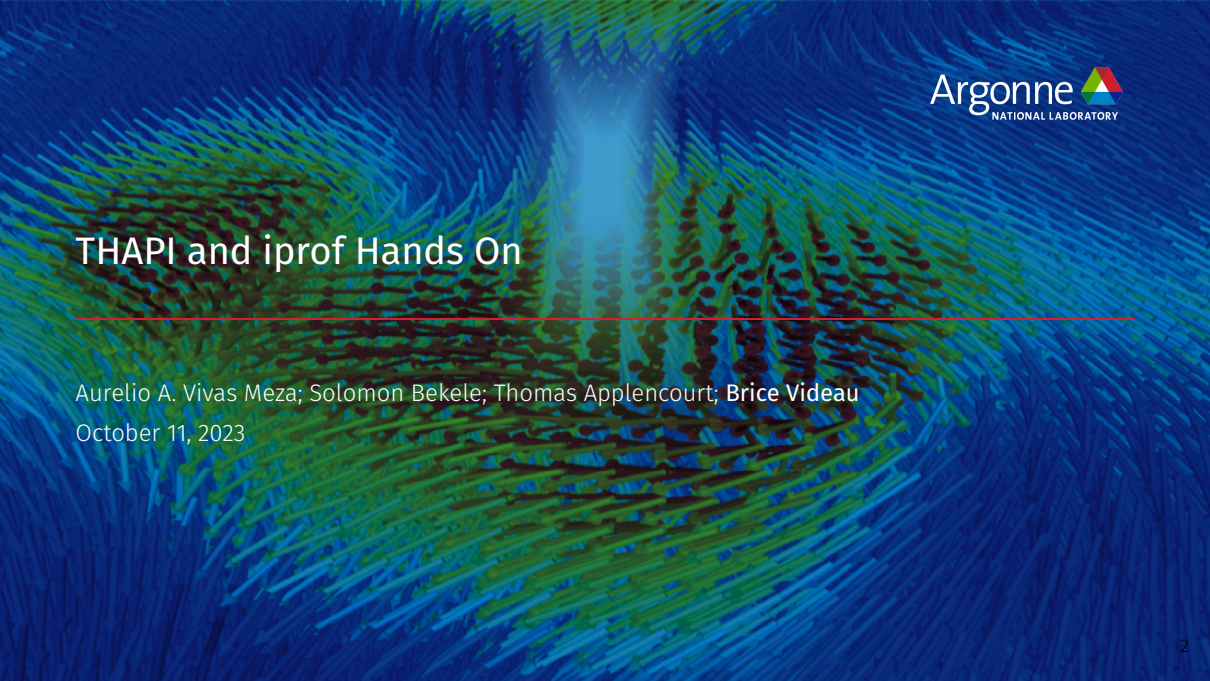
October 10-12, 2023

ALCF Hands-on HPC Workshop



THAPI and iprof Hands On

Aurelio A. Vivas Meza; Solomon Bekele; Thomas Applencourt; **Brice Videau**

October 11, 2023

Table of Contents

Installation

Install via Spack

If THAPI is not already installed on your system, You can easily install it via Spack:

- Clone Spack:

```
git clone -c feature.manyFiles=true -b releases/latest https://github.com/spack/spack.git
```

- Load Spack into your environment:

```
. spack/share/spack/setup-env.sh
```

- Clone THAPI-spack:

```
git clone https://github.com/argonne-lcf/THAPI-spack
```

- Add THAPI-spack to Spack and install:

```
spack repo add ./THAPI-spack/
spack install thapi
```

Install specific versions via Spack

Specific versions can be specified on the spack install line.

- Example using *0.0.10*:

```
spack install thapi@0.0.10
```

- Can also use git hash or branch name

For more detailed instruction see:
https://spack-tutorial.readthedocs.io/en/latest/tutorial_basics.html

Load THAPI via Spack

To use a package installed by Spack.

- Load Spack into your environment (if it is not already loaded):

```
. spack/share/spack/setup-env.sh
```

- Load the default THAPI version:

```
spack load thapi
```

- Or load a specific version, *0.0.10* for example:

```
spack load thapi@0.0.10
```

Building THAPI from Sources

Sometimes, you may want to build THAPI from source. It is easy to achieve using spack.

- Load Spack into your environment (if it is not already loaded):

```
. spack/share/spack/setup-env.sh
```

- Clone THAPI-spack:

```
git clone https://github.com/argonne-lcf/THAPI
```

- Create a shell with the THAPI build environment:

```
spack build-env thapi bash
```

- Build as a regular autotools project:

```
cd THAPI
./autogen.sh
mkdir build
cd build/
../configure [--prefix=...]
make install
```

Basic Usage: iprof

Loading THAPI for the Workshop

As the spack install can be long, installation on Polaris has been done already.

- Load Spack into your environment on Polaris:

```
cd /eagle/projects/fallwkshp23/THAPI/
. spack/share/spack/setup-env.sh
```

- Load THAPI:

```
spack load thapi
```

- Test that iprof is working:

```
> iprof --help
iprof: a tracer / summarizer of OpenCL, L0, CUDA, HIP, and OMPT driver calls
Usage:
 iprof -h | --help
 iprof [option]... <application> <application-arguments>
 iprof [option]... -r [<trace>]...
[...]
```

Profiling a Simple CUDA Applications with iprof: Polaris

- Several sample applications are located inside the `/eagle/projects/fallwkshp23/THAPI` directory.
- Each application is located in 3 sub-directories:
  - *CUDA*: as simple addition CUDA application
  - *SYCL*: as simple hello world SYCL application that runs on the GPU
  - *OMP*: as simple benchmark OPENMP offload application taht runs on the GPU
- Some directories may contain a *README.md* file with important information regarding the environment required to run the application.

Profiling a Simple CUDA Applications with iprof

Example: running the simple CUDA application:

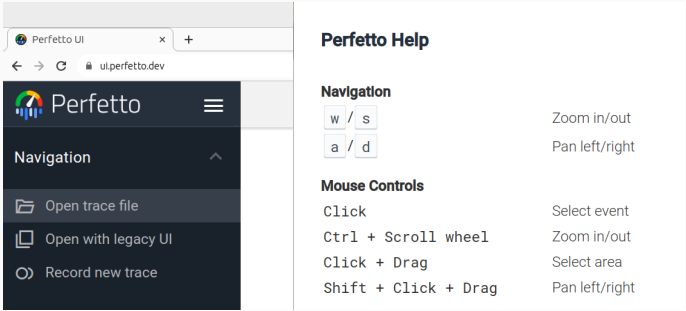
```
> cd /eagle/projects/fallwkshp23/THAPI/CUDA
> ./cuda_hello
Max error: 0
> iprof ./cuda_hello
Trace location: /home/videau/lttng-traces/iprof-20231010-025849

BACKEND_CUDA | 1 Hostnames | 1 Processes | 1 Threads |

                      Name |     Time | Time(%) | Calls |  Average |      Min |      Max | Error |
                   cuInit | 154.38ms |  43.41% |     1 | 154.38ms | 154.38ms | 154.38ms |     0 |
          cuCtxSynchronize | 122.67ms |  34.49% |     1 | 122.67ms | 122.67ms | 122.67ms |     0 |
 cuDevicePrimaryCtxRetain |  71.31ms |  20.05% |     1 |  71.31ms |  71.31ms |  71.31ms |     0 |
      cuDeviceTotalMem_v2 |   3.01ms |   0.85% |     4 | 753.07us | 731.66us | 772.24us |     0 |
     cuDeviceGetAttribute |   1.81ms |   0.51% |   392 |   4.61us |    838ns | 172.09us |     0 |
         cuGetProcAddress |   1.64ms |   0.46% |   373 |   4.39us |    908ns | 531.57us |     0 |
[...]
           cuCtxGetDevice |   2.44us |   0.00% |     1 |   2.44us |   2.44us |   2.44us |     0 |
         cuDeviceGetCount |   1.47us |   0.00% |     1 |   1.47us |   1.47us |   1.47us |     0 |
cuDevicePrimaryCtxRelease |          |         |     1 |          |          |          |     1 |
            cuModuleUnload |          |         |     1 |          |          |          |     1 |
                    Total | 355.67ms | 100.00% |   803 |                                      |     2 |

Device profiling | 1 Hostnames | 1 Processes | 1 Threads | 1 Devices | 1 Subdevices |

 Name |     Time | Time(%) | Calls |  Average |      Min |      Max |
  add | 122.68ms | 100.00% |     1 | 122.68ms | 122.68ms | 122.68ms |
Total | 122.68ms | 100.00% |     1 |
```

Replaying a Trace

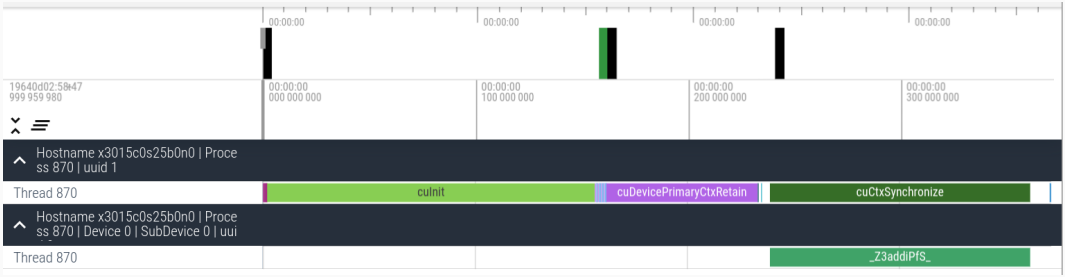
- By default traces are stored on the disc in the *~/lttng-traces/* directory.
- Last trace can be replayed using the *-r* option: *iprof -r*
- A specific trace can be replayed using the *-r* option followed by the trace location. If using the trace from the previous slide:
  iprof -r /home/videau/lttng-traces/iprof-20231010-025849

Generating a Timeline

- A timeline can be generated using the `-l` option.
- Often used while replaying a trace, like this:
  `iprof -l -r /home/videau/lttng-traces/iprof-20231010-025849`
- Generates a `out.pftrace` file that can be loaded in Perfetto UI:
  `https://ui.perfetto.dev/`



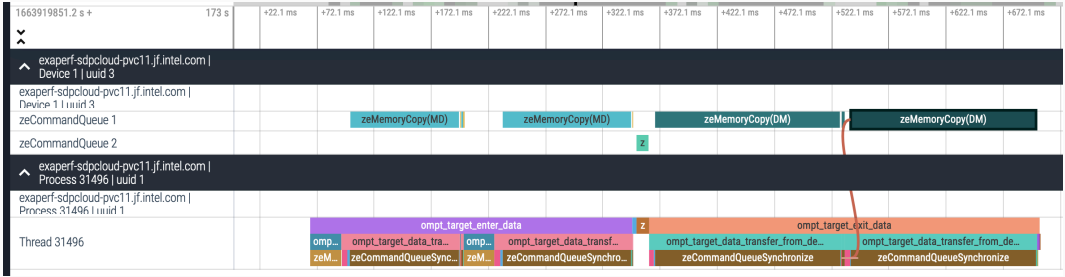
Perfetto Timeline



Argonne NATIONAL LABORATORY

Perfetto Timeline: OpenMP on Level Zero

The OpenMP compiler on Polaris does not support OMPT. But the Intel compiler on JLSE does:



Displaying a Trace

- A trace can be generated using the `-t` option.
- Often used while replaying a trace, like this:
  `iprof -l -t /home/videau/lttng-traces/iprof-20231010-025849`
- Generates a trace containing all the API calls (and more)

Simple CUDA Trace

```
1  [...]
2  02:58:48.234853435 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuModuleGetFunction_entry: { hfunc: 0x00007ffc1ebf8048,
   ↪ hmod: 0x00000000012e43d0, name: 0x0000000000046dcbd, name_val: "_Z3addiPfS_" }
3  02:58:48.234858185 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuModuleGetFunction_exit: { cuResult: CUDA_SUCCESS,
   ↪ hfunc_val: 0x00000000012e1b60 }
4  02:58:48.234867264 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemAllocManaged_entry: { dptr: 0x00007ffc1ebf8380,
   ↪ bytesize: 4194304, flags: 1 }
5  02:58:48.234900648 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemAllocManaged_exit: { cuResult: CUDA_SUCCESS,
   ↪ dptr_val: 0x00001471e4000000 }
6  02:58:48.234902604 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemAllocManaged_entry: { dptr: 0x00007ffc1ebf8378,
   ↪ bytesize: 4194304, flags: 1 }
7  02:58:48.234911613 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemAllocManaged_exit: { cuResult: CUDA_SUCCESS,
   ↪ dptr_val: 0x00001471e4400000 }
8  02:58:48.239007688 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuLaunchKernel_entry: { f: 0x00000000012e1b60, gridDimX:
   ↪ 1, gridDimY: 1, gridDimZ: 1, blockDimX: 1, blockDimY: 1, blockDimZ: 1, sharedMemBytes: 0, hStream: 0x0000000000000000,
   ↪ kernelParams: 0x00007ffc1ebf8300, extra: 0x0000000000000000, extra_vals: [  ] }
9  02:58:48.239034018 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda_profiling:event_profiling: { hStart: 0x00000000012e7810,
   ↪ hStop: 0x00000000012e1780 }
10 02:58:48.239035415 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuLaunchKernel_exit: { cuResult: CUDA_SUCCESS }
11 02:58:48.239038627 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuCtxSynchronize_entry: {  }
12 02:58:48.361712445 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuCtxSynchronize_exit: { cuResult: CUDA_SUCCESS }
13 02:58:48.370911152 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemFree_v2_entry: { dptr: 0x00001471e4000000 }
14 02:58:48.371248487 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemFree_v2_exit: { cuResult: CUDA_SUCCESS }
15 02:58:48.371249954 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemFree_v2_entry: { dptr: 0x00001471e4400000 }
16 02:58:48.371527993 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemFree_v2_exit: { cuResult: CUDA_SUCCESS }
17 02:58:48.371538120 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda_profiling:event_profiling_results: { hStart:
   ↪ 0x00000000012e7810, hStop: 0x00000000012e1780, startStatus: CUDA_SUCCESS, stopStatus: CUDA_SUCCESS, status: CUDA_SUCCESS,
   ↪ milliseconds: 122.68134307861328 }
18 [...]
```

Cleaning up

- When *iprof* seems to be acting weird, it is usually following a *Ctrl-C* or a crash.
- Cleaning up leftover daemons usually solves issues: *iprof --clean-up*

Advanced Usage: THAPI

THAPI Consists of Two Big Components

Open source at: *https://github.com/argonne-lcf/THAPI*

- The tracing of events
  - Use low level tracing: Linux Tracing Toolkit Next Generation (LTTng):
  - Tracepoints are generated from APIs' headers
- The parsing of the trace
  - Use Babeltrace2 library and tools
  - Pretty Printer, Tally, Timeline/Flamegraph, ...

Supported APIs

- OpenCL, Level Zero, Cuda Runtime/Driver, HIP
- OMPT

iprof is an orchestrator around THAPI, lttng, and babeltrace2.

Custom Trace Collection

iprof is a rather complex script, but simpler tools dedicated to each API exist. Example: *tracer_cuda.sh*

- Will capture a CUDA trace
- Provides additional options, useful for debugging applications
- Collected trace can be replayed with *iprof*
- Can be used as a base to design user tailored tracers

How Tracers Work - 1

- Use LTTng to setup a capture daemon

```
lttng-sessiond --daemonize --quiet
```

- Create a LTTng session

```
lttng create thapi-cuda-session
```

- Create a blocking channel and add context information. Context can also be perf counters.

```
lttng enable-channel --userspace --blocking-timeout=inf blocking-channel
lttng add-context --userspace --channel=blocking-channel -t vpid -t vtid
```

- Enable desired events, can be individual events or regular expressions:

```
lttng enable-event --channel=blocking-channel --userspace lttng_ust_cuda:*
```

How Tracers Work - 2

- Setup interposition libraries:

```
# Point tracer to CUDA driver library
export LTTNG_UST_CUDA_LIBCUDA=/usr/lib64/libcuda.so
# Put our interception version of the CUDA library in the path
export LD_LIBRARY_PATH=$pkglibdir/cuda:$LD_LIBRARY_PATH
# Preload it as well
export LD_PRELOAD=$libdir/libTracerCUDA.so:$LD_PRELOAD
# Enable LTTng client side blocking mode
export LTTNG_UST_ALLOW_BLOCKING=1
```

- Start LTTng

```
lttng start
```

- Launch the application

- Stop LTTng

```
lttng stop
```

Reading a Trace with Babeltrace2 - 1

Processing the trace can be done through Babeltrace 2 bindings. Here is a ruby example:

```ruby
require 'babeltrace2'
# Find babeltrace 2 plugins, the CTF reader, and the stream muxer
ctf_fs = BT2::BTPlugin.find("ctf").get_source_component_class_by_name("fs")
utils_muxer = BT2::BTPlugin.find("utils").get_filter_component_class_by_name("muxer")
# trace locations have the metadata file generated by lttng at runtime
trace_locations = ARGV
# Create a processing graph:
graph = BT2::BTGraph.new
# Add each trace to the graph:
sources = trace_locations.each_with_index.collect do |trace_location, i|
  graph.add_component(ctf_fs, "trace_#{i}", params: {"inputs" => [ trace_location ] })
end
# Mux the traces together
muxer = graph.add_component(utils_muxer, "mux")
sources.map(&:output_ports).flatten.each_with_index do |port, i|
  graph.connect_ports(port, muxer.input_port(i))
end
```

Reading a Trace with Babeltrace2 - 2

```
# Simple pretty printer
sink = graph.add_simple_sink("babeltrace_thapi", lambda do |it, _|
  it.next_messages.select { |m| m.type == :BT_MESSAGE_TYPE_EVENT }.each do |message|
    event = message.event
    str = message.get_default_clock_snapshot.ns_from_origin.to_s
    str << " - #{event.stream.trace.get_environment}"
    str << " - #{event.common_context_field.value}" if event.get_common_context_field
    str << " - #{event.name}: #{event.payload_field}"
    puts str
  end
end)
# Connect the muxer output
graph.connect_ports(muxer.output_port(0), sink.input_port(0))
# Execute the Babeltrace 2 graph
graph.run
```

Similar Python bindings exist, or plugins can be written in C.

Questions?