THAPI/Iprof: Design and Implementation of a Tracer for Heterogeneous API

Thomas Applencourt, Solomon Bekele, Brice Videau and many others

September 25, 2025

Before we start...

- ”The audience is very technical”, Mike
- So I made the slides in Beamer...

Table of Contents

What?

What is THAPI / Iprof

- `mpirun -n 60000 -- iprof ./a.out`
- THAPI (Tracing Heterogeneous API) is... a tracer for heterogeneous API.
  - We support OpeCL, L0, Cuda (Driver and Runtime) HIP, OpenMP, MPI
  - We Dump All Arguments. Traces should contain enough information to reconstruct the programming model state.
  - We do hardware counter sampling (frequency, power, traffic...)
- Iprof take the trace, and post-process it (high-level summary, timeline, ...)
- Scalable (tested on 10k GPUs), Low/Reasonable overhead (about 0.2us overhead ns per tracepoint)

What is not?

- It's not a full-blow performance analysis framework (Vtune, NSigh, HPC Toolkit, Tau)
- It's not a line-level profiler [1]

[1] We give you Kernel Time. And we know have sampling support of HW counter, but we stop here

Why?

Why a Tracer?

We work on runtime. So we Need to understand what is going on to solve bug.

- Why no data-transfer H2D, D2H overlap?!
- Why OpenMP Mapping take 10 min?!
- Why my SYCL queue in-order have so much submission overhead?!

Application:

- Does I'm GPU bound? MPI Bound? Data-transfer bound?
- What is my memory footprint?
- How does my scaling affect my offload
- And give me some timeline to see if I see some "bubble"

Why a new tracer?

- When starting working on GPU-Aurora, their was not tracer for Level Zero[2].
- We wanted just a tracer. Nothing else.
- Want to use ”industry standard” binary trace format so we can develop tools on top of it.
- Easy to maintain

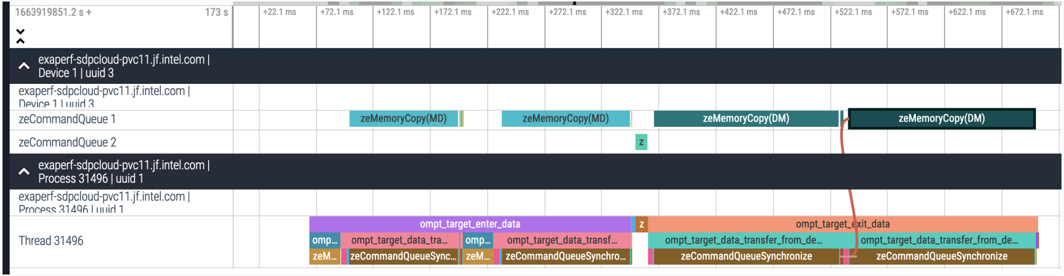
[2] L0 is the cuda-driver of Intel

Pretty Pictures

Profiling a Simple CUDA Applications with iprof

```
> 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 |
```

Perfetto Timeline: OpenMP on top of Level Zero



Simple CUDA Trace

```
[...]
02:58:48.234853435 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuModuleGetFunction_entry: { hfunc: 0x00007ffc1ebf8048, hmod: 0x00000000012e43d0, name: 0x000000000046dcbd, name_val: "_Z3addiPfS_" }
02:58:48.234858185 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuModuleGetFunction_exit: { cuResult: CUDA_SUCCESS, hfunc_val: 0x00000000012e1b60 }
02:58:48.234867264 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemAllocManaged_entry: { dptr: 0x00007ffc1ebf8380, bytesize: 4194304, flags: 1 }
02:58:48.234900648 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemAllocManaged_exit: { cuResult: CUDA_SUCCESS, dptr_val: 0x00001471e4000000 }
02:58:48.234902604 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemAllocManaged_entry: { dptr: 0x00007ffc1ebf8378, bytesize: 4194304, flags: 1 }
02:58:48.234911613 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemAllocManaged_exit: { cuResult: CUDA_SUCCESS, dptr_val: 0x00001471e4400000 }
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: [  ] }
02:58:48.239034018 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda_profiling:event_profiling: { hStart: 0x00000000012e7810, hStop: 0x00000000012e1780 }
02:58:48.239035415 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuLaunchKernel_exit: { cuResult: CUDA_SUCCESS }
02:58:48.239038627 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuCtxSynchronize_entry: {  }
02:58:48.361712445 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuCtxSynchronize_exit: { cuResult: CUDA_SUCCESS }
02:58:48.370911152 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemFree_v2_entry: { dptr: 0x00001471e4000000 }
02:58:48.371248487 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemFree_v2_exit: { cuResult: CUDA_SUCCESS }
02:58:48.371249954 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemFree_v2_entry: { dptr: 0x00001471e4400000 }
02:58:48.371527993 - x3015c0s25b0n0 - vpid: 870, vtid: 870 - lttng_ust_cuda:cuMemFree_v2_exit: { cuResult: CUDA_SUCCESS }
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 }
[...]
```

How?

Finally, the Interesting Part!

- With lot of love
- … and metaprograming.

THAPI (https://github.com/argonne-lcf/THAPI)

Two Big Components:

- The tracing of events
  - Using: Linux Tracing Toolkit Next Generation (LTTng) to generate CTF Trace.
  - Tracepoints are generated from APIs' headers
- The parsing of the trace
  - Using: Babeltrace2 [3]
  - Babaltrace2 plugin infrastructure generated by MetaBabel
  - Pretty Printer, Tally, Timeline/Flamegraph, ...

iprof is an orchestrator around THAPI, lTTng, and Babeltrace2.

LTTng, and Babeltrace2 are developed by EfficiOS (https://www.efficios.com/)

[3] Similar to ffmpeg pipeline approach

LTTng

State of the art tracing infrastructure for kernel and user-space.

- Well maintained and established (used in industry leading data-centers)
- Binary format (CTF: Common Trace Format) open standard
- About 0.2us overhead per Tracepoint (in our case: blocking mode)
    - can be relaxed if use case tolerate event losses
- LTTng relay daemons can be setup to stream over the network in complex topologies, or block for ”on-the-fly” analysis
    - ideal to deploy at scale
- Fine granularity, you can enable/disable individual tracepoint

Typical Tracer Strategy

To Intersect Symbol:

- we either use `LD_PRELOAD="fake_lib.so"` + y `dlopen("real_lib.so", RTLD_LAZY | RTLD_LOCAL)`
- or we use Native Support for Interception (OpenCL Layers, OpenMP [4])

Implementation of Tracing functions:

```
CUresult cuDeviceGetCount(int *count) {
  tracepoint(lttng_ust_cuda, cuDeviceGetCount_entry, count);
  CUresult _retval = CU_DEVICE_GET_COUNT_PTR(count);
  tracepoint(lttng_ust_cuda, cuDeviceGetCount_exit, count, _retval);
  return _retval;
}
```

```
21:03:53.070592532 - x3006c0s25b0n0 - vpid: 36056, vtid: 36056
- lttng_ust_cuda:cuDeviceGetCount_entry: { count: 0x00007ffe93bec390 }
21:03:53.070593929 - x3006c0s25b0n0 - vpid: 36056, vtid: 36056
- lttng_ust_cuda:cuDeviceGetCount_exit: { cuResult: CUDA_SUCCESS, count_val: 6 }
```

[4] But not PMPI, we want to trace people who use PMPI...

The Tracepoint Generation seem Tedious

- We trace all APIs entry points (multiple thousand tracepoint…).
- Tedious, error prone, and hard to maintain by hand
- Automatic generation from headers or API description (OpenCL)
  - C99 parser => YAML intermediary representation
  - YAML + user provided meta information + user provided tracepoints => wrapper functions + Trace Model
  - Trace Model => tracepoints

Example of Code-gen for LTTNG TracePoint

Cuda Header:

```
CUresult cuDeviceGetCount(int* count);
```

Metadata:

```
cuDeviceGetCount:
- [OutScalar, count]
```

```
- name: cuDeviceGetCount
  type:
    kind: custom_type
    name: CUresult
  params:
  - name: count
    type:
      kind: pointer
      type:
        kind: int
        name: int
```

The Final tracepoints:

```
- :name: cuDeviceGetCount_entry
  :payload:
  - :name: count
    :cast_type: int *
    :class: unsigned
    :class_properties:
      :field_value_range: 64
      :preferred_display_base: 16
```

```
- :name: cuDeviceGetCount_exit
  :payload:
  - :name: cuResult
    :cast_type: CUresult
    :class: signed
    :class_properties:
      :field_value_range: 32
    :be_class: CUDA::CUResult
  - :name: count_val
    :cast_type: int
    :class: signed
    :class_properties:
      :field_value_range: 32
```

And for the reading of the trace with Babeltrace2?

- Thanks you for asking!

Babeltrace2

Reference parser implementation of Common Trace Format

- Modular plugin infrastructure
- Compose Babeltrace 2 components into trace processing graphs:
  - Sources, Filters, Sources

```
babeltrace2 --plugin-path=$libdir \
            --component=filter.zeinterval.interval \
            --component=filter.ompinterval.interval \
            --component=sink.xprof.tally
```

THAPI use Pipeline of plugins

- Source are the generated traces
- Filters which aggregate messages
- Sinks which create outputs: Tally, Pretty Print Timeline

Automatic Plugins generation for Babeltrace 2 from the Trace Model

Metababel

- Problem: Writing Babeltrace 2 plugin by hand is tedious, error prone and hard to maintain.
  - Using Python bindings is too slow -> Use C or C++
- Main Idea: Attaching User-Callbacks to Trace Events
- Metababel generates Babeltrace 2 calls to read, write and dispatch events to User-Callbacks
  - Generate State Machine to handle Babeltrace 2 messages queues
- Open Source: `https://github.com/TApplencourt/metababel`

Code Generated for the Cu Exit

```
CUresult cuResult;
int count_val;
const bt_field *payload_field = bt_event_borrow_payload_field_const(bt_evt);
{
  const bt_field *_field = NULL;
  _field = bt_field_structure_borrow_member_field_by_index_const(payload_field, 0);
  cuResult = (CUresult)bt_field_integer_signed_get_value(_field);
}
{
  const bt_field *_field = NULL;
  _field = bt_field_structure_borrow_member_field_by_index_const(payload_field, 1);
  count_val = (int)bt_field_integer_signed_get_value(_field);
}
[...]
```

User code:

```
#include <metababel/metababel.h>
void cuDeviceGetCount_exit_callback(void *btx_handle, CUresult cuResult, int count_val) {
  std::cout << "cuResult: " << cuResult << ", count_val: "  << count_val << std::endl;
}
void btx_register_usr_callbacks(void *btx_handle) {
  btx_register_cuDeviceGetCount_exit(btx_handle, &cuDeviceGetCount_exit_callback);
}
```

Conclusion / Open Questions

THAPI/iprof

- Scalable Tracer with low-overhead
- Based on industry standard technology (LTTng, Babeltrace2)
- Lot of meta-programming (so maintainable but a small team)
- Should we converge on a meta-data description of API format?

One Last note about MPI deployment

How to Launch the LTTng Daemon and Post-Processing?

- Problem 1: We need to spawn one daemon per node.
- We need first to spawn the daemon and then launch user-application
- Problem 2: `iprof mpirun ./a.out` or `mpirun iprof ./a.out`
- I was afraid of doing 1. MPI launcher scare me[5].

So new Problem... Problem 3:

- Application will call `MPI_init`, but we need a barrier before app run. And We cannot call 2 `MPI_init`
- `MPI_Session_init` "aka" initialize a mpi communicator without call mpi init.

---

[5] vni, weird argument parsing, ...

Argonne NATIONAL LABORATORY