Using HPCToolkit to Measure and Analyze the Performance of CPU and GPU-accelerated Parallel Applications

John Mellor-Crummey
Rice University

ATPESC 2021
August 11, 2021

Download application examples to run, measure, and analyze on ascent
git clone https://github.com/HPCToolkit/hpctoolkit-tutorial-examples

See theta:/grand/ATPESC2021/EXAMPLES/track-6-tools/hpctoolkit (contains an example to run and analysis as well as measured data to explore)
Project Overview

• Goals
  – Develop effective tools to measure, attribute, analyze, and understand performance of applications, libraries, tools, and system software on extreme-scale parallel systems
  – Develop new strategies and mechanisms to measure and analyze performance and resource utilization of GPU-accelerated compute nodes
  – Lead evolution of hardware and software ecosystems to enable better tools

• Team
  – Lead Institution: Rice University (HPCToolkit performance tools)
    • PI: Prof. John Mellor-Crummey
    • Research staff: Dr. Laksono Adhianto, Dr. Mark Krentel, Dr. Xiaozhu Meng, Dr. Scott Warren
    • Grad students: Keren Zhou, Jonathon Anderson, Yumeng Liu, Aaron Cherian, Dejan Grubisic
  – Subcontractor: University of Wisconsin – Madison (Dyninst binary analysis toolkit)
    • Lead: Prof. Barton Miller

• Principal Funding:
  • DOE Exascale Computing Project, ANL, NNSA Tri-labs, AMD, Intel
Performance Analysis Challenges on Modern Supercomputers

• **Myriad performance concerns**
  – Computation performance on CPU and GPU
  – Data movement costs within and between memory spaces
  – Internode communication
  – I/O

• **Many ways to hurt performance**
  – insufficient parallelism, load imbalance, serialization, replicated work, parallel overhead …

• **Hardware and execution model complexity**
  – Multiple compute engines with vastly different characteristics, capabilities, and concerns
  – Multiple memory spaces with different performance characteristics
    • CPU and GPU have different complex memory hierarchies
  – Often, a large gap between programming model and implementation
    • e.g., OpenMP, template-based programming models
  – Asynchronous execution
Outline

• Overview of Rice’s HPCToolkit
• Understanding the performance of parallel programs using HPCToolkit’s GUIs
  – code centric views
  – time centric views
• Monitoring GPU-accelerated applications
• Work in progress
Rice University’s HPCToolkit Performance Tools

- Employs binary-level measurement and analysis
  - Observes executions of fully optimized, dynamically-linked applications
  - Supports multi-lingual codes with external binary-only libraries

- Collects sampling-based measurements of CPU
  - Controllable overhead
  - Minimize systematic error and avoid blind spots
  - Enable data collection for large-scale parallelism

- Measures GPU performance using APIs provided by vendors
  - Callbacks to monitor launch of GPU operations
  - Activity API to monitor and present information about asynchronous operations on GPU devices
  - PC sampling for fine-grain measurement

- Associates metrics with both static and dynamic context
  - Loop nests, procedures, inlined code, calling context on both CPU and GPU

- Specify and compute derived CPU and GPU performance metrics of your choosing
  - Diagnosis often requires more than one species of metric

- Supports top-down performance analysis
  - Identify costs of interest and drill down to causes: up and down call chains, over time
HPCToolkit’s Workflow for GPU-accelerated Applications

Source Files → Optimized Binary

Compile & Link

GPU Binary

hpcrun
Profile execution on CPUs and GPUs

Profile Files

Trace Files

hpcstruct
Analyze CPU/GPU program structure

Program Structure

hpcviewer
Present trace view and profile view

Database

hpcprof/hpcprof-mpi
Interpret profile Correlate w/ source
Step 1:
- Add `-g` to the host compiler
- Add `-g` and `-lineinfo` to the target compiler
Step 2:

- `hpcrun` collects call path profiles of events of interest
hpcrun - Measure CPU and GPU execution

• GPU profiling
  – hpcrun -e gpu=xxx <app> .... // xxx ∈ {nvidia,amd,opencl,level0}

• GPU tracing (-t)
  – hpcrun -e gpu=yyy -t <app> // yyy ∈ {nvidia,amd,opencl}

• GPU PC sampling (NVIDIA only)
  – hpcrun -e gpu=nvidia,pc -t <app>

• CPU and GPU profiling and tracing
  – hpcrun -e REALTIME -e gpu=yyy -t <app>

• Use hpcrun with job launchers
  – jsrun -n 32 -g 1 -a 1 hpcrun -e gpu=xxx <app>
  – srun -n 1 -G 1 hpcrun -e gpu=xxx <app>
  – aprun -n 16 -N 8 -d 8 hpcrun -e gpu=xxx <app>
HPCToolkit and GPU Software Stacks

HPCToolkit GPU-independent Measurement Substrate

Sanitizer

GT-Pin

NVIDIA GPU

Intel GPU

AMD GPU
Measurement for GPU-accelerated Supercomputers

- **Measurement interfaces**
  - **Hardware**
    - CPU hardware performance monitoring unit
    - GPU hardware counters and PC sampling
  - **Software**
    - Glibc LD_AUDIT for tracking dynamic loading of shared libraries
    - Linux perf_events for kernel measurement
    - GPU monitoring and instrumentation libraries from vendors

- **Multiple measurement modalities and interfaces**
  - Sampling on the CPU
  - Callbacks when GPU operations are launched and (sometimes) completed
  - GPU event stream, including PC sampling measurements
Step 3:
• *hpcstruct* recovers program structure about lines, loops, and inlined functions.
hpcstruct - Recover Program Structure

• Analyze CPU binaries
  – hpcstruct [-j <threads>] <app>
  – yields a file <app>.hpcstruct

• Analyze all GPU binaries in <measurements-dir>
  – hpcstruct [-j <threads>] [--gpucfg yes] <measurements-dir>
    • “gpucfg yes” means recover GPU loop nests, calling context information
    – augments the measurement directory with a hpcstruct file for each GPU binary
Step 4:
• *hpcprof/hpcprof-mpi* combines profiles from multiple threads and correlate metrics to static & dynamic program structure.
hpcprof/hpcprof-mpi - Correlate Measurements with Code

• Use a single process to combine performance data
  – hpcprof -S <app>.hpcstruct <measurements-dir>

• Use multiple processes to combine performance data
  – jsrun -n <np> hpcprof-mpi -S <app>.hpcstruct <measurements-dir>
  – srun -n <np> hpcprof-mpi -S <app>.hpcstruct <measurements-dir>
Step 4:
- *hpcviewer* - interactively explore profile and traces for GPU-accelerated applications
HPCToolkit’s Code-Centric Profiles of GPU-accelerated Code
Coarse- and Fine-grain Measurement on NVIDIA GPUs: ECP Quicksilver

**Compute Node**
- 2xPower9 + 4xNVIDIA GPUs
- Optimized (-O2) compilation with nvcc
- Detailed measurement and attribution using PC sampling
- Reconstruct approximate GPU calling context tree from flat PC samples
- Understand GPU loops and inlined code
- Attribute information to heterogeneous calling context

**Key Metrics**
- instructions executed
- instruction stalls and reasons
- GPU utilization

---

Coarse- and Fine-grain Measurement on NVIDIA GPUs: ECP Quicksilver

- HPCToolkit reconstructs approximate GPU calling context tree from flat PC samples
- Understand GPU loops and inlined code
- Attribute information to heterogeneous calling context

```
<gpu kernel>
  ▼ CycleTrackingKernel(MonteCarlo*, int, ParticleVault*, ParticleVault*)
    ▼ 132: CycleTrackingGuts(MonteCarlo*, int, ParticleVault*, ParticleVault*)
        ▼ loop at CycleTracking.cc: 118
    ▼ 63: CollisionEvent(MonteCarlo*, MC_Particle&, unsigned int)
        ▼ loop at CollisionEvent.cc: 67
        ▼ loop at CollisionEvent.cc: 71
    ▼ 73: macroscopicCrossSection(MonteCarlo*, int, int, int, int, int)
        ▼ [I] inlined from MacroscopicCrossSection.cc: 45
    ▼ 41: NuclearData::getReactionCrossSection(unsigned int, unsigned)
        ▼ [I] inlined from NuclearData.cc: 193

QS_Vector.hh: 94
```

Coarse- and Fine-grain Measurement on Intel GPUs: ECP PeleC

Hardware: JLSE Iris
- Intel E3-1585 v5
- Intel Iris Pro Graphics P580

Software
- DPC++
- oneAPI beta 10
- OpenCL runtime
- GT-Pin

Permission for open publication granted by Kent Moffatt, Intel
Coarse-grain Measurement on AMD GPUs: ECP PeleC

**Hardware:** Cray Tulip
- AMD Epyc 7601
- 4 x AMD MI60

**Software:**
- ROCM 3.8

Permission for open publication granted by Noah Reddell, HPE

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Aggregate Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU time</td>
</tr>
<tr>
<td>REALTIME (sec) Sum</td>
<td>1.47e-02 100%</td>
</tr>
<tr>
<td>CPU time (sec) Sum</td>
<td>4.01e-02 100%</td>
</tr>
<tr>
<td>CPU time (sec) Mean</td>
<td>1.00e-02 0.2%</td>
</tr>
</tbody>
</table>

GPU kernel time:
- 1.10e-02 100%
- 1.21e-02 100%
- 2.20e-02 100%

GPU copy time:
- 1.30e-03 100%
- 1.21e-02 100%
- 2.20e-02 100%

GPU sync time:
- 1.00e-02 100%
- 1.10e-02 100%
- 2.20e-02 100%

GPU copy counts:
- 1.00e-02 100%
- 1.21e-02 100%
- 2.20e-02 100%

---

**Note:**
- This table represents the execution times for different metrics, including CPU time, GPU kernel time, GPU copy time, GPU sync time, and GPU copy counts, measured on AMD GPUs using ECP PeleC software on Cray Tulip hardware.
- The data is collected using a GPU kernel, CPU time, GPU copy, and GPU sync evaluation framework.
Support for OpenMP TARGET: ECP miniqmc

- HPCToolkit implementation of OMPT OpenMP API
  - host monitoring
    - employs OMPT API for call stack introspection
  - GPU monitoring
    - leverages callbacks for device initialization, kernel launch, data operations
    - reconstruction of user-level calling contexts
- Leverages implementation of OMPT in LLVM OpenMP and libomptarget

Reconstruct full calling contexts that include:
- Outlined procedures for OpenMP parallel regions
- Offloaded OpenMP TARGET computation and synchronization

System: Power9 + NVIDIA V100
Understanding Temporal Behavior

- Profiling compresses out the temporal dimension
  - Temporal patterns, e.g. serial sections and dynamic load imbalance are invisible in profiles
- What can we do? Trace call path samples
  - N times per second, take a call path sample of each thread
  - Organize the samples for each thread along a time line
  - View how the execution evolves left to right
  - What do we view? assign each procedure a color; view a depth slice of an execution
Trace of Multi-rank Multi-GPU Executions: ECP Nyx on Summit

22 nodes (Power 9 + 6 x NVIDIA V100), 128 MPI ranks, 640 GPU streams
Trace of Multi-rank Multi-GPU Executions: ECP Nyx on Summit
## GPU Monitoring Capabilities of HPCToolkit

<table>
<thead>
<tr>
<th>Measurement Capability</th>
<th>NVIDIA</th>
<th>AMD</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel launches, explicit memory copies, synchronization</td>
<td>callbacks + activity API</td>
<td>callbacks + Activity API</td>
<td>callbacks</td>
</tr>
<tr>
<td>instruction-level measurement and analysis</td>
<td>PC sampling of GPU code</td>
<td>Future*: PC sampling (as seen on Github) of GPU code</td>
<td>GTPin; Future*: instruction-level measurement of GPU code</td>
</tr>
<tr>
<td>kernel characteristics</td>
<td>Activity API</td>
<td>(available statically)</td>
<td>(unknown)</td>
</tr>
</tbody>
</table>

- **Significant support in master branch**
- **Prototype support in master branch**
- **Prototype support in master branch**

*Approved for public release*
Work in Progress in HPCToolkit

- **GPU Enhancements**
  - Intel GPUs
    - Measurement support for Intel GPUs using OpenCL and Level 0
    - Fine-grain measurement using GTPin
    - Fine-grain attribution using binary analysis
  - AMD GPUs
    - Binary analysis and instrumentation for fine-grain measurement and attribution
  - NVIDIA GPUs
    - Interpret inlining information available (CUDA 11.2)
    - Reduce fine-grain measurement overhead with low-overhead PC sampling (CUDA 11.3)

- **Scalability**
  - Accelerate analysis of measurement data with hpcprof-mpi using multithreading
  - Use sparse formats to reduce size of measurement data and analysis results

- **User interface**
  - Overhauling metric view to enhance performance and scalability
  - Associate trace lines with metadata (node, GPU, MPI rank, GPU stream …)
  - Improve presentation of the many GPU metrics

- **Reliability and Completeness**
### Detailed Performance Analysis Requires Support at Many Levels

<table>
<thead>
<tr>
<th>Hardware and Software Stack Components</th>
<th>Partners</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware must include support for fine-grain measurement and attribution</strong></td>
<td>GPU vendors</td>
</tr>
<tr>
<td>• performance counters are not enough; NVIDIA's PC sampling approximates our needs</td>
<td></td>
</tr>
<tr>
<td><strong>System software must provide appropriate interfaces for introspection and analysis</strong></td>
<td>Red Hat</td>
</tr>
<tr>
<td>• e.g. Linux perf_events supports sample-based performance monitoring even in the kernel</td>
<td></td>
</tr>
<tr>
<td>• e.g. dynamic loader (ld.so) provides LD_AUDIT interface for monitoring and control of dynamic library operations</td>
<td></td>
</tr>
<tr>
<td>• elfutils must support NVIDIA's extended line maps in CUDA 11.2+ GPU binaries</td>
<td>GPU vendors</td>
</tr>
<tr>
<td>• GPU vendor software stacks (kernel driver, runtime, tools API)</td>
<td>GPU vendors</td>
</tr>
<tr>
<td><strong>Compiler must compute high-quality DWARF information</strong></td>
<td>Vendors and LLVM community</td>
</tr>
<tr>
<td>• associate each machine instruction with full call chains involving inlined templates and functions</td>
<td></td>
</tr>
<tr>
<td><strong>Runtime must maintain information needed to map computations to a source-level view</strong></td>
<td>OpenMP Language Committee and LLVM Community</td>
</tr>
<tr>
<td>• OpenMP's OMPT helps bridge the vast gap between the implementation and user-level view</td>
<td></td>
</tr>
<tr>
<td><strong>Performance tools must gather measurements using multiple modalities and map them to source</strong></td>
<td>Wisconsin's Dyninst Project</td>
</tr>
<tr>
<td>• precise attribution when possible</td>
<td></td>
</tr>
<tr>
<td>• reconstruct approximate attribution when precise attribution is unavailable</td>
<td></td>
</tr>
<tr>
<td>• GPU calling context</td>
<td></td>
</tr>
<tr>
<td>• loops in CPU and GPU code</td>
<td></td>
</tr>
<tr>
<td>• attribute inefficiencies from where they are observed back to their causes</td>
<td></td>
</tr>
</tbody>
</table>
Bonus Content
Download Hands-on Tutorial Examples on Ascent

- git clone https://github.com/hpctoolkit/hpctoolkit-tutorial-examples
- Configured for data collection on ascent
  - openmp example
    - miniqmc
      - CPU OpenMP: GCC, XL
  - gpu examples
    - quicksilver (dynamic monte-carol particle transport)
    - laghos (ultrashock code)
    - lammps (molecular dynamics)
    - pelec (compressible AMR combustion code)
Working with Hands-on Tutorial Examples on Ascent

• Locate the examples configured for measurement on ascent
  • from the directory hpctoolkit-tutorial-examples, you can run “find . -name ascent.sh” to see all of the examples prepared for ascent
  • change into one of the example directories, e.g. cd examples/gpu/quicksilver
  • source setup-env/ascent.sh
  • running make in a directory will show you the commands

• typical commands
  • make build          # build the example
  • make run            # collect profiles and traces
  • make run-pc         # collect detailed profiles using PC sampling
  • make view           # view profile and trace results
  • make view-pc        # view PC sampling results
Working with Hands-on Tutorial Examples on Theta-GPU

- See theta:/grand/ATPESC2021/EXAMPLES/track-6-tools/hpctoolkit
- README with directions
- code-examples
  - quicksilver.tgz - a dynamic monte carlo neutron transport miniapp
- databases
  - PeleC - 98 MPI ranks @ 5 GPU streams each (AMR turbulent combustion code)
  - hpcprof2-mpi - 8 MPI ranks @ 128 threads each (parallel analysis of 64K profiles)
  - Quicksilver - Single-threaded @ 1 GPU stream (dynamic Monte-Carlo neutron transport)
Measuring and Analyzing Quicksilver on Theta-GPU

• Create a directory where you want to work
• Unpack the tar file with the example
  /grand/ATPESC2021/EXAMPLES/track-6-tools/hpctoolkit/code-examples/quicksilver.tgz
• Follow the directions in
  /grand/ATPESC2021/EXAMPLES/track-6-tools/hpctoolkit/code-examples/README
• Pro tip: use two windows
  • one on a theta login node
  • one on a GPU compute node
• Why:
  • You can only compile and run the example on a GPU compute node
  • You can only run the GUI on a Theta login node
    • there is no X forwarding from GPU compute nodes
HPCToolkit’s Graphical User Interfaces

- Overview
- Tips for using them effectively
hpctraceviewer Panes and their Purposes

- **Trace View pane**
  - Displays a sequence of samples for each trace line rendered
  - Title bar shows time interval rendered, rank interval rendered, cross hair location
- **Call Path pane**
  - Show the call path of the selected thread at the cross hair
- **Depth View pane**
  - Show the call stack over time for the thread marked by the cross hair
  - Unusual changes or clustering of deep call stacks can indicate behaviors of potential interest
- **Summary View pane**
  - At each point in time, a histogram of colors above in a vertical column of the Trace View
Rendering Traces with hpctraceviewer

- hpctraceviewer renders traces by sampling the [rank x time] rectangle in the viewport
  - Don’t try to summarize activity in a time interval represented by a pixel
  - Just pick the last activity before the sample point in time

- Cost of rendering a large execution is $[H \times T \lg N]$ for traces of length $N$
  - The number of trace lines that can be rendered is limited by the number of vertical pixels $H$
  - Binary search along rendered trace lines to extract values for pixels

- It can be used to analyze large data: thousands of ranks and threads
  - Data is kept on disk, memory mapped, and read only as needed
Understanding How hpctraceviewer Paints Traces

- **CPU trace lines**
  - Given: (procedure f, t) (procedure g, t') (procedure h, t'')
    - Default painting algorithm
      - paint color “f” in [t, t'); paint color “g” in [t', t'')
    - Midpoint painting algorithm
      - paint color “f” in [t, (t+t')/2); paint color “g” in [(t+t')/2, (t'+t'')/2]

- **GPU trace lines**
  - Given GPU operations “f” in interval [t, t') and and “g” in interval [t'', t''')
    - paint color “f” in [t, t'); paint color white in [t', t''); paint color “g” in [t'', t''')
Analysis Strategies with Time-centric hpctraceviewer

- Use top-down analysis to understand the broad characteristics of the parallel execution
- Click on a point of interest in the Trace View to see the call path there
- Zoom in on individual phases of the execution or more generally subsets of [rank, time]
  - The mini-map tracks what subset of the execution you are viewing
- Home, undo, redo buttons allow you to move back and forth in a sequence of zooms
- Drill down the call path to see what is going on at the call path leaves
  - Hold your mouse over the call path depth selector. A tool tip will tell you the maximum depth
  - Type the maximum call stack depth number into the depth selector
- Use the summary view to see a histogram about what fraction of threads or ranks is doing at each time
- The summary view can facilitate analysis of how behavior changes over time
- The statistics view can show you the fraction of [rank x time] spent in each procedure at the selected depth level
Understanding the Navigation Pane in Code-centric hpcviewer

- **<program root>**: the top of the call chain for the executable
- **<thread root>**: the top of the call chain for any pthreads
- **<partial call paths>**
  - The presence of partial call paths indicates that hpcrun was unable to fully unwind the call stack
  - Even if a large fraction of call paths are “partial” unwinds, bottom-up and flat views can be very informative
- **Sometimes functions appear in the navigation pane and appear to be a root**
  - This means that hpcrun believed that the unwind was complete and successful
  - Ideally, this would have been placed under **<partial call paths>**
Understanding the Navigation Pane in Code-centric hpcviewer

- Treat inlined functions as if regular functions
- Calling an inlined function

[I] is a tag used to indicate that the called function is inlined

callsite is a hyperlink to the file and source line where the inlined function is called

callee is a hyperlink to the definition of the inlined function

- If no source file is available, the caller line number and the callee will be in black
Analysis Strategies with Code-centric hpcviewer

• Use top-down analysis to understand the broad characteristics of the execution
  • Are there specific unique subtrees in the computation that use or waste a lot of resources?
  • Select a costly node and drill down the “hottest path” rooted there with the flame button
  • One can select a node other than the root and use the flame button to look in its subtree
  • Hold your mouse over a long name in the navigation pane to see the full name in a tool tip

• Use bottom-up analysis to identify costly procedures and their callers
  • Pick a metric of interest, e.g. cycles
  • Sort by cycles in descending order
  • Pick the top routine and use the flame button to look up the call stack to its callers
  • Repeat for a few routines of particular interest, e.g. network wait, lock wait, memory alloc, …

• Use the flat view to explore the full costs associated with code at various granularities
  • Sort by a cost of interest; use the flame button to explore an interesting load module
  • Use the “flatten” button to melt away load modules, files, and functions to identify the most costly loop
Preparing a GPU-accelerated Program for HPCToolkit

- **HPCToolkit doesn’t need any modifications to your Makefiles**
  - it can measure fully-optimized code without special preparation
- **To get the most from your measurement and analysis**
  - Compile your program with line numbers
    - **CPU (all compilers)**
      - add "-g" to your compiler optimization flags
    - **NVIDIA GPUs**
      - compiling with nvcc
        - add "-lineinfo" to your optimization flags for GPU line numbers
        - adding -G provides full information about inlining and GPU code structure but disables optimization
      - compiling with xlc
        - line information is unavailable for optimized code
    - **AMD GPUs, no special preparation needed**
      - current AMD GPUs and ROCM software stack lack capabilities for fine-grain measurement and attribution
    - **Intel GPUs (prototypes not integrated into HPCToolkit master)**
      - monitors kernel launches, memory copies, synchronization
      - partial support for fine-grain monitoring with GTPin instrumentation; no source-level attribution yet
Using HPCToolkit to Measure an Execution

• Sequential program
  • `hpcrun [measurement options] program [program args]`

• Parallel program
  • `mpirun -n <nodes> [mpi options] hpcrun [measurement options] \ program [program args]`

• Similar launches with job managers
  • LSF: jsrun
  • SLURM: srun
  • Cray: aprun
CPU Time-based Sample Sources - Linux thread-centric timers

- **CPUTIME (DEFAULT if no sample source is specified)**
  - CPU time used by the thread in microseconds
  - Does not include time blocked in the kernel
    - disadvantage: completely overlooks time a thread is blocked
    - advantage: a blocked thread is never unblocked by sampling

- **REALTIME**
  - Real time used by the thread in microseconds
  - Includes time blocked in the kernel
    - advantage: shows where a thread spends its time, even when blocked
    - disadvantages
      - activates a blocked thread to take a sample
      - a blocked thread appears active even when blocked

Note: Only use one Linux timer to measure an execution
CPU Sample Sources - Linux perf_event monitoring subsystem

- Kernel subsystem for performance monitoring
- Access and manipulate
  - Hardware counters: cycles, instructions, ...
  - Software counters: context switches, page faults, ...

- Available in Linux kernels 2.6.31+
- Characteristics
  - Monitors activity in user space and in the kernel
    - Can see costs in GPU drivers
Case Study: Measurement and Analysis of GPU-accelerated Laghos

Laghos (LAGrangian High-Order Solver) is a LLNL ASC co-design mini-app that was developed as part of the CEED software suite, a collection of software benchmarks, miniapps, libraries and APIs for efficient exascale discretization based on high-order finite element and spectral element methods.

Figure credit: https://computing.llnl.gov/projects/co-design/laghos
Applying the GPU Operation Measurement Workflow to Laghos

# measure an execution of laghos
```
time mpirun -np 4 hpcrun -o $OUT -e cycles -e gpu=nvidia -t \$
LAGHOS_DIR)/laghos -p 0 -m $LAGHOS_DIR)/../data/square01_quad.mesh \-rs 3 -tf 0.75 -pa
```

# compute program structure information for the laghos binary
```
hpcstruct -j 16 laghos
```

# compute program structure information for the laghos cubins
```
hpcstruct -j 16 $OUT
```

# combine the measurements with the program structure information
```
mpirun -n 4 hpcprof-mpi -S laghos.hpcstruct $OUT
```
Computing Program Structure Information for NVIDIA cubins

• When a GPU-accelerated application runs, HPCToolkit collects unique GPU binaries
  • Currently, NVIDIA does not provide an API that provides a URI for cubins it launches
  • CUPTI presents cubins to tools as an interval in the heap (starting address, length)
  • HPCToolkit computes an MD5 hash for each cubin and saves one copy
    • stores save cubins in hpcrun’s measurement directory: <measurement directory>/cubins

• Analyze the cubins collected during an execution
  • hpcstruct -j 16 <measurement directory>
    • lightweight analysis based only on cubin symbols and line map
  • hpcstruct -j 16 -gpucfg yes <measurement directory>
    • heavyweight analysis based only on cubin symbols, line map, control flow graph
      • uses nvdisasm to compute control flow graph
      • fine-grain analysis only needed to interpret PC sampling experiments
  • hpcstruct analyzes cubins in parallel using thread count specified with -j
Initial hpctraceviewer view of Laghos (long) Execution

MPI Ranks

GPU Streams
Hiding the Empty MPI Helper Threads

Please type a pattern in the format minimum:maximum:stride. Any omitted or invalid sections will match as many processes or threads as possible.

For instance, 3:7:2 in the process box with the thread box empty will match all threads of processes 3, 5, and 7.

1 in the thread box with the process box empty will match thread 1 of all processes.

1::2 in the process box and 2:4:2 in the thread box will match 1.2, 1.4, 3.2, 3.4, 5.2 ...

Process

Thread 1:5:1
After Hiding the Empty MPI Helper Threads
A Detail of Only the MPI Threads
Only the MPI Threads - Analysis using the Statistics Panel

![Image of a graph and table showing MPI thread analysis results.]

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>mca_btl_vader_component_progress</td>
<td>22.21%</td>
</tr>
<tr>
<td>opal_progress</td>
<td>7.55%</td>
</tr>
<tr>
<td>cudbgMain</td>
<td>6.11%</td>
</tr>
<tr>
<td>cudbgApIDetach</td>
<td>5.55%</td>
</tr>
<tr>
<td>opal_timer_linux_get_cycles_sys_timer</td>
<td>5.38%</td>
</tr>
<tr>
<td>cuVDPAUctxCreate</td>
<td>5.11%</td>
</tr>
<tr>
<td>_phthread_mutex_lock</td>
<td>4.55%</td>
</tr>
<tr>
<td>cuMemGetAttribute_v2</td>
<td>4.49%</td>
</tr>
<tr>
<td>__ll_lock_wait</td>
<td>3.19%</td>
</tr>
<tr>
<td>cupbActivityDisable</td>
<td>2.68%</td>
</tr>
<tr>
<td>sigprocmask</td>
<td>2.57%</td>
</tr>
<tr>
<td>cupiOpenACCInitialize</td>
<td>2.52%</td>
</tr>
<tr>
<td>ioctl</td>
<td>2.33%</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>2.27%</td>
</tr>
<tr>
<td>__kernel_clock_gettime</td>
<td>1.81%</td>
</tr>
<tr>
<td>cupiEventGroupDisable</td>
<td>1.70%</td>
</tr>
<tr>
<td>malloc</td>
<td>1.16%</td>
</tr>
<tr>
<td>nvidia_ioctl [nvidia] [vmlinux]</td>
<td>1.06%</td>
</tr>
<tr>
<td>pfg_relock_read_lock</td>
<td>0.81%</td>
</tr>
<tr>
<td>__tls_get_addr</td>
<td>0.76%</td>
</tr>
<tr>
<td>cos</td>
<td>0.68%</td>
</tr>
<tr>
<td>__memcpy_power?</td>
<td>0.68%</td>
</tr>
<tr>
<td>exec_virt_0x4c00_system_call [vmlinux]</td>
<td>0.57%</td>
</tr>
<tr>
<td>cfree</td>
<td>0.51%</td>
</tr>
<tr>
<td>__copy_tofrom_user_power? [vmlinux]</td>
<td>0.49%</td>
</tr>
<tr>
<td>&lt;unknown procedure&gt; 0x609000 [libmpi]</td>
<td>0.49%</td>
</tr>
<tr>
<td>netack_nsh</td>
<td>0.46%</td>
</tr>
</tbody>
</table>
Only the GPU Threads - Inspecting the Callpath for a Kernel
Only the GPU Threads - Analysis Using the Statistics Panel
Some Cautions When Analyzing GPU Traces

• There are overheads introduced by NVIDIA’s monitoring API that we can’t avoid
• When analyzing traces from your program and compare GPU activity to [no activity]
  – Time your program without any tools
  – Time your program when tracing with HPCToolkit or nvprof
  – Re-weight <no activity> by the ratio of unmonitored time to monitored time
• While this is a concern for traces, this should be less a concern for profiles
  – On the CPU, HPCToolkit compensates for monitoring overhead in profiles by not measuring it
Using hpcviewer to See the Source-centric View

<table>
<thead>
<tr>
<th>Scope</th>
<th>cycles:Sum</th>
<th>cycles:Sum (E)</th>
<th>GKER (s):Sum</th>
<th>GKER (s):Sum</th>
<th>GXCOPY (s):St</th>
<th>GXCOPY (s):St</th>
<th>GXCOPY:H2D</th>
<th>GXCOPY:H2D</th>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;program root&gt;</td>
<td>1.82e+14  100.0</td>
<td>1.38e+01 100 %</td>
<td>2.31e+00 100 %</td>
<td>2.67e+07 100 %</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
</tr>
<tr>
<td>516: main</td>
<td>1.82e+14  100.0</td>
<td>1.38e+01 100 %</td>
<td>2.31e+00 100 %</td>
<td>2.67e+07 100 %</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
</tr>
<tr>
<td>loop at laghos.cpp: 427</td>
<td>1.80e+14  99.9 %</td>
<td>1.38e+01 99.9 %</td>
<td>2.30e+00 99.9 %</td>
<td>2.67e+07 99.9 %</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
</tr>
<tr>
<td>442: mfem::CudaRK4Solver::Step(mfem::CudaVector&amp;, double)</td>
<td>1.80e+14  99.9 %</td>
<td>1.38e+01 99.9 %</td>
<td>2.30e+00 99.9 %</td>
<td>2.67e+07 99.9 %</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
</tr>
<tr>
<td>loop at laghos_solver.cpp: 231</td>
<td>4.56e+13  25.1 %</td>
<td>3.45e+00 25.0 %</td>
<td>5.78e-01 25.0 %</td>
<td>6.68e+06 25.0 %</td>
<td>7.8</td>
<td>7.8</td>
<td>7.8</td>
<td>7.8</td>
<td>7.8</td>
</tr>
<tr>
<td>146: mfem::hydrodynamics::LagrangianHydroOperator::Multi</td>
<td>4.00e+13  24.2 %</td>
<td>3.30e+00 23.9 %</td>
<td>5.58e-01 24.2 %</td>
<td>6.33e+06 23.7 %</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
</tr>
<tr>
<td>loop at laghos_solver.cpp: 231</td>
<td>4.33e+13  23.8 %</td>
<td>3.30e+00 23.9 %</td>
<td>5.58e-01 24.2 %</td>
<td>6.33e+06 23.7 %</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
</tr>
<tr>
<td>252: [I].mfem::CudaCGSolver::Multi(mfem::CudaVector)</td>
<td>4.33e+13  23.8 %</td>
<td>3.25e+00 23.5 %</td>
<td>5.44e-01 23.6 %</td>
<td>6.10e+06 22.8 %</td>
<td>6.7</td>
<td>6.7</td>
<td>6.7</td>
<td>6.7</td>
<td>6.7</td>
</tr>
<tr>
<td>157: mfem::CudaCGSolver::h_Multi(mfem::CudaVector)</td>
<td>6.33e+13  23.8 %</td>
<td>3.25e+00 23.5 %</td>
<td>5.44e-01 23.6 %</td>
<td>6.10e+06 22.8 %</td>
<td>6.7</td>
<td>6.7</td>
<td>6.7</td>
<td>6.7</td>
<td>6.7</td>
</tr>
<tr>
<td>loop at solvers.cpp: 89</td>
<td>4.07e+13  22.4 %</td>
<td>3.07e+00 22.3 %</td>
<td>5.14e-01 22.3 %</td>
<td>5.73e+06 21.4 %</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
</tr>
<tr>
<td>137: mfem::hydrodynamics::CudaMassOperator</td>
<td>2.27e+13  12.5 %</td>
<td>3.75e+09 0.0 %</td>
<td>3.87e-01 16.8 %</td>
<td>5.73e+06 21.4 %</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
</tr>
<tr>
<td>135: mfem::CudaConstrainedOperator::Multi</td>
<td>2.21e+13  11.5 %</td>
<td>1.97e+09 14.3 %</td>
<td>3.47e-01 15.0 %</td>
<td>5.73e+06 21.4 %</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
</tr>
<tr>
<td>loop at solvers.cpp: 89</td>
<td>2.10e+13  11.5 %</td>
<td>1.97e+09 14.3 %</td>
<td>3.47e-01 15.0 %</td>
<td>5.73e+06 21.4 %</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
</tr>
<tr>
<td>210: mfem::CudaRAPOperator::Multi</td>
<td>7.51e+10  11.5 %</td>
<td>1.97e+09 14.3 %</td>
<td>3.47e-01 15.0 %</td>
<td>5.73e+06 21.4 %</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
</tr>
<tr>
<td>loop at solvers.cpp: 89</td>
<td>1.02e+13  5.6 %</td>
<td>6.42e-01 4.7 %</td>
<td>1.73e-01 7.5 %</td>
<td>2.86e+06 10.7 %</td>
<td>2.8</td>
<td>2.8</td>
<td>2.8</td>
<td>2.8</td>
<td>2.8</td>
</tr>
</tbody>
</table>
Selecting Metrics to Display Using the Column Selector
Using GPU Kernel Time to Guide Top-down Exploration

Select the header to select the column triangle indicates descending sort

<table>
<thead>
<tr>
<th>Scope</th>
<th>cycles:Sum (i)</th>
<th>cycles:Sum (j)</th>
<th>GKR (s):Sum</th>
<th>GXCOPY (s):Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;program root&gt;</td>
<td>1.82e+14 100.0</td>
<td>1.82e+14 100.0</td>
<td>1.38e+01 100</td>
<td>2.11e+00 100</td>
</tr>
<tr>
<td>main</td>
<td>1.80e+14 99.1</td>
<td>1.80e+14 99.0</td>
<td>1.38e+01 99.7</td>
<td>2.30e+00 99.9</td>
</tr>
<tr>
<td>loop at laghos.cpp: 427</td>
<td>1.80e+14 99.0</td>
<td>1.80e+14 99.7</td>
<td>1.38e+01 99.4</td>
<td>2.30e+00 99.5</td>
</tr>
<tr>
<td>442: mfm::CudaRK4Solver::Step[mfm::CudaVector&lt;6, double&gt;&amp;, double&amp;]</td>
<td>4.56e+13 25.1</td>
<td>4.56e+13 25.1</td>
<td>3.45e+00 25.0</td>
<td>5.78e-01 25.0</td>
</tr>
<tr>
<td>146: mfm::hydrodynamics::LagrangianHydroOperator::Mult[mfm::CudaVector const&amp; mfm::CudaVector const&amp;]</td>
<td>4.40e+13 24.2</td>
<td>4.40e+13 24.2</td>
<td>3.30e+00 23.9</td>
<td>5.78e-01 24.2</td>
</tr>
<tr>
<td>loop at laghos_solver.cpp: 231</td>
<td>3.33e+13 23.8</td>
<td>3.33e+13 23.8</td>
<td>2.35e+00 23.5</td>
<td>4.44e-01 23.6</td>
</tr>
<tr>
<td>252: [i] mfm::CudaCGSolver::Mult[mfm::CudaVector const&amp; mfm::CudaVector const&amp;]</td>
<td>3.33e+13 23.8</td>
<td>3.33e+13 23.8</td>
<td>2.35e+00 23.5</td>
<td>4.44e-01 23.6</td>
</tr>
<tr>
<td>157: mfm::CudaCGSolver::h Mult[mfm::CudaVector const&amp; mfm::CudaVector const&amp;]</td>
<td>4.07e+12 22.9</td>
<td>4.07e+12 22.9</td>
<td>3.07e+00 22.3</td>
<td>3.87e-01 22.3</td>
</tr>
<tr>
<td>loop at solvers.cpp: 89</td>
<td>2.73e+12 12.5</td>
<td>2.73e+12 12.5</td>
<td>1.85e+00 13.1</td>
<td>1.97e+00 14.3</td>
</tr>
<tr>
<td>137: mfm::hydrodynamics::CudaMassOperator::Mult[mfm::CudaVector const&amp; mfm::CudaVector const&amp;]</td>
<td>1.10e+12 11.5</td>
<td>1.10e+12 11.5</td>
<td>7.51e+00 13.3</td>
<td>3.47e+00 15.0</td>
</tr>
<tr>
<td>135: mfm::CudaConstrainedDOperator::Mult[mfm::CudaVector const&amp; mfm::CudaVector const&amp; mfm::CudaVector const&amp;]</td>
<td>2.10e+12 11.5</td>
<td>2.10e+12 11.5</td>
<td>7.51e+00 13.3</td>
<td>3.47e+00 15.0</td>
</tr>
<tr>
<td>210: mfm::CudaRAPOperator::Mult[mfm::CudaVector const&amp; mfm::CudaVector const&amp;]</td>
<td>2.50e+11 11.1</td>
<td>2.50e+11 11.1</td>
<td>7.03e-01 5.1</td>
<td>1.97e-01 1.4</td>
</tr>
<tr>
<td>85: mfm::CudaBilinearForm::Mult[mfm::CudaVector const&amp; mfm::CudaVector const&amp;]</td>
<td>6.62e+11 0.3</td>
<td>6.62e+11 0.3</td>
<td>1.97e-01 1.4</td>
<td>1.97e-01 1.4</td>
</tr>
<tr>
<td>loop at bilinearform.cpp: 136</td>
<td>6.62e+11 0.3</td>
<td>6.62e+11 0.3</td>
<td>1.97e-01 1.4</td>
<td>1.97e-01 1.4</td>
</tr>
<tr>
<td>138: mfm::CudaMassIntegrator::MultAdd[mfm::CudaVector const&amp; mfm::CudaVector const&amp;, double]</td>
<td>6.62e+11 0.3</td>
<td>6.62e+11 0.3</td>
<td>1.97e-01 1.4</td>
<td>1.97e-01 1.4</td>
</tr>
<tr>
<td>514: rMassMulAddInt, int, int, int, double const*, double</td>
<td>1.47e+11 2.7</td>
<td>1.47e+11 2.7</td>
<td>1.97e-01 1.4</td>
<td>1.97e-01 1.4</td>
</tr>
<tr>
<td>303: rMassMulAddD&lt;3, 3, 4&gt;(int, double const*, double&lt;</td>
<td>3.36e+10 2.2</td>
<td>3.36e+10 2.2</td>
<td>1.97e-01 1.4</td>
<td>1.97e-01 1.4</td>
</tr>
<tr>
<td>29: [0] __wrapper_device_stub rMassMulAddD&lt;3, 3, 4&gt;(</td>
<td>3.36e+10 2.2</td>
<td>3.36e+10 2.2</td>
<td>1.97e-01 1.4</td>
<td>1.97e-01 1.4</td>
</tr>
<tr>
<td>111: [i] __device_stub rMassMulAddD&lt;3, 3, 4&gt;(</td>
<td>3.36e+10 2.2</td>
<td>3.36e+10 2.2</td>
<td>1.97e-01 1.4</td>
<td>1.97e-01 1.4</td>
</tr>
<tr>
<td>110: [i] cudaLaunchKernel&lt;char&gt;</td>
<td>0.97e-01 1.4</td>
<td>0.97e-01 1.4</td>
<td>1.97e-01 1.4</td>
<td>1.97e-01 1.4</td>
</tr>
</tbody>
</table>

GPU Kernel Launch
Using GPU Kernel Time to Guide Bottom-up Exploration
### HPCToolkit’s GPU Instruction Sampling Metrics (NVIDIA Only)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GINST:STL_ANY</td>
<td>GPU instruction stalls: any (sum of all STL metrics other than NONE)</td>
</tr>
<tr>
<td>GINST:STL_NONE</td>
<td>GPU instruction stalls: no stall</td>
</tr>
<tr>
<td>GINST:STL_IFET</td>
<td>GPU instruction stalls: await availability of next instruction (fetch or branch delay)</td>
</tr>
<tr>
<td>GINST:STL_IDEP</td>
<td>GPU instruction stalls: await satisfaction of instruction input dependence</td>
</tr>
<tr>
<td>GINST:STL_GMEM</td>
<td>GPU instruction stalls: await completion of global memory access</td>
</tr>
<tr>
<td>GINST:STL_TMEM</td>
<td>GPU instruction stalls: texture memory request queue full</td>
</tr>
<tr>
<td>GINST:STL_SYNC</td>
<td>GPU instruction stalls: await completion of thread or memory synchronization</td>
</tr>
<tr>
<td>GINST:STL_CMEM</td>
<td>GPU instruction stalls: await completion of constant or immediate memory access</td>
</tr>
<tr>
<td>GINST:STLPIPE</td>
<td>GPU instruction stalls: await completion of required compute resources</td>
</tr>
<tr>
<td>GINST:STL_MTHR</td>
<td>GPU instruction stalls: global memory request queue full</td>
</tr>
<tr>
<td>GINST:STL_NSEL</td>
<td>GPU instruction stalls: not selected for issue but ready</td>
</tr>
<tr>
<td>GINST:STL_OTHR</td>
<td>GPU instruction stalls: other</td>
</tr>
<tr>
<td>GINST:STL_SLP</td>
<td>GPU instruction stalls: sleep</td>
</tr>
</tbody>
</table>
Approximation of GPU Calling Contexts to Understand Performance

- GPU code from C++ template-based programming models is complex
- NVIDIA GPUs collect flat PC samples
- Flat profiles for instantiations of complex C++ templates are inscrutable

- HPCToolkit reconstructs approximate GPU calling contexts
  - Reconstruct call graph from machine code
  - Infer calls at call sites
  - PC samples of call instructions indicate calls
  - Use call counts to apportion costs to call sites
  - PC samples in a routine
Approximation of GPU Calling Contexts to Understand Performance

- GPU code from C++ template-based programming models is complex.
- NVIDIA GPUs collect flat PC samples.
- Flat profiles for instantiations of complex C++ templates are inscrutable.
- HPCToolkit reconstructs approximate GPU calling contexts.
- PC samples of call instructions indicate calls.
- Use counts to split costs.
- PC samples in a routine.
- Infer caller or distribute costs equally to potential callers.
Approximation of GPU Calling Contexts to Understand Performance

• GPU code from C++ template-based programming models is complex
• NVIDIA GPUs collect flat PC samples
• Flat profiles for instantiations of complex C++ templates are inscrutable

• HPCToolkit reconstructs approximate GPU calling contexts
  – Reconstruct call graph from machine code
  – Infer calls at call sites
    • PC samples of call instructions indicate calls
    • Use call counts to apportion costs to call sites
  – PC samples in a routine
Approximation of GPU Calling Contexts to Understand Performance

• GPU code from C++ template-based programming models is complex
• NVIDIA GPUs collect flat PC samples
• Flat profiles for instantiations of complex C++ templates are inscrutable
• HPCToolkit reconstructs approximate GPU calling contexts
– PC samples of call instructions indicate calls
• Use counts to split costs
– PC samples in a routine
• Infer caller or distribute costs equally to potential callers

CPU Calling Context

GPU API Node

GPU Calling Context

GPU Loops

GPU Hotspot
Accuracy of GPU Calling Context Recovery: Case Studies

• Compute approximate call counts as the basis for partitioning the cost of function invocations across call sites
  • Use call samples at call sites, data flow analysis to propagate call approximation upward
    • if samples were collected in some function f, if no calls to f were sampled, equally attribute f to each of its call sites
    – How accurate is our approximation?

• Evaluation methodology
  – Use NVIDIA’s nvbit to
    – instrument call and return for GPU functions
    – instrument basic blocks to collect block histogram
Accuracy of GPU Calling Context Recovery: Case Studies

- Error partitioning a function’s cost among call sites

\[
\text{Error} = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} \left( \sqrt{\sum_{j=0}^{i-1} \left( \frac{f_N(i,j) - f_H(i,j)}{i_c} \right)^2} \right)^2}
\]

- Experimental study

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Unique Call Paths</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic_INIT_VIEW1D_OFFSET</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Basic_REDUCE3_INT</td>
<td>113</td>
<td>0.03</td>
</tr>
<tr>
<td>Stream_DOT</td>
<td>60</td>
<td>0.006</td>
</tr>
<tr>
<td>Stream_TRIAD</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Apps_PRESSURE</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Apps_FIR</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Apps_DEL_DOT_VEC_2D</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Apps_VOL3D</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Costs of GPU Functions Distributed Among Their Call Sites

• Use call site frequency approximation
• Use Gprof assumption: all calls to a function incur exactly the same cost
  – known to not be true in all cases, but a useful assumption nevertheless
GPU call site attribution example

- **Case study: call function GPU “vectorAdd”**
  - iter1 = N
  - iter2 = 2N

Note: the computation by the function is synthetic and is not a vector addition. The name came from code that was hacked to do perform an unrelated computation.
Profiling Result for GPU-accelerated Example

GPU kernel

loop 14
loop 11
device fn calls

device fn calls
Support for OpenMP TARGET

- HPCToolkit implementation of OMPT OpenMP API
  - host monitoring
    - leverages callbacks for regions, threads, tasks
  - GPU monitoring
    - leverages callbacks for device initialization, kernel launch, data operations
  - reconstruction of user-level calling contexts
- Leverages implementation of OMPT in LLVM OpenMP and libomptarget

ECP QMCPACK Project: miniqmc using OpenMP TARGET (Power9 + NVIDIA V100)

Reconstruct full calling contexts that include
- Outlined procedures for OpenMP parallel regions
- Offloaded OpenMP TARGET computation and synchronization
Support for RAJA and Kokkos C++ Template-based Models

• RAJA and Kokkos provide portability layers atop C++ template-based programming abstractions

• HPCToolkit employs binary analysis to recover information about procedures, inlined functions and templates, and loops
  – Enables both developers and users to understand complex template instantiation present with these models

ECP EXAALT Project: lammps using Kokkos over CUDA (Power9 + NVIDIA V100)

Reconstruct full calling contexts that include
• Inlined Kokkos templates
• Offloaded Kokkos CUDA computation
Prototype Integration with AMD’s Roctracer GPU Monitoring Framework

AMD MatrixTranspose Testcase for Roctracer
(AMD Ryzen + AMD 580 GPU)

• Use AMD Roctracer activity API to trace GPU activity
  – kernel launches
  – explicit memory copies

• Current prototype supports AMD’s HIP programming model

Attribute AMD GPU activity
• Kernel execution
• Memory copies
HPCToolkit Challenges and Limitations

- **Fine-grain measurement and attribution of GPU performance**
  - PC sampling overhead on NVIDIA GPUs is currently very high: a function of NVIDIA’s CUPTI implementation
  - No available hardware support for fine-grain measurement on Intel and AMD GPUs

- **GPU tracing in HPCToolkit**
  - Creates one tool thread per GPU stream when tracing
  - OK for a small number of streams but many streams can be problematic

- **Cost of call path sampling**
  - Call path unwinding of GPU kernel invocations is costly (~2x execution dilation for Laghos)
  - Best solution is to avoid some of it, e.g. sample GPU kernel invocations
  - **Currently, hpcprof and hpcprof-mpi compute dense vectors of metrics**
    - Designed for few CPU metrics, not O(100) GPU metrics: space and time problem for analysis
Analysis and Optimization Case Studies

- **Environments**
  - Summit
    - cuda/10.1.168
    - gcc/6.4.0
  - Local
    - cuda/10.1.168
    - gcc/7.3.0
Case 1: Locating expensive GPU APIs with profile view

- Laghos
  - 1 MPI process
  - 1 GPU stream per process
nvprof: missing CPU calling context

- **Goal:** Associate every GPU API with its CPU calling context
Context-aware optimizations

<table>
<thead>
<tr>
<th>Scope</th>
<th>XDMOV_IMPORTANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;cuda copy&gt;</td>
<td>13.23</td>
</tr>
<tr>
<td>72: mfem::memcpyDtoD(void*, void const*, unsigned long, bool)</td>
<td>6.83</td>
</tr>
<tr>
<td>34: [l] mfem::CudaVector::SetSize(unsigned long, void const*)</td>
<td>6.83</td>
</tr>
<tr>
<td>109: mfem::CudaVector::operator=(mfem::CudaVector const&amp;)</td>
<td>6.83</td>
</tr>
<tr>
<td>49: mfem::CudaProlongationOperator::Mult(mfem::CudaVector const&amp; mfem::CudaVector&amp;))</td>
<td>2.20</td>
</tr>
<tr>
<td>86: mfem::CudaRAPOperator::Mult(mfem::CudaVector const&amp; mfem::CudaVector&amp;))</td>
<td>2.14</td>
</tr>
<tr>
<td>245: mfem::hydrodynamics::LagrangianHydroOperator::Mult(mfem::CudaVector const&amp; mfem::CudaVector&amp;))</td>
<td>0.06</td>
</tr>
<tr>
<td>29: mfem::CudaProlongationOperator::Mult(mfem::CudaVector const&amp; mfem::CudaVector&amp;))</td>
<td>2.20</td>
</tr>
<tr>
<td>84: mfem::CudaRAPOperator::Mult(mfem::CudaVector const&amp; mfem::CudaVector&amp;))</td>
<td>2.14</td>
</tr>
<tr>
<td>256: mfem::hydrodynamics::LagrangianHydroOperator::Mult(mfem::CudaVector const&amp; mfem::CudaVector&amp;))</td>
<td>0.06</td>
</tr>
<tr>
<td>Case 1</td>
<td></td>
</tr>
<tr>
<td>Case 2</td>
<td></td>
</tr>
<tr>
<td>130: mfem::hydrodynamics::CudaMassOperator::Mult(mfem::CudaVector const&amp; mfem::CudaVector&amp;))</td>
<td>2.14</td>
</tr>
<tr>
<td>212: mfem::hydrodynamics::LagrangianHydroOperator::Mult(mfem::CudaVector const&amp; mfem::CudaVector&amp;))</td>
<td>0.15</td>
</tr>
<tr>
<td>39: mfem::CudaCGSolver::h_Mult(mfem::CudaVector const&amp; mfem::CudaVector&amp;)) cons</td>
<td>0.12</td>
</tr>
<tr>
<td>436: main</td>
<td>0.01</td>
</tr>
<tr>
<td>Case 3</td>
<td></td>
</tr>
<tr>
<td>61: cuVectorDot(unsigned long, double const*, double const*)</td>
<td>6.16</td>
</tr>
</tbody>
</table>
Performance insight: Pin host memory page

- A small amount of memory is transferred from device to host each time, repeated 197000 times

<table>
<thead>
<tr>
<th>Scope</th>
<th>GXCOPY (s):Sum (I)</th>
<th>GXCOPY:COUNT:Sum (I)</th>
<th>GXCOPY:D2H (B):Sum (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cuVectorDot(unsigned long, double const*, double const*)</code></td>
<td>3.67e-01 46.3%</td>
<td>1.97e+05 37.9%</td>
<td>7.81e+06 20.4%</td>
</tr>
</tbody>
</table>

- Avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory
  - Use pinned memory when data movement frequency is high but size is small
Case 2: Trace Applications at Large-scale

- **Nyx**
  - 6 MPI processes
  - 16 GPU stream per process

- **DCA++**
  - 60 MPI processes
  - 128 GPU stream per process
nvprof: Non-scalable Tracing of DCA++

• **nvprof**
  – With CPU profiling enabled, hangs on Summit
  – Without CPU profiling
    • Collects 1.1 GB data

• **Hpctoolkit**
  – CPU+GPU hybrid profiling with full calling context
    • Collects 0.13 GB data
    • Data can be further reduced by sampling GPU events
Nyx trace view
DCA++ trace view
Nyx insufficient GPU stream parallelism

- On GPU, streams are not working concurrently
Nyx cudaCallBack issue

• On CPU, amrex::Gpu::Exlixir::clear() invokes stream callbacks

```cpp
33 void
34 Elixir::clear () noexcept
35 {
36 #ifdef AMREX_USE_GPU
37   if (Gpu::inLaunchRegion())
38     {
39       if (m_p != nullptr) {
40         void** p = static_cast<void**>(std::malloc(2*sizeof(void*)));
41         p[0] = m_p;
42         p[1] = (void*)m_arena;
43         AMREX_HIP_OR_CUDA(
44           AMREX_HIP_SAFE_CALL ( hipStreamAddCallback(Gpu::gpuStream(),
45             amrex_elixir_delete, p, 0)));,
46           AMREX_CUDA_SAFE_CALL(cudaStreamAddCallback(Gpu::gpuStream(),
47             amrex_elixir_delete, p, 0)));;
48       }
49     }
50   else
51   #endif
```
Nyx performance insight

• A bug present in the current version of CUDA (10.1). If a callBack is called in a place where multiple streams are used, the device kernels artificially synchronize and have no overlap.
• Fixed in CUDA-10.2?
• Workaround
  – The Elixir object holds a copy of the data pointer to prevent it from being destroyed before the related device kernels are completed
  – Allocate new objects outside the compute loop and delete them after the completion of the work
Case 3: Fine-grained GPU Kernel Tuning

- Nekbone: A lightweight subset of Nek5000 that mimics the essential computational complexity of Nek5000
nvprof: Limited source level performance metrics

- No loop structure,
- No GPU calling context,
- No instruction mix
Nekbone Profile View
Performance insight 1: Execution dependency

- The hotspot statement is waiting for $j$ and $k$
Strength reduction

- **MISC.CONVERT: I2F, F2I, MUFU instructions**
  - NVIDIA GPUs convert integer to float for division
  - High latency and low throughput instruction
- **Replace** $j = \frac{it}{N}$ **by** $j = it \times (1/N)$ **and precompute** $1/N$
Separate GPU instructions into classes

- **Memory operations**
  - instruction (load, store)
  - size
  - memory kind (global memory, texture memory, constant memory)

- **Floating point**
  - instruction (add, mul, mad)
  - size
  - compute unit (tensor unit, floating point unit)

- **Integer operations**

- **Control operations**
  - branches, calls
Performance insight 2: Instruction Throughput

- Estimate instruction throughput based on pc samples

\[ \frac{INS}{TIME} \]

- GFLOPS = THROUGHPUT\_DP

- Arithmetic Intensity = \( \frac{\text{THROUGHPUT}\_\text{GMEM}}{\text{THROUGHPUT}\_\text{DP}} \)
Roofline analysis

- 83.9% of peak performance
Performance insight 3: unfused DMUL and DADD

- **DMUL**: $6.51 \times 10^5$
- **DADD**: $4.55 \times 10^5$

- If all paired DMUL and DADD instructions are fused to MAD instructions

\[
\left( 4.55 \times 10^5 + 3.08 \times 10^6 \right) - \frac{3.08 \times 10^6}{3.08 \times 10^6} = 14.7\%
\]

- $1663 \text{ GFLOPS} \times 114.7\% = 1908 \text{ GFLOPS} (99\% \text{ of peak})$
Case Study Acknowledgements

• ORNL
  – Ronnie Chatterjee

• IBM
  – Eric Liu

• NERSC
  – Christopher Daley
  – Jean Sexton
  – Kevin Gott
Installing HPCToolkit for Analysis of GPU-accelerated Codes

• Full instructions: http://hpctoolkit.org/software-instructions.html

• The short form
  • Clone spack
    – command: git clone https://github.com/spack/spack
  • Configure a packages.yaml file
    – specify your platform’s installation of CUDA or ROCM
    – specify your platform’s installation of MPI
    – use an appropriate GCC compiler
      • ensure that a GCC version >= 5 is on your path. typically, we use GCC 7.3
      • spack compiler find
  • Install software for your platform using spack
    – NVIDIA GPUs: spack install hpctoolkit@master +cuda +mpi
    – AMD GPUs: spack install hpctoolkit@master +rocm +mpi