `cudaMallocHost`.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL
- Default: True

num_blocks

The total number of blocks which are allocated by the pool. If more blocks are requested allocation requests will fail.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

storage_type

The memory storage type used by this allocator. Can be kHost (0) or kDevice (1) or kSystem (2).

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT32
- Default: 0

nvidia::gxf::UnboundedAllocator

Allocator that uses dynamic memory allocation without an upper bound.

- Component ID: c3951b16-a01c-539f-d87e-1dc18d911ea0
- Base Type: nvidia::gxf::Allocator

Parameters

do_not_use_cuda_malloc_host

If enabled operator new will be used to allocate host memory instead of `cudaMallocHost`.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL
- Default: True

nvidia::gxf::Tensor

A component which holds a single tensor.

- Component ID: 377501d6-9abf-447c-a617-0114d4f33ab8
- Defined in: gxf/std/tensor.hpp

nvidia::gxf::Timestamp

Holds message publishing and acquisition related timing information.

- Component ID: d1095b10-5c90-4bbc-bc89-601134cb4e03
- Defined in: gxf/std/timestamp.hpp

nvidia::gxf::Metric

Collects, aggregates, and evaluates metric data.

- Component ID: f7cef803-5beb-46f1-186a-05d3919842ac
- Base Type: nvidia::gxf::Component

Parameters

aggregation_policy

Aggregation policy used to aggregate individual metric samples. Choices:{mean, min, max}.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_STRING

lower_threshold

Lower threshold of the metric's expected range.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_FLOAT64

upper_threshold

Upper threshold of the metric's expected range.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_FLOAT64

nvidia::gxf::JobStatistics

Collects runtime statistics.

- Component ID: 2093b91a-7c82-11eb-a92b-3f1304ecc959
- Base Type: nvidia::gxf::Component

Parameters

clock

The clock component instance to retrieve time from.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE

- Handle Type: nvidia::gxf::Clock

codelet_statistics

If set to true, JobStatistics component will collect performance statistics related to codelets.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_BOOL

json_file_path

If provided, all the collected performance statistics data will be dumped into a json file.

- Flags: GXF_PARAMETER_FLAGS_OPTIONAL
- Type: GXF_PARAMETER_TYPE_STRING

nvidia::gxf::Broadcast

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

- Component ID: 3daadb31-0bca-47e5-9924-342b9984a014
- Base Type: nvidia::gxf::Codelet

Parameters

mode

The broadcast mode. Can be Broadcast or RoundRobin.

- Flags: GXF_PARAMETER_FLAGS_NONE

- Type: GXF_PARAMETER_TYPE_CUSTOM

source

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

nvidia::gxf::Gather

All messages arriving on any input channel are published on the single output channel.

- Component ID: 85f64c84-8236-4035-9b9a-3843a6a2026f
- Base Type: nvidia::gxf::Codelet

Parameters

sink

The output channel for gathered messages.

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

tick_source_limit

Maximum number of messages to take from each source in one tick. 0 means no limit.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_INT64

nvidia::gxf::TensorCopier

Copies tensor either from host to device or from device to host.

- Component ID: c07680f4-75b3-189b-8886-4b5e448e7bb6
- Base Type: nvidia::gxf::Codelet

Parameters

allocator

Memory allocator for tensor data

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

mode

Configuration to select what tensors to copy:

1. kCopyToDevice (0) - copies to device memory, ignores device allocation
2. kCopyToHost (1) - copies to pinned host memory, ignores host allocation
3. kCopyToSystem (2) - copies to system memory, ignores system allocation.

- Flags: GXF_PARAMETER_FLAGS_NONE

- Type: GXF_PARAMETER_TYPE_INT32

receiver

Receiver for incoming entities.

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

transmitter

Transmitter for outgoing entities.

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

nvidia::gxf::TimedThrottler

Publishes the received entity respecting the timestamp within the entity.

- Component ID: ccf7729c-f62c-4250-5cf7-f4f3ec80454b
- Base Type: nvidia::gxf::Codelet

Parameters

execution_clock

Clock on which the codelet is executed by the scheduler.

- Flags: GXF_PARAMETER_FLAGS_NONE

- Type: GXF_PARAMETER_TYPE_HANDLE
- Handle Type: nvidia::gxf::Clock

receiver

Channel to receive messages that need to be synchronized.

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

scheduling_term

Scheduling term for executing the codelet.

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

throttling_clock

Clock which the received entity timestamps are based on.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_HANDLE

- Handle Type: nvidia::gxf::Clock

transmitter

Transmitter channel publishing messages at appropriate timesteps.

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

nvidia::gxf::Vault

Safely stores received entities for further processing.

- Component ID: 1108cb8d-85e4-4303-ba02-d27406ee9e65
- Base Type: nvidia::gxf::Codelet

Parameters

drop_waiting

If too many messages are waiting the oldest ones are dropped.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_BOOL

max_waiting_count

The maximum number of waiting messages. If exceeded the codelet will stop pulling messages out of the input queue.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_UINT64

source

Receiver from which messages are taken and transferred to the vault.

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

nvidia::gxf::Subgraph

Helper component to import a subgraph.

- Component ID: 576eedd7-7c3f-4d2f-8c38-8baa79a3d231
- Base Type: nvidia::gxf::Component

Parameters

location

`Yaml` source of the subgraph.

- Flags: GXF_PARAMETER_FLAGS_NONE
- Type: GXF_PARAMETER_TYPE_STRING

nvidia::gxf::EndOfStream

A component which represents end-of-stream notification.

- Component ID: 8c42f7bf-7041-4626-9792-9eb20ce33cce

- Defined in: gxf/std/eos.hpp

nvidia::gxf::Synchronization

Component to synchronize messages from multiple receivers based on the `acq_time`.

- Component ID: f1cb80d6-e5ec-4dba-9f9e-b06b0def4443
- Base Type: nvidia::gxf::Codelet

Parameters

inputs

All the inputs for synchronization. Number of inputs must match that of the outputs.

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

outputs

All the outputs for synchronization. Number of outputs must match that of the inputs.

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

signed char

- Component ID: 83905c6a-ca34-4f40-b474-cf2cde8274de

unsigned char

- Component ID: d4299e15-0006-d0bf-8cbd-9b743575e155

short int

- Component ID: 9e1dde79-3550-307d-e81a-b864890b3685

short unsigned int

- Component ID: 958cbdef-b505-bcc7-8a43-dc4b23f8cead

int

- Component ID: b557ec7f-49a5-08f7-a35e-086e9d1ea767

unsigned int

- Component ID: d5506b68-5c86-fedb-a2a2-a7bae38ff3ef

long int

- Component ID: c611627b-6393-365f-d234-1f26bfa8d28f

long unsigned int

- Component ID: c4385f5b-6e25-01d9-d7b5-6e7cadc704e8

float

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

double

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

bool

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

Data Flow Tracking

> ⚠ **Warning**
>
> Data Flow Tracking is currently not supported between multiple fragments in a distributed application.

The Holoscan SDK provides the Data Flow Tracking APIs as a mechanism to profile your application and analyze the fine-grained timing properties and data flow between operators in the graph of a fragment.

Currently, data flow tracking is only supported between the root operators and leaf operators of a graph and in simple cycles in a graph (support for tracking data flow between any pair of operators in a graph is planned for the future).

- A *root operator* is an operator without any predecessor nodes
- A *leaf operator* (also known as a *sink operator*) is an operator without any successor nodes.

When data flow tracking is enabled, every message is tracked from the root operators to the leaf operators and in cycles. Then, the maximum (worst-case), average and minimum end-to-end latencies of one or more paths can be retrieved using the Data Flow Tracking APIs.

> **Tip**
>
> - The end-to-end latency between a root operator and a leaf operator is the time taken between the start of a root operator and the end of a leaf operator. Data Flow Tracking enables the support to track the end-to-end latency of every message being passed between a root operator and a leaf operator.

- The reported end-to-end latency for a cyclic path is the time taken between the start of the first operator of a cycle and the time when a message is again received by the first operator of the cycle.

The API also provides the ability to retrieve the number of messages sent from the root operators.

Tip

- The Data Flow Tracking feature is also illustrated in the flow_tracker
- Look at the
  `<a href="api/cpp/classholoscan_1_1DataFlowTracker.html#_CPPv4N8holoscan1 </a>`
  and
  `<a href="api/python/holoscan_python_api_core.html#holoscan.core.DataFlowTr`
  API documentation for exhaustive definitions

Enabling Data Flow Tracking

Before an application (`C++` / `python`) is run with the `run()` method, data flow tracking can be enabled by calling the `track()` method in `C++` and using the `Tracker` class in `python`.

Ingested Tab Module

Retrieving Data Flow Tracking Results

After an application has been run, data flow tracking results can be accessed by various functions:

1. `print()` (`C++` / `python`)
    - Prints all data flow tracking results including end-to-end latencies and the number of source messages to the standard output.
2. `get_num_paths()` (`C++` / `python`)
    - Returns the number of paths between the root operators and the leaf operators.
3. `get_path_strings()` (`C++` / `python`)
    - Returns a vector of strings, where each string represents a path between the root operators and the leaf operators. A path is a comma-separated list of operator names.
4. `get_metric()` (`C++` / `python`)
    - Returns the value of different metrics based on the arguments.
    - `get_metric(std::string pathstring, holoscan::DataFlowMetric metric)` returns the value of a metric `metric` for a path `pathstring`. The metric can be one of the following:
        - `holoscan::DataFlowMetric::kMaxE2ELatency` (`python`): the maximum end-to-end latency in the path
        - `holoscan::DataFlowMetric::kAvgE2ELatency` (`python`): the average end-to-end latency in the path
        - `holoscan::DataFlowMetric::kMinE2ELatency` (`python`): the minimum end-to-end latency in the path
        - `holoscan::DataFlowMetric::kMaxMessageID` (`python`): the message number or ID which resulted in the maximum end-to-end latency
        - `holoscan::DataFlowMetric::kMinMessageID` (`python`): the message number or ID which resulted in the minimum end-to-end latency

- `get_metric(holoscan::DataFlowMetric metric = DataFlowMetric::kNumSrcMessages)` returns a map of source operator and its edge, and the number of messages sent from the source operator to the edge.

In the above example, the data flow tracking results can be printed to the standard output like the following:

Ingested Tab Module

Customizing Data Flow Tracking

Data flow tracking can be customized using a few, optional configuration parameters. The `track()` method (`C++` / `Tracker class in python`) can be configured to skip a few messages at the beginning of an application's execution as a *warm-up* period. It is also possible to discard a few messages at the end of an application's run as a *wrap-up* period. Additionally, outlier end-to-end latencies can be ignored by setting a latency threshold value which is the minimum latency below which the observed latencies are ignored.

> **Tip**
>
> For effective benchmarking, it is common practice to include warm-up and cool-down periods by skipping the initial and final messages.

Ingested Tab Module

The default values of these parameters of `track()` are as follows:

- `kDefaultNumStartMessagesToSkip`: 10
- `kDefaultNumLastMessagesToDiscard`: 10
- `kDefaultLatencyThreshold`: 0 (do not filter out any latency values)

These parameters can also be configured using the helper functions: `set_skip_starting_messages`, `set_discard_last_messages` and `set_skip_latencies`.

Logging

The Data Flow Tracking API provides the ability to log every message's graph-traversal information to a file. This enables developers to analyze the data flow at a granular level. When logging is enabled, every message's received and sent timestamps at every operator between the root and the leaf operators are logged after a message has been processed at the leaf operator.

The logging is enabled by calling the `enable_logging` method in `C++` and by providing the `filename` parameter to `Tracker` in `python`.

Ingested Tab Module

The logger file logs the paths of the messages after a leaf operator has finished its `compute` method. Every path in the logfile includes an array of tuples of the form:

"(root operator name, message receive timestamp, message publish timestamp) -> ... -> (leaf operator name, message receive timestamp, message publish timestamp)".

This log file can further be analyzed to understand latency distributions, bottlenecks, data flow and other characteristics of an application.

Video Pipeline Latency Tool

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

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

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

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

Requirements

Hardware

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

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

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

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).

Installation

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
```

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
```

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 ..
```

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**:

   _images/ubuntu_automatic_login.png

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

   _images/ubuntu_lock_settings.png

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).

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. 24 *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
```

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. 25 *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
```

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.

Fig. 26 *SDI Cable Between Channel 1 and 2 of a Single AJA Corvid 44 Capture Card*

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:

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

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.

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.

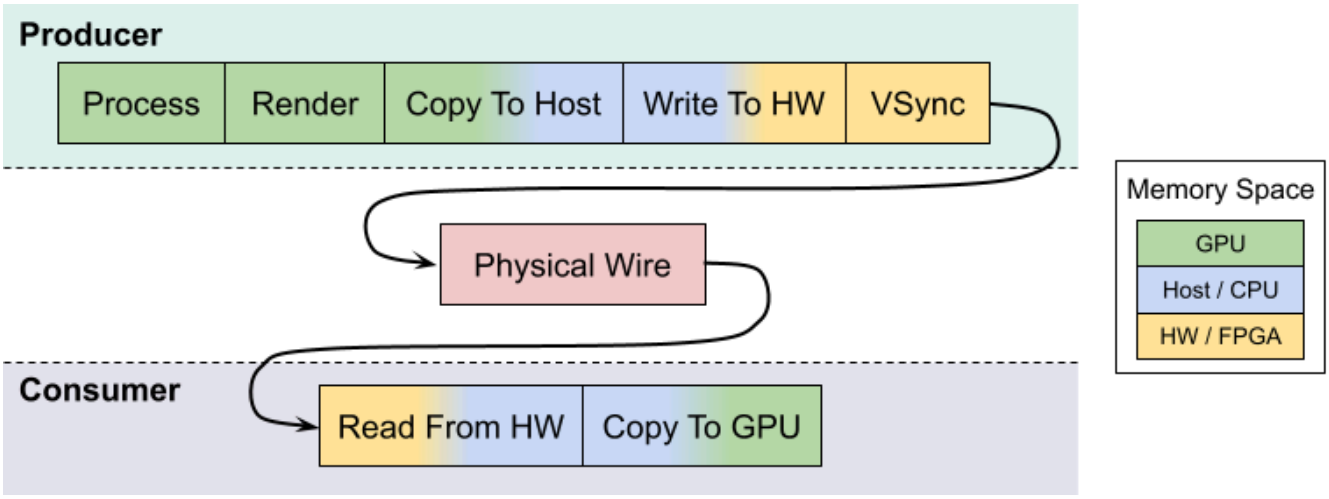


Fig. 27 *Latency Tool Frame Lifespan (RDMA Disabled)*

1. **CUDA Processing**

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

2. **Render on GPU**

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

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

3. **Copy To Host**

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

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

4. **Write to HW**

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

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

5. **VSync Wait**

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

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

6. **Wire Time**

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

7. **Read From HW**

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

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

8. **Copy to GPU**

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

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

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

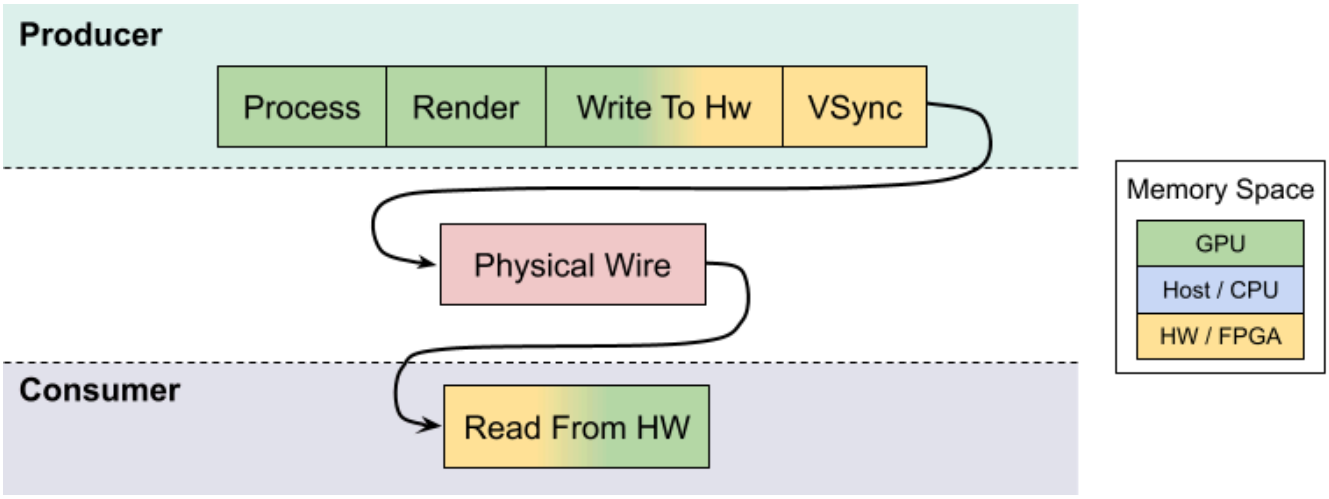


Fig. 28 *Latency Tool Frame Lifespan (RDMA Enabled)*

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)

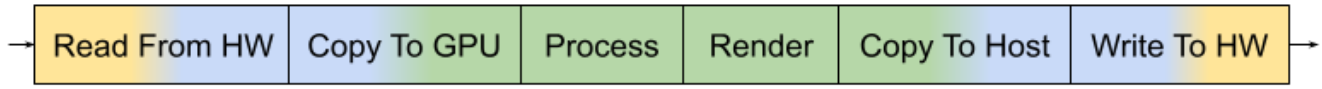


Fig. 29 *Real Application Frame Lifespan*

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)

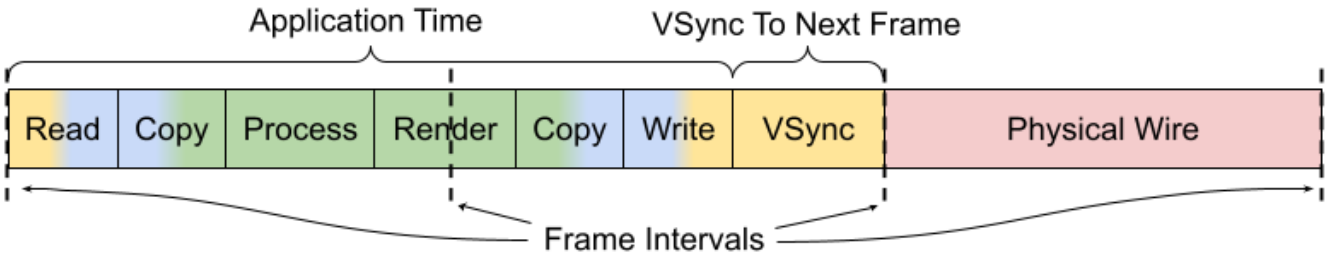


Fig. 30 *Final Estimated Latency with VSync and Physical Wire Time*

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).

Reducing Latency With RMDA

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

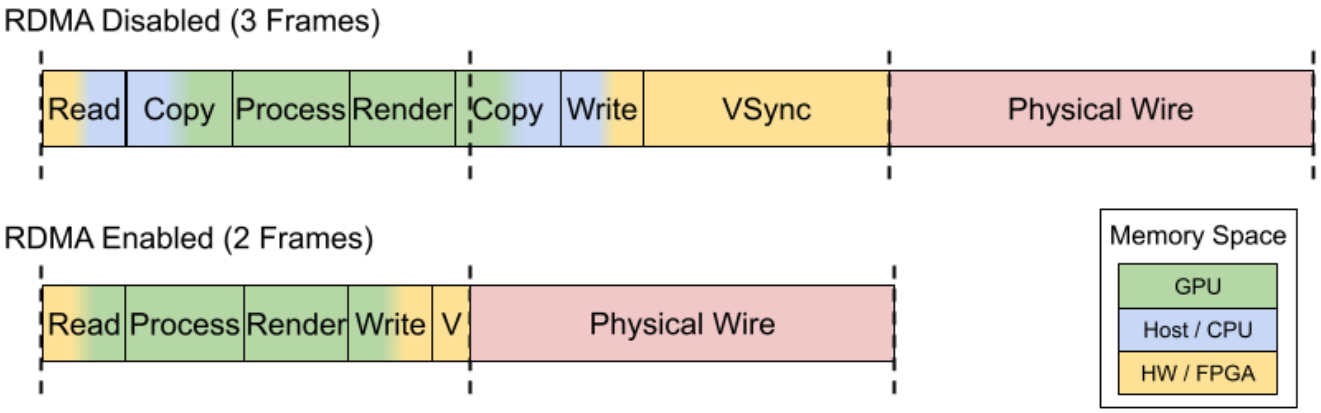


Fig. 31 *Reducing Latency With RDMA*

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
```

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
```

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.

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.

Producers

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

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.

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.

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).

Consumers

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

V4L2 (Onboard HDMI Capture Card)

This consumer (`v4l2`) uses the V4L2 API directly in order to capture frames using the HDMI capture card that is onboard some of the NVIDIA Developer Kits.

V4L2 Consumer Notes:

- The onboard HDMI capture card is locked to a specific frame resolution and 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.

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 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.

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).

Troubleshooting

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

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

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

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

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

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

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

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

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

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

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

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

Solution: Try the following:

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

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