ADVANCED OPTIMIZATION FOR SCALABLE HETEROGENEOUS CLUSTERS

Jeremy Meredith
February 21, 2012

http://keeneland.gatech.edu
Outline

• Single-GPU optimization techniques
  – Hand-written CUDA, OpenCL
  – Compiler directive approaches

• Optimization for heterogeneous systems
  – Non-uniform memory access
  – Data transfers between devices
  – Task layout examples
SINGLE-GPU OPTIMIZATION TECHNIQUES
Host Motherboard Layout

- CPU
- DDR DRAM
- PCIe slot
Discrete GPU PCB Layout

Image from http://techreport.com/articles.x/14168
CUDA, OpenCL Optimization

- Minimize data transfers across PCI-Express bus
  - Very expensive: e.g. 5GB/s PCIe versus 100GB/s for device
  - Can be asynchronous; overlap communication with computation
- Coalesce memory reads (and writes)
  - ensure threads simultaneously read adjacent values
  - effectively uses GPU memory bandwidth

GOOD: GPU can read values for all threads in a single chunk

BAD

BAD
CUDA, OpenCL Optimization

- Shared memory is fast, local to a group of threads
- When access patterns are irregular:
  - perform coalesced reads to shared memory
  - synchronize threads
  - then access in any pattern
CUDA, OpenCL Optimization

• Unroll loops to minimize overhead
  – GPU kernel compilation not yet mature here

• Execute more than one item per thread
  – further increase computational density
  – remember: maintain coalescing
    • e.g. stride by grid size

*Many presentations, whitepapers detail these aspects of optimization.
Accelerating Compiler Optimization

• Similar concepts apply
• Relying on compiler for a lot:
  – *coalescing*: you might be able to help by modifying your array layouts
  – *unrolling, tiling, shared memory*: some compilers are better than others, some offer unroll+jam pragmas, some offer shared memory pragmas
  – *minimizing data transfers*: most offer directives to specify allocation and transfer boundaries
OPTIMIZATIONS ON HETEROGENEOUS SYSTEM NODES
Keeneland’s Multi-GPU Nodes

- Keeneland is a dual-I/O-hub node architecture
  - Allows full PCIe bandwidth to 3 GPUs and 1 NIC
Sharing GPUs on Keeneland

• Simultaneous PCIe bandwidth to all 3 GPUs
Non-Uniform Memory Access

- Node architectures result in Non-Uniform Memory Access (NUMA)
  - Point-to-point connections between devices
  - Not fully-connected topologies
  - Host memory connected to sockets instead of across a bus
NUMA Can Affect GPUs and Network Too

Older node architecture with single I/O hub but no NUMA effects between CPU and GPU/HCA

- DL160
- Single I/O Hub
- PCIe switch connects GPUs

Keeneland node architecture with dual I/O hub but NUMA effects

- SL390
- Dual I/O Hub
- No PCIe switch
NUMA Control Mechanisms

• Process, data placement tools:
  – Tools like libnuma and numactl
  – Some MPI implementations have NUMA controls built in (e.g., Intel MPI, OpenMPI)

• numactl usage:
  ```
  numactl [--interleave=nodes] [--preferred=node]
  [--physcpubind=cpus] [--cpunodebind=nodes]
  [--membind=nodes] [--localalloc] command
  numactl [--show]
  numactl [--hardware]
  ```
numactl on Keeneland

[meredith@kid107]$ numactl -show
  policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11
cpubind: 0 1
nodebind: 0 1
membind: 0 1
“NUMA Nodes” on Keeneland nodes

- Physical CPUs 0-5
- Physical CPUs 6-11
- Infiniband
- GPU 0
- GPU 1
- GPU 2

RAM Node 0
RAM Node 1
CPU Node 0
CPU Node 1
IOH
numactl on Keeneland

[meredith@kid107]$ numactl --hardware

available: 2 nodes (0-1)
node 0 size: 12085 MB
node 0 free: 11286 MB
node 1 size: 12120 MB
node 1 free: 11648 MB
node distances:
node 0 1
  0: 10 20
  1: 20 10
OpenMPI with NUMA control

Use `mpirun` to execute a script:

```
mpirun ./prog_with_numa.sh
```

In that script (prog_with_numa.sh) launch under `numactl`:

```
if [[ $OMPI_COMM_WORLD_LOCAL_RANK == "0" ]]
then
  numactl --membind=0 --cpunodebind=0 ./prog -args
else
  numactl --membind=1 --cpunodebind=1 ./prog -args
fi
```
How much Does NUMA Impact Performance?

• Microbenchmarks to focus on individual node components
• Macrobenchmarks to focus on individual operations and program kernels
• Full applications to gauge end-user impact


Data Transfer Bandwidth

- Measured bandwidth of data transfers between CPU socket 0 and the GPUs

![Diagram showing data transfer bandwidth between CPU and GPUs](Image)

- CPU-to-GPU
- GPU-to-CPU

![Graph showing CPU-to-GPU bandwidth](Image)

- GPU #0 vs. GPU #1 or #2

![Graph showing GPU-to-CPU bandwidth](Image)

- GPU #0 vs. GPU #1 or #2

Size (KB): 1, 16, 256, 4096, 65536
SHOC Benchmark Suite

- What penalty for “long” mapping?
- Rough inverse correlation to computational intensity

<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>

Table 3: SHOC Benchmark Results
Full Applications

- With one application task, performance penalty for using incorrect mapping (e.g., CPU socket 0 with GPU 1)
- With two application tasks, performance penalty for using mapping that uses “long” paths for both (e.g., CPU socket 0 with GPU 1 and CPU socket 1 with GPU 0)
HPL Linpack

- Runtimes on Keeneland under 3 pinning scenarios
NUMA and Network Traffic

• Have to worry about not only process/data placement for CPU and GPU, but also about CPU and Infiniband HCA
Thread Splitting

- Instead of 1 thread that controls a GPU and issues MPI calls, split into two threads and bind to appropriate CPU sockets.
GPU DIRECT
GPU Direct

- Transferring data between GPUs in a scalable heterogeneous system like KIDS is expensive
  - Between GPUs in different nodes
  - Between GPUs in the same node
The Problem with Inter-Node Transfers

- Data is in device memory of GPU on one node, needs to be transferred to device memory of GPU on another node
- Several hops:
  - Data transferred from GPU memory to GPU buffer in host memory
  - Data copied from GPU buffer to IB buffer in host memory
  - Data read by IB HCA using RDMA transfer
  - Repeat in reverse on other end

GPUDirect

• NVIDIA and Mellanox developed an approach for allowing others to access the GPU buffer in host memory
• Eliminates the data copy from GPU buffer to IB buffer
  – Eliminates two system memory data copy operations (one on each end)
  – Keeps host CPU out of the data path
  – Up to 30% performance improvement (according to NVIDIA)

GPUDirect 2.0: Improving Transfer Performance Within a Node

- Similar problem when transferring data from one GPU to another within the same node
- Old way:
  - Copy data from GPU 1 to host memory
  - Copy data from host memory to GPU 2
- New way:
  - Copy data from GPU 1 to GPU 2 without host CPU involvement
- Integrates well with Unified Virtual Addressing feature (single address space for CPU and 1+ GPUs)
- Available in CUDA 4.0
Deploying GPUDirect on Keeneland

• Initially, GPUDirect packaging made deploying it unattractive
  – Required running a specific kernel version, with RPMs provided by Mellanox
  – Kernel version was quite old
  – Concerns about keeping updated to avoid security risks
  – Have not yet tested GPUDirect kernel patches with PAPI patches for our CentOS 5.5 kernel

• Mellanox now makes kernel patches available for some Enterprise distributions

• Mellanox working on getting GPUDirect support included as part of stock Linux kernel

• GPUDirect 2.0 included with CUDA 4.0
Currently active on Keeneland for GPU1 ↔ GPU2
- 2.8 GB/s normally, 4.9 GB/s with GPUDirect

Not yet possible for GPU0 or NIC
Using GPUDirect

• General strategy:
  – GPU-GPU copies
    • Use cudaMemcpy with two device pointers
    • Enable peer access in CUDA to allow direct GPU-GPU
      – even allows inter-GPU access within CUDA kernels
  – Host-device copies
    • Allocated any host memory as pinned in CUDA
    • CUDA driver puts this in user-pageable memory, virtual address space
      – May need to “export CUDA_NIC_INTEROP=1” for InfiniBand to share this with CUDA
Checking GPUDirect for GPU1 ⇔ GPU2

1. Are devices using Tesla Compute Cluster driver?
   - cudaDeviceProp prop1, prop2;
   - cudaGetDeviceProperties(&prop1, 1);
   - cudaGetDeviceProperties(&prop2, 2);
   - check prop1.tccDriver==1 and prop2.tccDriver==1

2. Do devices support peer access to each other?
   - int access2from1, access1from2;
   - cudaDeviceCanAccessPeer(&access2from1, 1, 2);
   - cudaDeviceCanAccessPeer(&access1from2, 2, 1);
   - check access2from1==1 and access1from2==1
Enabling GPUDirect for GPU1 ⇔ GPU2

3. Enable device peer access both directions:
   • cudaSetDevice(1);
   • cudaDeviceEnablePeerAccess(2,flags); //flags=0
   • cudaSetDevice(2);
   • cudaDeviceEnablePeerAccess(1,flags); //flags=0

4. Example: send data directly from GPU2 to GPU1:
   • float *gpu1data, *gpu2data;
   • cudaSetDevice(1);
   • cudaMalloc(&gpu1data, nbytes);
   • cudaSetDevice(2);
   • cudaMalloc(&gpu2data, nbytes);
   • cudaMemcpy(gpu1data, gpu2data, cudaMemcpyDefault);
MPI AND GPU TASK MAPPING
How to combine GPUs and MPI?

• **Use 1 MPI task per CPU core?**
  – Simplest for an existing MPI code
    • particularly if they are not threaded
  – Either time share GPUs ...  
    • performance can vary, especially with more tasks/GPU
  – ... or only use GPUs from some MPI tasks  
    • introduce load balance problem
How to combine GPUs and MPI?

• Use 1 MPI task per GPU? Per CPU socket?
  – thread/OpenMP/OpenCL to use more CPU cores
  – ratios like 3GPU:2CPU add complexity
    • pinning 3 tasks to 2 CPU sockets makes using 12 cores hard
    • optimal NUMA mapping may not be obvious
  – can use 1 task for 2 GPUs, leave 3rd GPU idle
    • with 2 I/O hubs, bandwidth is probably sufficient
  – can leave CPU cores idle
    • for codes that match GPUs well, this can be a win
    • recent NVIDIA HPL results show benefits of this approach
How to combine GPUs and MPI?

• Use 1 MPI task per compute node?
  – With work, can be highly optimized:
    • Best use of GPUDirect transfers (GPU-GPU, GPU-NIC)
    • Can use numactl library within the task
  – Very complex – must handle:
    • multiple GPUs in one task
    • offload work for all CPU cores
    • NUMA mapping is a challenge
      – especially for automated threading like OpenMP
Summary

• GPU optimization techniques remain valid in the context of scalable heterogeneous systems
• More CPUs and GPUs add complexity
• Be aware of the machine architecture
  – simultaneous bandwidths
  – NUMA penalties
  – peer data transfers
• Pay attention to your CPU,GPU task mapping
For more information

- [http://keeneland.gatech.edu](http://keeneland.gatech.edu)

- Project PI: Jeffrey S. Vetter
  [vetter@cc.gatech.edu](mailto:vetter@cc.gatech.edu)