HPCToolkit Performance Tools

Performance analysis of CPU and GPU-accelerated applications at Scale

HPCToolkit at Exascale on Frontier: 8K nodes, 64K MPI ranks + GPU Tiles

John Mellor-Crummey
Professor, Rice University

e xtremecomputingtraining.anl.gov
Government
- Exascale Computing Project 17-SC-20-SC
- Lawrence Livermore National Laboratory Subcontract B658833
- Argonne National Laboratory Subcontract 9F-60073

Corporate
- Advanced Micro Devices
- TotalEnergies EP Research & Technology USA, LLC.
Rice University’s HPCToolkit Performance Tools

Measure and analyze performance of CPU and GPU-accelerated applications

Easy: profile unmodified application binaries
Fast: low-overhead measurement
Informative: understand where an application spends its time and why
  - call path profiles associate metrics with application source code contexts
  - optional hierarchical traces to understand execution dynamics

Broad audience
  - application developers
  - framework developers
  - runtime and tool developers
HPCToolkit’s Workflow for CPU Applications

Source Files → Compile & Link → Optimized Binary

hpcrun
Profile execution on CPUs

Profile Files → Trace Files → Program Structure

hpcstruct
Analyze CPU program structure

hpcviewer
Present trace view and profile view

Database

hpcprof/hpcprof-mpi
Interpret profile Correlate w/ source
HPCToolkit’s Workflow for GPU-accelerated Applications

Source Files → Compile & Link → Optimized Binary → hpcrun → Profile execution on CPUs and GPUs → Profile Files

Optimized Binary → GPU Binary → hpcstruct → Analyze CPU/GPU program structure → Trace Files → Program Structure

GPU Binary → hpcviewer → Present trace view and profile view

hpcviewer → Database

hpcprof/hpcprof-mpi → Interpret profile Correlate w/ source
Step 1:
- Ensure that compilers record line mappings
- host compiler: -g
- nvcc: -lineinfo
Step 2:
- `hpcrun` collects call path profiles of events of interest
Measurement of CPU and GPU-accelerated Applications

**CPU**
- Sampling on timer interrupts and hardware counter overflows on the CPU

**GPU**
- Callbacks when GPU operations are launched/completed
- GPU event stream for GPU operations
- PC Samples in GPU kernels (NVIDIA)
- Instruction-level instrumentation (Intel)
Unwind when timer or hardware counter overflows
measurement overhead proportional to sampling frequency rather than call frequency
Unwind to capture context for GPU events such as kernel launches and data copies

Call path sample
- return address
- return address
- return address
- instruction pointer

Calling context tree
**hpcrun: Measure CPU and/or GPU activity**

### GPU profiling

```
hpcrun -e gpu=xxx <app> 
```

### GPU instrumentation (Intel GPU only)

```
hpcrun -e gpu=level0,inst=count,latency <app>
```

### GPU PC sampling (NVIDIA GPU only)

```
hpcrun -e gpu=nvidia,pc <app>
```

### CPU and GPU Tracing (in addition to profiling)

```
hpcrun -e CPUTIME -e gpu=xxx -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>
```

Profiles: aggregated on the fly
- a calling context tree per thread
- a calling context tree per GPU stream
- instruction level measurements

**CPU traces**
- trace of call stack samples

**GPU traces**
- trace of call stacks that initiate GPU operations
HPCToolkit’s Workflow for GPU-accelerated Applications

Step 3:
- **hpcstruct** recovers program structure about lines, loops, and inlined functions
hpcstruct: Analyze CPU and GPU Binaries Using Multiple Threads

Usage

hpcstruct [--gpucfg yes] <measurement-directory>

What it does

Recover program structure information

Files, functions, inlined templates or functions, loops, source lines

In parallel, analyze all CPU and GPU binaries that were measured by HPCToolkit

default: use size(CPU set)/2 threads

analyze large application binaries with 16 threads

analyze multiple small application binaries concurrently with 2 threads each

Cache binary analysis results for reuse when analyzing other executions

NOTE: --gpucfg yes needed only for analysis of GPU binaries when NVIDIA PC samples were collected
Step 4:
- `hpcprof/hpcprof-mpi` combines profiles from multiple threads and correlate metrics to static & dynamic program structure
hpcprof/hpcprof-mpi: Associate Measurements with Program Structure

Analyze data from modest executions with threaded parallelism

```
hpcprof <measurement-directory>
```

Analyze data from large executions using both distributed-memory and shared-memory parallelism

```
jsrun -n 32 -a 1 hpcprof-mpi <measurement-directory>
srun -n 32 hpcprof-mpi <measurement-directory>
aprun -n 128 -N 8 hpcprof-mpi <measurement-directory>
```
Step 4:
- **hpcviewer** - interactively explore profile and traces for GPU-accelerated applications
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 hpcviewer

The color at a particular point in a timeline indicates the CPU procedure or GPU kernel active at that time at the selected call stack depth.

A multi-level call stack based view of execution over time.

Minimap indicates part of execution trace shown.

A depth view showing the history of calling contexts for the thread with the cursor.

Call stack pane shows full calling context for the cursor.
White Intervals in Traces Indicate Blocking of CPU Threads or GPU Streams

Miniqmc: OpenMP on 32 CPU threads
hpcstruct Example: Analyze 7.7GB TensorFlow library (170MB text) in 77s
hpcprof-mpi: Analyze Measurements of LAMMPS @ 2K threads + 2K GPUs

Analysis on 8 nodes using 504 threads!

Completes in 41s!
Coarse- and Fine-grain Measurement on NVIDIA GPUs: LLNL’s Quicksilver

Compute Node
-2xPower9 + 4xNVIDIA GPUs

Optimized (-O2) compilation with nvcc

Detailed measurement and attribution using PC sampling

Attribute information to heterogeneous calling context

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

Analysis of PeleC using PC Sampling on an NVIDIA GPU

Improvement:
pass udata components as scalars
https://github.com/AMReX-Combustion/PelePhysics/pull/192
4% speedup on PeleC PMF drm19 test case

Cause:
passed udata structure pointer to lambda capture

9.4% GPU stalls
outside the loop
mostly memory stalls

9.4% GPU stalls
outside the loop
mostly memory stalls
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter

GAMESS original

All CPU threads and GPU streams
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter

GAMESS original

All GPU streams, whole execution
GPU load imbalance due to triangular iteration spaces

GAMESS original

GPU streams: 1 iteration
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter

GAMESS original
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter

GAMESS improved with better manual distribution of work in input.
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter

GAMESS improved adding Rank 0 Thread 0 to GPU streams.
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter

1 CPU Stream, 2 GPU Streams: 6 Iterations
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter

```
1096 C
1097 IJ=1-INC
1098 DO 150 I=2,NA
1099 IJ=IJ+INC
1100 IM1=I-1
1101 DO 140 J=1,IM1
1102 IJ=IJ+INC
1103 AIJ=A(IJ)
1104 IF(AIJ.EQ.ZERO) GO TO 140
1105 CALL DAXPY(MB, AIJ, B(I,1), NA, AB(J,1), NAB)
1106 CALL DAXPY(MB, AIJ, B(J,1), NA, AB(I,1), NAB)
1107 140 CONTINUE
1108 150 CONTINUE
1109 RETURN
1110 END
```
Measure and Attribute OpenMP Offloading

![Image of code and performance metrics]

<table>
<thead>
<tr>
<th>Scope</th>
<th>GPUOP (sec):Sum (E)</th>
<th>GPUOP (sec):Sum (I)</th>
<th>GKER (sec):Sum (I)</th>
<th>GKER (sec):Sum (E)</th>
<th>GMEM (sec):Sum (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiment Aggregate Metrics</td>
<td>6.23e+00 100.0%</td>
<td>6.23e+00 100.0%</td>
<td>5.50e+00 100.0%</td>
<td>5.50e+00 100.0%</td>
<td>7.10e+03 100.0%</td>
</tr>
<tr>
<td>&lt;program root&gt;</td>
<td>6.23e+00 100.0%</td>
<td>6.23e+00 100.0%</td>
<td>5.50e+00 100.0%</td>
<td>5.50e+00 100.0%</td>
<td>7.10e+03 100.0%</td>
</tr>
<tr>
<td>main</td>
<td>6.23e+00 100.0%</td>
<td>6.23e+00 100.0%</td>
<td>5.50e+00 100.0%</td>
<td>5.50e+00 100.0%</td>
<td>7.10e+03 100.0%</td>
</tr>
<tr>
<td>loop at miniqmc.cpp: 402</td>
<td>6.02e+00 96.6%</td>
<td>6.33e+00 96.6%</td>
<td>5.33e+00 96.6%</td>
<td>5.33e+00 96.6%</td>
<td>7.10e+03 100.0%</td>
</tr>
<tr>
<td>404 = _<em>omp_outlined</em>.62</td>
<td>6.02e+00 96.6%</td>
<td>6.33e+00 96.6%</td>
<td>5.33e+00 96.6%</td>
<td>5.33e+00 96.6%</td>
<td>7.10e+03 100.0%</td>
</tr>
<tr>
<td>404 = [1].omp_outlined_.61</td>
<td>6.02e+00 96.6%</td>
<td>6.33e+00 96.6%</td>
<td>5.33e+00 96.6%</td>
<td>5.33e+00 96.6%</td>
<td>7.10e+03 100.0%</td>
</tr>
<tr>
<td>loop at miniqmc.cpp: 485</td>
<td>5.01e+00 80.5%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
</tr>
<tr>
<td>loop at miniqmc.cpp: 472</td>
<td>5.01e+00 80.5%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
</tr>
<tr>
<td>loop at miniqmc.cpp: 478</td>
<td>5.01e+00 80.5%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
</tr>
<tr>
<td>485 = qmplus::Wavefunction::ratioq...</td>
<td>5.01e+00 80.5%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
</tr>
<tr>
<td>485 = qmplus::DiracDeterminant::qmplus...</td>
<td>5.01e+00 80.5%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
</tr>
<tr>
<td>108 = qmplus::einspline_spo_omp...</td>
<td>5.01e+00 80.5%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
</tr>
<tr>
<td>107 = qmplus::einspline_spo_omp...</td>
<td>5.01e+00 80.5%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
</tr>
<tr>
<td>loop at einspline_spo_omp.cpp: 157</td>
<td>5.01e+00 80.5%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
<td>4.90e+00 81.6%</td>
</tr>
<tr>
<td>162 = __omp tgt kernel&gt;</td>
<td>4.90e+00 72.0%</td>
<td>4.90e+00 72.0%</td>
<td>4.90e+00 72.0%</td>
<td>4.90e+00 72.0%</td>
<td>4.90e+00 72.0%</td>
</tr>
<tr>
<td>&lt;gpu kernel&gt;</td>
<td>4.90e+00 72.0%</td>
<td>4.90e+00 72.0%</td>
<td>4.90e+00 72.0%</td>
<td>4.90e+00 72.0%</td>
<td>4.90e+00 72.0%</td>
</tr>
</tbody>
</table>

Source: extremecomputingtraining.anl.gov
LAMMPS on Frontier: 8K nodes, 64K MPI ranks + GPU times
HPCToolkit Status on GPUs

NVIDIA
- heterogeneous profiles
- GPU instruction-level execution and stalls using PC sampling traces

AMD
- heterogeneous profiles
- no GPU instruction-level measurements within kernels
- measure OpenMP offloading using OMPT interface traces

Intel
- heterogeneous profiles
- GPU instruction-level measurements with instrumentation; heuristic latency attribution to instructions
- measure OpenMP offloading using OMPT interface traces
Ongoing Work

Enhancing measurement to identify root causes of scalability losses
  Better measurement of delays caused by GPU and communication
Improving the scalability of hpcprof-mpi
  Avoid unnecessary serialization of I/O
Adding a Python-based interface for analysis of performance results
  Python API supports arbitrary queries and analysis of profiles and traces
  Automatic analysis to identify notable features in executions
    e.g. load imbalance, trace line equivalence classes
HPCToolkit Resources

Documentation
  User manual
  Tutorial videos
    http://hpctoolkit.org/training.html

Software
  Download hpcviewer GUI binaries for your laptop, desktop, cluster, or supercomputer
    OS: Linux, Windows, MacOS
    Processors: x86_64, aarch64, ppc64le
    http://hpctoolkit.org/download.html
  Install HPCToolkit on your Linux desktop, cluster, or supercomputer using Spack
    http://hpctoolkit.org/software-instructions.html
HPCToolkit Hands-On Directions

Performance analysis of CPU and GPU-accelerated applications at Scale

John Mellor-Crummey
Professor, Rice University

extremecomputingtraining.anl.gov
Sample Performance Databases for You to Analyze on Polaris

• Setup on polaris
  • module use /soft/perftools/hpctoolkit/polaris/modulefiles
  • module load hpctoolkit/default
• Data on theta and polaris: /grand/ATPESC2023/track-6-tools-hpctoolkit/data
  • CPU
    • QMCPACK - quantum Monte Carlo electronic structure calculations (early experiment)
  • GPU-accelerated
    • GAMESS - ab initio quantum chemistry
      • 1.singlegroup-unbalanced
      • 2.singlegroup-balanced
      • 3.multigroup-unbalanced-mtarbr
      • 4.multigroup-balanced
      • 5.multigroup-unbalanced-pc
      • 6.scale
    • PeleC - AMR Solver (AMREx) for compressible reacting flows
    • Pytorch-deepwave - GPU-accelerated reverse-time migration using Pytorch
    • Quicksilver - proxy application for dynamic Monte Carlo transport
GPU: Profiling Quicksilver with HPCToolkit on Polaris or Perlmutter

- git clone https://github.com/hpctoolkit/hpctoolkit-tutorial-examples
- cd hpctoolkit-tutorial-examples/examples/gpu/quicksilver
- polaris:
  - export HPCTOOLKIT_TUTORIAL_PROJECTID=ATPESC2023
  - export HPCTOOLKIT_TUTORIAL_RESERVATION=default
  - source setup-env/polaris.sh
- perlmutter:
  - export HPCTOOLKIT_TUTORIAL_PROJECTID=ntrain5_g
  - export HPCTOOLKIT_TUTORIAL_RESERVATION=default
  - source setup-env/perlmutter.sh
- make build
- make run
- make run-pc
- make view
- make view-pc
CPU: Profiling AMG2013 with HPCToolkit on Theta

- git clone https://github.com/hpctoolkit/hpctoolkit-tutorial-examples
- cd hpctoolkit-tutorial-examples/examples/cpu/mpi+openmp/amg2013
- export HPCTOOLKIT_TUTORIAL_PROJECTID=ATPESC2023
- export HPCTOOLKIT_TUTORIAL_RESERVATION=debug-cache-quad
- source setup-env/theta.sh
- make build
- make run
  - # wait for $COBALT_JOBID.done to appear in your directory
- make analyze
- Alternatives
  - make view
  - hpcviewer hpctoolkit-amg2013.d