Debugging

Table of contents

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
```