Fundamental Optimizations
Paulius Micikevicius | NVIDIA
Outline

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

• Optimization of CPU-GPU interaction
  – Maximizing PCIe throughput
  – Overlapping kernel execution with memory copies
Launch Configuration
Launch Configuration

• How many threads/threadblocks to launch?
• Key to understanding:
  – Instructions are issued in order
  – A thread stalls when one of the operands isn’t ready:
    • Memory read by itself doesn’t stall execution
  – Latency is hidden by switching threads
    • GMEM latency: 400-800 cycles
    • Arithmetic latency: 18-22 cycles

• Conclusion:
  – Need enough threads to hide latency
Launch Configuration

Hiding arithmetic latency:
- Need ~18 warps (576) threads per Fermi SM
- Fewer warps for pre-Fermi GPUs (Fermi SM more than doubled issue rate)
- Or, latency can also be hidden with independent instructions from the same warp
  - For example, if instruction never depends on the output of preceding instruction, then only 9 warps are needed, etc.

Maximizing global memory throughput:
- Depends on the access pattern, and word size
- Need enough memory transactions in flight to saturate the bus
  - Independent loads and stores from the same thread
  - Loads and stores from different threads
  - Larger word sizes can also help (float2 is twice the transactions of float, for example)
Maximizing Memory Throughput

• Increment of an array of 64M elements
  – Two accesses per thread (load then store)
  – The two accesses are dependent, so really 1 access per thread at a time
• Tesla C2050, ECC on, theoretical bandwidth: ~120 GB/s

Several independent smaller accesses have the same effect as one larger one.

For example:
Four 32-bit  ~  one 128-bit
Launch Configuration: Summary

• Need enough total threads to keep GPU busy
  – Typically, you’d like 512+ threads per SM
    • More if processing one fp32 element per thread
  – Of course, exceptions exist

• 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

• For more details:
  – Vasily Volkov’s GTC2010 talk “Better Performance at Lower Occupancy”
Global Memory Throughput
Fermi Memory Hierarchy Review
Fermi Memory Hierarchy Review

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

• Shared memory / L1
  – Program configurable: 16KB shared / 48 KB L1 OR 48KB shared / 16KB L1
  – Shared memory is accessible by the threads in the same threadblock
  – Very low latency
  – Very high throughput: 1+ TB/s aggregate

• L2
  – All accesses to global memory go through L2, including copies to/from CPU host

• Global memory
  – Accessible by all threads as well as host (CPU)
  – Higher latency (400-800 cycles)
  – Throughput: up to 177 GB/s
Programming for L1 and L2

• Short answer: DON’T
  – GPU caches are not intended for the same use as CPU caches
    • Smaller size (especially per thread), so not aimed at temporal reuse
    • Intended to smooth out some access patterns, help with spilled registers, etc.
  – Don’t try to block for L1/L2 like you would on CPU
    • You have 100s to 1,000s of run-time scheduled threads hitting the caches
    • If it is possible to block for L1 then block for SMEM
      – Same size, same bandwidth, hw will not evict behind your back

• Optimize as if no caches were there
  – No Fermi-only techniques to learn per se (so, all you know is still good)
  – Some cases will just run faster
Fermi GMEM Operations

- **Two types of loads:**
  - Caching
    - Default mode
    - Attempts to hit in L1, then L2, then GMEM
    - Load granularity is 128-byte line
  - Non-caching
    - Compile with `-Xptxas -dlcm=cg` option to nvcc
    - Attempts to hit in L2, then GMEM
      - Do not hit in L1, invalidate the line if it’s in L1 already
    - Load granularity is 32-bytes

- **Stores:**
  - Invalidate L1, write-back for L2
Load Caching and L1 Size

• Non-caching loads can improve perf when:
  – Loading scattered words or only a part of a warp issues a load
    • Benefit: transaction is smaller, so useful payload is a larger percentage
    • Loading halos, for example
  – Spilling registers (reduce line fighting with spillage)

• Large L1 can improve perf when:
  – Spilling registers (more lines so fewer evictions)
  – Some misaligned, strided access patterns
  – 16-KB L1 / 48-KB smem OR 48-KB L1 / 16-KB smem
    • CUDA call, can be set for the app or per-kernel

• How to use:
  – Just try a 2x2 experiment matrix: \{CA,CG\} x \{48-L1, 16-L1\}
    • Keep the best combination - same as you would with any HW managed cache, including CPUs
Load Operation

- Memory operations are issued per warp (32 threads)
  - Just like all other instructions
  - Prior to Fermi, memory issues were per half-warp

- Operation:
  - Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments
Caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

Memory addresses

© NVIDIA 2010
Non-caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

addresses from a warp

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448

Memory addresses
Caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

addresses from a warp

Memory addresses
Non-caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
Caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within 2 cache-lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%

addresses from a warp

Memory addresses

© NVIDIA 2010
Non-caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
  - Warp needs 128 bytes
  - At most 160 bytes move across the bus
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization
Caching Load

• All threads in a warp request the same 4-byte word
• Addresses fall within a single cache-line
  – Warp needs 4 bytes
  – 128 bytes move across the bus on a miss
  – Bus utilization: 3.125%
Non-caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
  - Warp needs 4 bytes
  - 32 bytes move across the bus on a miss
  - Bus utilization: 12.5%

addresses from a warp

| Memory addresses | 0 | 32 | 64 | 96 | 128 | 160 | 192 | 224 | 256 | 288 | 320 | 352 | 384 | 416 | 448 |
|------------------|---|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within \( N \) cache-lines
  - Warp needs 128 bytes
  - \( N \times 128 \) bytes move across the bus on a miss
  - Bus utilization: \( \frac{128}{(N \times 128)} \)
Non-caching Load

• Warp requests 32 scattered 4-byte words
• Addresses fall within $N$ segments
  – Warp needs 128 bytes
  – $N \times 32$ bytes move across the bus on a miss
  – Bus utilization: $\frac{128}{N \times 32}$
Impact of Address Alignment

- Warps should access aligned regions for maximum memory throughput
  - Fermi L1 can help for misaligned loads if several warps are accessing a contiguous region
  - ECC further significantly reduces misaligned store throughput

Experiment:
- Copy 16MB of floats
- 256 threads/block

Greatest throughput drop:
- GT200: 40%
- Fermi:
  - CA loads: 15%
  - CG loads: 32%
GMEM Optimization Guidelines

• Strive for perfect coalescing per warp
  – Align starting address (may require padding)
  – A warp should access within a contiguous region

• Have enough concurrent accesses to saturate the bus
  – Launch enough threads to maximize throughput
    • Latency is hidden by switching threads (warps)
  – Process several elements per thread
    • Multiple loads get pipelined
    • Indexing calculations can often be reused

• Try L1 and caching configurations to see which one works best
  – Caching vs non-caching loads (compiler option)
  – 16KB vs 48KB L1 (CUDA call)
Shared Memory
Shared Memory

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

• **Fermi organization:**
  – 32 banks, 4-byte wide banks
  – Successive 4-byte words belong to different banks

• **Performance:**
  – 4 bytes per bank per 2 clocks per multiprocessor
  – smem accesses are issued per 32 threads (warp)
    • per 16-threads for GPUs prior to Fermi
  – **serialization:** if $n$ threads in a warp access different 4-byte words in the same bank, $n$ accesses are executed serially
  – **multicast:** $n$ threads access the same word in one fetch
    • Could be different bytes within the same word
    • Prior to Fermi, only broadcast was available, sub-word accesses within the same bank caused serialization
Bank Addressing Examples

- No Bank Conflicts

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7

Thread 31

Bank 31

= No Bank Conflicts

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7

Thread 31

Bank 31
Bank Addressing Examples

• 2-way Bank Conflicts

• 8-way Bank Conflicts
Shared Memory: Avoiding Bank Conflicts

- **32x32** SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)

```
<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>...</th>
<th>Bank 31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>31</td>
</tr>
</tbody>
</table>
```

© NVIDIA 2010
Shared Memory: Avoiding Bank Conflicts

• Add a column for padding:
  – 32x33 SMEM array

• Warp accesses a column:
  – 32 different banks, no bank conflicts
Additional “memories”

• *Texture* and *constant*
• Read-only
• Data resides in global memory
• Read through different caches
Constant Memory

- Ideal for coefficients and other data that is read uniformly by warps
- Data is stored in global memory, read through a constant-cache
  - `__constant__` qualifier in declarations
  - Can only be read by GPU kernels
  - Limited to 64KB

- Fermi adds uniform accesses:
  - Kernel pointer argument qualified with `const`
  - Compiler must determine that all threads in a threadblock will dereference the same address
  - No limit on array size, can use any global memory pointer

- Constant cache throughput:
  - 32 bits per warp per 2 clocks per multiprocessor
  - To be used when all threads in a warp read the same address
    - Serializes otherwise
Constant Memory

- Ideal for coefficients and other data that is read uniformly by warps.
- Data is stored in global memory.
  - `__constant__` qualifier in declarations.
  - Can only be read by GPU kernels.
  - Limited to 64KB.
- Fermi adds uniform access:
  - Kernel pointer argument qualified with `const`.
  - Compiler must determine that all threads in a threadblock will dereference the same address.
  - No limit on array size, can use any global memory pointer.
- Constant cache throughput:
  - 32 bits per warp per 2 clocks per multiprocessor.
  - To be used when all threads in a warp read the same address.
  - Serializes otherwise.

```c
__global__ void kernel( const float *g_a )
{
    ... 
    float x = g_a[15]; /* uniform */
    float y = g_a[blockIdx.x+5]; /* uniform */
    float z = g_a[threadIdx.x]; /* non-uniform */
    ...
}
```
Constant Memory

• Ideal for coefficients and other data that is read uniformly by warps
• Data is stored in global memory, read through a constant-cache
  – `__constant__` qualifier in declarations
  – Can only be read by GPU kernels
  – Limited to 64KB
• Fermi adds uniform accesses:
  – Kernel pointer argument qualified with `const`
  – Compiler must determine that all threads in a threadblock will dereference the same address
  – No limit on array size, can use any global memory pointer
• Constant cache throughput:
  – 32 bits per warp per 2 clocks per multiprocessor
  – To be used when all threads in a warp read the same address
    • Serializes otherwise
Constant Memory

- Kernel executes 10K threads (320 warps) per SM during its lifetime
- All threads access the same 4B word
- Using GMEM:
  - Each warp fetches 32B -> 10KB of bus traffic
  - Caching loads potentially worse - 128B line, very likely to be evicted multiple times
Constant Memory

- Kernel executes 10K threads (320 warps) per SM during its lifetime
- All threads access the same 4B word
- Using constant/uniform access:
  - First warp fetches 32 bytes
  - All others hit in constant cache -> 32 bytes of bus traffic per SM
    - Unlikely to be evicted over kernel lifetime - other loads do not go through this cache

Addresses from a warp

0  32  64  96  128  160  192  224  256  288  320  352  384  416  448
Texture

• Separate cache
• Dedicated texture cache hardware provides:
  – Out-of-bounds index handling
    • clamp or wrap-around
  – Optional interpolation
    • Think: using fp indices for arrays
    • Linear, bilinear, trilinear
      – Interpolation weights are 9-bit
  – Optional format conversion
    • {char, short, int} -> float
  – All of these are “free”
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 \((\text{threadIdx.x}) > 2\) \{...\} else \{...\}
    - Branch granularity < warp size
  - Example without divergence:
    - if \((\text{threadIdx.x} / \text{WARP\_SIZE}) > 2\) \{...\} else \{...\}
    - Branch granularity is a whole multiple of warp size
Control Flow

if ( ... )
{
  // then-clause
}
else
{
  // else-clause
}
Execution within warps is coherent
Execution diverges within a warp
CPU-GPU Interaction
Pinned (non-pageable) memory

• Pinned memory enables:
  – faster PCIe copies
  – 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 may be interleaved
  – A kernel and memcopy from different streams can be overlapped
Overlap kernel and memory copy

• Requirements:
  – D2H or H2D memcpy from pinned memory
  – Device with compute capability $\geq 1.1$ (G84 and later)
  – Kernel and memcpy 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.

- Fermi can concurrently execute:
  - Upto 16 kernels
  - Upto 2 memcopies, as long as they are in different directions (D2H and H2D)

- A call is dispatched if both are true:
  - Resources are available
  - Preceding calls in the same stream have completed

- 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

K1,M1,K2,M2:

K1  K2
M1  M2

K1,K2,M1,M2:

K1  K2  M1  M2

K1,M1,M2:

K1  M1  M2

K1,M2,M1:

K1  M2  M1

K1,M2,M2:

K1  M2  M2

K: kernel  
M: memcpy  
Integer: stread ID
More on Fermi Concurrent Kernels

• Kernels may be executed concurrently if they are issued into different streams

• Scheduling:
  – Kernels are executed in the order in which they were issued
  – Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available
More on Fermi Dual Copy

• Fermi is capable of duplex communication with the host
  – PCIe bus is duplex
  – The two memcopies must be in different streams, different directions

• Not all current host systems can saturate duplex PCIe bandwidth:
  – Likely limitations of the IOH chips
  – If this is important to you, test your host system
Duplex Copy: Experimental Results

10.8 GB/s

- PCIe, x16
  16 GB/s
- QPI, 6.4 GT/s
  25.6 GB/s
- 3xDDR3, 1066 MHz
  25.8 GB/s

7.5 GB/s

© NVIDIA 2010
Duplex Copy: Experimental Results

10.8 GB/s

- PCIe, x16
  - 16 GB/s
- QPI, 6.4 GT/s
  - 25.6 GB/s
- 3xDDR3, 1066 MHz
  - 25.8 GB/s

11 GB/s

- PCIe, x16
  - 16 GB/s
- QPI, 6.4 GT/s
  - 25.6 GB/s
- 3xDDR3, 1066 MHz
  - 25.8 GB/s
Summary

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

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

• Use shared memory when applicable (over 1 TB/s bandwidth)

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

• Use analysis/profiling when optimizing:
  – “Analysis-driven Optimization” talk next
Additional Resources

• **Basics:**
  – CUDA webinars on NVIDIA website (just google for CUDA webinar)
  – CUDA by Example” book by J. Sanders and E. Candrot

• **Profiling, analysis, and optimization for Fermi:**
  – GTC-2010 session 2012: “Analysis-driven Optimization” (tomorrow, 3-5pm)

• **GT200 optimization:**
  – GTC-2009 session 1029 (slides and video)
    • Slides:  
    • Materials for all sessions:

• **CUDA Tutorials at Supercomputing:**

• CUDA Programming Guide
• CUDA Best Practices Guide
Questions?