Model-Centric Tracing of Low-Level HPC APIs

Brice Videau Thomas Applencourt

Argonne National Laboratory

10th September 2020

Section 1

Context

Intro

Programming languages and models for HPC have never been more diverse:

Languages

- FORTRAN
- C
- C++
- Python

Prospective languages

- Julia
- Lua
- Chapel
- PGAS approaches

Programming models

- MPI
- OpenMP
- CUDA, L0, ROCm, HIP, OpenCL
- SYCL/DPC++
- Kokkos
- Raja

Domain Based Programming Models

- Linear algebra: BLAS/LAPACK
- FFTs: cuFFT, FFTWx, mkl FFT
- Low level AI: cuDNN, clDNN, Intel DNNL
- AI/ML: TensorFlow/Caffe/PyTorch

Problematic

This plethora of alternatives are entwined, especially since heterogeneous computing is the norm.

Possible Dependencies

- SYCL:
  - HIP
  - OpenCL
  - L0
- OpenMP:
  - OpenCL
  - CUDA
  - L0
- OpenCL:
  - L0
  - CUDA
- HIP:
  - CUDA
  - OpenCL
  - ROCm
- Kokkos
  - OpenMP
  - CUDA
  - SYCL
- ...

How to Introspect Those?

- Analyze applications based on those models;
- Understand application performances;
- Understand interactions between applications / compilers / run-times / system / hardware;
- Influence/optimize application at any point:
  - writing,
  - optimization,
  - execution.

Section 2

Low / High level Programming Models

Low / High level Programming Models

First thing we can do is sort the programming models in three categories:

- Low level: CUDA driver, OpenCL, ROCm, L0...
- High level: Kokkos, Raja, SYCL, CUDA, OpenMP (intermediate level?)
- Library Level: BLAS, LAPACK, FFT, Neural Networks, etc...

Library are based on the underlying low or high-level programming models

Objective

What can we do that would have an impact on as many layer of the stack as possible:

- Application writing,
- Application optimization,
- Application modeling/simulation,
- Application monitoring,
- Node resource management,
- Platform management.

Focus on low-level programming models

Section 3

Low-Level Programming Models

Model Centric Debugging / Tracing

Low level programming models usually rely on APIs:

- Traceable (library preloading);
- Injectable;
- Usually task based, with clear dependencies;
- State is reconstructible, on the fly or post-mortem;
- Enables verification,
- and modeling.

Simulation

Accurate tracing and accessible modeling of low level programming models and tasks allows for accurate simulation:

- Scalability studies,
- Performance extrapolation,
- Performance debugging.

High-level Programming Models Introspection

Low level programming models are a window into high-level programming models:

- Debug high level programming models,
- Optimization of high level programming model,
- Could be used to override high level programming model's behavior.

Node Resource Management

Live tracing of low level programming models can be used as input to solve control issues at the node level. Especially correlated with other kind of performance metric tracing.

- Check progress/liveliness
- Check appropriate utilization
- Use as input for balancing power between Threads / Processes / Tasks / etc. . .

Platform Management

Post mortem analysis of traces (and inputs?) of HPC applications can be used to make platform-wide decisions about resources allocations, driving:

- Global power repartition between different applications,
- Higher priority for efficient applications?
- Anticipation of application power/resource usage allowing for better platform tuning (amount of cooling, application interferences avoiding)

Section 4

THAPI: Tracing Heterogeneous APIs

THAPI is a Collection of Tracers

Use low level tracing: Linux Tracing Toolkit Next Generation (LTTng):

- Low Overhead
- Binary format
- Well maintained and established (used in industry leading data-centers)

Supported APIs

- OpenCL (Full)
- Level Zero (Full API trace, profiling WIP)
- Cuda (WIP)

Open source at: https://xgitlab.cels.anl.gov/heteroflow/tracer

Example

- Wrapping the API entry points to be able to reconstruct the context.
- SYCL example:

```
#include <sycl.hpp>

int main() {
  sycl::default_selector selector;
  sycl::queue myQueue(selector);
  myQueue.submit([&](sycl::handler &cgh) {
    sycl::stream sout(1024, 256, cgh);
    cgh.single_task<class hello_world>([=]() {
      sout << "Hello, World!" << sycl::endl;
    });
  });
  return 0;
}
```

OpenCL Tracing of SYCL

- 69 OpenCL calls:

```
cl:clCreateProgramWithIL_start: context: 0x2522780, il: 0x461ec0, length: 41996,
                                errcode_ret: 0x7ffd9763f8cc
cl:clCreateProgramWithIL_stop: program: 0x24ed980, errcode_ret_val: CL_SUCCESS
cl:clBuildProgram_start: program: 0x24ed980, num_devices: 0, device_list: 0x0,
                         options: 0x7ffd9763fdf0, pfn_notify: 0x0, user_data: 0x0,
                         device_list_vals: [], options_val: ""
cl_build:infos: program: 0x24ed980, device: 0x24394c0, build_status: CL_BUILD_SUCCESS,
                build_options: "", build_log: ""
cl_build:infos_1_2: program: 0x24ed980, device: 0x24394c0,
                    binary_type: CL_PROGRAM_BINARY_TYPE_EXECUTABLE
cl_build:infos_2_0: program: 0x24ed980, device: 0x24394c0,
                    build_global_variable_total_size: 0
cl:clBuildProgram_stop: errcode_ret_val: CL_SUCCESS
cl:clCreateKernel_start: program: 0x24ed980, kernel_name: 0x24ee890,
                         errcode_ret: 0x7ffd9763fe64,
                         kernel_name_val: "_ZTSZZ4mainENK3$_0clERN2cl4sycl7handlerEE11hello_world"
cl_arguments:kernel_info: kernel: 0x24ed170,
                          function_name: "_ZTSZZ4mainENK3$_0clERN2cl4sycl7handlerEE11hello_world",
                          num_args: 9, context: 0x2522780, program: 0x24ed980, attibutes: ""
cl:clCreateKernel_stop: kernel: 0x24ed170, errcode_ret_val: CL_SUCCESS
```

Example Tool: clprof

```
API calls | 1 Hostnames | 1 Processes | 1 Threads
                     Name |      Time | Time(%) | Calls |   Average |       Min |       Max |
            clBuildProgram | 219.26ms |  99.08% |     1 | 219.26ms | 219.26ms | 219.26ms |
   clEnqueueNDRangeKernel |   1.44ms |   0.65% |     1 |   1.44ms |   1.44ms |   1.44ms |
      clEnqueueReadBuffer | 370.14us |   0.17% |     1 | 370.14us | 370.14us | 370.14us |
                  clFinish | 149.72us |   0.07% |     1 | 149.72us | 149.72us | 149.72us |
            clGetDeviceIDs |  22.90us |   0.01% |     2 |  11.45us |   1.44us |  21.46us |
            clCreateBuffer |   7.68us |   0.00% |     1 |   7.68us |   7.68us |   7.68us |
           clReleaseKernel |   6.86us |   0.00% |     1 |   6.86us |   6.86us |   6.86us |
            clCreateKernel |   6.60us |   0.00% |     1 |   6.60us |   6.60us |   6.60us |
           clCreateContext |   6.27us |   0.00% |     1 |   6.27us |   6.27us |   6.27us |
          clGetPlatformIDs |   4.15us |   0.00% |     2 |   2.07us | 251.00ns |   3.90us |
            clSetKernelArg |   4.01us |   0.00% |     2 |   2.00us | 544.00ns |   3.46us |
         clGetPlatformInfo |   3.21us |   0.00% |     1 |   3.21us |   3.21us |   3.21us |
clCreateProgramWithSource |   2.29us |   0.00% |     1 |   2.29us |   2.29us |   2.29us |
     clCreateCommandQueue |   1.71us |   0.00% |     1 |   1.71us |   1.71us |   1.71us |
    clReleaseCommandQueue |   1.32us |   0.00% |     1 |   1.32us |   1.32us |   1.32us |
         clReleaseContext |   1.01us |   0.00% |     1 |   1.01us |   1.01us |   1.01us |
         clReleaseProgram | 737.00ns |   0.00% |     1 | 737.00ns | 737.00ns | 737.00ns |
          clGetDeviceInfo | 712.00ns |   0.00% |     1 | 712.00ns | 712.00ns | 712.00ns |
                    Total | 221.29ms | 100.00% |    21 |

Device profiling | 1 Hostnames | 1 Processes | 1 Threads | 1 Devices
                Name |    Time | Time(%) | Calls | Average |     Min |     Max |
         hello_world | 18.54us |  59.04% |     1 | 18.54us | 18.54us | 18.54us |
clEnqueueReadBuffer | 12.86us |  40.96% |     1 | 12.86us | 12.86us | 12.86us |
               Total | 31.41us | 100.00% |     2 |
Explicit memory trafic | 1 Hostnames | 1 Processes | 1 Threads
                Name |   Byte | Byte(%) | Calls | Average |    Min |    Max |
clEnqueueReadBuffer | 40.00B | 100.00% |     1 |  40.00B | 40.00B | 40.00B |
               Total | 40.00B | 100.00% |     1 |
```

Section 5

Conclusion and Future Work

Conclusion

Working, robust and efficient tracers:

- Used for simulation purposes,
- Used for lightweight profiling,
- Used for kernel extractions,
- Used for debugging.

Future work

Deployment strategies and use-cases on HPC infrastructures:

- Lightweight monitoring for continuous usage,
- Enabling full tracing of distributed applications,
- Develop specialized trace analysis tools.

More tracers:

- Finish CUDA tracer,
- ROCm
- OpenMP ?