



CS/EE 217
Midterm

ANSWER ALL QUESTIONS

TIME ALLOWED 60 MINUTES

Question	Possible Points	Points Scored
1	24	
2	32	
3	20	
4	24	
Total	100	

Question 1] [24 Points]

Given a GPGPU with 14 streaming multiprocessor each supporting up to 1536 threads in up to 8 blocks. Each block can have up to 1024 threads. The GPU has 500 GFLOP peak performance. Assume further that the DRAM system has 8 channels, and can provide up to 150 GB/s.

For any three of the following, explain how it could harm performance and possible ways the program can be modified to reduce this effect. **Please be specific.**

- (a) The application needs to access global memory to get one floating point value for every operation.

How this harms performance: *If 4 bytes are required for every floating point operation, then our performance is limited by the number of bytes which we can read from memory. With 150GB/s, we can read 150/4 or 37.5 billion floating point numbers per second, which means our performance is limited at 37.5 GFlops—much lower than the nominal 500 GFlops.*

Technique/change that could reduce this effect: *Take advantage of locality across threads by loading data into shared memory (e.g., using tiling).*

- (b) Memory accesses are not coalesced.

How this harms performance: *each DRAM access returns 128 bytes taking advantage of bursting and multiple channels. If data is not coalesced, most of these 128 bytes are not used, and we obtain much lower than the maximum memory bandwidth, limiting our performance. A warp that has uncoalesced accesses will have to access memory multiple times to get the necessary data.*

Technique/change that could reduce this effect: *try to change the global memory reference pattern to be coalesced. There are several techniques to do this, but one general technique is tiling. Tiling allows us to load the tile data in a coalesced way into shared memory. After that the algorithm reference pattern is less important since the data is fetched from shared memory.*

- (c) Control divergence

How this harms performance: *control statements (such as if statements) that cause multiple warps to execute different paths within the code. Since there is only one controller (SIMD), it has to sequentially issue the different branches, lowering overall performance.*

Technique/change that could reduce this effect: *Try to map threads to data in a way that each warp has identically behaving threads. We saw many examples of this technique.*

- (a) Suboptimal block size selection in grid decomposition

How this harms performance:

Case 1: if block size is too big, we can end up not taking advantage of all the available threads on an SM because we cannot map multiple blocks to the SM. At a block size of 1024,

we can only map one block to an SM on Fermi. In addition to underutilization, we could end up with poor utilization during barriers since there are no other blocks to schedule from.

Case 2: block sizes are too small, we run into the limit of 8 blocks per SM, with the total number of threads significantly lower than capacity.

Technique/change that could reduce this effect: Picking the right block size would allow us to use all the threads, and have multiple blocks mapped per SM.

Question 2] [32 Points]

We would like to launch a matrix multiplication kernel to multiply an 80X96 matrix A with a 96X40 matrix B with the simple matrix multiplication kernel using 16X16 thread blocks. Answer the following questions:

- (a) (6 points) How many blocks will be launched if each thread is responsible for one element?

*The output matrix size is 80x40. Since we need one thread to compute each output element, this is our grid size. The other dimension (96) affects the number of iterations through the dot product loop, but not the block decomposition. With 16x16 blocks, we need $\text{ceil}(80/16) * \text{ceil}(40/16) = 5x3$ or 15 blocks.*

I worked with your solution if you assumed a different size output matrix.

- (b) (6 points) How many blocks if each thread is responsible for four elements?

*Assuming each thread is responsible for 4 elements arranged 2x2, we need 40x20 threads to cover the matrix. With 16x16 blocks we need $\text{ceil}(40/16) * \text{ceil}(20/16) = 3x2$ or 6 blocks.*

I worked with your solution if you assumed a different configuration of the 4 elements (e.g., 4x1).

- (c) (10 points) Assume we use tiled with case (b) where each thread is responsible for four elements. For a non-boundary tile what is the ratio of global memory accesses in the tiled version to that in the simple matrix multiplication?

We load a tile of size 32x32 data elements for M and the same for N. For each output element we carry out 32 multiplies and 32 adds for a total number of 32x32x64 operations (number of threads x 64 operations per thread). These used to require 32x32x64 memory accesses, and with tiling need only 32x32x2 for a 1/32 reduction in the number of memory accesses.

You could also reach this number by finding the loads per thread; in the tiled implementation each thread loads 8 elements now, 4 from M and 4 from N. It also does 32 adds and 32

multiplies for each of the four output elements it is responsible for. This used to require 64 memory accesses for each of the 4 output elements. So, the reduction is $8/(4 \times 64)$ or $1/32$.

- (d) (10 points) What are the considerations in selecting the tile size for matrix multiplication?

We would like to make the size of the tile bigger to get more global memory access reduction. However, we may be limited by the shared memory size. We also may need too many threads which means the block size is large, and we cannot schedule multiple blocks to the same SM, or hide barrier synchronization delays. So, we want to find a balance between reduction of global memory accesses (bigger tile) and reasonable block sizes that allow multiple blocks to be scheduled to the same SM. Finally, you may want a block size that minimizes boundary effects/thread divergence.

Question 3] [20 points]: Consider a tiled 3D convolution kernel on data of size $512 \times 512 \times 512$. Assume mask size is $5 \times 5 \times 5$. Assume that the size of the output set in the tile is $8 \times 8 \times 8$.

- (a) We considered two implementations that load the full input set into shared memory: the first had one thread per output set element, with some of the threads also helping in loading the halo elements. The second had one thread per input element. How many threads does each configuration need? Are they both feasible?

In the first configuration, we have $8 \times 8 \times 8$ threads per block (matching the output set), which is 512. This configuration is feasible. In the second configuration, we have $12 \times 12 \times 12$ threads per block, matching the input set ($8 + 2$ halo on each side for 12 in each dimension). This total is 1728 which exceeds the maximum block size and the number of threads per block. Both configurations are fine with respect to the size of the required shared memory.

Some solutions computed the total number of threads for the full 3D array. This is simply 512^3 for the first configuration (noting that 1 thread is needed for each output element). For the second configuration, we have 12^3 threads per tile, and $512/8$ tiles in every dimension. So, the total is $512^3 \times 12^3 / 8^3 = 768^3$. This is fine but it is hard to judge feasibility. They both fit in global memory assuming floats.

- (b) What is the reduction in the number of global memory accesses per full tile compared to a non-tiled implementation?

*We load $12 \times 12 \times 12$ per tile and perform $8^3 * 5^3$ operations. These operations require 40^3 memory accesses without tiling, so the reduction ratio is $12^3 / 40^3 = 27/1000$ or approximately 40x reduction.*

Question 4 [24 points]

Work inefficient Prefix Sum kernel

```
__shared__ float XY[BLOCK_SIZE];
int i = blockIdx.x * blockDim.x + threadIdx.x;
//load into shared memory
if (i < InputSize) { XY[threadIdx.x] = X[i];}
//perform iterative scan on XY
for (unsigned int stride = 1; stride <= threadIdx.x; stride
*=2) {
    __syncthreads();
    if(i>=stride)
        in1 = XY[threadIdx.x - stride];
    __syncthreads();
    if(i>=stride)
        XY[threadIdx.x]+=in1;
}
```

Work efficient Prefix Sum kernel

```
// XY[2*BLOCK_SIZE] is in shared memory
for(int stride=1; stride <= BLOCK_SIZE;
stride=stride*2)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];
    stride = stride*2;
    __syncthreads();
}
//post reduction step
for(int stride=1; stride <= BLOCK_SIZE;
stride=stride*2)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];
    stride = stride*2;
    __syncthreads();}
```

I made a mistake in the cut and paste of the work efficient kernel – the post-reduction step code is identical to the reduction step. The question targeted the reduction step, which is correct, but I understand if there is confusion and as a result, am giving free credit for part c that has to do with the work efficient code. If you got it correctly despite my error, you got a bonus.

- (a) Explain work efficiency in the context of the two prefix sum implementations above

Work efficiency refers to the number of operations in the parallel implementation compared to the best sequential implementation. The best sequential implementation is $O(n)$. The work inefficient implementation is $O(n \log n)$ operations while the work efficient one is $\sim 2xn$ or $O(n)$ as well, making it more work efficient.

- (b) For the work inefficient kernel, assuming a BLOCK_SIZE of 1024 threads, how many warps will have control divergence at the step when stride is 16.

All threads < stride are not active. All threads >= stride are active. When stride = 16, the bottom 16 threads are not active. So, the first warp has threads 0 to 15 inactive, while 16-31 are active (control divergence). All other warps have all threads active. Answer: 1 warp with divergence.

- (c) For the work efficient kernel, assume that we have 2048 elements (each block has BLOCK_SIZE=1,024 threads) in each section and warp size is 32, how many warps in each block will have control divergence during the reduction tree phase iteration where stride is 16?

When stride = 16 only the bottom $1/16^{\text{th}}$ of the threads which are 64 threads will be active. These form two consecutive warps because of the remapping of the index, and we have no thread divergence.