Creating Operators

Table of contents

C++ Operators

Python Operators

Advanced Topics

List of Figures

Figure 0. Graphviz 75ff72e35736eb2b8ccd3233ed065e0b088d5b17

Figure 1. Holoscan Tensor Interoperability

Figure 2. Graphviz 1c255a991739fa90e384e904715221679a5f67ca

Figure 3. Graphviz D5bf8fa022b9f3584e20248ff54a8f06a49e3b63

Figure 4. Graphviz 8cdb373b9b4010a5e5ba0fd13bbf5f2d32b72fc9

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.

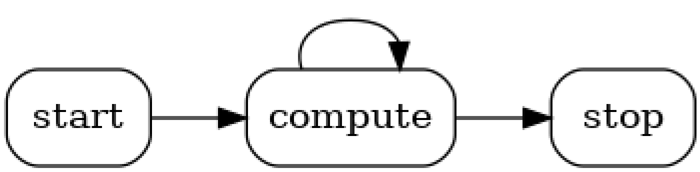

Fig. 15 *The sequence of method calls in the lifecycle of a Holoscan Operator*

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
>

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<IOSpec*>> (Parameter<std::vector<IOSpec*>> receivers_) and calling spec.param(receivers_, "receivers", "Input Receivers", "List of input receivers.", {}); as done for PingRxOp in the native operator ping example.

Listing 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.

Tip

The manual codelet wrapping mechanism described below is no longer necessary in order to make use of a GXF Codelet as a Holoscan operator. There is a new <a href="api/cpp/classholoscan_1_1ops_1_1GXFCodeletOp.html#_CPPv4N8holoscan3 which allows directly using an existing GXF codelet via <a href="api/cpp/classholoscan_1_1Fragment.html#_CPPv4I00Dp0EN8holoscan8Frag without having to first create a wrapper class for it. Similarly there is now also a <a href="api/cpp/classholoscan_1_1GXFComponentResource.html#_CPPv4N8holosca class which allows a GXF Component to be used as a Holoscan resource via <a href="api/cpp/classholoscan_1_1Fragment.html#_CPPv4I00Dp0EN8holoscan8Frag . A detailed example of how to use each of these is provided for both C++ and Python applications in the **examples/import_gxf_components** folder.

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<nvidia::gxf::Receiver>>`
	or
	`nvidia::gxf::Handle<nvidia::gxf::Transmitter>>`

`std::shared_ptr<holoscan::Resource>>`	`nvidia::gxf::Handle<T>>` 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.

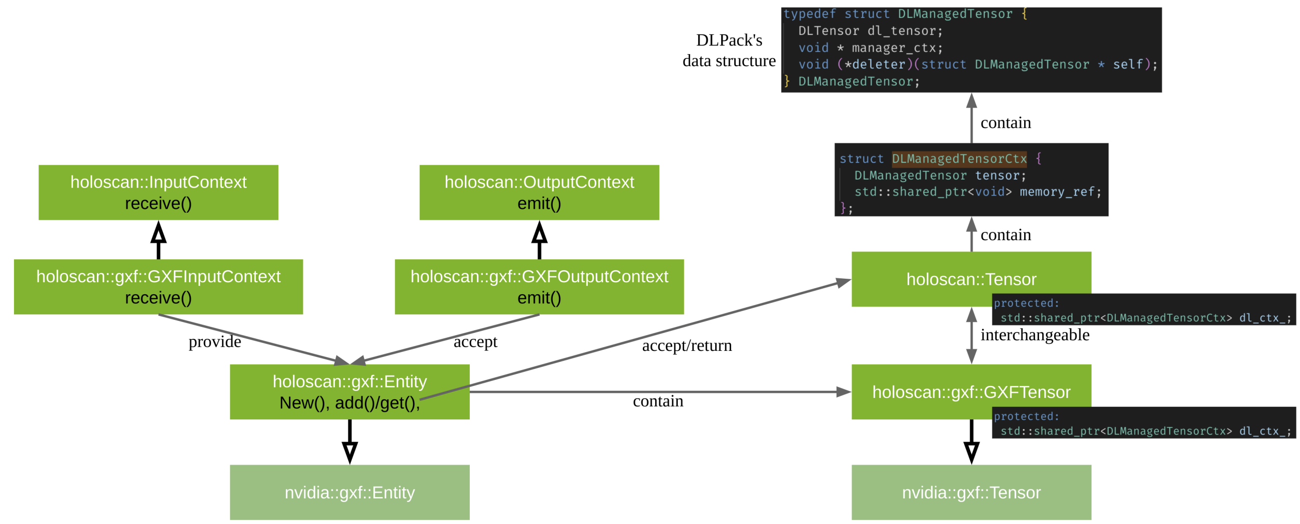

Fig. 16 *Supporting Tensor Interoperability*

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

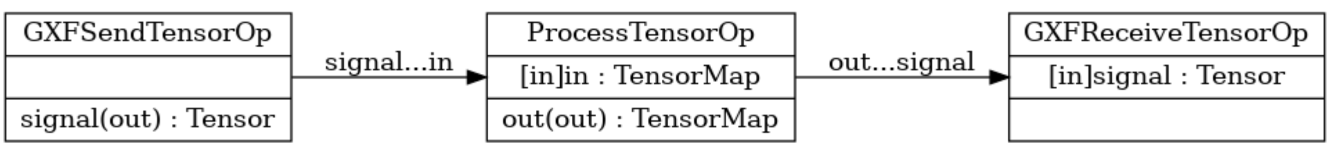

Fig. 17 *The tensor interoperability between C++ native operator and GXF operator*

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.

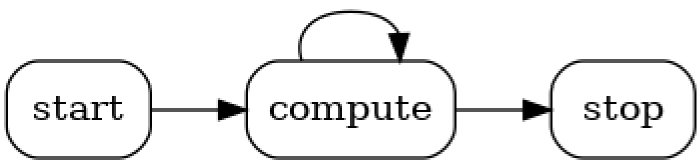

Fig. 18 *The sequence of method calls in the lifecycle of a Holoscan Operator*

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"

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

Wrapping an operator developed in C++ for use from Python is covered in a separate section on creating C++ operator Python bindings.

Tip

As of Holoscan 2.1, there is a <a href="api/python/holoscan_python_api_operators.html#holoscan.operators.GXFC class which can be used to easily wrap an existing GXF codelet from Python without having to first write an underlying C++ wrapper class for it. Similarly there is now also a <a href="api/python/holoscan_python_api_resources.html#holoscan.resources.GXFC class which allows a GXF Component to be used as a Holoscan resource from Python applications. A detailed example of how to use each of these is provided for Python applications in the **examples/import_gxf_components** folder.

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:

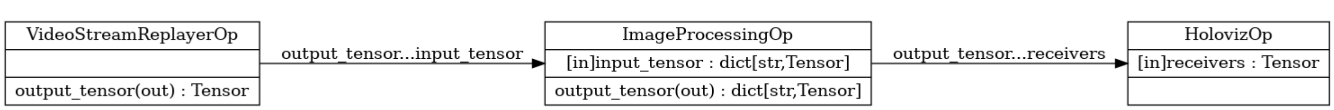

Fig. 19 *The tensor interoperability between Python native operator and C++-based Python GXF operator*

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

Listing 17 *examples/tensor_interop/python/tensor_interop.py*

```
def compute(self, op_input, op_output, context): # in_message is a dict of tensors 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 will be a dict of tensors 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 Entity (see the holoviz entry in apps/endoscopy_tool_tracking/python/endoscopy_tool_tracking.yaml).

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

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.

Advanced Topics

Further customizing inputs and outputs

This section complements the information above on basic input and output port configuration given separately in the C++ and Python operator creation guides. The concepts described here are the same for either the C++ or Python APIs.

By default, both the input and output ports of an Operator will use a double-buffered queue that has a capacity of one message and a policy that is set to error if a message arrives while the queue is already full. A single `MessageAvailableCondition` (`C++` / `Python`)) condition is automatically placed on the operator for each input port so that the `compute` method will not be called until a single message is available at each port. Similarly each output port has a `DownstreamMessageAffordableCondition` (`C++` / `Python`) condition that does not let the operator call `compute` until any operators connected downstream have space in their receiver queue for a single message. These default conditions ensure that messages never arrive at a queue when it is already full and that a message has already been received whenever the `compute` method is called. These default conditions make it relatively easy to connect a pipeline where each operator calls compute in turn, but may not be suitable for all applications. This section covers how the default behavior can be overridden on request.

> **Note**
>
> Overriding operator port properties is an advanced topic. Developers may want to skip this section until they come across a case where the default behavior is not sufficient for their application.

To override the properties of the queue used for a given port, the `connector` (`C++` / `Python`) method can be used as shown in the example below. This example also shows how the `condition` (`C++` / `Python`) method can be used to change the condition type placed on the Operator by a port. In general, when an operator has multiple conditions, they are AND combined, so the conditions on **all** ports must be satisfied before an operator can call `compute`.

Ingested Tab Module

To learn more about overriding connectors and/or conditions there is a multi_branch_pipeline example which overrides default conditions to allow two branches of a pipeline to run at different frame rates. There is also an example of increasing the queue sizes available in this Python queue policy test application.