CUDA Performance Optimization

Patrick Legresley
Optimizations

Kernel optimizations
- Maximizing global memory throughput
- Efficient use of shared memory
- Minimizing divergent warps
- Intrinsic instructions

Optimizations of CPU/GPU interaction
- Maximizing PCIe throughput
- Asynchronous memory copies
Coalescing global memory accesses

A coordinated load/store by a **half-warp** (16 threads)

A contiguous region of global memory
- **64 bytes** - each thread accesses a **32-bit word**: int, float, …
- **128 bytes** - each thread accesses a **double-word**: int2, float2, …
- **256 bytes** - each thread accesses a **quad-word**: int4, float4, …

Additional restrictions
- Starting address for a region must be a multiple of region size
- The $k^{th}$ thread in a half-warp must access the $k^{th}$ element in a block

Exception: not all threads must be participating
- Predicated access, divergence within a half-warp
Coalesced Access: floats

All threads participate

Some threads do not participate
Uncoalesced Access: floats

Permuted access by threads

Misaligned starting address (not a multiple of 64)
Coalesced Access: float3

- **Use shared memory to allow coalescing**
  - Threads read a block of `floats` into shared memory in a coalesced way
  - Need `sizeof(float3) * (threads per block)` bytes of shared memory

- **Processing**
  - Each thread retrieves its `float3` from shared memory
  - Rest of the compute code does not change
Coalescing: Timing Results

**Experiment**
- Kernel: read a float, increment, write back
- 3M floats (12MB)
- Times averaged over 10K runs

**12K blocks x 256 threads reading floats**
- 356\(\mu\)s – coalesced
- 357\(\mu\)s – coalesced, some threads don’t participate
- 3,494\(\mu\)s – permuted/misaligned thread access

**4K blocks x 256 threads reading float3s**
- 3,302\(\mu\)s – float3 uncoalesced
- 359\(\mu\)s – float3 coalesced through shared memory
Coalescing:
Structures of size ≠ 4, 8, or 16 bytes

- Use a Structure of Arrays (SoA) instead of Array of Structures (AoS)

- If SoA is not viable
  - Force structure alignment: __align(X), where X = 4, 8, or 16
  - Use shared memory to achieve coalescing

Point structure

AoS

SoA
Coalescing (Compute 1.2+ GPUs)

- Much improved coalescing capabilities in 10-series architecture
- Hardware combines addresses within a half-warp into one or more aligned segments:
  - 32, 64, or 128 bytes
- All threads with addresses within a segment are serviced with a single memory transaction:
  - Regardless of ordering or alignment within the segment
Compute 1.2+ **Coalesced Access:**
Reading floats

Permuted access by threads

Misaligned starting address (not a multiple of 64)
Compute 1.2+ Coalesced Access: Reading floats

32-byte segment

64-byte segment

Misaligned starting address (not a multiple of 64)

Transaction size recursively reduced to minimize size
Textures in CUDA

Texture is an object for reading data

Benefits
- Data is cached (optimized for 2D locality)
  - Helpful when coalescing is a problem
- Filtering
  - Linear / bilinear / trilinear
  - dedicated hardware
- Wrap modes (for “out-of-bounds” addresses)
  - Clamp to edge / repeat
- Addressable in 1D, 2D, or 3D
  - Using integer or normalized coordinates

Usage
- CPU code binds data to a texture object
- Kernel reads data by calling a fetch function
Texture Addressing

**Wrap**
- Out-of-bounds coordinate is wrapped (modulo arithmetic)

**Clamp**
- Out-of-bounds coordinate is replaced with the closest boundary
Two CUDA Texture Types

- **Bound to linear memory**
  - Global memory address is bound to a texture
  - Only 1D
  - Integer addressing
  - No filtering, no addressing modes

- **Bound to CUDA arrays**
  - CUDA array is bound to a texture
  - 1D, 2D, or 3D
  - Float addressing (size-based or normalized)
  - Filtering
  - Addressing modes (clamping, repeat)

- **Both**
  - Return either element type or normalized float
CUDA Texturing Steps

**Host (CPU) code**
- Allocate/obtain memory (global linear, or CUDA array)
- Create a texture reference object
  - Currently must be at file-scope
- Bind the texture reference to memory/array
- When done:
  - Unbind the texture reference, free resources

**Device (kernel) code**
- Fetch using texture reference
- Linear memory textures
  - `tex1Dfetch()`
- Array textures
  - `tex1D()` or `tex2D()` or `tex3D()`
Global memory optimization

- Coalescing greatly improves throughput
  - Prefer Structures of Arrays over AoS
  - If SoA is not viable, read/write through shared memory
  - Try textures for uncoalescible read patterns

- Batching can help performance
  - A thread reads/writes multiple elements
  - Increases overlap opportunities

- Strive for 50% or higher occupancy
  - Occupancy is number of threads running concurrently divided by maximum number of threads that can run concurrently
Latency is hidden by using many threads per multiprocessor

Factors limiting the number of concurrent threads
- Number of registers
  - 8192 or 16384 per multiprocessor, partitioned among concurrent threads
- Amount of shared memory
  - 16KB per multiprocessor, partitioned among concurrent thread blocks

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

Use `--maxrregcount=N` flag to NVCC
- \( N = \) desired maximum registers / thread
- At some point “spilling” into local memory will occur
  - Reduces performance – local memory is slow (physically in DRAM)
Shared Memory

- Orders of magnitude faster than global memory

- Uses
  - Inter-thread communication within a block
  - Use it to avoid non-coalesced access
    - See “Matrix Transpose” SDK example

- Organization
  - 16 banks
  - Bank width: 32 bits
  - Successive 32-bit words belong to different banks
Shared memory bank conflicts

- **No bank conflicts**
  - all threads in a half-warp access different banks
  - all threads in a half-warp read the same address

- **Bank conflicts**
  - Multiple threads in a half-warp access the same bank
  - Access is serialized
  - Detecting
    - `warp_serialize` profiler signal
    - bank checker macro in the SDK

- **Performance impact**
  - Shared memory intensive apps: up to 30%
  - Little to no impact if performance limited by global memory
Bank Addressing Examples

No Bank Conflicts
- Linear addressing
  - stride == 1

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Bank 1</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Bank 2</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Bank 3</td>
</tr>
<tr>
<td>Thread 4</td>
<td>Bank 4</td>
</tr>
<tr>
<td>Thread 5</td>
<td>Bank 5</td>
</tr>
<tr>
<td>Thread 6</td>
<td>Bank 6</td>
</tr>
<tr>
<td>Thread 7</td>
<td>Bank 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Bank 1</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Bank 2</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Bank 3</td>
</tr>
<tr>
<td>Thread 4</td>
<td>Bank 4</td>
</tr>
<tr>
<td>Thread 5</td>
<td>Bank 5</td>
</tr>
<tr>
<td>Thread 6</td>
<td>Bank 6</td>
</tr>
<tr>
<td>Thread 7</td>
<td>Bank 7</td>
</tr>
</tbody>
</table>

No Bank Conflicts
- Random 1:1 Permutation

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Bank 1</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Bank 2</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Bank 3</td>
</tr>
<tr>
<td>Thread 4</td>
<td>Bank 4</td>
</tr>
<tr>
<td>Thread 5</td>
<td>Bank 5</td>
</tr>
<tr>
<td>Thread 6</td>
<td>Bank 6</td>
</tr>
<tr>
<td>Thread 7</td>
<td>Bank 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Bank 1</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Bank 2</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Bank 3</td>
</tr>
<tr>
<td>Thread 4</td>
<td>Bank 4</td>
</tr>
<tr>
<td>Thread 5</td>
<td>Bank 5</td>
</tr>
<tr>
<td>Thread 6</td>
<td>Bank 6</td>
</tr>
<tr>
<td>Thread 7</td>
<td>Bank 7</td>
</tr>
</tbody>
</table>
Bank Addressing Examples

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

8-way Bank Conflicts
- Linear addressing
- stride == 8
Assessing Memory Performance

- **Global memory**
  - how close to the maximum bandwidth?

- **Textures**
  - throughput alone can be tricky
    - some codes exceed theoretical bus bandwidth (due to cache)
  - how close to the theoretical fetch rate?
    - G80: ~18 billion fetches per second

- **Shared memory**
  - check for bank conflicts -> profiler, bank-macros
Two types of runtime math library 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)

A number of additional intrinsics
- __sincos(), __rcp(), ...
Control Flow

Divergent branches

- Threads within a single warp take different paths
  - if-else, ...
- Different execution paths are serialized

Divergence avoided when branch condition is a function of thread ID

- 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
Grid Size Heuristics

- # of blocks > # of multiprocessors
  - All multiprocessors execute at least one block

- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently on a multiprocessor
  - Blocks that aren’t waiting at a __syncthreads() keep the hardware busy

- # of blocks > 100 to scale to future devices
  - 1000s blocks per grid will scale across multiple generations
Optimizing threads per block

- Thread block size should be a multiple of warp size
  - Avoid wasting computation and on chip resources on under-populated warps

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 128, 192, or 256 threads a better choice
    - Usually still enough registers to compile and invoke successfully
  - This all depends on your computation, so experiment!
Page-Locked System Memory

- Enables highest cudaMemcpy performance
  - 3.2 GB/s on PCI-e x16 Gen1
  - 5.2 GB/s on PCI-e x16 Gen2
  - cudaMallocHost() / cudaFreeHost() calls
    - See the “bandwidthTest” CUDA SDK sample

- Use with caution
  - reduces RAM available to the virtual memory system
Asynchronous memory copy

- Asynchronous host ↔ device memory copy for pinned memory (allocated with `cudaMallocHost` in C) frees up CPU on all CUDA capable devices
- Overlap implemented by using a stream
  - Stream = Sequence of operations that execute in order
  - Stream API
    - `0 = default stream`
    - `cudaMemcpyAsync(dst, src, size, direction, 0);`
Kernel and memory copy overlap

Concurrent execution of a kernel and a host ↔
device memory copy for pinned memory
- Devices with compute capability ≥ 1.1 (G84 and up)
- Overlaps kernel execution in one stream with a memory copy from another stream

Stream API

cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, dir, stream1);
kernel<<<grid, block, 0, stream2>>>(…);
cudaStreamQuery(stream2);