Using HPCToolkit to Measure and Analyze the Performance of Parallel Applications

John Mellor-Crummey
Rice University

ATPESC 2020
August 5, 2020

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

• **Current funding**
  - DOE Exascale Computing Project (Subcontract 4000151982)
  - NSF Software Infrastructure for Sustained Innovation (Collaborative Agreement 1450273)
  - DOE Labs: ANL (Subcontract 9F-60073), Tri-labs (LLNL Subcontract B633244)
  - Industry: AMD

• **Team**
  - Rice University
    • HPCToolkit PI: Prof. John Mellor-Crummey
    • Research staff: Laksono Adhianto, Mark Krentel, Xiaozhu Meng, Scott Warren
    • Contractor: Marty Itzkowitz
    • Students: Jonathon Anderson, Aaron Cherian, Dejan Grubisic, Yumeng Liu, Keren Zhou
    • Recent summer interns: Vladimir Indjic, Tijana Jovanovic, Aleksa Simovic
  - University of Wisconsin – Madison
    • Dyninst PI: Prof. Barton Miller
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 Workflow

 Compile & link

 Source code → Optimized binary → Profile execution [hpcrun] → Call path profile

 Optimized binary → Binary analysis [hpcstruct] → Program structure

 Interpret profile correlate w/ source [hpcprof/hpcprof-mpi] → Database

 Presentation [hpcviewer/hpctraceviewer]
HPCToolkit Workflow

Measure execution unobtrusively with **hpcrun**

— Launch optimized dynamically-linked application binaries
— Collect statistical call path profiles of events of interest
— Where necessary, intercept interfaces for control and measurement
Call Path Profiling

- Measure and attribute costs in context
  - Sample timer or hardware counter overflows
  - Gather CPU calling context using stack unwinding

Overhead proportional to sampling frequency, not call frequency
HPCToolkit Workflow

Analyze binary with **hpcstruct**: recover program structure

— Analyze machine code, line map, debugging information
— Extract loop nests & identify inlined procedures
— Map transformed loops and procedures to source
HPCToolkit Workflow

• Combine multiple profiles
  — Multiple threads; multiple processes; multiple executions

• Correlate metrics to static & dynamic program structure
HPCToolkit Workflow

Presentation
— Explore performance data from multiple perspectives
  – Rank order by metrics to focus on what’s important
    e.g., cycles, instructions, GPU instructions, GPU stalls
  – Compute derived metrics to help gain insight
    e.g. scalability losses
— Explore evolution of behavior over time
Code-centric Analysis with hpcviewer

- function calls in full context
- inlined procedures
- inlined templates
- outlined OpenMP loops
- loops
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
Time-centric Analysis with hpctraceviewer

Experimental version of QMCPack on Blue Gene Q

- 32 ranks
- 32 threads each

Ranks/Threads

Call Path at Cross Hair
Demo: QMCPACK

QMCPACK in ECP

• Goal
  – Find, predict, and control materials and properties at the quantum level with an unprecedented and systematically improvable accuracy using quantum Monte Carlo methods

• Focus:
  – transition metal oxide systems where the additional capability over existing methods is essential

• Hope
  – have a major impact on materials science
    • e.g., help to uncover the mechanisms behind high-temperature superconductivity
Measurement and Analysis of GPU-accelerated Applications

- **What HPCToolkit GUIs present for GPU-accelerated applications**
  - Profile views displaying call paths that integrate CPU and GPU call paths
  - Trace views that attribute CPU threads and GPU streams to full heterogeneous call paths

- **What HPCToolkit collects**
  - Heterogeneous call path profiles and call path traces

- **How HPCToolkit collects information**
  - **CPU**
    - Sampling-based measurement of application thread activity in user space and in the kernel
    - Measurement of blocking time using Linux perf_events context switch notifications
  - **GPU**
    - Coarse-grain measurement of GPU operations (memory copies, kernel launches, …)
    - Fine-grain measurement of GPU kernels using PC Sampling (NVIDIA only)
## 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, analysis of GPU binaries</td>
<td>no</td>
<td>GTPin</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 a development branch**
Miniqmc GPU OpenMP Example: A Trace View

Compute Node
- 2 Power9
- 6xNVIDIA GPU

Compiled with IBM XL
- 1 rank
- 10 OMP threads
- 32 GPU streams

Trace view shows
- master thread
- OMP worker threads,
- GPU streams
- all activities attributed to full calling context
Miniqmc GPU OpenMP Example: A Profile View

Compute Node
- 2 Power9
- 6xNVIDIA GPU

Compiled with IBM XL
- 1 rank
- 10 OMP threads
- 32 GPU streams

Profile view shows OMP target offload in full calling context

hpctoolkit-tutorial-examples/examples/gpu/openmp/miniqmc
Quicksilver GPU CUDA Example: Detailed Profile View

- **Compute Node**
  - 2 Power9
  - 6xNVIDIA GPU
- **Optimized (-O2) compilation with nvcc**
- **1 GPU stream**
- **Detailed measurement and attribution using PC sampling**
- **Reconstruct approximate call graph on GPU from flat PC samples**
- **Attribute information to heterogeneous calling context including**
  - CPU calling context
  - GPU kernel
  - GPU calling context
  - GPU loops
  - GPU statements
- **Metrics**
  - instructions executed
  - instruction stalls and reasons
  - GPU utilization
Quicksilver GPU CUDA Example: Detailed Profile View

- Optimized (-O2) compilation with nvcc
- 1 GPU stream
- Detailed measurement and attribution using PC sampling
- Reconstruct approximate call graph on GPU from flat PC samples
- Attribute information to heterogeneous calling context including
  - CPU calling context
  - GPU kernel
  - GPU calling context
  - GPU loops
  - GPU statements
- Metrics
  - instructions executed
  - instruction stalls and reasons
  - GPU utilization

Detailed Attribution on GPUs

```
<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)
      [1] inlined from MacroscopicCrossSection.cc: 45
    41: NuclearData::getReactionCrossSection(unsigned int, unsigned int)
      [1] inlined from NuclearData.cc: 193
    QS_Vector.hh: 94
```

hpctoolkit-tutorial-examples/examples/gpu/quicksilver
Work in Progress

• **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

• **Scalability**
  • Add multithreading to hpcprof-mpi to accelerate analysis
  • Overhaul representations used for recording measurement and analysis results to use sparse forms
  • Overhaul file management to use few large files instead of two per thread

• **User interface**
  • Integrated hpcviewer and hpctraceviewer
  • Modernized implementation using latest Eclipse and Java
Bonus Content
Download Hands-on Tutorial Examples

- **git clone** [https://github.com/hpctoolkit/hpctoolkit-tutorial-examples](https://github.com/hpctoolkit/hpctoolkit-tutorial-examples)
- **Configured for use on**
  - ANL’s Theta
    - AMG2006
      - MPI + OpenMP
  - ORNL’s Ascent
    - miniqmc
      - CPU OpenMP: GCC, XL
      - GPU OpenMP Target: XL
    - quicksilver
      - GPU CUDA: nvcc
Installing HPCToolkit: Configuration and Installation on Ascent

Use spack for installation

- `git clone https://github.com/spack/spack`
- `module load gcc`
  - ensure that a GCC version >= 5 is on your path. Typically, we use GCC 7 to compile hpctoolkit
- `export SPACK_ROOT=`pwd`/spack`
- `export PATH=${SPACK_ROOT}/bin:$PATH`
- `source ${SPACK_ROOT}/share/spack/setup-env.sh`
- `spack compiler find`
- `configure ~/.spack/packages.yaml for custom build`
- `spack install hpctoolkit`
- see [http://hpctoolkit.org/software-instructions.html](http://hpctoolkit.org/software-instructions.html) for additional details and troubleshooting
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 \([H \times T \log N]\) 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: `aprunc`
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 ...
After Hiding the Empty MPI Helper Threads
A Detail of Only the MPI Threads
Only the MPI Threads - Analysis using the Statistics Panel
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
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

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 STALL 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:STL_PIPE</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
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

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

geometric mean across GPU functions of (root mean square error of call attribution across all of a function’s call sites comparing our approximation vs. attribution using exact nvbit measurements)

• 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 libompttarget

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
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::memcpy::DtoD(void*, void const*, unsigned long, bool)</td>
<td>6.83%</td>
</tr>
<tr>
<td>34: [I] 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::MultTranspose(mfem::CudaVector const&amp;, mfem::CudaVector)</td>
<td>2.20%</td>
</tr>
<tr>
<td>86: mfem::CudaRAPOperator::Mult(mfem::CudaVector const&amp;, mfem::CudaVector)</td>
<td>2.14%</td>
</tr>
<tr>
<td>245: mfem::hydrodynamics::LagrangianHydroOperator::Mult(mfem::CudaVector const&amp;)</td>
<td>0.06%</td>
</tr>
<tr>
<td>29: mfem::CudaProlongationOperator::Mult(mfem::CudaVector const&amp;, mfem::CudaVector)</td>
<td>2.20%</td>
</tr>
<tr>
<td>84: mfem::CudaRAPOperator::Mult(mfem::CudaVector const&amp;, mfem::CudaVector)</td>
<td>2.14%</td>
</tr>
<tr>
<td>256: mfem::hydrodynamics::LagrangianHydroOperator::Mult(mfem::CudaVector const&amp;)</td>
<td>0.06%</td>
</tr>
<tr>
<td>130: mfem::hydrodynamics::CudaMassOperator::Mult(mfem::CudaVector const&amp;, mfem::CudaVector)</td>
<td>2.14%</td>
</tr>
<tr>
<td>212: mfem::hydrodynamics::LagrangianHydroOperator::Mult(mfem::CudaVector const&amp;)</td>
<td>0.15%</td>
</tr>
<tr>
<td>39: mfem::CudaCGSolver::h_Mult(mfem::CudaVector const&amp;, mfem::CudaVector&amp;) const</td>
<td>0.12%</td>
</tr>
<tr>
<td>436: main</td>
<td>0.01%</td>
</tr>
<tr>
<td>61: cuVectorDot(unsigned long, double const*, double const*)</td>
<td>6.16%</td>
</tr>
</tbody>
</table>

**Case 1**

**Case 2**

**Case 3**
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>cuVectorDot(unsigned long, double const*, double const*)</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 Exlixir::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 #endif
51   else
52 #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

```
int i, j, k;
for (int it = threadIdx.x; it < e.size; it += blockDim.x) {
    j = it / N;
    i = it - j * N;
    k = j / N;
    j -= k * N;
    double wr = 0.0;
}
```

### Scope

<table>
<thead>
<tr>
<th>Scope</th>
<th>GiNS:Sum (I)</th>
<th>GiNS:Sum (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>516: main</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150: [l] nekbone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: __device_stub_Z7nekbonesPdS_S_S_S_i(double*, double*, double*, double*, double*, int)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13: [l] cudaLaunchKernel&lt;char&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>209: &lt;gpu kernel&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>174: nekbone(double*, double*, double*, double*, double*, int)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop at cuda1: 18</td>
<td>3.17e+00 48.1%</td>
<td>3.17e+00 48.1%</td>
</tr>
<tr>
<td>loop at cuda1: 39</td>
<td>2.21e+00 33.6%</td>
<td>2.21e+00 33.6%</td>
</tr>
<tr>
<td>loop at cuda1: 11</td>
<td>6.47e+07 9.8%</td>
<td>6.47e+07 9.8%</td>
</tr>
<tr>
<td>cuda1: 39</td>
<td>3.30e+07 5.0%</td>
<td>3.30e+07 5.0%</td>
</tr>
<tr>
<td>cuda1: 11</td>
<td>1.31e+07 2.0%</td>
<td>1.31e+07 2.0%</td>
</tr>
<tr>
<td>cuda1: 15</td>
<td>2.76e+06 0.4%</td>
<td>2.76e+06 0.4%</td>
</tr>
</tbody>
</table>
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** \( j = it \times \left(\frac{1}{N}\right) \) **and precompute** \( \frac{1}{N} \)
Coming Attraction: Instruction-level Analysis

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

\[
\text{THROUGHPUT} = \frac{\text{INS}}{\text{TIME}}
\]

- \(GFLOPS = \text{THROUGHPUT}_{DP}\)

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

$$\left(4.55 \times 10^5 + 3.08 \times 10^6\right) \div 3.08 \times 10^6 = 14.7\%$$

- 1663 GFLOPS $\times 114.7\% = 1908$ GFLOPS (99% 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](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 \( \geq 5 \) 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`