Midterm Review
Logistics

• Quizzes:
  • I decided to keep 15% but to help: I will remove the lowest grade among all quizzes before calculating the final grade.
  • Next quizzes (or quiz) will be just multiple choices, true/false, or super short answer questions

• Lab 2 due Friday May 7th
• Midterm next class
  • computer architecture background, gpu architecture, CUDA Parallelism, Memory coalescing, warp divergence, thread synchronization, Reduction, Scan, and Matrix Multiplication parallel algorithms
Quiz 2 – Question 1

• Allocate
  • `cudaMalloc((void**)&d_img, sizeof(float)*width*height);`
  • Do not determine size like `sizeof(float)*h_img`

• Copy to device
  • `cudaMemcpy(d_img, h_img, sizeof(float)*width*height, cudaMemcpyHostToDevice);`
  • Destination, source, size, direction

• Launch
  • `BlockDim = (32,32,1) – given in question`
  • `gridDim = (ceil(width/32),ceil(height/32),1) – gridDim also needs to be 2D`
  • `ProcessImage<<<gridDim,BlockDim>>>(d_img,height,width)`

• Copy to host
  • `cudaMemcpy(h_img, d_img, sizeof(float)*width*height, cudaMemcpyDeviceToHost);`
Quiz 2 – Question 2

- We have matrix of 80 by 100
- We want to cover it to index it with:
  - A) thread blocks of size 32 by 32
  - B) thread blocks of size 16 by 16

- Performance impact of divergent warps:
  - \( \frac{\text{(# of divergent warps)}}{\text{(# of total warps)}} \times 100 \)
Quiz 2 – Question 3

Implementation 1

Implementation 2
Quiz 2 – Question 3

• Implementation 1 is better for any size
• It has less warp divergence and exhibits memory coalescing
• Implementation 2 every other thread becomes inactive thus has warp divergence after the first phase
• Implementation 1 active threads are contiguous and do not have divergence until the last 5 stages (16, 8, 4, 2, 1)
• First five stages of no divergence only occurs if the size is 1024
• For size 128; the first three steps (128, 64, 32) have no divergence
Quiz 2 – Question 4-9

✗ Functions annotated with the __device__ qualifier may be called on the host or the device

✓ Shared memory in CUDA is accessible by all threads in a single block

✗ Shared memory in CUDA is accessible to both the host and GPU

✗ threads from different block have access to the same shared memory

✓ In the case of warp divergence; all possible execution paths are run by all threads in a warp serially so that thread instructions do not diverge

✓ All thread blocks involved in the same computation use the same kernel
Midterm Review
Computer Architecture

- Threads and processes
  - What they contain and how they relate in hardware and software

- Cache hierarchy
  - Understand the memory gap
  - SW leads to HW design

- Principles of spacial and temporal locality
  - How to write code to apply them
  - HW leads to SW design

- Specialization towards parallel processing

- These are foundational concepts questions will not be explicitly mentioning them but will have implied understanding
**GPU Architecture**

- Warps contain 32 threads and execute on a SIMD unit
- SM Cores contain multiple SIMD Units run entire Thread Blocks
- GPU Contains multiple SMs

<table>
<thead>
<tr>
<th></th>
<th>Scalar</th>
<th>Vector</th>
<th>Core</th>
<th>Card</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware</strong></td>
<td>ALU</td>
<td>ALU</td>
<td>SIMD</td>
<td>SM</td>
</tr>
<tr>
<td>ALU Unit</td>
<td></td>
<td></td>
<td>SM</td>
<td>GPU</td>
</tr>
<tr>
<td><strong>Threads</strong></td>
<td>Thread</td>
<td>Warp</td>
<td>Thread Block</td>
<td>Block Grid</td>
</tr>
<tr>
<td>Memory</td>
<td>Register File</td>
<td>L1 Cache</td>
<td>L2 / Memory</td>
<td></td>
</tr>
<tr>
<td>Address Space</td>
<td>Local per thread</td>
<td>Shared Memory</td>
<td>Global</td>
<td></td>
</tr>
</tbody>
</table>
GPU Architecture

- Hardware constraints
- Limit to number of threads and thread block per SM

Table 2.  Compute Capabilities:  GK180 vs GM200 vs GP100 vs GV100

<table>
<thead>
<tr>
<th>GPU</th>
<th>Kepler GK180</th>
<th>Maxwell GM200</th>
<th>Pascal GP100</th>
<th>Volta GV100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
<td>5.2</td>
<td>6.0</td>
<td>7.0</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps / SM</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / SM</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks / SM</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max 32-bit Registers / SM</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Block</td>
<td>65536</td>
<td>32768</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>255</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Max Thread Block Size</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>FP32 Cores / SM</td>
<td>192</td>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Ratio of SM Registers to FP32 Cores</td>
<td>341</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size / SM</td>
<td>16 KB/32 KB/48 KB</td>
<td>96 KB</td>
<td>64 KB</td>
<td>Configurable up to 96 KB</td>
</tr>
</tbody>
</table>

1 The per-thread program counter (PC) that forms part of the improved SIMT model typically requires two of the register slots per thread.
GPU Architecture

- Hardware constraints examples
- An SM is fully occupied if it is running the maximum number of threads
- 2 blocks with 1024 threads – Fully occupied
- 32 blocks with 32 threads – not fully occupied
- Typically you want the number of threads per block to be divisible by 32 and have at least 64 threads
- Multidimensional blocks get linearized
- Block size of (16,16,4) = 16*16*4 = 1024 threads
CUDA Programming

- Allocate, Copy to Device, Launch, Copy to Host
  - `Cudamemcopy(dest,src,size,direction)`
  - `globalFunction<<<gridDim,BlockDim>>>(args)`

- Allocate and copy data only pointed to by pointers
- Block and Grid size are 3 Dimensional
- Threads are assigned a Thread id and Block id in each dimension
  - Determine proper block and grid size for any input size
  - How to assign data with thread and block ids e.g...
    - `Row = blockIdx.y*blockDim.y + threadIdx.y;`
    - `Col = blockIdx.x*blockDim.x + threadIdx.x;`
Memory coalescing

- When all threads of a warp execute a load instruction, if all accessed locations are contiguous, only one DRAM request will be made and the access is fully coalesced.

<table>
<thead>
<tr>
<th>Coalesced Loads</th>
<th>Un-coalesced Loads</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₀</td>
<td>T₁</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

- When the accessed locations spread across burst section boundaries, coalescing fails and multiple DRAM requests are made.
Memory coalescing

• Be able to spot and modify code to address memory coalescing concerns
• This affect thread access patterns
• Loads across threads access memory contiguously
• Threads read across a row and access down a column
• Or load into shared memory if your access pattern cannot be easily altered
Warp Divergence

- Divergence only occurs when threads within a warp go through different control paths
  - 1) all threads are active
  - 2) All warps have divergence
  - 3) Some threads are inactive but no warp divergence
  - 4) Some warps have divergence
Warp Divergence

- Be able to calculate the number of warps that exhibit divergence for a particular input and block size

- Spot and modify code to reduce the amount of divergence
  - Pad outer bounds with 0 and get rid of any control instructions
  - Resize block or change thread access pattern to land on warp boundaries
  - Compact active threads to contiguous warps (reduction implementation)
Shared memory

Accessing memory is expensive, reduce the number of global memory loads
Shared Memory

Global Memory

On-chip Memory

Thread 1

Thread 2

Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time
Shared Memory

- Declare with __Shared__ var[size]
- Load into shared var then read from it
- Shared memory is only useful if you access it multiple times
- How to use it with tiling
Reduction

- Parallel reduction uses tree algorithm for $O(\log n)$
- Two implementations
  - Understand the difference in implementation and performance
- Understand as an example of warp divergence, memory coalescing, and thread synchronization
Scan
- Parallel scan either strided array or tree algorithm
- Two implementations
  - Understand the difference in implementation and performance
  - Understand as an example of work efficiency and thread synchronization
Tiled Matrix Multiplication

- Great example of tiling algorithm, use of shared memory, and thread synchronization
- Relation between tile size and block size
- Number of tiled phases for any height and width of matrix
- 2D Thread and block ids