Performance Analysis of GPU-accelerated OpenMP Applications using HPCToolkit

John Mellor-Crummey
Rice University

August 25, 2023

OpenMP Users Monthly Telecon
Outline

• Introduction to HPCToolkit
  – Overview of HPCToolkit components and their workflow
  – HPCToolkit's graphical user interfaces
• Analyzing the performance of GPU-accelerated codes with HPCToolkit
  – GAMESS
  – GEM
• Status
• Ongoing work
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

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

2. **hpcrun**
   - Profile execution on CPUs

3. **hpcstruct**
   - Analyze CPU program structure

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

5. **hpcprof/hpcprof-mpi**
   - Interpret profile
   - Correlate w/ source

6. **Profile Files**
7. **Trace Files**
8. **Program Structure**

---

HPCToolkit is a suite of tools designed for performance analysis and tuning of CPU applications in high-performance computing environments.
HPCToolkit’s Workflow for GPU-accelerated Applications

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

2. **hpcrun**
   - Profile execution on CPUs and GPUs

3. **GPU Binary**
   - **hpcstruct**
     - Analyze CPU/GPU program structure

4. **Program Structure**
   - **Trace Files**
   - **Profile Files**

5. **Database**
   - **hpcviewer**
     - Present trace view and profile view
   - **hpcprof/hpcprof-mpi**
     - Interpret profile
     - Correlate w/ source
HPCToolkit’s Workflow for GPU-accelerated Applications

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

Step 2:
- `hpcrun` collects call path profiles (and optionally, traces) of events of interest
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

![Calling context tree](image)

- return address
- return address
- return address
- instruction pointer
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**
  - `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>`
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:
- \texttt{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
  
  ```
  jsrun -n 2 -a 1 -c 22 -b packed hpcprof-mpi <measurement-directory>
  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

- 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.

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.
hpcstruct Example: Analyze 7.7GB TensorFlow library (170MB text) in 77s
Case Studies

- GAMESS - an ab initio quantum chemistry package: Fortran + MPI + OpenMP offloading
- GEM - a gyrokinetic turbulence code that simulates both ions and electrons
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 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
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter
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

![Graph and table data related to GAMESS performance on Perlmutter.]

**GAMESS improved**
Time-centric Analysis: GAMESS improved

GAMESS improved

CPU Threads and GPU Streams
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter

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

GAMESS improved with PC Sampling
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter

<table>
<thead>
<tr>
<th>Scope</th>
<th>CGNE (Sec)</th>
<th>Sum (E)</th>
<th>CGNE (STL)</th>
<th>Sum (E)</th>
<th>CGNE (STL, ABP)</th>
<th>Sum (E)</th>
<th>CGNE (STL, ABP, E)</th>
<th>Sum (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop at gnu_dircf_tdhf.pro: 136</td>
<td>1.16e-13</td>
<td>3.7e-13</td>
<td>1.13e-13</td>
<td>3.5e-13</td>
<td>5.3e-13</td>
<td>3.3e-13</td>
<td>4.4e-10</td>
<td>0.01</td>
</tr>
<tr>
<td>gnu_dircf_tdhf.pro: 137</td>
<td>1.05e-13</td>
<td>2.08e-13</td>
<td>1.02e-13</td>
<td>2.08e-13</td>
<td>5.2e-13</td>
<td>3.2e-13</td>
<td>4.2e-10</td>
<td>0.01</td>
</tr>
<tr>
<td>gnu_dircf_tdhf.pro: 141</td>
<td>9.66e-12</td>
<td>2.66e-12</td>
<td>9.35e-12</td>
<td>2.7e-12</td>
<td>9.0e-12</td>
<td>3.2e-12</td>
<td>9.0e-12</td>
<td>3.2e-12</td>
</tr>
</tbody>
</table>
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

```c
136 !$omp atomic update
FA(IJ,IXYZ) = FA(IJ,IXYZ) + VAL4*DA(KL,IXYZ)
138 !$omp atomic update
FA(KL,IXYZ) = FA(KL,IXYZ) + VAL4*DA(IJ,IXYZ)
140 !$omp atomic update
FA(IK,IXYZ) = FA(IK,IXYZ) - VAL1*DA(JL,IXYZ)
142 !$omp atomic update
FA(JL,IXYZ) = FA(JL,IXYZ) - VAL1*DA(IK,IXYZ)
144 !$omp atomic update
FA(IL,IXYZ) = FA(IL,IXYZ) - VAL1*DA(JK,IXYZ)
146 !$omp atomic update
FA(JK,IXYZ) = FA(JK,IXYZ) - VAL1*DA(IL,IXYZ)
ENDDO
```
Case Study: GEM (Gyrokinetic Turbulence Code)

- GEM: a comprehensive electromagnetic delta-f particle-in-cell code that includes the full dynamics of gyrokinetic ions and drift-kinetic electrons
  - Developed by University of Colorado at Boulder, part of ECP WDMApp project
- Code is written in Fortran 90 + MPI + OpenACC, with ongoing porting efforts to OpenMP target offload (https://dl.acm.org/doi/abs/10.1007/978-3-030-97759-7_7)
- Tested platforms: Perlmutter, Crusher, and Frontier using Cray compiler
  - Frontier: 16 nodes, 8 MPI ranks per node, 4 OpenMP threads per rank, 1 GPU per rank, 2 GPU streams per GPU device

<table>
<thead>
<tr>
<th>Frontier</th>
<th>Wall-clock Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without GPU offloading</td>
<td>290.88s</td>
<td>1 (base)</td>
</tr>
<tr>
<td>Naive GPU offloading</td>
<td>41.80s</td>
<td>6.96</td>
</tr>
<tr>
<td>Optimized GPU offloading</td>
<td>39.52s</td>
<td>7.36</td>
</tr>
</tbody>
</table>
First attempt: not all parallel loops should be offloaded

Too much data movement between CPU & GPU
First attempt: not all parallel loops should be offloaded

Procedures test_init_pmove and test_pmove have high data movement compared to GPU computation.
Use CPU threads to reduce GPU idleness

Most OpenMP threads are idle

10.6% of GPU idle occurs when the main CPU thread executes `fltm_` procedure.

Parallelizing this procedure should reduce GPU idleness.
Final step: parallelizing `f1tm_` procedure to reduce GPU idleness
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
  - hardware counters to measure kernels
  - 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
  - identify measurement of delays caused by GPU and communication
- Developing comprehensive support for NVTX/ROCTX/Caliper/Kokkos Labels
- Support for instruction-level measurement and attribution on AMD and Intel GPUs
- Improving the scalability of hpcprof-mpi
  - avoid unnecessary serialization of I/O
- Developing new GUI support for analysis of remote data
- Adding a Python-based interface for analysis of performance results
  - developing a Python API to support arbitrary queries and analysis of profiles and traces
  - developing a tool that presents high-level performance reports
  - exploring automated analysis to identify notable features in executions
    - e.g. load imbalance, trace line equivalence classes