CUDA Optimization
Paulius Micikevicius | NVIDIA
Outline

• Kernel optimizations
  – Global memory throughput
  – Launch configuration
  – Instruction throughput / control flow
  – Shared memory access

• Optimizations of CPU-GPU interaction
  – Maximizing PCIe throughput
  – Overlapping kernel execution with memory copies
Global Memory Throughput
Memory Review

• Local storage
  – Each thread has own local storage
  – Mostly registers (managed by the compiler)

• Shared memory
  – Each thread block has own shared memory
  – Very low latency (a few cycles)
  – Very high throughput: 38-44 GB/s per multiprocessor
    • 30 multiprocessors per GPU -> over 1.1 TB/s

• Global memory
  – Accessible by all threads as well as host (CPU)
  – High latency (400-800 cycles)
  – Throughput: 140 GB/s (1GB boards), 102 GB/s (4GB boards)
GMEM Coalescing: Compute Capability 1.2, 1.3

• Possible GPU memory bus transaction sizes:
  – 32B, 64B, or 128B
  – Transaction segment must be aligned
    • First address = multiple of segment size

• Hardware coalescing for each half-warp (16 threads):
  – Memory accesses are handled per half-warps
  – Carry out the smallest possible number of transactions
  – Reduce transaction size when possible
HW Steps when Coalescing

• Find the memory segment that contains the address requested by the lowest-numbered active thread:
  – 32B segment for 8-bit data
  – 64B segment for 16-bit data
  – 128B segment for 32, 64 and 128-bit data.

• Find all other active threads whose requested address lies in the same segment

• Reduce the transaction size, if possible:
  – If size == 128B and only the lower or upper half is used, reduce transaction to 64B
  – If size == 64B and only the lower or upper half is used, reduce transaction to 32B
    • Applied even if 64B was a reduction from 128B

• Carry out the transaction, mark serviced threads as inactive

• Repeat until all threads in the half-warp are serviced
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127

![Diagram showing memory segment](image-url)
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 64B)
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 32B)
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255 (reduce to 64B)
Experiment: Impact of Address Alignment

• Assume half-warp accesses a contiguous region

• Throughput is maximized when region is aligned on its size boundary
  – 100% of bytes in a bus transaction are useful

• Impact of misaligned addressing:
  – 32-bit words, streaming code, Quadro FX5800 (102 GB/s)
  – 0 word offset: 76 GB/s (perfect alignment, typical perf)
  – 8 word offset: 57 GB/s (75% of aligned case)
  – All others: 46 GB/s (61% of aligned case)
Address Alignment, 64-bit words

• Can be analyzed similarly to 32-bit case:
  – 0B offset: 80 GB/s (perfectly aligned)
  – 8B offset: 62 GB/s (78% of perfectly aligned)
  – 16B offset: 62 GB/s (78% of perfectly aligned)
  – 32B offset: 68 GB/s (85% of perfectly aligned)
  – 64B offset: 76 GB/s (95% of perfectly aligned)

• Compare 0 and 64B offset performance:
  – Both consume 100% of the bytes
    • 64B: two 64B transactions
    • 0B: a single 128B transaction, slightly faster
Comparing Compute Capabilities

• **Compute capability < 1.2**
  – Requires threads in a half-warp to:
    • Access a single aligned 64B, 128B, or 256B segment
    • Threads must issue addresses in sequence
  – If requirements are not satisfied:
    • Separate 32B transaction for each thread

• **Compute capability 1.2 and 1.3**
  – Does not require sequential addressing by threads
  – Perf degrades gracefully when a half-warp addresses multiple segments

• **Compute capability 2.0 (Fermi)**
  – Memory access is per warp (32 threads), L1/L2 caches help with alignment
GMEM Optimization Guidelines

• Strive for perfect coalescing
  – Align starting address (may require padding)
  – Warp should access within contiguous region

• Process several elements per thread
  – Multiple loads get pipelined
  – Indexing calculations can often be reused

• Launch enough threads to cover access latency
  – GMEM accesses are not cached
  – Latency is hidden by switching threads (warps)
Data Layout for Optimal Memory Throughput

• Prefer Structure of Arrays instead of Array of Structures:
  – A warp (32 threads) should be accessing a contiguous memory region
    • As opposed to a thread accessing a contiguous region (as is often the case on CPU)
    – Stride between threads in a warp will ideally be 1, 2, 4, 8, or 16B
      (goes back to how hw coalescing is done)
  • Not that different from what has to be done for CPU vectorization:
    – SSE: 4-wide vectors (for fp32)
    – One can think of GPU accesses as 32-wide vectors
Global Memory Throughput Metric

• Many applications are memory throughput bound

• When coding from scratch:
  – Start with memory operations first, achieve good throughput
  – Add the arithmetic, measuring perf as you go

• When optimizing:
  – Measure effective memory throughput
  – Compare to the theoretical bandwidth
    • 70-80% is very good, ~50% is good if arithmetic is nontrivial

• Measuring throughput
  – From the app point of view (“useful” bytes)
  – From the hw point of view (actual bytes moved across the bus)
  – The two are likely to be different
    • Due to coalescing, discrete bus transaction sizes
Measuring Memory Throughput

- Visual Profiler reports memory throughput
  - From HW point of view
  - Based on counters for one TPC (3 multiprocessors)
  - Need compute capability 1.2 or higher GPU
Measuring Memory Throughput

- **Visual Profiler** reports memory throughput
  - From HW point of view
  - Based on counters for one TPC (3 multiprocessors)
  - Need compute capability 1.2 or higher GPU

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
<th>GPU usec</th>
<th>%GPU time</th>
<th>glob mem read throughput (GB/s)</th>
<th>glob mem write throughput (GB/s)</th>
<th>glob mem overall throughput (GB/s)</th>
<th>instruction throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fwd_3D_16x16_order8</td>
<td>3.09382E+06</td>
<td>82.15</td>
<td>95.9465</td>
<td>11.6771</td>
<td>58.6236</td>
<td>0.763973</td>
</tr>
<tr>
<td>2</td>
<td>memcpyHtoD</td>
<td>503394</td>
<td>13.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>memcpyDtoH</td>
<td>168906</td>
<td>4.48</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Measuring Memory Throughput

- Latest **Visual Profiler** reports memory throughput
  - From **HW** point of view
  - Based on counters for one **TPC** (3 multiprocessors)
  - Need **compute capability 1.2 or higher GPU**

<table>
<thead>
<tr>
<th>Profiler Output</th>
<th>Summary Table</th>
<th>GPU usec</th>
<th>%GPU time</th>
<th>glob mem read throughput (GB/s)</th>
<th>glob mem write throughput (GB/s)</th>
<th>glob mem overall throughput (GB/s)</th>
<th>instruction throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>fwd_3D_16x16_order8</td>
<td></td>
<td>3.08326+06</td>
<td>82.15</td>
<td>45.9465</td>
<td>11.6771</td>
<td>58.6236</td>
<td>0.763973</td>
</tr>
<tr>
<td>memcpyHtoD</td>
<td></td>
<td>503094</td>
<td>13.35</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>memcpyDtoH</td>
<td></td>
<td>168906</td>
<td>4.18</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- **How throughput is computed:**
  - Count load/store bus transactions of each size (32, 64, 128B) on the TPC
  - Extrapolate from one TPC to the entire GPU
    - Multiply by (total threadblocks / threadblocks on TPC)
    - (grid size / cta launched)
Launch Configuration
Launch Configuration

• How many threads/threadblocks to launch?

• Key to understanding:
  – Instructions are issued in order
  – A thread blocks when one of the operands isn’t ready:
    • Memory read doesn’t block
  – Latency is hidden by switching threads
    • Not by cache
    • GMEM latency is 400-800 cycles

• Conclusion:
  – Need enough threads to hide latency
Hiding Latency

Arithmetic:
- Need at least 6 warps (192) threads per SM
Hiding Latency

**Arithmetic:**
- Need at least 6 warps (192) threads per SM

**Memory:**
- Depends on the access pattern
- For GT200, 50% occupancy (512 threads per SM) is often sufficient
  - Occupancy = fraction of the maximum number of threads per multiprocessor

Streaming 16M words: each thread reads, increments, writes 1 element
Launch Configuration: Summary

• Need enough total threads to keep GPU busy
  – Currently (GT200), 512+ threads per SM is ideal
  – Fewer than 192 threads per SM WILL NOT hide arithmetic latency

• Threadblock configuration
  – Threads per block should be a multiple of warp size (32)
  – SM can concurrently execute up to 8 threadblocks
    • Really small threadblocks prevent achieving good occupancy
    • Really large threadblocks are less flexible
    • I generally use 128-256 threads/block, but use whatever is best for the application
Instruction Throughput / Control Flow
Runtime Math Library and Intrinsics

- Two types of runtime math library functions
  - __func(): many map directly to hardware ISA
    - Fast but lower accuracy (see CUDA Programming Guide for full details)
    - Examples: __sinf(x), __expf(x), __powf(x, y)
  - func(): compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x, y)

- A number of additional intrinsics:
  - __sincosf(), __frcp_rz(), ...
  - Explicit IEEE rounding modes (rz,rn,ru,rd)
Control Flow

• Instructions are issued per 32 threads (warp)
• Divergent branches:
  – Threads within a single warp take different paths
    • if-else, ...
  – Different execution paths within a warp are serialized
• Different warps can execute different code with no impact on performance
• Avoid diverging within a warp
  – Example with divergence:
    • if (threadIdx.x > 2) {...} else {...}
    • Branch granularity < warp size
  – Example without divergence:
    • if (threadIdx.x / WARP_SIZE > 2) {...} else {...}
    • Branch granularity is a whole multiple of warp size
Profiler and Instruction Throughput

• Profiler counts per multiprocessor:
  – Divergent branches
  – Warp serialization
  – Instructions issues

• Visual Profiler derives:
  – Instruction throughput
    • Fraction of SP arithmetic instructions that could have been issued in the same amount of time
      – So, not a good metric for code with DP arithmetic or transcendentals
    – Extrapolated from one multiprocessor to GPU
### Profiler and Instruction Throughput

- Divergent branches
- Warp serialization
- Instructions issues

**Visual Profiler derives:**

- Instruction throughput
  - Fraction of SP arithmetic instructions that could have been issued in the same amount of time
    - So, not a good metric for code with DP arithmetic or transcendentals
  - Extrapolated from one multiprocessor to GPU

<table>
<thead>
<tr>
<th>Method</th>
<th>GPU usec</th>
<th>%GPU time</th>
<th>glob mem read throughput (GB/s)</th>
<th>glob mem write throughput (GB/s)</th>
<th>glob mem overall throughput (GB/s)</th>
<th>Instruction throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>fwd_3D_16x16_order8</td>
<td>3.09352e+06</td>
<td>62.15</td>
<td>46.9465</td>
<td>11.6771</td>
<td>58.6236</td>
<td>0.763973</td>
</tr>
<tr>
<td>memcpyHtoD</td>
<td>5.03094</td>
<td>13.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcpyDtoH</td>
<td>16.8906</td>
<td>4.48</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table

<table>
<thead>
<tr>
<th>Method</th>
<th>GPU usec</th>
<th>%GPU time</th>
<th>glob mem read throughput (GB/s)</th>
<th>glob mem write throughput (GB/s)</th>
<th>glob mem overall throughput (GB/s)</th>
<th>Instruction throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>fwd_3D_16x16_order8</td>
<td>3.09352e+06</td>
<td>62.15</td>
<td>46.9465</td>
<td>11.6771</td>
<td>58.6236</td>
<td>0.763973</td>
</tr>
<tr>
<td>memcpyHtoD</td>
<td>5.03094</td>
<td>13.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcpyDtoH</td>
<td>16.8906</td>
<td>4.48</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shared Memory
Shared Memory

• **Uses:**
  – Inter-thread communication within a block
  – Cache data to reduce redundant global memory accesses
  – Use it to avoid non-coalesced access

• **Organization:**
  – 16 banks, 32-bit wide banks
  – Successive 32-bit words belong to different banks

• **Performance:**
  – 32 bits per bank per 2 clocks per multiprocessor
  – smem accesses are per 16-threads (half-warp)
  – **serialization:** if \( n \) threads (out of 16) access the same bank, \( n \) accesses are executed serially
  – **broadcast:** \( n \) threads access the same word in one fetch
Bank Addressing Examples

- No Bank Conflicts

- No Bank Conflicts
Bank Addressing Examples

- 2-way Bank Conflicts

- 8-way Bank Conflicts
Trick to Assess Impact On Performance

- Change all SMEM reads to the same value
  - All broadcasts = no conflicts
  - Will show how much performance can be improved by eliminating bank conflicts

- The same doesn’t work for SMEM writes
  - So, replace SMEM array indices with `threadIdx.x`
  - Can also be done to the reads
CPU-GPU Interaction
Pinned (non-pageable) memory

• Pinned memory enables:
  – faster PCIe copies (~2x throughput on FSB systems)
  – memcopies asynchronous with CPU
  – memcopies asynchronous with GPU

• Usage
  – cudaHostAlloc / cudaFreeHost
    • instead of malloc / free

• Implication:
  – pinned memory is essentially removed from host virtual memory
Streams and Async API

• Default API:
  – Kernel launches are asynchronous with CPU
  – Memcopies (D2H, H2D) block CPU thread
  – CUDA calls are serialized by the driver

• Streams and async functions provide:
  – Memcopies (D2H, H2D) asynchronous with CPU
  – Ability to concurrently execute a kernel and a memcopy

• Stream = sequence of operations that execute in issue-order on GPU
  – Operations from different streams can be interleaved
  – A kernel and memcopy from different streams can be overlapped
Overlap kernel and memory copy

• **Requirements:**
  – D2H or H2D memcopy from pinned memory
  – Device with compute capability ≥ 1.1 (G84 and later)
  – Kernel and memcopy in different, non-0 streams

• **Code:**
  ```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(...);
```
Call Sequencing for Optimal Overlap

• CUDA calls are dispatched to the hw in the sequence they were issued
• A call is dispatched if both are true:
  – Resources are available
  – Preceding calls in the same stream have completed
• One kernel and one memcpy can be executed concurrently

• Note that if a call blocks, it blocks all other calls of the same type behind it, even in other streams
  – Type is one of { kernel, memcpy }
Stream Examples (current HW)

K1,M1,K2,M2:

K1, K2, M1, M2:

K1, K2, M1, M2:

K1, M1, M2:

K1, M2, M1:

K1, M2, M2:
Summary

• **GPU-CPU interaction:**
  – Minimize CPU/GPU idling, maximize PCIe throughput

• **Global memory:**
  – Maximize throughput (GPU has lots of bandwidth, use it effectively)

• **Kernel Launch Configuration:**
  – Launch enough threads per SM to hide latency
  – Launch enough threadblocks to load the GPU
Summary

• **GPU-CPU interaction:**
  – Minimize CPU/GPU idling, maximize PCIe throughput

• **Global memory:**
  – Maximize throughput (GPU has lots of bandwidth, use it effectively)

• **Kernel Launch Configuration:**
  – Launch enough threads per SM to hide latency
  – Launch enough threadblocks to load the GPU

• **Measure!**
  – Use the Profiler, simple code modifications
  – Compare to theoretical peaks