408: Applied Parallel Programming

Spring 2019 – Midterm Exam 1

February 26th, 2019

1. This is a closed book exam except for 1 sheet of hand-written notes
2. You may not use any personal electronic devices except for a calculator
3. Absolutely no interaction between students is allowed
4. Illegible answers will likely be graded as incorrect

Good Luck!

Name:____

NetID:____

Exam Room:___________________________________

Question 1 (24 points): _________________

Question 2 (18 points): _________________

Question 3 (30 points): _________________

Question 4 (28 points): _________________

Total Score: _________________

Name: ______________________________ NetID:_____________________

Problem 1 (26 points): Multiple Choice

Choose the proper response, and **if multiple responses are correct, choose all**. No partial credit will be provided if the answer is partially correct, or wrong.

Part 1a (2 points) A particular CUDA device's streaming multiprocessor (SM) can take up to 1536 threads and up to 4 thread blocks. Which of the following block configurations would result in the most number of total threads in the SM?

□ a. 256 threads per block
□ b. 384 threads per block
□ c. 512 threads per block
□ d. 1024 threads per block
□ e. Either (a) or (b) depending on whether block dimension is a power of 2.

Part 1b (2 points) For a vector addition, assume that the vector length is 4000, each thread calculates 10 output elements, and each block contains 64 threads. The programmer configures that kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be created in the grid?

□ 384
□ 448
□ 640
□ 3840
□ 4000
□ 4480
□ None of the above

Part 1c (2 points) A CUDA kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?

□ 1
□ 512
□ 1000
□ 512000
□ None of the above

Part 1d (2 points) Consider the following code in a CUDA kernel.

```
__global__ void do_work(int i, int *A)
{
        int result = 0;
        if (i < 5)
                result = threadIdx.x;

        A[threadIdx.x] = result;
}
```

□ There is control divergence in this code
□ There is no control divergence in this code
□ The control divergence depends on the value of `i`

Part 1e (2 points) Consider the following code in a CUDA kernel.

```
__global__ void do_work(int i, int *A)
{
        int result = 0;

        for (j = 0; j<blockIdx.x; j++)
                result += j;

        A[threadIdx.x] = result;
}
```

□ There is control divergence in this code
□ There is no control divergence in this code
□ Control divergence depends on the number of blocks in the x dimension

Part 1f (2 points) Consider the following code in a CUDA kernel.

```
__global__ void do_work(int i, int *A)
{
        int result = 0;

        for (j = 0; j<threadIdx.x; j++)
                result += j;

        A[threadIdx.x] = result;
}
```

□ There is control divergence in this code
□ There is no control divergence in this code
□ Control divergence depends on the number of threads in the x dimension

Name: ____________________________ NetID:___________________

Part 1g (2 points) Consider the following statements, then select those are correct:

i. All the threads in a CUDA warp execute the same instruction at the same time
ii. Only one block on an SM can use the shared memory in that SM
iii. CUDA constant memory is cached
iv. Memory coalescing is an optimization to maximize memory utilization
v. A `__syncthreads()` call synchronizes across all threads in all blocks

□ ii, iii, iv, v
□ i, iii, iv
□ i. iii, iv, v
□ ii, iii, iv
□ i, ii, iii, iv, v

Part 1h (2 points) Consider a 3D video filtering (convolution) code in CUDA with a 3x3x5 mask, which is stored in constant memory. Shared memory is used to fully store the input tile required for a 16x16x16 output tile. What is the ratio of global memory loads to shared memory accesses for one output tile? For this question, only consider interior tiles with no ghost elements.

□ 16*16*16 to 3*3*5*16*16*16
□ 18*18*20 to 16*16*16
□ 15*15*12 to 16*16*16
□ 15*15*12 to 3*3*5*16*16*16
□ 18*18*20 to 3*3*5*16*16*16
□ None of the above

Part 1i (2 points) Consider 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 that **A** is a float array in the global memory. What is the maximal memory data access throughput we can hope to achieve in the following access to **A**?

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
float temp = A[4*i]+A[4*i+1];
```

□ 240 GB/s
□ 120 GB/s
□ 60 GB/s
□ 30GB/s

Part 1j (2 points) The following vector addition kernel and launch code applies to parts (j) through (l)

```
1  __global__ void vecAddKernel(float* A,float* B,float* C, int n)
2  {
3    int i = threadIdx.x + blockDim.x * blockIdx.x * 2;
4
5    if (i < n) C_d[i] = A_d[i] + B_d[i];
6    i += blockDim.x;
7    if (i < n) C_d[i] = A_d[i] + B_d[i];
8  }
9
10 int vectAdd (float* A, float* B, float* C, int n)
11 {
12   int size = n * sizeof (float);
13   cudaMalloc ((void **)&A_d, size);
14   cudaMalloc ((void **)&B_d, size);
15   cudaMalloc ((void **)&C_d, size);
16   cudaMemcpy (A_d, A, size, cudaMemcpyHostToDevice);
17   cudaMemcpy (B_d, B, size, cudaMemcpyHostToDevice);
18
19   vecAddKernel<<<ceil(n/1024.0), 512>>> (A_d, B_d, C_d, n);
20   cudaMemcpy (C, C_d, size, cudaMemcpyDeviceToHost);
21 }
```

If the size of the vectors is 50,000 elements, identify the block number(s) that will have control divergence. (Assume the index is starting from 0)

□ Block 47
□ Block 48
□ Both A and B
□ Neither A or B

Part 1k (2 points) If the size of the vectors is 50,000 elements, which lines in the kernel code will experience control divergence?

□ line 5
□ line 7
□ both line 5 and line 7
□ None of the above

Part 1l (2 points) Again, consider the vector add kernel from part j. If the size of the vectors is *num* elements, identify the number of warps that will have control divergence.

□ 0 or 1
□ 1 or 2
□ 0 or 2
□ ceil(*num*/1024)
□ None of the above

Name: ___________________________ NetID:__________________

Problem 2 (18 points): Matrix Multiply

Following is part of a tiled 2D matrix multiplication CUDA kernel, similar to the one in MP3. However, instead of calculating a single element, each thread calculates a 2x2 section of the output matrix. Adjacent threads calculate adjacent sections. For example, thread (0, 0) in the block (0, 0) would calculate the (0, 0), (0, 1), (1, 0), (1, 1) elements in the output matrix. Likewise thread (1, 0) in the block (0, 0) would calculate the (2, 0), (2, 1), (3, 0), (3, 1) elements in the output matrix. When loading the tiles into the shared memory, each thread in the block will also load a 2x2 section of the corresponding tile.

```
1.  #define BLOCK_WIDTH 8
2.  #define TILE_WIDTH 16
3.  __global__
4.  void matrixMultiplyShared(float *M, float *N, float *P,
5.                            int numMRows, int numMColumns, int numNRows,
6.                            int numNColumns, int numPRows, int numPColumns)
7.  {
8.    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
9.    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
10.
11.   int bx = blockIdx.x;  int by = blockIdx.y;
12.   int tx = threadIdx.x; int ty = threadIdx.y;
13.
14.   // Identify the row and column of the first P element to work on
15.   int Row = by * TILE_WIDTH + ty * 2;
16.   int Col = bx * TILE_WIDTH + tx * 2;
17.
18.   // Thread-local array to store 2x2 result of P
19.   float Pvalue[2][2];
20.   for (int j = 0; j < 2; ++j)
21.     for (int i = 0; i < 2; ++i)
22.       Pvalue[j][i] = 0;
23.
24.   // number of iterations needed to loop over all tiles required
25.   int ph_count = ceil(numMColumns/(float)TILE_WIDTH);
26.
27.   // loop over the M and N tiles required to compute the P elements
28.   for (int ph = 0; ph < ph_count; ++ph) {
29.     // Collaborative loading of M and N tiles into shared memory
30.     for (int j = 0; j < 2; ++j)
31.       for (int i = 0; i < 2; ++i){
32.         if ((Row+j) < numMRows && (ph*TILE_WIDTH+tx*2+i) < numMColumns)
33.           Mds[ty*2+j][tx*2+i] = M[(Row+j)*numMColumns+ph*TILE_WIDTH+tx*2+i];
34.         else
35.           Mds[ty*2+j][tx*2+i] = 0;
36.         if ((ph*TILE_WIDTH+ty*2+j) < numNRows && (Col+i) < numNColumns)
37.           Nds[ty*2+j][tx*2+i] = N[(ph*TILE_WIDTH+ty*2+j)*numNColumns+Col+i];
38.         else
39.           Nds[ty*2+j][tx*2+i] = 0;
40.       }
41.     __syncthreads();
42.
```

```
    // Calculate partial dot product for 2x2 array per thread
    for (int k = 0; k < TILE_WIDTH; ++k) {
      for (int j = 0; j < 2; ++j)
        for (int i = 0; i < 2; ++i)

          Pvalue[j][i] += Mds[__________][__________] * Nds[__________][__________];
    }
    __syncthreads();
  } // ph loop

  for (int j = 0; j < 2; ++j)
    for (int i = 0; i < 2; ++i)
      if (Row + j < numPRows && Col + i < numPColumns)
        P[(Row + j) * numNColumns + Col + i] = Pvalue[j][i];
}

// Below are some host code to calculate grid_dim and block_dim

dim3 grid_dim(________________________________,________________________________, 1);
dim3 block_dim(BLOCK_WIDTH, BLOCK_WIDTH, 1);
```

Part 2a (12 points) Fill in the missing code in the 6 blanks above

Part 2b (2 points) Suppose we are using the above code to multiply two matrices M and N, where M is a 80 * 104 matrix and N is a 104* 80 matrix (here, an m * n matrix means a matrix with m elements in the y or vertical direction and n elements in the x or horizontal direction), how many warps in the whole grid will have control divergence during the loading of **Mds**? (Warp size is 32 threads)

□ 0
□ 25
□ 50
□ 100
□ None of the above

Part 2c (2 points) How many floating point operations (ADDs and MULTs) will each thread perform?

□ 2 * numMColumns ADDs and 2 * numMColumns MULTs
□ 2 * numNColumns ADDs and 2 * numNColumns MULTs
□ 4 * numMColumns ADDs and 4 * numMColumns MULTs
□ 4 * numNColumns ADDs and 4 * numNColumns MULTs
□ None of the above

Part 2d (2 points) How many loads from global memory will each thread perform?

□ ph_count * 2
□ ph_count * 4
□ ph_count * 8
□ ph_count * 16
□ None of the above

Problem 3 (30 points): Separable 2D Convolution

Under certain conditions, the 2D mask of a 2D convolution can be decomposed to two 1-D masks as shown in the below example. This is called a separable convolution.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} * A$$

For convolutions that are separable, we can apply the two 1-D masks sequentially to the input matrix: first apply the horizontal 1-D mask to each element, generating an intermediate result. The apply the vertical 1-D mask on the intermediate result to generate the output matrix. In the following questions, you will work with kernel code that uses shared memory tiles to compute a separable 2D convolution. It uses Strategy 1 to load a tile's worth of the input to shared memory, then it computes the horizontal 1-D convolution of size MASK_WIDTHx1, and then the vertical convolution of 1xMASK_WIDTH. In the code, **`mask1`** is the horizontal mask, **`mask2`** is the vertical mask.

Part 3a (2 points): Consider the following input array A, and a 3x3 mask, decomposed into two 1D masks

$$mask = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix},\ decomposed\ masks = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}, \text{A} = \begin{bmatrix} 3 & 2 & 5 & 8 & 1 & 4 \\ 2 & 4 & 7 & 1 & 2 & 6 \\ 6 & 9 & 6 & 5 & 9 & 1 \\ 0 & 4 & 7 & 2 & 8 & 5 \\ 9 & 3 & 1 & 3 & 4 & 8 \\ 7 & 4 & 9 & 2 & 1 & 4 \end{bmatrix}$$

Assume we are using a CUDA kernel has an output tile width of 2. Calculate values for tile (1,1), i.e., tile= $\begin{bmatrix} 6 & 5 \\ 7 & 2 \end{bmatrix}$ after the horizontal mask = $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$ is applied.

Part 3b (2 points): Calculate the output values for tile (1, 1). That is, apply the vertical mask to the results from **part 3a**. Hint: ensure that all values used for this convolution have been first convolved by the horizontal mask.

Name: ___________________________ NetID:__________________

Part 3c (20 points) In the following questions, you will work with kernel code that uses shared memory tiles to compute a separable 2D convolution by applying two 1-D convolutions. It uses Strategy 1 to load a tile's worth of the input to shared memory, then it computes the horizontal 1-D convolution of size MASK_WIDTHx1, and then the vertical convolution of 1xMASK_WIDTH. In the code, **`mask1`** is the horizontal mask, **`mask2`** is the vertical mask.Fill in the blanks in the code to complete the kernel. There are 10 blanks in total.

```
#define MASK_WIDTH 5
#define MASK_RADIUS 2
#define INPUT_TILE_WIDTH ___
#define OUTPUT_TILE_WIDTH 8


__constant__ float mask1[MASK_WIDTH]; //horizontal mask
__constant__ float mask2[MASK_WIDTH]; //vertical mask
__global__
void Separable2DConv(float *input, float *output, int x_size, int y_size)
{
  __shared__ float input_tile[INPUT_TILE_WIDTH][INPUT_TILE_WIDTH];
  int tx = threadIdx.x; int ty = threadIdx.y;
  int bx = blockIdx.x; int by = blockIdx.y;

  //output index
  int row_o = by * OUTPUT_TILE_WIDTH + ty;
  int col_o = bx * OUTPUT_TILE_WIDTH + tx;

  // load input tile into shared memory
  int num_iters = _______________________________________________
  for(int i = 0; i< num_iters; i++){
    for(int j = 0; j<num_iters; j++){
      int row_i = _____________________________________________

      int col_i = _____________________________________________
      int tile_y_idx = ty + i* OUTPUT_TILE_WIDTH;
      int tile_x_idx = tx + j* OUTPUT_TILE_WIDTH;
      if(tile_y_idx < INPUT_TILE_WIDTH && tile_x_idx < INPUT_TILE_WIDTH){
        if(row_i>= 0 && row_i < y_size && col_i >= 0 && col_i < x_size)
          input_tile[tile_y_idx][tile_x_idx] = input[row_i*x_size +col_i];
        else
          input_tile[tile_y_idx][tile_x_idx] = 0;
      }
    }
  }
  __syncthreads();

  float val;
  for(int iter = 0; iter < num_iters; iter++){
    val = 0.0;
    int y_index = ty + iter*OUTPUT_TILE_WIDTH;
    for(int k =0; k< MASK_WIDTH; k++)
      if( y_index < INPUT_TILE_WIDTH)
        val += mask1[k] * input_tile[_______][____];
    if( y_index < INPUT_TILE_WIDTH )
47.
48.       input_tile[________][__] = val;
49.    }
50.    val = 0.0;
51.    for(int k =0; k< MASK_WIDTH; k++){
52.      val += mask2[k] * input_tile[____][__]  ;
53.    }
54.    if(row_o < y_size && col_o < x_size)
55.      output[row_o * x_size + col_o] = val;
56. }
57.
58.
```

Part 3d (4 points) The code above is incorrect in that it lacks synchronization. Please specify where **__syncthreads()** is required for correct execution, by stating which line numbers in the code the **__syncthreads()** should appear after. For the sake of efficiency, we want to execute as few **__syncthreads()** as possible. Hint: more than one **__syncthreads()** is required.

Part 3e (2 points) Provide 1 possible advantage and 1 possible disadvantages of using separable masks over a standard 2D convolution?

Name: ______________________________ NetID:____________________

Problem 4 (28 points): Machine Learning

Part 4a (6 points): Choose the proper responses for the questions below. No partial credit will be provided if the answer is partially correct, or wrong.

1. Mark **all** statements below that are true.

   □ A multi-layer perceptron can learn the XOR function
   □ A single-layer perceptron can learn the XOR function
   □ A convolutional layer has a smaller receptive field than a fully connected layer
   □ The learning process involves finding weights and biases that minimize the loss function

2. Mini-batch stochastic gradient descent generally converges to the optimized point faster than batched stochastic gradient descent.

   □ True
   □ False

3. Unlike classical machine learning algorithms, deep learning does not require hand-designed features.

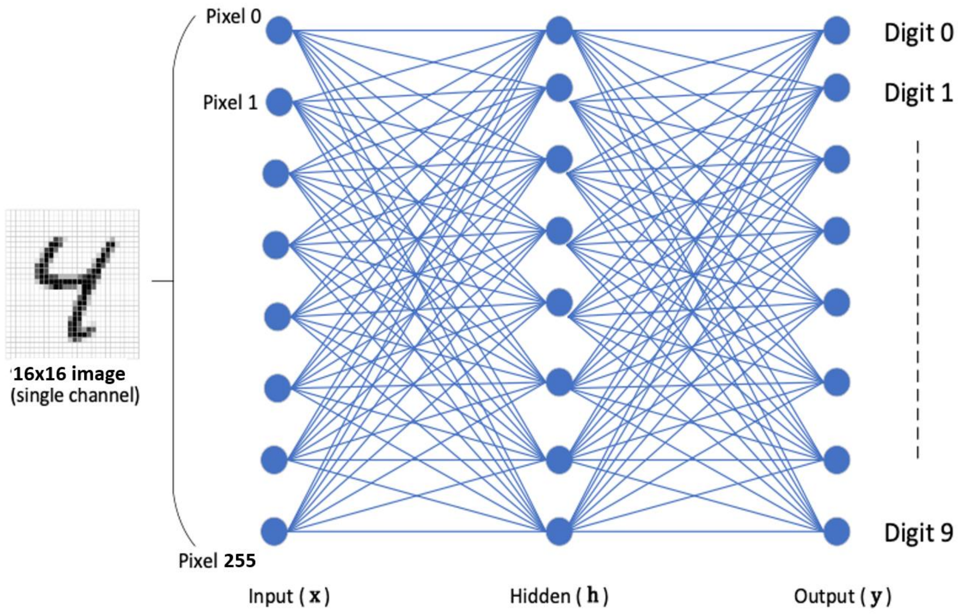
   □ True
   □ False

Name: ______________________________ NetID:____________________

Part 4b (8 points): You are required to implement optimized multi-layer perceptron with 3 layers as shown in the figure. (NOTE: Image is not scaled to dimension). It takes input $\boldsymbol{x}$ gray scale image of size 16x16 and has 10 classes for output $\boldsymbol{y}$, each representing a digit. The inputs and outputs are represented as linearized vectors, $\boldsymbol{x}$ and $\boldsymbol{y}$. The hidden layer $\boldsymbol{h}$ has 100 neurons in it. The overall equation of the model can be given by

$$\boldsymbol{h} = \boldsymbol{\sigma}(\boldsymbol{W_1 x} + \boldsymbol{b_1})$$
$$\boldsymbol{y} = \boldsymbol{\sigma}(\boldsymbol{W_2 h} + \boldsymbol{b_2})$$

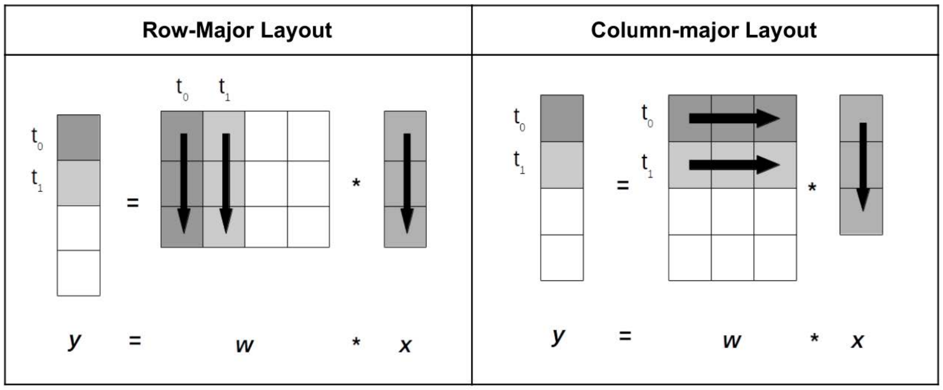
Where $\boldsymbol{b_1}$ and $\boldsymbol{b_2}$ are vectors holding the bias values, and $\boldsymbol{W_1}$ and $\boldsymbol{W_2}$ are weight matrices, and the function $\boldsymbol{\sigma}$ is the sigmoid function.



Fill in the dimensions in the following table, based on the architecture of the network:

Q1	Dimension of $\boldsymbol{b_1}$	[____, 1]
Q2	Dimension of $\boldsymbol{b_2}$	[___, 1]
Q3	Dimension of $\boldsymbol{W_1}$	[____, ____]
Q4	Dimension of $\boldsymbol{W_2}$	[___, ____]

Part 4c (8 points): You realize the both forward-pass equations (for ***h*** and ***y***) are the same computation, but with different input dimensions. You want to use a single GPU kernel general enough to perform both. You can disregard the sigmoid function ***σ*** for this question. Please complete below code base to complete the implementation in column-major layout weight matrix. The figure below shows the thread access pattern for the column major layout in a 4x3 example.



```
__global__
void fc_col(float *y, const float *x, const float *w, const float *b, const
 int ySize, const int xSize) {

   int tx = blockDim.x * blockIdx.x + threadIdx.x;
   int gx = gridDim.x * blockDim.x;

   for( int o = tx; o < ySize; o += gx){
      float sum =0;
      for( int i =0; i< xSize; i++){

               sum += x[_] * w[_______________];
       }

       y[_] = sum + b[_];
   }
}
```

Part 4d (2 points): Based on the code in 4c, are the weight matrix accesses coalesced?

Name: ______________________________ NetID:_____________________

Part 4e (4 points): Your partner for ECE 408 optimized the code in 4c by using shared memory for the input matrix (x). However, his implementation has bugs. What changes do you need to make on the code below to make it correct? You can modify, add, or delete lines in the code. Please note that you may not need all the empty lines below and overly complex answers will result in lost points. Assume all the indices with $ means the same indices you answered in 4c.

```
1.  __global__
2.  void fc_col_shared(float *y, const float *x, const float *w, const float *b,
    const int ySize, const int xSize) {
3.
4.      __shared__ x_shared[xSize];
5.
6.      int tx = blockDim.x * blockIdx.x + threadIdx.x;
7.      int gx = gridDim.x * blockDim.x;
8.
9.      for( int s = tx; s < xSize; s += gx){
10.       if (s < xSize)
11.         x_shared[s] = x[s];
12.       else
13.         x_shared[s] = 0;
14.     }
15.
16.     for( int o = tx; o < ySize; o += gx){
17.       float sum =0;
18.       for( int i =0; i< xSize; i++){
19.
20.         sum += x_shared[$] * w[$];
21.       }
22.
23.       y[$] = sum + b[$];
24.    }
25.  }
```

Example:
[Add between line 6 and line 7]: `int bx = blockIdx.x;`

[ ]:

[ ]:

[_____________________________]: ____.

[_____________________________]: ____.