ECE408/CS483 Exam #1, Fall 2016

Tuesday, October 25, 2016

- You are allowed to use any notes, books, papers, and other reference material as you desire.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 120 minutes to complete. To eliminate time pressure and allow for any unforeseen difficulties, we will give everyone 180 minutes.
- This exam is based on lectures as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered up to and including parallel scan.
- You can write down the reasoning behind your answers for possible partial credit.
- **Good luck!**

Question 1 (30 points, suggested time allocation 30 minutes): Short Answer

Answer each of the following questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

(Question about very basic CUDA programming skills)

1. (3 points) If we want to copy host array h_A (h_A is a pointer to element 0 of the source array) to device array d_A (d_A is a pointer to element 0 of the destination array), what would be an appropriate API call for this in CUDA? Assume that array h_A and array d_A each consists of 2,048 single-precision floating elements.

    (A) cudaMemcpy(2048*size(float), h_A, d_A, cudaMemcpyHostToDevice);
    (B) cudaMemcpy(h_A, d_A, 2048, cudaMemcpyDeviceTHost);
    (C) cudaMemcpy(d_A, h_A, 2048, cudaMemcpyHostToDevice);
    (D) cudaMemcpy(d_A, h_A, 2048*size(float), cudaMemcpyHostToDevice);

(Question about 1D kernel launch)

2. (3 points) We want to use each thread to calculate eight (8) output elements of a vector addition. Each thread block process 8*blockDim.x consecutive elements that form 8 sections. All threads in each block will first process a section first with each processing one element. They will then all move to the next section with each processing one element. What would be the kernel code expression for forming the value of i, the data index of the first element to be processed by each thread?

    (A) i=blockIdx.x*blockDim.x + threadIdx.x +2;
    (B) i=blockIdx.x*threadIdx.x*2
    (C) i=blockIdx.x*blockDim.x*8 + threadIdx.x
    (D) i=blockIdx.x*blockDim.x*2 + threadIdx.x*8

3. (3 points) For a vector addition, assume that the vector length is 8000, each thread calculates eight (8) output element, and the thread block size is 512 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?

    (A) 1024
    (B) 8196
    (C) 8192
    (D) 8200

4. (3 points) We are to process an 800X565 (784 pixels in the x or horizontal direction, 565 pixels in the y or vertical direction) picture with the PictureKernel below:

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {

 // Calculate the row # of the d_Pin and d_Pout element to process
 int Row = blockIdx.y*blockDim.y + threadIdx.y;

 // Calculate the column # of the d_Pin and d_Pout element to process
 int Col = blockIdx.x*blockDim.x + threadIdx.x;

 // each thread computes one element of d_Pout if in range
 if ((Row < m) && (Col < n)) {
   d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
 }
}
```

Assume that each block is organized as a 2D 16X16 array of threads. Which of the following statements sets up the kernel configuration properly? Assume that int variable n has value 784 and int variable m has value 565. The kernel is launched with the statement PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, n, m);

(A) dim3 DimGrid(ceil(1, ceil(n/16), ceil(m/16); dim3 DimBlock(1, 16, 16);
(B) dim3 DimGrid(ceil(n/16.0), ceil(m/16.0), 1); dim3 DimBlock(16, 16, 1);
(C) dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1); dim3 DimBlock(16,16,1);
(D) dim3 DimGrid(ceil(m/16), ceil(n/16), 1); dim3 DimBlock(16, 16, 1);

5. In Question 4, how many warps will have control divergence?
   (A) 37*16 + 50
   (B) 38*16
   (C) 49
   (D) 0

(3 points) Write some CUDA kernel code in which the threads in the same block rotate the elements of an array **`list`** in shared memory in-place. That is, element list[i] after rotation should be element list[i-1] before rotation and element list[0] should be element list[255] before You can assume that that each block has exactly 256 threads. Make your code as efficient in time and space as possible.

```
__shared__ float list[256]
```

3. (5 points) For a 1D tiled convolution kernel (Lecture 9), assume that we use a block size of 1024 and a mask_width of 9. What is the average number of times each data element of an internal tile is reused from the Shared Memory?

4. (5 points) Your kernel runs on a device with compute capability 2.0. The kernel resource usage is 30 registers/thread and 5KB of shared memory per block. The kernel has 1,920,000 threads in total and a square grid with dimension 50 on each side. What is the maximum number of simultaneous blocks that will run on a single SM?

5. (5 points) Assume that a kernel has an atomic operation on a variable in the global memory. If you know that each load or store access to the global memory takes 1,000 clock cycles and the clock runs at the 1 GHz. What is the maximal throughput one can hope for the atomic operations on this variable?

6. (5 points) Assume a DRAM system with a burst size of 512 bytes and a peak bandwidth of 240 GB/s. Assume a thread block size of 1024 and warp size of 32 and A as a double-precision array in the global memory. What is the maximal memory bandwidth we can hope to achieve? Explain.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;

double temp = A[8*i];
```

7. (bonus 3 points) List the errors and typos that you reported via Piazza postings, e-mails, or in person communication with Prof. Hwu.

Question 2 (15 points, suggested time allocation 20 minutes): CUDA Basics. For the vector addition kernel and the corresponding kernel launch code, answer each of the sub-questions below.

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    1. int i = threadIdx.x + blockDim.x * blockIdx.x;
    2. if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
   //assume that size has been set to the actual length of
   //arrays A, B, and C
    3.  int size = n * sizeof(float);
    4.
    5.  cudaMalloc((void **) &A_d, size);
    6.  cudaMalloc((void **) &B_d, size);
    7.  cudaMalloc((void **) &C_d, size);
    8.  cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    9.  cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

10. vecAddKernel<<<ceil(n/1024.0),1024>>>(A_d, B_d, C_d, n);

11. cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

}
```

2(a). (2 point) Assume that the size of A, B, and C is 10,000 elements each. How many thread blocks will be generated?

Answer:

2(b). (2 point) Assume that the size of A, B, and C is 10,000 elements each. How many warps are there in each block?

Answer:

2(c) (2 point) Assume that the size of A, B, and C is 10,000 elements. How many threads will be created in the grid?
Answer:

2(d) (4 points) Assume that the size of A, B, and C is 10,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the block number and warp number that causes the control divergence. Explain why or why not.

2(e). (5 points) Assume that the size of A, B, and C is 20,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the line number of the statement that causes the control divergence. Explain why or why not.

Question 3. (15 points, suggested time allocation 20 minutes): The following CUDA code is intended to perform a sum of all elements of the **`partialSum`** array. Assume that the kernel is launched with 1024 threads in each block. First, provide the missing operation, and second estimate the fraction of the iterations of the **`for`** loop that will have branch divergence.

```
__shared__ float partialSum[2048];
unsigned int tid = threadIdx.x;

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

3(a). (2 pts) How many versions of partialSum[2048] will be created for each block?

Answer:

3(b). (2 pts) How many iterations will the for-loop take?

Answer:

3(b). (3 pts) Under what condition will there be control divergence in the if-statement? Assume that the warp size is 32.

Answer:

3(c). (2 pts) How many iterations of the for-loop will have control divergence in the if-statement?

Answer:

3(d). (3 pts) How many warps have control divergence in the for-loop where stride value is 16?

Answer:

3(e). (3 pts) In each block, how many warps have control divergence in the for-iteration where stride value is 4?

Answer:

Question 4. (20 points, suggested time allocation 40 minutes): Consider the following fragment of C code:

```
for(unsigned int x = 0; x < 512; ++x) {
   for(unsigned int y = 0; y < 512; ++y) {
      for(unsigned int z = 0; z < 512; ++z) {
         out[x][y] = out[x][y] <OP> in[x][y][z];
      }
   }
}
```

Explain how you would optimally parallelize this code and why if:
(a) <OP> was not associative
(b) <OP> was associative

You need to state to what you will assign your blocks and your threads to and why, then write out the kernel function code.

Assume out[x][y] is initialized correctly.

Part (a):

Kernel code:

Part (b):

Kernel code:

Question 5. (20 points, suggested allocation of time 30 minutes). This question tests your ability to handle boundary conditions in a tiled matrix multiplication algorithm. For simplicity, we will only handle square matrices.

Consider the following tiled matrix multiplication code:

```
01  #define TILE_WIDTH
02
03  __global__ void sgemm(float* M, float* N, float* P, int Width) {
04
05      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
```

```
06      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
07
08      int bx = blockIdx.x;  int by = blockIdx.y;
09      int tx = threadIdx.x; int ty = threadIdx.y;
10
11      // Identify the row and column of the P element to work on
12      int Row = by * TILE_WIDTH + ty;
13      int Col = bx * TILE_WIDTH + tx;
14
15      float Pvalue = 0;
16      // Loop over the M and N tiles required to compute P element
17      for (int m = 0; m < ____________________; ++m) {
18
19          // Collaborative load of M and N tiles into shared memory
20          if(____________________) {
21              Mds[ty][tx] = M[Row*Width + m*TILE_WIDTH + tx];
22              Nds[ty][tx] = N[(m*TILE_WIDTH + ty)*Width + Col];
23          }
24          __syncthreads();
25
26          if(____________________) {
27              for (int k = 0; k < ____________________; ++k) {
28                  Pvalue += Mds[ty][k] * Nds[k][tx];
29              }
30          }
31          __syncthreads();
32      }
33      P[Row*Width + Col] = Pvalue;
34  }
```

Fill in the missing boundary checks on lines 17, 20, 26, and 27 to make this code run correctly. If you think it is impossible to make this code to run correctly as is, state why, and then make the necessary changes to fix it by inserting a maximum of 6 lines.