CSCI-UA.0480-003

Parallel Computing

Homework Assignment 3

1. [2] Suppose you want to write a kernel that operates on an image of size 400x900 pixels. You also want to assign one GPU thread to each pixel. Your thread blocks are square and you want to use the maximum number of threads per block possible on the device. The maximum number of threads per block is 1024. How would you select the grid dimensions and block dimensions of your kernel?

2. [2] Consider a block with 8 threads executing a section of code before reaching a barrier. The threads require the following amount of time (in micro seconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, 2.9 respectively, and spend the rest of their time waiting for the barrier. What percentage of the threads' summed up execution times (i.e. cumulative time for all threads) is spent waiting for the barrier?

3. [3] What factors can make two threads corresponding to two different warps but of the same block take different amount of time to finish?

4. [2] What is the difference between shared memory and L1 cache?

5. Assume the following piece of code (next page) is running on GPU with the following specs:
Each SM can have up to:

- 8 blocks
- 768 thread
- 8192 registers

a. [1] How many threads are there in total?

b. [1] How many threads are there in a warp?

c. [1] How many threads are there in a block?

d. [1] How many global memory loads and stores are done for each thread?

e. [1] How many accesses to shared memory are done for each block?

f. [3] How many iterations of the for loop (Line 23) will have branch divergence? Show your derivation.

g. [3] Identify an opportunity to significantly reduce the bandwidth requirement on the global memory. How would you achieve this? How many accesses can you eliminate?

```
 1  #define VECTOR_N 1024
 2  #define ELEMENT_N 256
 3  const int DATA_N          = VECTOR_N * ELEMENT_N;
 4  const int DATA_SZ         = DATA_N * sizeof(float);
 5  const int RESULT_SZ       = VECTOR_N * sizeof(float);
    ...
 6  float *d_A, *d_B, *d_C;
    ...
 7  cudaMalloc((void **)&d_A, DATA_SZ);
 8  cudaMalloc((void **)&d_B, DATA_SZ);
 9  cudaMalloc((void **)&d_C, RESULT_SZ);
    ...
10  scalarProd<<<VECTOR_N, ELEMENT_N>>>(d_C, d_A, d_B, ELEMENT_N);
11
12  __global__ void
13  scalarProd(float *d_C, float *d_A, float *d_B, int ElementN)
14  {
15      __shared__ float accumResult[ELEMENT_N];
16      //Current vectors bases
17      float *A = d_A + ElementN * blockIdx.x;
18      float *B = d_B + ElementN * blockIdx.x;
19      int tx = threadIdx.x;
20
21      accumResult[tx] = A[tx] * B[tx];
22
23      for(int stride = ElementN /2; stride > 0; stride >>= 1)
24      {
25          __syncthreads();
26          if(tx < stride)
27          accumResult[tx] += accumResult[stride + tx];
28      }
30      d_C[blockIdx.x] = accumResult[0];
31  }
```