TOOLS FOR GPU COMPUTING
With focus on NVIDIA GPUs

03.12.2019 | MICHAEL KNOBLOCH
MOTIVATION

Make it work, make it right, make it fast.

Kent Beck

Debugger:
- CUDA-MEMCHECK
- CUDA-GDB
- TotalView
- DDT

Performance Tools:
- NVIDIA Visual Profiler
- NVIDIA Nsight System
- NVIDIA Nsight Compute
- Score-P
- Vampir
- Performance Reports
- TAU
- HPCToolkit
GPU PROGRAMMING MODELS
CURRENT STATE OF THE MESS
TRADITIONAL HPC

• Inter-node:
  • MPI

• Intra-node:
  • OpenMP
  • Pthreads

• C/C++ and Fortran
MODERN HPC

• Inter-node:
  • MPI
  • PGAS (SHMEM, GASPI, …)

• Intra-node:
  • OpenMP
  • Pthreads
  • Tasking, C++11 threads, TBB, …

• C/C++, Fortran and Python
GPU PROGRAMMING

• **Low-level:**
  • CUDA (NVIDIA), ROCm (AMD)
  • OpenCL

• **Pragma-based:**
  • OpenACC
  • OpenMP target

• On top: SYCL, oneAPI, HIP, KOKKOS, …
MAKE IT WORK
DEVELOPMENT OF GPU APPS
IDE
Integrated Development Environment

- Integrates Editor, Build system, Debugger, and Profiler
- NVIDIA Nsight (Linux: Eclipse, Windows: Visual Studio)
- Nsight Code Editor
  - CUDA aware code completion and inline help
  - CUDA code highlighting
  - CUDA aware refactoring
PORTING

• Several tools exist helping expose parallelism
  • Example: Appentra Parallelware Trainer
    • Identifies parallelizable sections in sequential applications
    • Supports OpenMP and OpenACC
    • Supports versioning of changes
    • Start program directly from GUI
MAKE IT RIGHT
DEBUGGER AND MEMORY ANALYZER
DEBUGGING – A PAINFUL PROCESS

ROOT CAUSE

WHAT ARE YOU DOING?
DEBUGGING

GOTCHA!

REAL PROBLEM
OH SHIT

JÜLICH
Forschungszentrum

Mitglied der Helmholtz-Gemeinschaft

STARECAT.COM

MONKEYUSER.COM
## TOOL COMPATIBILITY MATRIX

<table>
<thead>
<tr>
<th>Tool</th>
<th>CUDA</th>
<th>OpenACC</th>
<th>OMPD</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA-MEMCHECK</td>
<td>✓</td>
<td>(✗)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CUDA-GDB</td>
<td>✓</td>
<td>(✗)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>TotalView</td>
<td>✓</td>
<td>✓</td>
<td>(✗)</td>
<td>✗</td>
</tr>
<tr>
<td>DDT</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
CUDA-MEMCHECK

- Valgrind for GPUs
- Monitors hundreds of thousands of threads running concurrently on each GPU
- Reports detailed information about global, local, and shared memory access errors (e.g. out-of-bounds, misaligned memory accesses)
- Reports runtime executions errors (e.g. stack overflows, illegal instructions)
- Reports detailed information about potential race conditions
- Displays stack back traces on host and device for errors
- And much more
- Included in the CUDA Toolkit
CUDA-GDB

- Extension to gdb
- CLI and GUI (Nsight)
- Simultaneously debug on the CPU and multiple GPUs
- Use conditional breakpoints or break automatically on every kernel launch
- Can examine variables, read/write memory and registers and inspect the GPU state when the application is suspended
- Identify memory access violations
  - Run CUDA-MEMCHECK in integrated mode to detect precise exceptions.
• UNIX Symbolic Debugger
  for C/C++, Fortran, Python, PGI HPF, assembler programs
• JSC’s “standard” debugger
• Special, non-traditional features
  • Multi-process and multi-threaded
  • Multi-dimensional array data visualization
  • Support for parallel debugging (MPI: automatic attach, message queues, OpenMP, Pthreads)
• Scripting and batch debugging
• Advanced memory debugging
• CUDA and OpenACC support

• http://www.roguewave.com
• **NOTE:** JSC license limited to 2048 processes (shared between all users)
DDT UNIX Graphical Debugger for C/C++, Fortran, and Python programs

- Modern, easy-to-use debugger

- Special, non-traditional features
  - Multi-process and multi-threaded
  - Multi-dimensional array data visualization
  - Support for MPI parallel debugging (automatic attach, message queues)
  - Support for OpenMP (Version 2.x and later)
  - Support for CUDA and OpenACC
  - Job submission from within debugger

- https://developer.arm.com

- **NOTE:** JSC license limited to 64 processes (shared between all users)
DDT: MAIN WINDOW

- **Process controls**
- **CUDA Thread control**
- **Source code**
- **CUDA Thread stepping**
- **Variables**
- **Stack trace**
- **GPU Device information**
- **Expression evaluator**
### MAKE IT FAST

**PERFORMANCE ANALYSIS TOOLS**

---

**Example Table:**

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Execution</td>
<td>1062.70 sec</td>
</tr>
<tr>
<td>Overhead</td>
<td>0.00 sec</td>
</tr>
<tr>
<td>Total MBytes</td>
<td>5.88e4</td>
</tr>
<tr>
<td>Bytes Transferred</td>
<td>2.59e6</td>
</tr>
<tr>
<td>MPI File Operations</td>
<td>0.00</td>
</tr>
<tr>
<td>Computational Imbalance</td>
<td>31.52 sec</td>
</tr>
<tr>
<td>Minimum Inclusive Time</td>
<td>0.00 sec</td>
</tr>
<tr>
<td>Maximum Inclusive Time</td>
<td>251.21 sec</td>
</tr>
<tr>
<td>Allocation Size</td>
<td>0.00 Bytes</td>
</tr>
<tr>
<td>Deallocation Size</td>
<td>0.00 Bytes</td>
</tr>
<tr>
<td>Bytes Leaked</td>
<td>0.00 Bytes</td>
</tr>
<tr>
<td>Maximum Heap Memory Allocated</td>
<td>1.74e12 Bytes</td>
</tr>
<tr>
<td>PAPI L2 L2_INS (#)</td>
<td>5.15e11</td>
</tr>
<tr>
<td>PAPI L2_TCM (#)</td>
<td>2.36e10</td>
</tr>
</tbody>
</table>

---

**Example Diagram:**

- Network diagram showing connections and data flow.
- Node n03588 with tasks and processes.
- Task breakdown: add sources, compute.psi0_corrDisp, averageFields, push, solve_fem, ksetc, update_pict_and_work2, update_work3, bpart, MPI_Barrier, diagnose, allreduce, parmove, collisions, quadtree_smoothing, epilogue, MPI_Init, MPI_Barrier, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank.

---

**Jülich Forschungszentrum**

Mitglied der Helmholtz-Gemeinschaft
# TOOL COMPATIBILITY MATRIX

<table>
<thead>
<tr>
<th>Tool</th>
<th>CUDA</th>
<th>OpenACC</th>
<th>OMPT</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score-P</td>
<td>✓</td>
<td>✓</td>
<td>(x)</td>
<td>✓</td>
</tr>
<tr>
<td>NVIDIA Tools</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Perf. Reports</td>
<td>✓</td>
<td>(x)</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>TAU</td>
<td>✓</td>
<td>✓</td>
<td>(x)</td>
<td>✓</td>
</tr>
<tr>
<td>HPC Toolkit</td>
<td>✓</td>
<td>x</td>
<td>(x)</td>
<td>x</td>
</tr>
</tbody>
</table>
ARM PERFORMANCE REPORTS

- Single page report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows CPU, GPU, memory, network and I/O utilization
- Supports MPI, multi-threading and accelerators
- Saves data in HTML, CVS or text form

- [link](https://www.arm.com/products/development-tools/server-and-hpc/performance-reports)
- **Note:** JSC license limited to 512 processes (with unlimited number of threads)
EXAMPLE PERFORMANCE REPORTS

Summary: cp2k.popt is **CPU-bound** in this configuration

The total wallclock time was spent as follows:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>56.5%</td>
</tr>
<tr>
<td>MPI</td>
<td>43.5%</td>
</tr>
<tr>
<td>I/O</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

This application run was CPU-bound. A breakdown of this time and advice for investigating further is in the CPU section below.

**CPU**

A breakdown of how the 56.5% total CPU time was spent:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar numeric ops</td>
<td>27.7%</td>
</tr>
<tr>
<td>Vector numeric ops</td>
<td>11.3%</td>
</tr>
<tr>
<td>Memory accesses</td>
<td>60.9%</td>
</tr>
<tr>
<td>Other</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance. Little time is spent in **vectorized instructions**. Check the compiler’s vectorization advice to see why key loops could not be vectorized.

**MPI**

Of the 43.5% total time spent in MPI calls:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in collective calls</td>
<td>8.2%</td>
</tr>
<tr>
<td>Time in point-to-point calls</td>
<td>91.8%</td>
</tr>
<tr>
<td>Estimated collective rate</td>
<td>169 Mb/s</td>
</tr>
<tr>
<td>Estimated point-to-point rate</td>
<td>50.6 Mb/s</td>
</tr>
</tbody>
</table>

The point-to-point transfer rate is low. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait. Use an MPI profiler to identify the problematic calls and ranks.

**I/O**

A breakdown of how the 0.0% total I/O time was spent:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in reads</td>
<td>0.0%</td>
</tr>
<tr>
<td>Time in writes</td>
<td>0.0%</td>
</tr>
<tr>
<td>Estimated read rate</td>
<td>0 bytes/s</td>
</tr>
<tr>
<td>Estimated write rate</td>
<td>0 bytes/s</td>
</tr>
</tbody>
</table>

No time is spent in I/O operations. There's nothing to optimize here!

**Memory**

Per-process memory usage may also affect scaling:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean process memory usage</td>
<td>82.5 Mb</td>
</tr>
<tr>
<td>Peak process memory usage</td>
<td>89.3 Mb</td>
</tr>
<tr>
<td>Peak node memory usage</td>
<td>7.4%</td>
</tr>
</tbody>
</table>

The peak node memory usage is low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.
Accelerators

A breakdown of how accelerators were used:

- GPU utilization: 47.8%
- Global memory accesses: 1.6%
- Mean GPU memory usage: 0.8%
- Peak GPU memory usage: 0.8%

**GPU utilization** is low; identify CPU bottlenecks with a profiler and offload them to the accelerator.

The **peak GPU memory usage** is low. It may be more efficient to offload a larger portion of the dataset to each device.
NVIDIA VISUAL PROFILER

• Part of the CUDA Toolkit
• Supports all CUDA enabled GPUs
• Supports CUDA and OpenACC on Windows, OS X and Linux

• Unified CPU and GPU Timeline
• CUDA API trace
  • Memory transfers, kernel launches, and other API functions
• Automated performance analysis
  • Identify performance bottlenecks and get optimization suggestions
• Guided Application Analysis
• Power, thermal, and clock profiling
NVIDIA VISUAL PROFILER: EXAMPLE

Timeline view

Detailed information on Kernel execution

Automatic analysis of performance bottlenecks
NVIDIA NSIGHT SYSTEMS

- System wide performance analysis tool
- High-level, low overhead
- Similar functionality as NVVP
  - No automated/guided analysis
  - Can launch Nsight Compute for in-depth kernel analysis
- CLI and GUI
NVIDIA NSIGHT COMPUTE

- Interactive kernel profiler
- Detailed performance metrics
- Guided analysis
- Baseline feature to compare versions
- Customizable and data-driven UI
- Supports analysis scripts for post-processing results
- CLI and GUI
SCORE-P

- Community instrumentation and measurement infrastructure
- Developed by a consortium of performance tool groups
  - Scalasca 2.x
  - Vampir
  - TAU
  - Periscope
- Next generation measurement system of
- Common data formats improve tool interoperability
- http://www.score-p.org
SCORE-P OVERVIEW
SCORE-P GPU MEASUREMENTS

- **OpenACC**
  - Prefix compiler and linker command with scorep --openacc
  - `export ACC_PROFLIB=${SCOREP_ROOT}/lib/libscorep_adapter_openacc_event.so`
  - `export SCOREP_OPENACC_ENABLE=yes`
  - yes refers to: regions, wait, enqueue
  - Full list of options in User Guide

- **CUDA**
  - Prefix compiler and linker command with scorep --cuda
  - `export SCOREP_CUDA_ENABLE=yes`
  - yes refers to: runtime, kernel, memcpy
  - Full list of options in User Guide

- **OpenCL similar** (use `SCOREP_OPENCL_ENABLE=yes`)
CUBE OVERVIEW

What kind of performance metric?

Where is it in the source code? In what context?

How is it distributed across the processes/threads?
EXAMPLE: OPENACC

Pure OpenACC measurements give host-side events only
Enabling CUDA also shows kernels on the GPU
VAMPIR EVENT TRACE VISUALIZER

- **Offline** trace visualization for Score-P’s OTF2 trace files
- **Visualization of MPI, OpenMP, GPU and application events:**
  - All diagrams highly customizable (through context menus)
  - Large variety of displays for **ANY** part of the trace
- http://www.vampir.eu

- **Advantage:**
  - Detailed view of dynamic application behavior
- **Disadvantage:**
  - Requires event traces (huge amount of data)
  - Completely manual analysis
VAMPIR COMPLEX APPLICATION
REMARK: NO SINGLE SOLUTION IS SUFFICIENT!

- A combination of different methods, tools and techniques is typically needed!
  - Analysis
    - Statistics, visualization, automatic analysis, data mining, ...
  - Measurement
    - Sampling / instrumentation, profiling / tracing, ...
  - Instrumentation
    - Source code / binary, manual / automatic, ...

Mitglied der Helmholtz-Gemeinschaft
WHAT NOW?

- The tools are there – what now?

- Development phase:
  - Use NVDIA tools
  - Debug: CUDA-MEMCHECK/CUDA-GDB
  - Performance: Nsight Systems and Compute

- Scaling up:
  - Use 3rd-party tools
  - Debug: TotalView/DDT
  - Performance: Score-P, Vampir
NEED HELP?

- Talk to the experts
  - NVIDIA Application Lab
  - CST Performance Analysis
  - CST Application Optimization
  - Apply for a POP audit

Successful performance engineering often is a collaborative effort
QUESTIONS