ECE408/CS483/CSE408 Exam #1, Spring 2018

Tuesday, February 27, 2018

- You are allowed one 8.0x11.5 cheat sheet with notes on both sides. The minimal font size for your text on the cheat sheet should be 8pts.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 155 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, textbook chapters, as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered up to and including Deep Learning.
- You can write down the reasoning behind your answers for possible partial credit.
- You must write your answers with pen, not pencil, in order to request regrade.
- **Good luck!**

Name: ___________________________

Netid: __________________________

UIN: _________________________

Question 1: ______________________

Question 2: ______________________

Question 3: ______________________

Question 4: ______________________

Question 5: ______________________

Question 1 (27 points, suggested time allocation 40 minutes): multiple-choice and short-answer questions. If you get more than 27 points by answering all questions (1-8), your score will saturate at 27 points.

For multiple-choice questions, give a concise explanation for your answer for possible partial credit. Answer each of the short-answer questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

1. (4 points) If we want to allocate an array of `float` in the GPU **global memory** and have the pointer variable `device_array` to point to the array, what is the correct call to `cudamalloc()` on the host side?

```
// pointers to host & device arrays
float *device_array;
int size = num_bytes * sizeof(float);
```

(A) `cudaMalloc(device_array, size);`
(B) `cudaMalloc((void *)device_array, size);`
(C) `cudaMalloc((void *)&device_array, size);`
(D) `cudaMalloc((void **)&device_array, size);`

2. (4 points) If we want to copy an array of `float` from the host memory to the GPU **constant memory** and have the pointer variable `device_array` to point to the array, what is the correct call on the host side?

```
// pointers to host & device arrays
__constant__ float device_array[100];

float host[100]; // Assume that host array is already filled with data
int size = 100 * sizeof(float);
```

(A) `cudaMemcpy(device_array, host, size, cudaMemcpyHostToDevice);`
(B) `cudaMemcpyToSymbol(device_array, host, size, 0,`
`cudaMemcpyHostToDevice);`
(C) `cudaMemcpy(host, device_array, size);`
(D) `cudaMemcpyToSymbol(host, device_array, size);`

3. (4 points) We want to use each thread to calculate four (4) output elements of a vector addition. Each thread block processes four sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, with each thread processing one more element. This repeats until all four sections are processed. 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 of the first element?

```
(A) i=blockIdx.x*blockDim.x*4 + threadIdx.x;
(B) i=blockIdx.x*threadIdx.x*2;
(C) i=(blockIdx.x*blockDim.x + threadIdx.x)*4;
(D) i=blockIdx.x*blockDim.x*2 + threadIdx.x;
```

4. (4 points) You would like to run a kernel on a GPU device with compute capability 6.0. There are a total of 1,000,000 threads needed to launch the kernel. The kernel uses 10 registers per thread and 10KB shared memory per block. The kernel is launched as a cubic grid of cubic blocks. The grid dimension is 10 in X, Y and Z direction. What is the maximum number of simultaneous blocks that will run on a single SM?
   (A) 1
   (B) 2
   (C) 3
   (D) 6

Technical specifications of CUDA compute capability 6.0 below. Note that you might not need

all information from this table.

Maximum number of resident grids per device (Concurrent Kernel Execution)	3
Maximum number of threads per block	1024
Maximum number of resident blocks per multiprocessor	32
Maximum number of resident warps per multiprocessor	64
Maximum number of resident threads per multiprocessor	2048
Number of 32-bit registers per multiprocessor	64K

Maximum amount of shared memory per multiprocessor	64KB
Maximum amount of shared memory per thread block	48KB

5. (4 points) For a tiled 2D convolution kernel with 30x30 output tiles and 3x3 mask (and thus 32x32 input tile), how many warps in each thread block have control divergence? (Assume strategy 2: **block size covers input tile**)

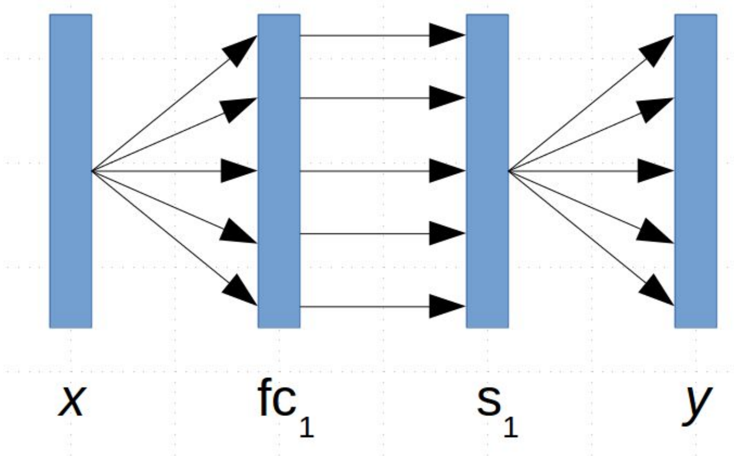
(A) 3
(B) 30
(C) 32
(D) 30*30

6. (4 points) How do we declare a 5x5 constant memory float array Mc?

(A) const float[5][5] Mc; in the kernel that needs to use Mc
(B) const float Mc[5][5]; outside all kernels
(C) __const__ float[5][5] Mc; outside all kernels.
(D) __const__ float Mc[5][5]; outside all kernels.

7. (4 points) True or false: A single perceptron can learn the XOR function. Explain

8. (5 points) The following figure represents a feed-forward multi-layer network operating on input x, with a fully-connected layer, a sigmoid layer, and a fully connected layer, to produce an output y. Let w_1, b_1 and w_2, b_2 be the weight matrices and bias vectors for the first and second fully-connected layers, respectively. In the figure below, fc_1 is the output of the first fully-connected layer, and s_1 is the output of the sigmoid layer



Recall that for a fully-connected layer, the output vector *o* is related to the input vector *i*

o = W * *i* + b

And for a sigmoid layer,

o = σ(*i*)

Note that for the sigmoid layer, $\frac{\partial \sigma(x)}{\partial x} = \sigma(x) * (1 - \sigma(x))$

Using the chain rule, show that if the error gradient at y is $\frac{\partial E}{\partial y}$, then the gradient of the error with respect to b_1 ,i.e., $\frac{\partial E}{\partial b_1}$, is $\frac{\partial E}{\partial y} * W_1 * \sigma(fc_1) * (1 - \sigma(fc_1))$

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

Kernel Code:

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConvertion(unsigned char *grayImage, unsigned char
*rgbImage, int width, int height) {
   int Col = threadIdx.x + blockIdx.x * blockDim.x;
   int Row = threadIdx.y + blockIdx.y * blockDim.y;
   if (Col < width && Row < height) {
         // get 1D coordinate for the grayscale image
     int greyOffset = Row*width + Col;
   // one can think of the RGB image having
         // CHANNEL times columns of the gray scale image
     int rgbOffset = greyOffset*CHANNELS;
     unsigned char r = rgbImage[rgbOffset    ]; // red value for pixel
     unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
     unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
   // perform the rescaling and store it
         // We multiply by floating point constants
     grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
   }
}
```

Host Code:

```
      // the following host code is buggy
      // you need to fix the bugs in question 2(a)
dim3 dimGrid(ceil(height/64), ceil(width/32), 1);
dim3 dimBlock(64, 32, 1);
colorToGreyscaleConvertion<<<dimGrid, dimBlock>>>(d_grayImage, d_rgbImage,
width, height);
```

2(a). (6 points) Assume the above kernel code is correctly implemented. Based on the kernel code, there are three bugs in the host code which launches the kernel. Please clearly state where the bugs are and state how to fix these bugs.

Bug 1:

Fix 1:

Bug 2:

Fix 2:

Bug 3:

Fix 3:

2(b). (3 points) Assume that the height of the grey image is 600 pixels and the width is 800 pixels. Instead of the settings in 2(a), assume that we decide to use a grid of 16X16 blocks. That is, each block is organized as a 2D 16X16 array of threads. How many warps will be generated during the execution of the kernel? **Please show your work.** (Hint: Draw a picture on how pixels are covered by thread blocks.)

2(c). (3 points) Based on 2(b), how many warps will have control divergence? **Please show your work.**

2(d). (3 points) Based on 2(b), if we now have a grey image whose height is 800 pixels and the width is 600 pixels, how many warps will have control divergence? (Assume that we decide to use a grid of 16X16 blocks. That is, each block is organized as a 2D 16X16 array of threads.) **Please show your work.**

Question 3. (14 points, suggested time allocation 20 minutes): Deep Learning.

You are implementing a fully-connected neural-network layer. It takes an input vector ***x***, multiplies it by a weight matrix ***w***, adds a bias vector ***b***, and produces an output vector ***y***. ***w*** [i,j] corresponds to how the i-th element of the input affects the j-th element of the output.

Your layer will be used as the first fully-connected layer in a network that reads in 28x28 input image data treated as a linearized input vector and produces an output vector of 500 values.

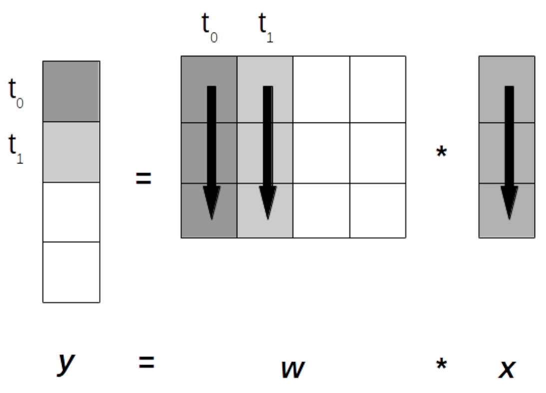
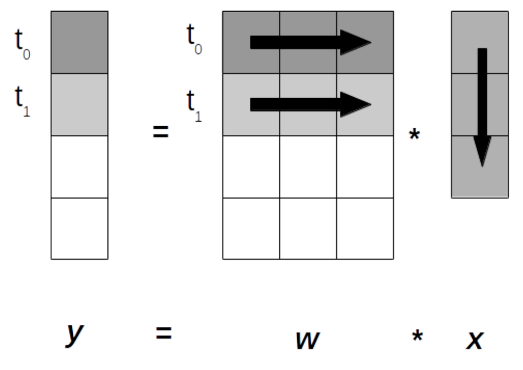
3(a). (2 points) Recall our formulation for a fully-connected layer as a matrix-vector multiplication:

$$\boldsymbol{y} = \boldsymbol{w} * \boldsymbol{x} + \boldsymbol{b}$$

Fill in the following table with the vector and matrix dimensions

Data	**Dimensions**
x	
y	
w	
b	

3(b). (4 points) You have two different implementations of the layer. In the first, the weight matrix data is stored in row-major order, and in the second, in column-major order. The figure below shows the thread access pattern for each storage type in a 4x3 example.

Row-Major Layout	Column-major Layout
	

Fill in the blanks in the following implementation of the fully-connected layer for row-major weight matrix.

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

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

  for (int o = tx; o < ySize; o += gx) {
    float acc = 0;
    for (int i = 0; i < xSize; ++i) {
      acc += x[                ] * w[                ];
    }
    y[                ] = acc + b[                ];
  }
}
```

3(c). (4 points) Fill in the blanks in the following implementation of the fully-connected layer for col-major weight matrix.

```
__global__ void fc_col(float *y, const float *x,
                          const float *w, const float *b,
```

```
                   const int ySize, const int xSize) {

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

  for (int o = tx; o < ySize; o += gx) {
    float acc = 0;
    for (int i = 0; i < xSize; ++i) {
      acc += x[                ] * w[                ];
    }
    y[                ] = acc + b[                ];
  }
}
```

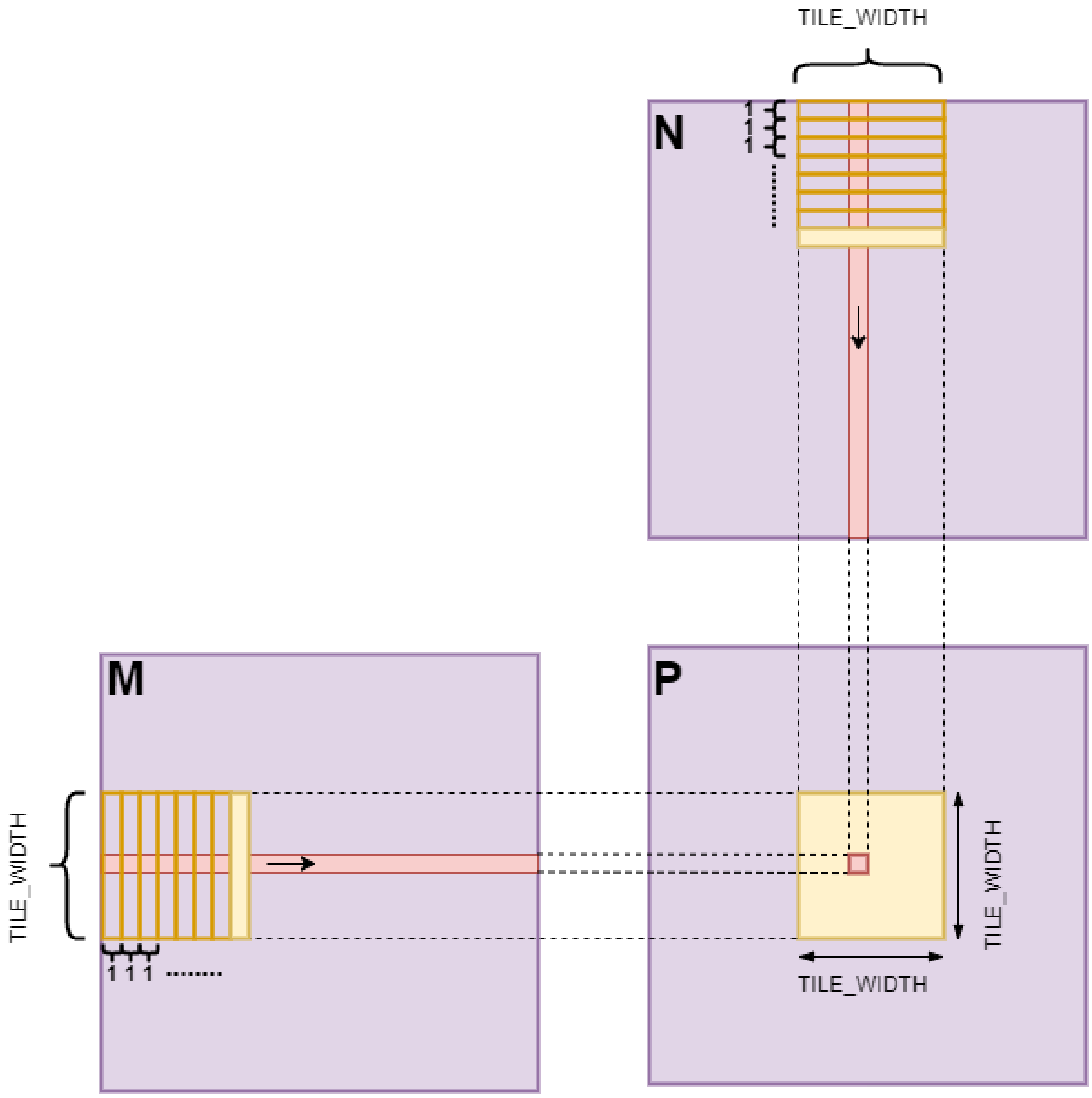
3(d). (2 point) How many threads can simultaneously contribute to y in this kernel? You may state your answer as an algebraic expression in terms of the variables in the code.

3(e). (2 points) You observe that the performance of the row-major weight kernel is substantially higher. Why?

Question 4. (24 points, suggested time allocation 35 minutes): Matrix Multiplication.

You need to do a matrix multiplication between a **632 x 632 matrix M** and a **632 x 632 matrix N**. You write a tiled matrix multiplication kernel you learned in class with a **8 x 8 square tile**. Your friend suggests you to instead use a **1 x 16 strip input tile** with **16x16 output tile** (see picture below) and gives you a partially finished code. You decided to implement the rest of code and compare the two.

The strip tiled matrix multiplication is visualized as below



```
 #define TILE_WIDTH 16
__global__ void strip_sgemm(float *M, float *N, float *P, Width) {
  // Assume that TILE_WIDTH is set to 16
  __shared__ float Mds[TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Identify the row and column of the P element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  // Loop over the M and N tiles required to compute P element
  for (int m = 0; m < Width; ++m) {
      // use the first 16 threads in each block to load Nds and Mds
      if(threadIdx.y== 0) {
        if(_______________________< Width)
          Mds[____________] = M[__________________];
        else
          Mds[____________] = 0;
        if(_______________________< Width)
          Nds[____________] = N[__________________];
        else
          Nds[____________] = 0;
      }

      __syncthreads();

      Pvalue += Mds[______] * Nds[______];
      __syncthreads();
  }

  if(Row < Width && Col < Width)
      P[Row*Width + Col] = Pvalue;
}
```

```
 /* strip_sgemm is launched with the following parameters
 dim3 gridDim(ceil(632/(float)TILE_WIDTH),ceil(632/(float)TILE_WIDTH);
 dim3 blockDim(TILE_WIDTH, TILE_WIDTH);
 */
```

4(a). (8 points) Fill in the blank to make this code run correctly. If you think it is impossible to fill in any of the blanks to make this code to run correctly, state why. (Hint: The thread index to data index mapping for the output is still the same as we used in MP3. Use the provided picture to identify the expressions for the thread index to data index mapping for loading and using M and N tiles.)

4(b). (4 points) For the strip tile matrix multiplication kernel, which of the input matrices have/has a coalesced memory access pattern? Please explain. (Hint: use the provided picture to analyze the memory access patterns by adjacent threads when loading M and N elements.)

(A) M
(B) N
(C) Both
(D) None

Answer:
Explanation:

4(c). (3 points) If the computation is limited by global memory bandwidth and the global memory bandwidth is 150GB/s bandwidth, what are the highest achievable FLOPS for the two kernel? Assume 32-bit floating point (single precision) for both kernels.(Hint: use the provided picture to analyze the number of reuses for M and N elements in the 1x16 tiling design.)

8 x 8 square tile:
1x 16 strip tile:

4(d). (6 points) Calculate the number of warps that will have control divergence for the two code during kernel execution for a 632x632 P matrix. Count a warp as having divergence if there is any control divergence at any of the statements during the execution of the kernel. Also, count a warp as one diverging warp even if there is control divergence at multiple statements for the warp. (Hint: use the provided picture to analyze the divergence patterns when loading M and N tiles.)

8 x 8 square tile:
1 x 16 strip tile:

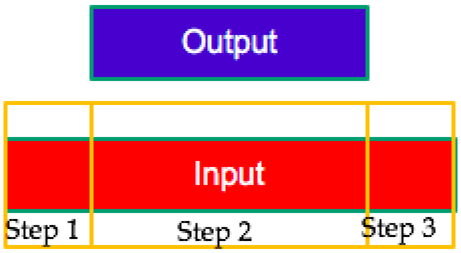
4(e). (3 points) If the SM (streaming multiprocessor) can take up to 1,536 threads and up to 12 thread blocks and have 5,120 Bytes of shared memory, How many pending global memory reads to M and N can the two code have?

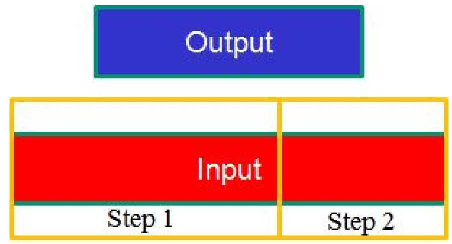
8 x 8 square tile:

1 x 16 strip tile:

Question 5. (20 points, suggested allocation of time 40 minutes): Convolution.

Recall tiling strategy 1 that we introduced in lecture where the threads were mapped to the output tile and the entire input tile was loaded into shared memory in multiple stages. In the strategy it was presented that for the 1D case you could load the input tile in three stages as presented by the left diagram below, one for the core elements and then one for each side with halo elements. This strategy inefficiently utilizes the number of threads we have and could actually implemented in two stages like in the right diagram.





5(a). (2 points) Now consider the 2D case and determine the **minimum number of stages** it would require to load an **input** tile with the dimensions 16x16 and a mask with dimensions 7x7. Make sure to explain your reasoning. Feel free to draw a diagram as part of your explanation.

5(b). (12 points) Write out the indexing and declarations needed to load the input tile into shared memory below for all three sections. You may assume **x_size** and **y_size** are known variables that give the number of pixels in the horizontal and vertical dimensions when declaring block and grid dimensions.

Defines:

```
#define MASK_SIZE 49
#define MASK_WIDTH 7
#define OUT_TILE_WIDTH ______
#define IN_TILE_WIDTH 16
__constant__ float kernel[MASK_WIDTH][MASK_WIDTH];
```

Define block and grid dimensions here:

```
// Somewhere in the host code
dim3 DimBlock(_____________, _____________, _);
dim3 DimGrid(__________________________, __________________________
, __);
```

Device Code:

```
__global__ void conv3d(float *input, float *output, const int


                          y_size, const int x_size) {
```

```
int tx = threadIdx.x, ty = threadIdx.y;
int bx = blockIdx.x, by = blockIdx.y;
// x_i and y_i are the x and y indices of the upper upper-left
// corner of the input tile
int x_i = bx * OUT_TILE_WIDTH - MASK_WIDTH/2;
int y_i = by * OUT_TILE_WIDTH - MASK_WIDTH/2;
__shared__ float inputTile[IN_TILE_WIDTH][IN_TILE_WIDTH];
for(int i = 0; i < __; i++){
  // Below are helper variables
  // idx is the linearized index of an element in the input tile
  int idx = __________________________;
  // xidx and yidx are the x and y positions of the corresponding
  // element in the tile
  int yidx = __________________________;
  int xidx = __________________________;
  // xpos and ypos are the x and y positions of the corresponding
  // element in the input
  int ypos = _____________;
  int xpos = _____________;
  if(idx < IN_TILE_WIDTH * IN_TILE_WIDTH){
    if (________________________________________________________)
      inputTile[yidx][xidx] = input [__________________________];
    else
      inputTile[yidx][xidx] = 0.0;
  }
}
….
}
```

5(c). (3 points) Does this strategy have any advantage over strategy 2 with regards to how the hardware is utilized? Recall that strategy 2 is where the threads were mapped to the input tile. Explain why or why not. Think about the kind of work each thread would do in either case and any hardware limitations. You may assume CUDA compute capability 3.0.

5(d). (3 points) Assuming the same grid and block dimensions and that the input we perform convolution on is perfectly tiled (one of the internal tiles), how many warps will experience control divergence for an internal tile using this modified strategy? You may assume that if the number of threads don't divide evenly by the warp size, the last warp is underutilized but does **not** experience control divergence because of this. Explain your answer.