CS/EE 217

GPU Architecture and Parallel Programming

Midterm Review

Material on exam

- Lectures 1 to 12 inclusive
- Chapters 3-6, 8 and 9
- Understand the CUDA C programming model
- Understand the architecture limitations and how to navigate them to improve the performance of your code
- Parallel programming patterns.
  - Analyze for run-time, memory performance (global memory traffic; memory coalescing), work efficiency, resource efficiency, …

Review problems

- Problem 3.5 from the book. If we need to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to data index.

Review problems

- Problem 3.6. We want to use each thread to calculate two adjacent elements of a vector addition. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index.

- Assume that the vector length is 2000, and each thread calculates one output element, with a block size of 512. How many threads will there be in the grid?
- How many warps will have divergence?

- 4.4: You need to write a kernel that operates on an image of size 400x900. You would like to allocate one thread to each pixel. You would like the thread blocks to be square and to use the maximum number of threads per block possible on the device (assume the device has compute capability 3.0). How would you select the grid and block dimensions?

- Assuming next that we use blocks of size 16x16, how many warps would experience thread divergence?

- For the simple reduction kernel, if the block size is 1024, how many warps will have thread divergence during the fifth iteration? How many for the improved kernel?

Recall the more efficient reduction kernel

```
for (unsigned int stride = blockDim.x;
     stride > 0;  stride /= 2)
{
  __syncthreads();
  if (t < stride)
     partialSum[t] += partialSum[t+stride];
}
```

- A bright engineer wanted to optimize this kernel by unrolling the last five steps as follows.

```
for (unsigned int stride = blockDim.x;
     stride >= 32;  stride >>= 1) {
  __synchthreads();
  if (t < stride)
     partialSum[t] += partialSum[t+stride];}
__synchthreads();

if(t < 32) {
   partialSum[t]+= partialSum[t+16];
   partialSum[t]+= partialSum[t+8];
   partialSum[t]+= partialSum[t+4];
   partialSum[t]+= partialSum[t+2];
   partialSum[t]+= partialSum[t+1];
}
```

What are they thinking? Will this work? Will performance be better?

- 8.5: Consider performing a 2D convolution on a square matrix of size nxn with a mask of size mxm.
  - How many halo elements will there be?
  - What percentage of the multiplications involves halo elements?
  - What is the saving in memory accesses for an internal tile (no ghost elements) vs. an untiled implementation?
  - Assuming the implementation where every element has a thread to load into shared memory, how many warps will there be per block?

- 9.7: Consider the following array: [4 6 7 1 2 8 5 2]
  - Perform inclusive prefix sum on the array using the work inefficient algorithm. Report the intermediate results at every step.
  - Repeat with the work efficient kernel