Data Flow Tracking

Table of contents

> ⚠ **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.