HPCToolkit: Performance Analysis of GPU-accelerated Kokkos Applications on NVIDIA GPUs

John Mellor-Crummey
Rice University

April 26, 2024
Current Funding for HPCToolkit

- Government
  - Lawrence Livermore National Laboratory Subcontract B658833
  - DOE Software Tools Ecosystem Project - UT-Battelle Subcontract CW54422
  - Argonne National Laboratory Subcontract 4F-60094
- Corporate
  - Advanced Micro Devices
  - TotalEnergies EP Research & Technology USA, LLC.
A Hands-on Example for the Tutorial: ArborX

A library written in Kokkos that provides performance portable algorithms for geometric search

% git clone https://github.com/hpctoolkit/hpctoolkit-tutorial-examples
% cd hpctoolkit-tutorial-examples/examples/gpu/arborx
% source setup/perlmutter.sh
% make all # downloads, builds, measures, and analyzes two executions
% make view
% make view-pc

Note: precomputed databases available on Perlmutter at /global/cfs/cdirs/m3977/data/arborx
Outline

• Introduction to HPCToolkit performance tools
  – Overview of HPCToolkit components and their workflow
  – HPCToolkit's graphical user interfaces
• Analyzing the performance of GPU-accelerated codes with HPCToolkit
  – GAMESS
  – ArborX
  – LAMMPS at Exascale
• Coming attractions
• Troubleshooting
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
- Measures complex programs on a broad range of platforms
  - CPU: x86_64, Power, ARM
  - GPU: NVIDIA, AMD, Intel
How does HPCToolkit Differ from NVIDIA’s Tools?

- NVIDIA NSight Systems
  - tracing of CPU and GPU streams
  - analyze traces when you open them with the GUI
    - long running traces are huge and thus extremely slow to analyze, limiting scalability
  - designed for measurement and analysis within a node

- NVIDIA NSight Compute
  - detailed measurement of kernels with counters and execution replay
  - very slow measurement
  - flat display of measurements within GPU kernels

- HPCToolkit
  - supports more scalable tracing than Nsight Systems
    - measure exascale executions across many GPUs and nodes
  - scalable, parallel post-mortem analysis vs. non-scalable in-GUI analysis
  - detailed reconstruction of estimates for calling context profiles within GPU kernels
HPCToolkit’s Workflow for CPU Applications

- **Source Files** → **Optimized Binary**
  - Compile & Link

- **hpcrun**
  - Profile execution on CPUs
  - Profile Files

- **hpcstruct**
  - Analyze CPU program structure
  - Trace Files
  - Program Structure

- **hpcviewer**
  - Present trace view and profile view

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

1. Source Files → Compile & Link → Optimized Binary
2. hpcrun:
   - Profile execution on CPUs and GPUs
3. GPU Binary
4. hpcstruct:
   - Analyze CPU/GPU program structure
5. hpcviewer:
   - Present trace view and profile view
6. Database
7. hpcprof/hpcprof mpi:
   - Interpret profile
   - Correlate with source
8. Profile Files
9. Trace Files
10. Program Structure
HPCToolkit’s Workflow for GPU-accelerated Applications

Step 1:
- Ensure that compilers record line mappings
- host compiler: -g
- nvcc: -lineinfo
HPCToolkit’s Workflow for GPU-accelerated Applications

Step 2:
- `hpcrun` collects call path profiles (and optionally, traces) of events of interest
Measurement of CPU and GPU-accelerated Applications

- Sampling using Linux timers and hardware counter overflows on the CPU
- Callbacks when GPU operations are launched and (sometimes) completed
- Event stream for GPU operations; PC Samples (NVIDIA)
- Binary instrumentation of GPU kernels on Intel GPUs for fine-grain measurement
Call Stack Unwinding to Attribute Costs in Context

- Unwind when timer or hardware counter overflows
  - measurement overhead proportional to sampling frequency rather than call frequency
- Unwind to capture context for events such as GPU kernel launches

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>` ....  \[xxx \in \{\text{nvidia,amd,opencl,level0}\}\]

- 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
  - `srun -n 1 -G 1 hpcrun -e gpu=xxx <app>`
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 for interpreting PC samples on NVIDIA GPUs
HPCToolkit’s Workflow for GPU-accelerated Applications

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 multithreading
  
  hpcprof <measurement-directory>

- Analyze data from large executions with distributed-memory parallelism + multithreading
  
  srun -N 2 -n 2 -c 126 hpcprof-mpi <measurement-directory>
HPCToolkit’s Workflow for GPU-accelerated Applications

Step 4:
- `hpcviewer` - interactively explore profile and traces for GPU-accelerated applications
Code-centric Analysis with hpcviewer

- 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
  - function calls in full context
  - inlined procedures
  - inlined templates
  - outlined OpenMP loops
  - loops

- source pane
- view control
- metric display
- navigation pane
- metric pane
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 hpctviewer

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

Call stack pane shows full calling context for the cursor

Depth view showing the history of calling contexts for the thread/GPU stream with the cursor

Minimap indicates part of execution trace shown

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

MPI ranks, OpenMP Threads, GPU streams
Summary of ECP Developments

• Measurement
  • profile and trace GPU-accelerated applications on AMD, Intel, and NVIDIA GPUs
  • source-level measurement of Python frameworks, e.g. Pytorch
  • record measurement data in sparse formats: benefits GPU-accelerated programs with many metrics
  • implement of OMPT performance tools interface in AMD OpenMP and LLVM
• Binary analysis
  • binary analysis of AMD, Intel, NVIDIA GPU binaries
  • parallel analysis of application binaries to speed recovery of program structure
• Performance analysis and attribution
  • MPI + OpenMP highly parallel analysis of measurement data at exascale
  • sparse representations observed to reduce performance analysis results by > 1000x
  • detailed attribution of PC samples to rich calling contexts within GPU kernels
• Presentation
  • interactive display profiles and terabytes of traces from exascale executions
hpcstruct Example: Analyze 7.7GB TensorFlow library (170MB text) in 77s
Analyze 38.1GB data for 2K MPI ranks + 2K GPUs using 1K threads in 41s
Case Studies

- GAMESS (OpenMP)
- ArborX (Kokkos)
- LAMMPS (Kokkos) at exascale
Case Study: GAMESS

- General Atomic and Molecular Electronic Structure System (GAMESS)
  - general *ab initio* quantum chemistry package
- Calculates the energies, structures, and properties of a wide range of chemical systems

- Experiments
  - GPU-accelerated nodes at a prior Perlmutter hackathon
    - Single node with 4 GPUs
    - Five nodes with 20 GPUs

Perlmutter node at a glance
AMD Milan CPU
4 NVIDIA A100 GPUs
256 GB memory
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter
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

GAMESS original

All GPU streams, whole execution
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter

GPU load imbalance due to triangular iteration spaces

GAMESS original

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

GAMESS improved

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

GAMESS improved
All GPU streams, whole execution
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter

- Improved GPU load balance
- GAMESS improved
- All GPU streams: 2 iterations
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter

GAMESS improved
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
```
Case Study: ArborX

• A library written in Kokkos that provides performance portable algorithms for geometric search
ArborX Trace: Lots of irrelevant CPU Trace Lines for Idle Threads

- Solution: Filter out trace lines with very small numbers of samples
ArborX Trace: Filter to Focus on Relevant CPU and GPU Traces

- Use Filter→Filter Ranks: select Rank 0 and GPU trace lines
ArborX Trace: PC sampling of ArborX
Key Metrics Available for GPU Kernels

- **GPUOP**: GPU operation time (kernel launch, copies, etc.)
- **GXCOPY**: GPU copies of various kinds
- **GKER**: GPU kernel time
- **GKER:FGP_ACT**: fine grain parallelism actual (number of threads used)
- **GKER:FGP_MAX**: maximum possible fine-grain parallelism (number of threads possible)
- **GKER:BLK_THR**: threads per block
- **GKER:BLK_SM**: block shared memory
- **GKER:OCC_THR**: theoretical thread occupancy
What Metrics Are Available for GPU Kernels with PC Sample

- GINS: GPU instructions
- GINS:STL_ANY: GPU instruction stalls for any reason
- GINS:STL_IFET: GPU instruction stalls for instruction fetch
- GINS:STL_GMEM: GPU instruction stalls for global memory
- GINS:STL_CMEM: GPU instruction stalls for constant memory
- GINS:STL_IDEP: GPU instruction stalls for instruction dependences
- GINS:STL_PIPE: GPU instruction pipeline stalls
- GINS:STL_MTHR: GPU instruction stalls for memory throttling

- GSAMP:EXP: expected number of samples
- GSAMP:TOT: total number of samples recorded
- GSAMP:UTIL: GPU utilization = (PC samples expected) / (PC samples total)
LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds
LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds
LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds
LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds
LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds
LAMMPS on Frontier: 8K nodes, 64K MPI ranks + 64K GPU tiles

Kernel duration of microseconds
LAMMPS on Frontier: 8K nodes, 64K MPI ranks + GPU times

Kernel duration of microseconds
Coming Attractions

- Developing comprehensive support for NVTX/ROCTX/Caliper/Kokkos Labels
- Support for instruction-level measurement and attribution on AMD and Intel GPUs
- New GUI support for analysis of remote data
- Python-based interface for analysis of performance results
Troubleshooting: Only GPU kernel Name

- Need to measure with PC sampling to measure within GPU kernels
Troubleshooting: No GPU source code lines with PC sampling

- If you don’t see source code with PC sampling on NVIDIA GPUs: compile with “-lineinfo” option
Troubleshooting: Compiling ArborX with GPU Line Map Info

- ArborX cmake isn't set up to include GPU line mappings
- Force the compiler to record GPU line mappings

```bash
% cmake -DARBORX_ENABLE_EXAMPLES=true \
  -DCMAKE_INSTALL_PREFIX=`pwd`/../install \
  -DCMAKE_CXX_COMPILER=g++ \
  -DCMAKE_BUILD_TYPE=RelWithDebInfo \
  -DCMAKE_CXX_FLAGS_RELWITHDEBINFO="-O2 -g -DNDEBUG -lineinfo"
```
HPCToolkit Resources

- **Documentation**
  - User manual
  - Tutorial videos
    - [http://hpctoolkit.org/training.html](http://hpctoolkit.org/training.html)
    - Recorded demo of GPU analysis of Quicksilver: [https://youtu.be/vixa3hGDUgG](https://youtu.be/vixa3hGDUgG)
    - Recorded tutorial presentation including demo with GPU analysis of GAMESS: [https://vimeo.com/781264043](https://vimeo.com/781264043)
  - Cheat sheet
    - [https://gitlab.com/hpctoolkit/hpctoolkit/-/wikis/home](https://gitlab.com/hpctoolkit/hpctoolkit/-/wikis/home)

- **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](http://hpctoolkit.org/download.html)
  - Install HPCToolkit on your Linux desktop, cluster, or supercomputer using Spack
    - [http://hpctoolkit.org/software-instructions.html](http://hpctoolkit.org/software-instructions.html)