Performance Tools for GPU-Powered Scalable Heterogeneous Systems

Allen D. Malony, Scott Biersdorff, Sameer Shende
{malony,scott,sameer}@cs.uoregon.edu

University of Oregon
Outline

- Heterogeneous performance methods and tools
  - Measurement approaches
  - Implementation mechanisms
  - Tools with GPU measurement support
  - Examples

- Applications case studies
  - GTC
  - NAMD

- New features in TAU 2.21.2 for heterogeneous performance measurement

- LiveDVD!!!
Heterogeneous Parallel Systems and Performance

- Heterogeneous parallel systems are highly relevant today
  - Multi-CPU, multicore shared memory nodes
  - Manycore (throughput) accelerators with high-BW I/O
  - Cluster interconnection network

- Performance is the main driving concern
  - Heterogeneity is an important (the?) path to extreme scale

- Heterogeneous software technology to get performance
  - More sophisticated parallel programming environments
  - Integrated parallel performance tools
    - support heterogeneous performance model and perspectives
Implications of Heterogeneity for Performance Tools

- Current status quo is somewhat comfortable
  - Mostly homogeneous parallel systems and software
  - Shared-memory multithreading – OpenMP
  - Distributed-memory message passing – MPI

- Parallel computational models are relatively stable (simple)
  - Corresponding performance models are relatively tractable
  - Parallel performance tools can keep up and evolve

- Heterogeneity creates richer computational potential
  - Results in greater performance diversity and complexity

- Heterogeneous systems will utilize more sophisticated programming and runtime environments

- Performance tools have to support richer computation models and more versatile performance perspectives
Heterogeneous Performance Views

- Want to create performance views that capture heterogeneous concurrency and execution behavior
  - Reflect interactions between heterogeneous components
  - Capture performance semantics relative to computation model
  - Assimilate performance for all execution paths for shared view
- Existing parallel performance tools are CPU (host)-centric
  - Event-based sampling (not appropriate for accelerators)
  - Direct measurement (through instrumentation of events)
- What perspective does the host have of other components?
  - Determines the semantics of the measurement data
  - Determines assumptions about behavior and interactions
- Performance views may have to work with reduced data
Heterogeneous Performance Measurement

- Multi-level heterogeneous performance perspectives

- *Inter-node communication*
  - Message communication, overhead, synchronization

- *Intra-node execution*
  - Multicore thread execution and interactions

- *Host-GPU interactions* (general CPU – “special” device)
  - Kernel setup, memory transfer, concurrency overlap, synchronization

- *GPU kernel execution*
  - Use of GPU compute and memory resources
Host (CPU) - GPU Scenarios

- **Single GPU**
  - Host (CPU):
    - Open device
    - Move data
    - Launch kernel(s)
    - Wait
    - Move data
  - GPU:
    - Run kernel(s)
  - Implemented as asynchronous calls

- **Multi-stream**
  - Host (CPU):
    - Open device
    - Move data
    - Launch kernel(s)
    - Wait
    - Move data
  - GPU:
    - Open device
    - Move data
    - Launch kernel(s)
    - Wait
    - Move data
    - Run kernel(s)
  - Stream 1
  - Stream 2

- **Multi-CPU, Multi-GPU**
  - Thread (CPU 1):
    - Open device
    - Move data
    - Launch kernel(s)
    - Wait
    - Move data
  - GPU 1
  - GPU k
  - Thread (CPU k):
    - Open device
    - Move data
    - Launch kernel(s)
    - Wait
    - Move data
  - Run kernel(s)
  - Run kernel(s)

Time
Consider three measurement approaches:
- **Synchronous**, Event queue, Callback

- **Synchronous** approach treats Host-GPU interactions as synchronous events that are measured on the CPU.

- Approximate measurement of actual kernel start/stop.
Host-GPU Measurement – Event Queue Method

- Event queue methods inserts events in GPU stream
- Events are measured by GPU
- Performance information read at sync points on CPU
- Support an asynchronous performance view
- Events must be placed around kernel launch !!!
Host-GPU Measurement – Callback Method

- **Callback method** is based on GPU driver and runtime support for exposing certain routines and runtime actions
- Measurement tool registers the callbacks
  - Application code is not modified !!!
- Where measurements occur depends on implementation
  - Measurement might be made on CPU or GPU
  - Measurements are accessed at callback point (CPU)
Method Support and Implementation

- **Synchronous method**
  - Place instrumentation around GPU calls
  - Wrap (synchronous) library with performance tool

- **Event queue method**
  - Utilize CUDA and OpenCL event support
  - Need instrumentation to create / insert events in the streams with kernel launch and process events
  - Can be implemented with driver library wrapping

- **Callback method**
  - Utilize language-level callback support in OpenCL
  - Use NVIDIA CUDA Performance Tool Interface (CUPTI)
  - Need to appropriately register callbacks
GPU Performance Measurement Tools

- Focus on development of measurement tools to support the Host-GPU performance perspective

- Objectives:
  - Provide integration with existing measurement system to promote/facilitate tool use
  - Utilize (where possible) support in GPU driver/runtime libraries and GPU device

- Tools with GPU measurement support
  - TAU performance system
  - VampirTrace measurement and Vampir analysis
  - PAPI (PAPI CUDA)
  - NVIDIA CUPTI

TAU for Heterogeneous Measurement

- TAU Performance System® (http://tau.uoregon.edu)
  - Instrumentation, measurement, analysis for parallel systems
  - Extended to support heterogeneous performance analysis

- Integrate Host-GPU support in TAU measurement
  - Enable host-GPU measurement approach
    - CUDA, OpenCL, PyCUDA as well as support for PGI and HMPP accelerator code generation capabilities
    - Utilize PAPI CUDA and CUPTI
  - Provide both heterogeneous profiling and tracing support
    - Contextualization of asynchronous kernel invocation

- Additional support
  - TAU wrapping of libraries (tau_gen_wrapper)
  - Work with binaries using library preloading (tau_exec)
Performance Tools for GPU-Powered Scalable Heterogeneous Systems

- Performance API (PAPI) (http://icl.cs.utk.edu/papi)

  - PAPI CUDA component
    - PAPI component to support measurement of GPU counters
    - Based on CUPTI (works with NVIDIA GPUs and CUDA)
    - Device-level access to GPU counters (different devices)

<table>
<thead>
<tr>
<th>Event Code</th>
<th>Symbol</th>
<th>Long Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x44000000</td>
<td>CUDA.GeForce_GTX_480.gpc0.local_load</td>
<td># executed local load instructions per warp on a multiprocessor</td>
</tr>
<tr>
<td>0x44000001</td>
<td>CUDA.GeForce_GTX_480.gpc0.local_store</td>
<td># executed local store instructions per warp on a multiprocessor</td>
</tr>
<tr>
<td>0x44000002</td>
<td>CUDA.GeForce_GTX_480.gpc0.gld_request</td>
<td># executed global load instructions per warp on a multiprocessor</td>
</tr>
<tr>
<td>0x44000003</td>
<td>CUDA.GeForce_GTX_480.gpc0.gst_request</td>
<td># executed global store instructions per warp on a multiprocessor</td>
</tr>
<tr>
<td>0x44000004</td>
<td>CUDA.GeForce_GTX_480.gpc0.shared_load</td>
<td># executed shared load instructions per warp on a multiprocessor</td>
</tr>
<tr>
<td>0x44000005</td>
<td>CUDA.GeForce_GTX_480.gpc0.shared_store</td>
<td># executed shared store instructions per warp on a multiprocessor</td>
</tr>
<tr>
<td>0x44000006</td>
<td>CUDA.GeForce_GTX_480.gpc0.branch</td>
<td># branches taken by threads executing a kernel</td>
</tr>
<tr>
<td>0x44000007</td>
<td>CUDA.GeForce_GTX_480.gpc0.divergent_branch</td>
<td># divergent branches within a warp</td>
</tr>
<tr>
<td>0x4400000b</td>
<td>CUDA.GeForce_GTX_480.gpc0.active_cycles</td>
<td># cycles a multiprocessor has at least one active warp</td>
</tr>
<tr>
<td>0x4400000c</td>
<td>CUDA.GeForce_GTX_480.gpc0.sm_cta_launched</td>
<td># thread blocks launched on a multiprocessor</td>
</tr>
<tr>
<td>0x4400000d</td>
<td>CUDA.GeForce_GTX_480.gpc0.l1_local_load_hit</td>
<td># local load hits in L1 cache</td>
</tr>
<tr>
<td>0x4400000e</td>
<td>CUDA.GeForce_GTX_480.gpc0.l1_local_load_miss</td>
<td># local load misses in L1 cache</td>
</tr>
<tr>
<td>0x44000011</td>
<td>CUDA.GeForce_GTX_480.gpc0.l1_global_load_hit</td>
<td># global load hits in L1 cache</td>
</tr>
<tr>
<td>0x4400002e</td>
<td>CUDA.Tesla_C870.domain_a.tex_cache_hit</td>
<td># texture cache hits</td>
</tr>
<tr>
<td>0x4400002f</td>
<td>CUDA.Tesla_C870.domain_a.tex_cache_miss</td>
<td># texture cache misses</td>
</tr>
<tr>
<td>0x44000034</td>
<td>CUDA.Tesla_C870.domain_b.local_load</td>
<td># local memory load transactions</td>
</tr>
<tr>
<td>0x44000037</td>
<td>CUDA.Tesla_C870.domain_b.branch</td>
<td># branches taken by threads executing a kernel</td>
</tr>
<tr>
<td>0x44000038</td>
<td>CUDA.Tesla_C870.domain_b.divergent_branch</td>
<td># divergent branches within a warp</td>
</tr>
<tr>
<td>0x44000039</td>
<td>CUDA.Tesla_C870.domain_b.instructions</td>
<td># instructions executed</td>
</tr>
</tbody>
</table>
Vampir / VampirTrace for GPU

- Vampir / VampirTrace (http://www.vampir.eu/)
  - Trace measurement and analysis of parallels application
  - Extend to support GPU performance measurement

- Integrate Host-GPU measurement in trace measurement
  - Based on the event queue method
  - Library wrapping for CUDA and OpenCL
  - Per kernel thread recording asynchronous events
  - Use of CUPTI to capture performance counters
  - Translation of GPU trace information to valid Vampir form

- Visualization of heterogeneous performance traces
  - Presentation of memory transfer and kernel launches
  - Includes calculation of counter statistics and rates
NVIDIA CUDA Performance Tool Interface (CUPTI)

- NVIDIA is developing CUPTI to enable the creation of profiling and tracing tools
  - CUPTI support was released with CUDA 4.0
  - Current version is released with CUDA 4.2
- CUPTI is delivered as a dynamic library
NVIDIA CUPTI APIs

- **Callback API**
  - interject tool callback code at the entry and exist to each CUDA runtime and driver API call
  - registered tools are invoked for selected events

- **Counter API**
  - query, configure, start, stop, read counters on CUDA devices
  - device-level counter access

- **Activity API**
  - GPU kernel and memory copy timing information is stored in a buffer until a synchronization point is encountered and these timings are recorded by the CPU
  - Synchronization can be either be within a device, stream or occur during some synchronous memory copies and event synchronizations
GPU Performance Tool Interoperability

CUDA → OpenCL

CUDA → OpenCL

CUPITI

CUDA → OpenCL

CUDA → OpenCL

TAU → PAPI → VampirTrace

parallel profile

tau2otf

Paraprof

Event queue
Callback

Vampir

parallel trace
CUDA SDK simpleMultiGPU

- Demonstration of multiple GPU device use
- Program structure:
  
  \[ \text{main} \rightarrow \text{solverThread} \rightarrow \text{reduceKernel} \]

- One node with three GPUs
- Performance profile for:
  
  - One \textit{main} thread
  - Three \textit{solverThread} threads
  - Three \textit{reduceKernel} “threads”
Identified a known overhead in GPU context creation:
Allocating memory blocks other host-device interactions like `cudaSetDevice()`
# simpleMultiGPU Profile

## Main Thread

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>main [{simpleMultiGPU.cpp} {105,0}]</td>
<td>1,543.18</td>
<td>20,997.786</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>cutWaitForThreads [{multithreading.cpp} {65,0}]</td>
<td>0.006</td>
<td>19,450.196</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>cutEndThread [{multithreading.cpp} {55,0}]</td>
<td>19,450.19</td>
<td>19,450.19</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaGetDeviceCount(int *) C</td>
<td>4.342</td>
<td>4.342</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cutStartThread [{multithreading.cpp} {48,0}]</td>
<td>0.064</td>
<td>0.064</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaThreadExit(void) C</td>
<td>0.004</td>
<td>0.004</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

## Solver Thread

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>solverThread(TGPUplan*) [{simpleMultiGPU.cpp} {47,0}]</td>
<td>0.103</td>
<td>18,767.447</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td>cudaError_t cudaMalloc(void **, size_t) C</td>
<td>18,743.914</td>
<td>18,743.914</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaMemcpy(void *, const void *, size_t, enum cudaMemcpyKind) C</td>
<td>20.108</td>
<td>20.108</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaFree(void *) C</td>
<td>2.007</td>
<td>2.007</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaThreadSynchronize(void) C</td>
<td>1.286</td>
<td>1.286</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaSetDevice(int) C</td>
<td>0.013</td>
<td>0.013</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaLaunch(const char *) C</td>
<td>0.012</td>
<td>0.012</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaConfigureCall(dim3, dim3, size_t, cudaStream_t) C</td>
<td>0.002</td>
<td>0.002</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaSetupArgument(const void *, size_t, size_t) C</td>
<td>0.002</td>
<td>0.002</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

## GPU Execution

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>.TAU application</td>
<td>0</td>
<td>0.865</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>reduceKernel</td>
<td>0.865</td>
<td>0.865</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
SHOC Benchmark Suite

- **Scalable HeterOgeneous Computing** benchmarks (ORNL)
  - Programs to test performance on heterogeneous systems
  - Benchmark suite with a focus on scientific computing workloads, including common kernels like SGEMM, FFT, Stencils

- Parallelized with MPI, with support for multi-GPU and cluster scale comparisons

- Implemented in CUDA and OpenCL for a 1:1 performance comparison

- Includes stability tests

Consider SHOC FFT benchmark

- Three kernels: `ifft1D_512`, `fft1D_512`, `chk1D_512`
- Called from single or double precession step

TAU can associate callsite information with kernel launch

- Enabled with callpath profiling (CALLPATH env. variable)
- Callsite information links the kernels on the GPU with functions that launch them on the CPU
- Callsite paths can be thought of as an extension of a callpath spanning both CPU and GPU
SHOC Stencil2D

- Compute 2D, 9-point stencil
  - Multiple GPUs using MPI
  - CUDA and OpenCL versions
- Experiments:
  - One node with 3 GPUs (Keeneland)
  - Two nodes with 4 GPUs (TU Dresden)
  - Eight nodes with 24 GPUs (Keeneland)
- Performance profile (TAU) and trace (Vampir)
  - Application events
  - Communication events
  - Kernel execution
Stencil2D Trace (Vampir / VampirTrace)

- Four MPI processes each with one GPU
- VampirTrace measurements
Stencil2D Parallel Profile (TAU)

Metric: TAUGPU_TIME
Value: Exclusive

Std. Dev.
Mean
node 0, thread 0
node 0, thread 1
node 1, thread 0
node 1, thread 1
node 2, thread 0
node 2, thread 1

3 GPUs

Name: StencilKernel
Metric Name: TAUGPU_TIME
Value: Exclusive
Units: microseconds
Stencil2D Trace (TAU, 512 iterations, 4 CPUxGPU)

- Visualization using Jumpshot (Argonne)

CUDA memory transfer (white)
CUDA Linpack Profile (4 processes, 4 GPUs)

- GPU-accelerated Linpack benchmark (NVIDIA)
Evolving CUPTI Features

- CUPTI 4.1 delivered new features of importance
- Activity API
  - Facilitates gathering of per kernel performance information
  - Using CUDA events previously to record kernel times effectively sequentialized kernel execution
  - Kepler 2 will fix this going forward
- Tracking of the GPU portion of memory copy transactions
  - Allows for memory copy transaction pointers in traces (as seen in OpenCL)
  - Allows for performance analysis of asynchronous memory copies techniques
    - Overlapping memory copies with kernel execution
Synchronous / Asynchronous Memory Copy

Synchronous Memory Copies

Kernel Memory copy

Asynchronous Memory Copies (can now be observed in CUPTI 4.1)
- Compute a symmetric matrix vector (SYMV) product
- Symmetry exploitation is more challenging
  - Computation would involve irregular data access
- MAGMA (LAPACK for GPUs) SYMV implementation
  - Access each element of lower (or upper) triangular part of the matrix only once (N2/2 element reads (vs. N2))
  - Since SYMV is memory-bound, exploiting symmetry is expected to be twice as fast
  - To accomplish this, additional global memory workspace is used to store intermediate results
- Experiments on Tesla S2050 (Fermi)):
  - CUBLAS_dsymv (general)
  - MAGMA_dsymv (exploits symmetry)
  - Use PAPI CUDA to measure algorithm effects
CUDA Performance Counters for Read Behavior

- Green: # of read requests from L1 to L2
- Orange: # of read misses in L2 (= # read requests L1-L2)
- Black: read requests from L2 to DRAM
- # requests/misses halved in MAGMA due to symmetry
CUDA Performance Counters for Write Behavior

- Green: # of write requests from L1 to L2 (green)
- Orange: # of write misses in L2 (orange)
- Black: # of write requests from L2 to DRAM

- # requests/misses doubled in MAGMA
  - Need additional memory for intermediate results
CUDA Performance Counter for L1 Behavior

- # of L1 shared bank conflicts for medium to large matrices
- Performance with and without shared bank conflicts
- Shared cache bank conflicts were eliminated with array padding
- Results in performance improvement of 1 Gflops/s
NSF Keeneland Heterogeneous System

- Keeneland system (initial delivery)
  - 120x HP SL390 GPU cluster node
    - 2 Intel Xeon CPUs, 3 NVIDIA GPUs (M2070/M2090)
  - InfiniBand QDR network
  - Contains GPU nodes with non-uniform PCI performance

- Dr. Jeff Vetter, Keeneland project PI
  - http://keeneland.gatech.edu
Gyrokinetic Toroidal Simulations (GTC)

- GTC is used for fusion simulation
  - DOE SciDAC and INCITE application
- GTC CUDA version has been developed
  - OpenMP + CUDA
  - Three CUDA kernels
GPU threads and OpenMP threads are integrated into trace
GTC on 16 Keeneland Nodes (48 MPI ranks)

- 48 GPUs
- 198 OpenMP threads (240 total threads)
Nano Molecular Dynamics (NAMD)

- NAMD is an object-oriented MD code using Charm++
  - University of Illinois at Urbana-Champaign
- GPU version uses three kernels
  - Slow Energy
  - Pairlist
  - (Fast) Energy
  - Slow Energy
NAMD Profile

- NAMD's GPU kernels
  - Single node view shows execution time distribution

  ![Graph showing execution time distribution](image)

  **Metric: TAUPGPU_TIME**
  **Value: Exclusive**
  **Units: seconds**

  12.483
  6.725

  **dev_nonbonded_energy(patch_pair const*,**

  **Histogram across nodes for non-bounded energy calculation**

  ![Histogram across nodes](image)
LAMMPS

- Large-scale Automic/Molecular Massively Parallel Simulator (LAMMPS)

- Two different packages extend LAMMPS to the GPU
  - Both packages accelerate pair interactions and neighbor list construction
  - “GPU” packages is designed with smaller systems (atoms per processor) in mind and atoms are copied between the host and device each time step
  - “CUDA” packages is designed with large systems in mind and multiple time-steps can be run on GPUs minimizing Host-Device memory overhead
LAMMPS's "CUDA" implementation is generally faster.
“CUDA” versus “GPU” Runtime

- How much speedup is gained by using the “CUDA” package?
- Does it vary with the number of atoms per node or number of nodes used?

**CUDA improvement over GPU**

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Number of Atoms per Node</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32000</td>
<td>87.5089</td>
</tr>
<tr>
<td>4</td>
<td>230400</td>
<td>68.7767</td>
</tr>
<tr>
<td>7</td>
<td>428800</td>
<td>50.0445</td>
</tr>
<tr>
<td>10</td>
<td>627200</td>
<td>31.3123</td>
</tr>
<tr>
<td>13</td>
<td>825600</td>
<td>12.5801</td>
</tr>
<tr>
<td>16</td>
<td>1.024e+06</td>
<td>-6.15216</td>
</tr>
</tbody>
</table>
Compare Computing of Pair Interactions

- “CUDA” remains faster for computing pair interactions
- However for neighbor list construction the “GPU” implementation is faster

Number of Atoms scales with the number of processors (16k per node)

For iteration GPU Kernels

- CUDA remains faster for computing pair interactions
- However for neighbor list construction the “GPU” implementation is faster

Number of Atoms scales with the number of processors (16k per node)
LAMMPS Utilization of GPU

- GPU Idle time is when the GPU is not computing, but memory copying may still be taking place
- "CUDA" better utilizes the GPU
  - Comes at the price of having the CPU wait for the GPU to complete its computations

Number of Atoms scales with the number of processors (16k per node)

![Graphs showing time the CPUs are waiting for the GPU and time the GPUs are idle](Graphs.png)
### NUMA Effects on Keeneland

- NSF Keeneland contains GPU nodes with non-uniform communication costs between CPU and GPU.
- Penalty in SHOC benchmarks when this is not considered.

<table>
<thead>
<tr>
<th>Test</th>
<th>Units</th>
<th>Correct NUMA</th>
<th>Incorrect NUMA</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEMM</td>
<td>GFLOPS</td>
<td>535.640</td>
<td>519.581</td>
<td>3%</td>
</tr>
<tr>
<td>DGEMM</td>
<td>GFLOPS</td>
<td>239.962</td>
<td>230.809</td>
<td>4%</td>
</tr>
<tr>
<td>FFT</td>
<td>GFLOPS</td>
<td>30.501</td>
<td>26.843</td>
<td>12%</td>
</tr>
<tr>
<td>FFT-DP</td>
<td>GFLOPS</td>
<td>15.181</td>
<td>13.352</td>
<td>12%</td>
</tr>
<tr>
<td>MD</td>
<td>GB/s</td>
<td>12.519</td>
<td>11.450</td>
<td>9%</td>
</tr>
<tr>
<td>MD-DP</td>
<td>GB/s</td>
<td>19.063</td>
<td>17.654</td>
<td>7%</td>
</tr>
<tr>
<td>Reduction</td>
<td>GB/s</td>
<td>5.631</td>
<td>4.942</td>
<td>12%</td>
</tr>
<tr>
<td>Scan</td>
<td>GB/s</td>
<td>0.007</td>
<td>0.005</td>
<td>31%</td>
</tr>
<tr>
<td>Sort</td>
<td>GB/s</td>
<td>1.081</td>
<td>0.983</td>
<td>9%</td>
</tr>
<tr>
<td>Stencil</td>
<td>seconds</td>
<td>8.749</td>
<td>11.895</td>
<td>36%</td>
</tr>
</tbody>
</table>
Bus speed is dependent on NUMA

If incorrectly set a penalty of 15% for Host to Device and 46% for Device to Host transfers is observed

Memory copy Host to Device <= 536670912 bytes
Memory copy Host to Device <= 2684354656 bytes
Memory copy Host to Device <= 1342177268 bytes
Memory copy Host to Device <= 67108664 bytes
Memory copy Host to Device <= 33554432 bytes
Memory copy Host to Device <= 16777216 bytes
Memory copy Host to Device <= 8388608 bytes
Memory copy Host to Device <= 4194304 bytes
Memory copy Host to Device <= 2097152 bytes
Memory copy Host to Device <= 1048576 bytes
Memory copy Host to Device <= 524288 bytes
Memory copy Host to Device <= 262144 bytes
Memory copy Host to Device <= 131072 bytes
Memory copy Host to Device <= 65536 bytes
Memory copy Host to Device <= 32768 bytes

Memory copy Device to Host <= 536670912 bytes
Memory copy Device to Host <= 2684354656 bytes
Memory copy Device to Host <= 1342177268 bytes
Memory copy Device to Host <= 67108664 bytes
Memory copy Device to Host <= 33554432 bytes
Memory copy Device to Host <= 16777216 bytes
Memory copy Device to Host <= 8388608 bytes
Memory copy Device to Host <= 4194304 bytes
Memory copy Device to Host <= 2097152 bytes
Memory copy Device to Host <= 1048576 bytes
Memory copy Device to Host <= 524288 bytes
Memory copy Device to Host <= 262144 bytes
Memory copy Device to Host <= 131072 bytes
Memory copy Device to Host <= 65536 bytes
Memory copy Device to Host <= 32768 bytes
GPU package will suffer from incorrect placement

- Memory transfers between Host and Device through one QPI hop (correct) or two QPI hops (incorrect)
New Features in TAU 2.21.2

- CUDA device memory tracking
- Improved HMPP support
  - Utilize HMPP callback interface
- OpenACC support
- OpenCL queue wait times recording
CUDA Device Memory Tracking

- New feature in TAU 2.21.2
  - Requires CUDA 4.1 or greater
- Local, shared and device registers usage can be tracked for each kernel
- Familiar technique of sharing blocks of memory on the GPU is capture by this feature
**Multiply Matrices**

```c
multiply_matrices(float *d_a, float *d_b, float *d_c, int lda)
...
for (unsigned int j=0; j<M; j++) {
    ctemp = ctemp + d_a[idx(row,j,lda)] * d_b[idx(j,col,lda)];
}
d_c[id] = ctemp;
...
```

- **Simple**
  - No use of shared memory

- **Improved**
  - Utilize the shared memory
  - Better use of registers

```c
multiply_matrices_shared_blocks(float *d_a, float *d_b, float *d_c, int lda)
...
for (int k = 0; k < (M / bs); k++) {
    // form submatrices
    a[sub_row][sub_col] = sub_a[idx(sub_row, sub_col, lda)];
    b[sub_row][sub_col] = sub_b[idx(sub_row, sub_col, lda)];
    // wait for all threads to complete copy to shared memory.
    __syncthreads();
    // multiply each submatrix
    for (int j=0; j < bs; j++) {
        c = c + a[sub_row][j] * b[j][sub_col];
    }
    // move results to device memory.
    d_c[id] = c;
    // wait for multiplication to finish before moving onto the next submatrix
    __syncthreads();
}
```
CUDA Device Memory Profile

Use of shared memory  Increase in register usage

Improved

Simple

2.4x speedup by using shared memory
Directive assisted acceleration in HMPP compiler

```c
!$HMPP multiply codelet, target=CUDA, args[a;b;matsize].io=in, args[c].io=out
subroutine multiply_matrices(a, b, c, matsize)
```
PGI compiler

```c
!$acc region
do j=1,m
do k = 1,m
do l = 1,m
   a(l,j) = a(l,j) + b(i,k) * c(k,j)
```

#### TAU: ParaProf: Thread Statistics: n, c, t, 0, 0, 0 - /Users/sameer/rs/taudata/mm

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>_pgi_cu_downloadx multiply_matrices var=a, dims=2, desc.devx=0, desc.devstride=1, desc.hoststrid</td>
<td>55.4%</td>
<td>2.721</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>_pgi_cu_init multiply_matrices [/mnt/netapp/home1/sameer/mm/mm2.f90][9]</td>
<td>39.3%</td>
<td>1.933</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>mymatrixmultiply [/mnt/f90][1,0]</td>
<td>1.8%</td>
<td>0.09</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>_pgi_cu_uploadx multiply_matrices var=c, dims=2, desc.devx=0, desc.devstride=1, desc.hoststrid</td>
<td>1.6%</td>
<td>0.081</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>_pgi_cu_uploadx multiply_matrices var=b, dims=2, desc.devx=0, desc.devstride=1, desc.hoststrid</td>
<td>1.6%</td>
<td>0.079</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>_pgi_cu_free multiply_matrices [/mnt/netapp/home1/sameer/mm/mm2.f90]</td>
<td>0.1%</td>
<td>0.004</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>_pgi_cu_alloc multiply_matrices [/mnt/netapp/home1/sameer/mm/mm2.f90][9]</td>
<td>0.1%</td>
<td>0.003</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>multiply_matrices [/mnt/f90][5,0]</td>
<td>0.0%</td>
<td>0.002</td>
<td>5</td>
<td>65</td>
</tr>
<tr>
<td>_pgi_cu_module multiply_matrices [/mnt/netapp/home1/sameer/mm/mm2.f90][9]</td>
<td>0.0%</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>_pgi_cu_launch multiply_matrices (multiply_matrices_11_gpu,gx=168,gy=168,gz=1,bx=16,by=16)</td>
<td>0.0%</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>_pgi_cu_paramset multiply_matrices [/mnt/netapp/home1/sameer/mm/mm2.f90]</td>
<td>0.0%</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>_pgi_cu_launch multiply_matrices (multiply_matrices_15_gpu,gx=168,gy=168,gz=1,bx=16,by=16)</td>
<td>0.0%</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>_pgi_cu_module_function2 multiply_matrices name=multiply_matrices_11_gpu, argname=(null), arg</td>
<td>0.0%</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
OpenCL Queue Monitoring

- New in TAU 2.21.2
- Tracks the time spent for each kernel in the OpenCL command queue
  - Time from when the kernel was placed into the queue …
  - To when it is run on the GPU
- Measured in microseconds
  - max, min, total and standard deviation
- Information obtained from the OpenCL API
Effects of Multiple Command Queue

- Compare one command queue versus dual queue
  - Effects the time spent the queue for each kernel
- Look at NVIDIA SDK `oclCopyComputeOverlap` program

One Command Queue
(no overlap)

Time Spent in Queue

VectorHypot Kernel

Dual Command Queue
(overlap memory / compute)
Profiles show time spent in the queue in the profile directly

One Queue Mean Time: 289 (ms)

Dual Queue Mean Time: 145 (ms)
Vancouver: Heterogeneous Exascale Software

- DOE X-stack project
- Partners:
  - Oak Ridge National Laboratory
  - University of Oregon
  - University of Illinois
  - Georgia Institute of Technology
- Components
  - Compilers
  - Scheduling and runtime resource management
  - Libraries
  - Performance measurement, analysis, modeling
For More Information …

- TAU Website: [http://tau.uoregon.edu](http://tau.uoregon.edu)
  - Software
  - Release notes
  - Documentation

- TAU LiveDVD: [http://tau.uoregon.edu/point.iso](http://tau.uoregon.edu/point.iso)
  - Boot by typing `<tab>`, `‘drm.modeset=0’`
  - Includes TAU, VampirTrace/Vampir, and variety of other packages
  - Include documentation and a CUDA 4.1 pre-release driver for those of you with NVIDIA GPU cards
  - By using the LiveDVD you agree to all software licenses therein
Downloading TAU to Desk/Laptop

- **Windows** ([http://tau.uoregon.edu/tau.exe](http://tau.uoregon.edu/tau.exe))
  
  Executable is self-extracting
  
  Launch ParaProf and Jumpshot from
  
  C:\Program Files\Tau directory

- **Mac** ([http://tau.uoregon.edu/tau.dmg](http://tau.uoregon.edu/tau.dmg))
  
  Mount DMG and drag to copy
  
  TAU to /Applications directory
  
  Launch ParaProf and Jumpshot from /Application/TAU directory

- **Linux** ([http://tau.uoregon.edu/tau.tgz](http://tau.uoregon.edu/tau.tgz))
  
  Untar and run ./configure from tau directory
  
  Launch ParaProf and Jumpshot from
  
  tau/<arch>/bin directory
  
  (<arch> likely x86_64 or i386)
TAU Build Instructions

- TAU is released with CUDA support
- Download TAU and configure it

```
wget http://tau.uoregon.edu/tau.tgz
tar xzf tau.tgz
cd tau-2.21.2
./configure -cuda=<path to CUDA>
make install
```

```
set your PATH to tau-2.21.2/<arch>/bin and
LD_LIBRARY_PATH to tau-2.21.2/<arch>/lib
```
TAU Run Instructions

- TAU uses library preloading to interact with CUPTI
- Use with any executable (no re-compiling or re-linking)
- TAU wraps OpenCL by library preloading as well

tau_exec tools will do the preloading:

```
tau_exec -T <config> -<library> <exe>
```

<config> can be one or a combination of serial, mpi, cupti (matches your TAU configuration)

<library> can be either cupti or opencl
Support Acknowledgements

- Department of Energy (DOE)
  - Office of Science
  - ASC/NNSA
- Department of Defense (DoD)
  - HPC Modernization Office (HPCMO)
- NSF Software Development for Cyberinfrastructure (SDCI)
- Research Centre Juelich
- Argonne National Laboratory
- Technical University Dresden
- ParaTools, Inc.
- NVIDIA