Outline

- Overview
- Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary
Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it’s better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host
Optimize Memory Access

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory

- Optimize for spatial locality in cached texture memory

- In shared memory, avoid high-degree bank conflicts

- Partition camping
  - When global memory access not evenly distributed amongst partitions
  - Problem-size dependent
Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory
- Use one / a few threads to load / compute data shared by all threads
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalescable addressing
Use Parallelism Efficiently

Partition your computation to keep the GPU multiprocessors equally busy
- Many threads, many thread blocks

Keep resource usage low enough to support multiple active thread blocks per multiprocessor
- Registers, shared memory
Outline

- Overview
- **Hardware**
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary
10-Series Architecture

- 240 thread processors execute kernel threads
- 30 multiprocessors, each contains
  - 8 thread processors
  - One double-precision unit
  - Shared memory enables thread cooperation
Execution Model

Software

- Thread

Hardware

- Thread Processor

- Thread blocks are executed on multiprocessors
- Thread blocks do not migrate
- Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

- Grid

- Device

- Only one kernel can execute on a device at one time
Warps and Half Warps

A thread block consists of 32-thread warps.

A warp is executed physically in parallel (SIMD) on a multiprocessor.

A half-warp of 16 threads can coordinate global memory accesses into a single transaction.
Memory Architecture

Host

- CPU
- Chipset
- DRAM

Device

- DRAM
  - Local
  - Global
  - Constant
  - Texture

GPU

- Multiprocessor
- Multiprocessor
- Multiprocessor
  - Registers
  - Shared Memory
- Constant and Texture Caches
## Memory Architecture

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>All threads in a block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
</tbody>
</table>
Outline

- Overview
- Hardware
- **Memory Optimizations**
  - Data transfers between host and device
  - Device memory optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary
Host-Device Data Transfers

- Device to host memory bandwidth much lower than device to device bandwidth
  - 4GB/s peak (PCI-e x16 Gen 1) vs. 102 GB/s peak (Tesla C1060)

- Minimize transfers
  - Intermediate data can be allocated, operated on, and deallocated without ever copying them to host memory

- Group transfers
  - One large transfer much better than many small ones
Page-Locked Data Transfers

- `cudaMallocHost()` allows allocation of page-locked (“pinned”) host memory

- Enables highest `cudaMemcpy` performance
  - 3.2 GB/s on PCI-e x16 Gen1
  - 5.2 GB/s on PCI-e x16 Gen2

- See the “bandwidthTest” CUDA SDK sample

- Use with caution!
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits
Overlapping Data Transfers and Computation

Async and Stream APIs allow overlap of H2D or D2H data transfers with computation

- CPU computation can overlap data transfers on all CUDA capable devices
- Kernel computation can overlap data transfers on devices with “Concurrent copy and execution” (roughly compute capability >= 1.1)

Stream = sequence of operations that execute in order on GPU

- Operations from different streams can be interleaved
- Stream ID used as argument to async calls and kernel launches
Asynchronous Data Transfers

Asynchronous host-device memory copy returns control immediately to CPU
- cudaMemcpyAsync(dst, src, size, dir, stream);
- requires pinned host memory (allocated with “cudaMallocHost”)

Overlap CPU computation with data transfer
- 0 = default stream

\[
\begin{align*}
\text{cudaMemcpyAsync(a\_d, a\_h, size, cudaMemcpyHostToDevice, 0);} \\
\text{cpuFunction();} \\
\text{cudaThreadSynchronize();} \\
\text{kernel<<<grid, block>>>(dst);} \\
\end{align*}
\]

overlapped
GPU/CPU Synchronization

- **Context based**
  - `cudaThreadSynchronize()`
    - Blocks until all previously issued CUDA calls from a CPU thread complete

- **Stream based**
  - `cudaStreamSynchronize(stream)`
    - Blocks until all CUDA calls issued to given stream complete
  - `cudaStreamQuery(stream)`
    - Indicates whether stream is idle
    - Returns `cudaSuccess`, `cudaErrorNotReady`, ...
    - Does not block CPU thread
GPU/CPU Synchronization

Stream based using events
- Events can be inserted into streams:
  - `cudaEventRecord(event, stream)`
- Event is recorded then GPU reaches it in a stream
  - Recorded = assigned a timestamp (GPU clocktick)
  - Useful for timing

- `cudaEventSynchronize(event)`
  - Blocks until given event is recorded

- `cudaEventQuery(event)`
  - Indicates whether event has recorded
  - Returns `cudaSuccess`, `cudaErrorNotReady`, ...
  - Does not block CPU thread
Overlapping kernel and data transfer

**Requires:**
- "Concurrent copy and execute"
- `deviceOverlap` field of a `cudaDeviceProp` variable
- Kernel and transfer use different, **non-zero** streams
- A CUDA call to stream-0 blocks until all previous calls complete and cannot be overlapped

**Example:**
```c
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, dir, stream1);
kernel<<<grid, block, 0, stream2>>>(…);
cudaStreamSynchronize(stream2);
```
Outline

- Overview
- Hardware
- Memory Optimizations
  - Data Transfers between host and device
  - Device memory optimizations
    - Coalescing
    - Shared memory bank conflicts
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary
Coalescing

Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met.

Depends on compute capability

1.0 and 1.1 have stricter access requirements

Examples – float (32-bit) data

Global Memory

64B aligned segment (16 floats)

128B aligned segment (32 floats)

Half-warp of threads
Coalescing Constraints
Compute capability 1.0 and 1.1

- K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate

Coalesces – 1 transaction

Out of sequence – 16 transactions

Misaligned – 16 transactions
Coalescing Constraints
Compute capability 1.2 and higher

- Coalescing is achieved for any pattern of addresses that fits into a segment of size: 32B for 8-bit words, 64B for 16-bit words, 128B for 32- and 64-bit words

- Smaller transactions may be issued to avoid wasted bandwidth due to unused words
Shared Memory

- Approximately hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Coalescing
    - Shared memory bank conflicts
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary
Shared Memory Architecture

- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Successive 32-bit words assigned to successive banks

- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks

- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized
Bank Addressing Examples

**No Bank Conflicts**
- Linear addressing
  - stride == 1

- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Thread 6
- Thread 7
- Thread 15

- Bank 0
- Bank 1
- Bank 2
- Bank 3
- Bank 4
- Bank 5
- Bank 6
- Bank 7
- Bank 15

**No Bank Conflicts**
- Random 1:1 Permutation

- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Thread 6
- Thread 7
- Thread 15

- Bank 0
- Bank 1
- Bank 2
- Bank 3
- Bank 4
- Bank 5
- Bank 6
- Bank 7
- Bank 15
Bank Addressing Examples

2-way Bank Conflicts
- Linear addressing
  stride == 2

8-way Bank Conflicts
- Linear addressing
  stride == 8
Shared memory bank conflicts

- Shared memory is ~ as fast as registers if there are no bank conflicts

- `warp_serialize` profiler signal reflects conflicts

**The fast case:**
- If all threads of a half-warp access different banks, there is no bank conflict
- If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)

**The slow case:**
- Bank Conflict: multiple threads in the same half-warp access the same bank
- Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank
Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Matrix transpose study
      - Measuring performance - effective bandwidth
      - Coalescing
      - Shared memory bank conflicts
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary
Outline

- Overview
- Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary
Occupancy

- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy.

- **Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently.

- Limited by resource usage:
  - Registers
  - Shared memory
Grid/Block Size Heuristics

- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute

- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren’t waiting at a __syncthreads() keep the hardware busy
  - Subject to resource availability – registers, shared memory

- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations
Register Dependency

Read-after-write register dependency
- Instruction’s result can be read ~11 cycles later
- Scenarios:

<table>
<thead>
<tr>
<th>CUDA</th>
<th>PTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y + 5; )</td>
<td>( \text{add.f32 } $f3, $f1, $f2 )</td>
</tr>
<tr>
<td>( z = x + 3; )</td>
<td>( \text{add.f32 } $f5, $f3, $f4 )</td>
</tr>
<tr>
<td>( \text{s_data}[0] += 3; )</td>
<td>( \text{ld.shared.f32 } $f3, \lfloor r31 + 0 \rfloor )</td>
</tr>
<tr>
<td>( \text{add.f32 } $f3, $f3, $f4 )</td>
<td>( \text{add.f32 } $f3, $f3, $f4 )</td>
</tr>
</tbody>
</table>

To completely hide the latency:
- Run at least 192 threads (6 warps) per multiprocessor
- At least 25% occupancy
- Threads do not have to belong to the same thread block
Register Pressure

- Hide latency by using more threads per SM

Limiting Factors:
- Number of registers per kernel
  - 8K/16K per SM, partitioned among concurrent threads
- Amount of shared memory
  - 16KB per SM, partitioned among concurrent threadblocks

Compile with `--ptxas-options=-v` flag

Use `--maxrregcount=N` flag to NVCC
- \(N\) = desired maximum registers / kernel

At some point “spilling” into local memory may occur
  - Reduces performance – local memory is slow
CUDA GPU Occupancy Calculator

Click here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

### Varying Block Size

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>128</td>
<td>12</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
</tr>
</tbody>
</table>

### Varying Register Count

<table>
<thead>
<tr>
<th>Register Count</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>12</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
</tr>
</tbody>
</table>

### Varying Shared Memory Usage

<table>
<thead>
<tr>
<th>Shared Memory Usage</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>64</td>
<td>20</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td>256</td>
<td>12</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
</tr>
</tbody>
</table>

---

### GPU Occupancy Data is displayed here and in the graphs:

- Active Threads per Multiprocessor: 384
- Active Warps per Multiprocessor: 48
- Active Thread Blocks per Multiprocessor: 2
- Occupancy of each Multiprocessor: 50%
- Maximum Simultaneous Blocks per GPU: 20

### Physical Limits for GPU:

- Multiprocessors per GPU: 16
- Threads/Warp: 32
- Warp/Multiprocessor: 32
- Threads/Multiprocessor: 768
- Thread Blocks / Multiprocessor: 8
- Total # of 32-bit registers / Multiprocessor: 8192
- Shared Memory / Multiprocessor (Bytes): 6352

---

### Allocation Per Thread Block

- Warps: 16
- Registers: 3344
- Shared Memory: 512

---

### CUDA Occupancy Calculator

Version: 1.1
Optimizing threads per block

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
- More threads per block == better memory latency hiding
- But, more threads per block == fewer registers per thread
  - Kernel invocations can fail if too many registers are used

Heuristics

- Minimum: 64 threads per block
  - Only if multiple concurrent blocks
- 128 to 256 threads a better choice
  - Usually still enough regs to compile and invoke successfully
- This all depends on your computation, so experiment!
Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

  **BUT …**

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels

  (It all comes down to arithmetic intensity and available parallelism)
Parameterize Your Application

Parameterization helps adaptation to different GPUs

GPUs vary in many ways
- # of multiprocessors
- Memory bandwidth
- Shared memory size
- Register file size
- Max. threads per block

You can even make apps self-tuning (like FFTW and ATLAS)
- “Experiment” mode discovers and saves optimal configuration
Outline

- Overview
- Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary
CUDA Instruction Performance

- Instruction cycles (per warp) = sum of
  - Operand read cycles
  - Instruction execution cycles
  - Result update cycles

- Therefore instruction throughput depends on
  - Nominal instruction throughput
  - Memory latency
  - Memory bandwidth

- “Cycle” refers to the multiprocessor clock rate
  - 1.3 GHz on the Tesla C1060, for example
Maximizing Instruction Throughput

- Maximize use of high-bandwidth memory
  - Maximize use of shared memory
  - Minimize accesses to global memory
  - Maximize coalescing of global memory accesses

- Optimize performance by overlapping memory accesses with HW computation
  - High arithmetic intensity programs
    - i.e. high ratio of math to memory transactions
  - Many concurrent threads
Arithmetic Instruction Throughput

- **int and float** add, shift, min, max and float mul, mad: 4 cycles per warp
  - int multiply (*) is by default 32-bit
    - requires multiple cycles / warp
  - Use `__mul24() / __umul24()` intrinsics for 4-cycle 24-bit int multiply

- **Integer divide and modulo are more expensive**
  - Compiler will convert literal power-of-2 divides to shifts
    - But we have seen it miss some cases
  - Be explicit in cases where compiler can’t tell that divisor is a power of 2!
  - Useful trick: `foo % n == foo & (n-1)` if n is a power of 2
Arithmetic Instruction Throughput

- The intrinsics reciprocal, reciprocal square root, sin/cos, log, exp prefixed with "__" 16 cycles per warp
  - Examples: __rcp(), __sin(), __exp()

- Other functions are combinations of the above
  - $y / x = rcp(x) \times y$ takes 20 cycles per warp
  - $sqrt(x) = x \times rsqrt(x)$ takes 20 cycles per warp
There are two types of runtime math operations

- `__func()`: direct mapping to hardware ISA
  - Fast but lower accuracy (see prog. guide for details)
  - Examples: `__sin(x)`, `__exp(x)`, `__pow(x,y)`

- `func()`: compile to multiple instructions
  - Slower but higher accuracy (5 ulp or less)
  - Examples: `sin(x)`, `exp(x)`, `pow(x,y)`

The `-use_fast_math` compiler option forces every `func()` to compile to `__func()`
GPU results may not match CPU

- Many variables: hardware, compiler, optimization settings

- CPU operations aren’t strictly limited to 0.5 ulp
  - Sequences of operations can be more accurate due to 80-bit extended precision ALUs

- Floating-point arithmetic is not associative!
FP Math is Not Associative!

- In symbolic math, \((x+y)+z == x+(y+z)\)
- This is not necessarily true for floating-point addition
  - Try \(x = 10^{30}, y = -10^{30}\) and \(z = 1\) in the above equation

- When you parallelize computations, you potentially change the order of operations
- Parallel results may not exactly match sequential results
  - This is not specific to GPU or CUDA – inherent part of parallel execution
Control Flow Instructions

Main performance concern with branching is divergence
- Threads within a single warp take different paths
- Different execution paths must be serialized

Avoid divergence when branch condition is a function of thread ID
- Example with divergence:
  - if (threadIdx.x > 2) { }
  - Branch granularity < warp size
- Example without divergence:
  - if (threadIdx.x / WARP_SIZE > 2) { }
  - Branch granularity is a whole multiple of warp size
Summary

GPU hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:

- Use parallelism efficiently
- Coalesce memory accesses if possible
- Take advantage of shared memory
- Explore other memory spaces
  - Texture
  - Constant
- Reduce bank conflicts
- Avoid partition camping