Resources

Table of contents

Allocator

Clock

Transmitter (advanced)

Receiver (advanced)

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. This memory pool is easy to use and is recommended for initial prototyping. Once an application is working, switching to a `BlockMemoryPool` instead may help provide additional performance.

BlockMemoryPool

This is a memory pool which provides a user-specified number of equally sized blocks of memory. Using this memory pool provides a way to allocate memory blocks once and reuse the blocks on each subsequent call to an Operator's `compute` method. This saves overhead relative to allocating memory again each time `compute` is called. For the built-in operators which accept a memory pool parameer, there is a section in it's API docstrings titled "Device Memory Requirements" which provides guidance on the `num_blocks` and `block_size` needed for use with this memory pool.

- 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&lt;Rep, Period&gt;` 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.