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);`

(D) since we need a reference to the pointer itself and CUDA defines the first argument as void** type.
1 point partial credit for (C)

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);`

(B) correct syntax for cudaMemcpyToSymbol. (2 point partial credit if the student mentions cudaMemcpyToSymbol)

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;
```

(A) Explanation: All previous blocks cover (blockIdx.x*blockDim.x)*4. The index of the first element processed by each thread can be derived by adding threadIdx.x to it.

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

(4 points for (B)) 10^6/10^3 = 10^3 threads per block. Thus maximum of two blocks can run simultaneously on a single SM.
(2 point for (D)) Registers: 10 * 10^3 = 10K, 64K/10K = 6, not the limiting factor. Shared memory: 64KB/10KB=6, not the limiting factor.

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

Answer: (B) each row in a block is a warp. The first 30 rows will have control divergence. The last two warps are completely turned off during the calculation of inner products.

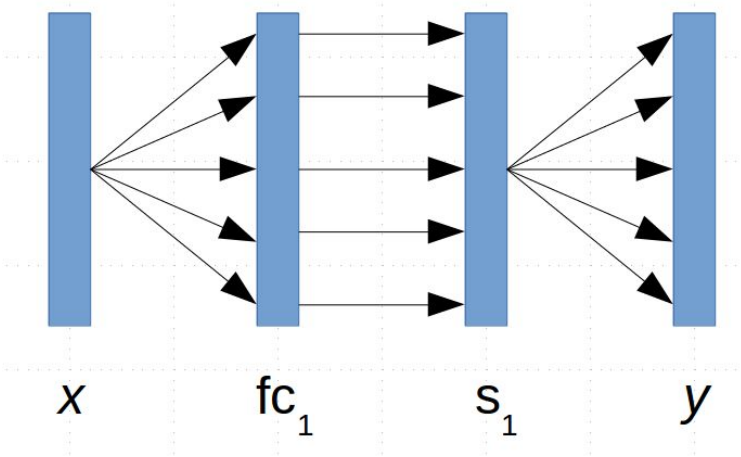
6. (4 points) How do we declare a 5x5 constant memory float array Mc?
(A) const float[5][5] Mc; in the kernel that needs to use Mc
(B) const float Mc[5][5]; outside all kernels
(C) __const__ float[5][5] Mc; outside all kernels.
(D) __const__ float Mc[5][5]; outside all kernels.

Answer: (D) keyword is __const__ and must be outside all kernels

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

False - XOR is not a linearly-separable function. Perceptrons work by finding a hyperplane that splits the input data.

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 $\frac{\partial E}{\partial y} * W_2 * \sigma(fc_1) * (1 - \sigma(fc_1))$

$$\frac{\partial E}{\partial b_1} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial s_1}\frac{\partial s_1}{\partial fc_1}\frac{\partial fc_1}{\partial b_1} = \frac{\partial E}{\partial y} * W_2 * \sigma(fc_1) * (1 - \sigma(fc_1)) * 1$$

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]
01. __global__
02. void colorToGreyscaleConvertion(unsigned char *grayImage, unsigned char
*rgbImage, int width, int height) {
03.    int Col = threadIdx.x + blockIdx.x * blockDim.x;
04.    int Row = threadIdx.y + blockIdx.y * blockDim.y;
05.    if (Col < width && Row < height) {
06.          // get 1D coordinate for the grayscale image
07.      int greyOffset = Row*width + Col;
08.    // one can think of the RGB image having
09.          // CHANNEL times columns of the gray scale image
10.      int rgbOffset = greyOffset*CHANNELS;
11.      unsigned char r = rgbImage[rgbOffset    ]; // red value for pixel
12.      unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
13.      unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
14.    // perform the rescaling and store it
15.          // We multiply by floating point constants
16.      grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
17.    }
18.}
```

Host Code:

```
      // the following host code is buggy
      // you need to fix the bugs in question 2(a)
01. dim3 dimGrid(ceil(height/64), ceil(width/32), 1);
02. dim3 dimBlock(64, 32, 1);
03. 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: 64*32 > 1024 which excesses the max available threads in a block
Fix 1: change 64->32 or other changes make generated threads less than 1024

Bug 2: ceil(height/64) is problematic when height/64 has a fractional part, as the fractional part of integer division in C is automatically truncated
Fix 2: ceil(height/64.0) or (height-1)/64 + 1 or other correct answers

Bug 3: the x dimension matches the column or width in the device code, while the host code assigns height into the x dimension
Fix 3: swap height and width in the host code

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.)

38*50*8 = 15200

Each thread block contains 16*16/32 = 8 warps
On vertical axis, there are in total ceil(600/16) = 38 blocks
On horizontal axis, there are in total ceil(800/16) = 50 blocks
Total warps = 38*50*8 = 15200

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

0

As warps are assigned in row-major order, and the number of thread blocks in x dimension is an integer value with no fractional part, so there is no divergence in the x dimension.
On the bottom, the threads in half of the warps are in use while all those in others are not. There is no divergence here.

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.**

50*8 = 400

As the warps are assigned in row-major order, all warps in last block of every row have divergence.
The number of ceil(800/16)*8 = 400 warps.

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.

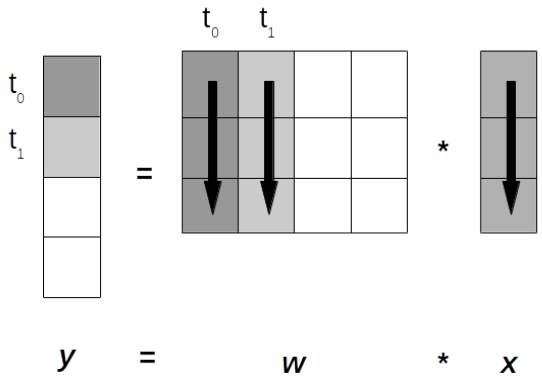
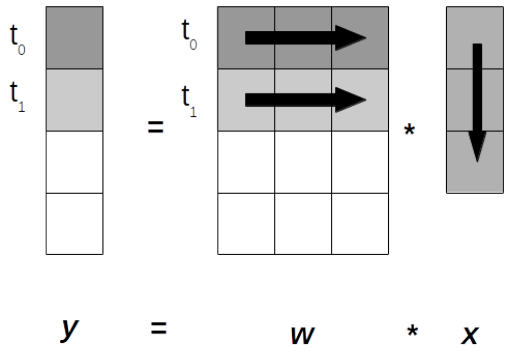
3(a). (2 points) Recall our formulation for a fully-connected layer as a matrix-vector multiplication:

$$y = w * x + b$$

Fill in the following table with the vector and matrix dimensions

Data	**Dimensions**
x	784
y	500
w	[784, 500]
b	500

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.

```
1.  __global__ void fc_row(float *y, const float *x,
2.                     const float *w, const float *b,
3.                     const int ySize, const int xSize) {
4.
5.    const int tx = blockDim.x * blockIdx.x + threadIdx.x;
6.    const int gx = gridDim.x * blockDim.x;
7.
8.    for (int o = tx; o < ySize; o += gx) {
9.      float acc = 0;
10.     for (int i = 0; i < xSize; ++i) {
11.       acc += x[                ] * w[                ];
12.     }
13.     y[                ] = acc + b[                ];
14.   }
15. }
```

- 11: i
- 11: i * ySize + o
- 13: o
- 13: o

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

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

```
3.                     const int ySize, const int xSize) {
4.
5.    const int tx = blockDim.x * blockIdx.x + threadIdx.x;
6.    const int gx = gridDim.x * blockDim.x;
7.
8.    for (int o = tx; o < ySize; o += gx) {
9.      float acc = 0;
10.     for (int i = 0; i < xSize; ++i) {
11.       acc += x[                ] * w[                ];
12.     }
13.     y[                ] = acc + b[                ];
14.   }
15. }
```

- 11: i
- 11: o * xSize + i
- 13: o
- 13: o

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.

- gx

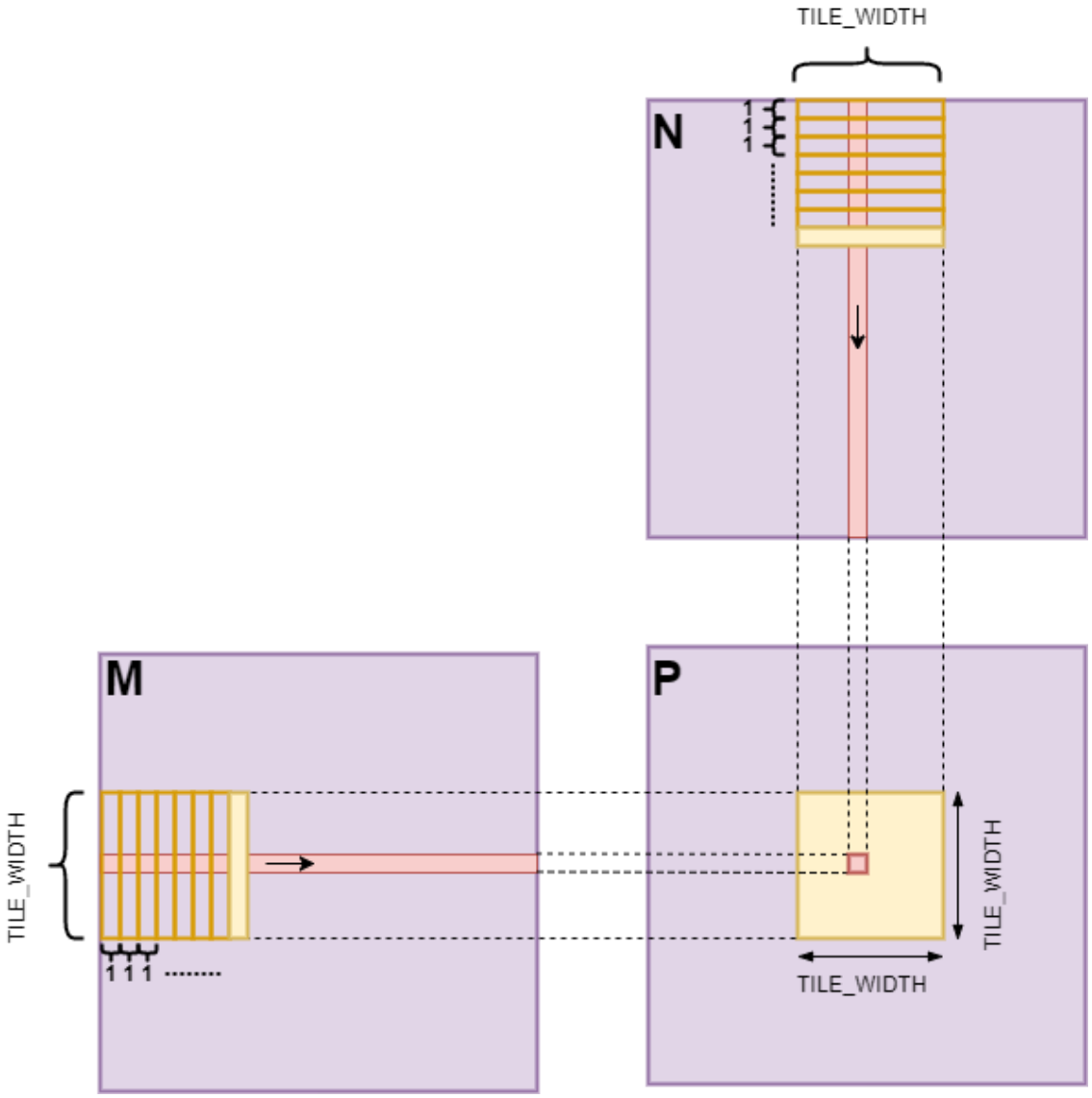
3(e). (2 points) You observe that the performance of the row-major weight kernel is substantially higher. Why?

- Matrix accesses are coalesced

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



```
1.      #define TILE_WIDTH 16
2.     __global__ void strip_sgemm(float *M, float *N, float *P, Width) {
3.       // Assume that TILE_WIDTH is set to 16
4.       __shared__ float Mds[TILE_WIDTH];
5.       __shared__ float Nds[TILE_WIDTH];
6.
7.       int bx = blockIdx.x;  int by = blockIdx.y;
8.       int tx = threadIdx.x; int ty = threadIdx.y;
9.
10.      // Identify the row and column of the P element to work on
11.      int Row = by * TILE_WIDTH + ty;
12.      int Col = bx * TILE_WIDTH + tx;
13.
14.      float Pvalue = 0;
15.      // Loop over the M and N tiles required to compute P element
16.      for (int m = 0; m < Width; ++m) {
17.          // use the first 16 threads in each block to load Nds and Mds
18.          if(threadIdx.y== 0) {
19.            if(_____________________< Width)
20.              Mds[____________] = M[_________________];
21.            else
22.              Mds[____________] = 0;
23.            if(_____________________< Width)
24.              Nds[____________] = N[_________________];
25.            else
26.              Nds[____________] = 0;
27.          }
28.
29.          __syncthreads();
30.
31.          Pvalue += Mds[______] * Nds[______];
32.          __syncthreads();
33.      }
34.
35.      if(Row < Width && Col < Width)
36.          P[Row*Width + Col] = Pvalue;
37.  }
```

```
1.      /* strip_sgemm is launched with the following parameters
2.      dim3 gridDim(ceil(632/(float)TILE_WIDTH),ceil(632/(float)TILE_WIDTH);
3.      dim3 blockDim(TILE_WIDTH, TILE_WIDTH);
4.      */
```

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.)

```
19. if( by * TILE_WIDTH + tx  < Width) + 1
20. Mds[tx] = M[(by * TILE_WIDTH + tx) * Width + m]; + 2
22. Mds[tx] = 0;  + .5
23. if( bx * TILE_WIDTH + tx  < Width) + 1
24. Nds[tx] = N [bx * TILE_WIDTH + tx + Width * m]; + 2
26. Nds[tx] = 0;  + .5
31. Pvalue += Mds[ty] * Nds[tx]; + 1
```

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:

(B) For M, the strip elements accessed by the adjacent threads (in the x dimension) belong to the same column and are not adjacent to each other in the layout. Thus M accesses are not coalesced. For N, the strip elements accessed by the adjacent threads (in the x dimension)) belong to the same row and are adjacent to each other. Thus N accesses are coalesced.

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: (150/4) * 8 = 300 GFlops
1x 16 strip tile: (150/4) * 16 = 600 GFlops

For 8x8 tiling, each input element is reused 8 times.
For 1x16 tiling, each input element is reused 16 times.

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: 0
1 x 16 strip tile: (40 * 40) +(39 * 7 + 3) = 1876

For 8x8 tiling, the number of pixels are multiples of the block width and height in both x and y dimensions. All threads will be active so there is no divergence.

For 1x16 tiling, there are four cases.

(1) For every block, the first warp will always have control divergence - 40*40*1 in this case.
(2) In addition. The blocks that calculate the right edge (except for the lower right corner) of the output P matrix have half of their threads active. There are 39 of them and each has 7 additional diverging warps in each of these blocks. All of these warps will have control divergence. 39*7warps in this case
(3) The blocks that calculate the bottom edge (except for the lower right corner) of the output P matrix have the last 4 warps inactive. There are no additional diverging warps.
(4) The block that calculates the lower right corner of the output P matrix has both active and inactive threads in all its warps. There are 3 additional diverging warps in this case.
(5) The rest of the blocks have all their threads active.

There are 40*40 + 39*7 + 3 = 1876 warps

In the second case, the blocks that process the bottom edge of the matrices have

memory

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: 10 * 64 * 2 = 1280
1 x 16 strip tile: 6 * 16 * 2 = 192

For 8x8 tiles, M and N tiles require 8*8*4*2 = 512 bytes so 10 thread blocks can go into the SM
So there will be 10*64*2 pending loads.

For 1x16 tiles, M and N tiles require 1*16*4*2 = 128 bytes so shared memory will not be a limitation. Each block has 256 threads so we can have up to 1536/256 = 6 blocks. So there will be 6*16*2 pending loads

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.

Answer: Output tile must be (16-6)^2 = 10 x 10. (Input / Output tile sizes) = 256 / 100 = 2.56, rounds up to 3. Can be done in 3 stages.

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:

```
1.  #define MASK_SIZE 49
2.  #define MASK_WIDTH 7
3.  #define OUT_TILE_WIDTH 10
4.  #define IN_TILE_WIDTH 16
5.  __constant__ float kernel[MASK_WIDTH][MASK_WIDTH];
```

Define block and grid dimensions here:

```
1.  // Somewhere in the host code
2.  dim3 DimBlock(OUT_TILE_WIDTH, OUT_TILE_WIDTH, 1);
3.  dim3 DimGrid(ceil(x_size  * 1.0 / OUT_TILE_WIDTH), ceil(y_size  *
1.0 / OUT_TILE_WIDTH), 1);
```

Device Code:

```
1.  __global__ void conv3d(float *input, float *output, const int
2.                         y_size, const int x_size) {
```

```
3.  int tx = threadIdx.x, ty = threadIdx.y;
4.  int bx = blockIdx.x, by = blockIdx.y;
5.  // x_i and y_i are the x and y indices of the upper upper-left
6.  // corner of the input tile
7.  int x_i = bx * OUT_TILE_WIDTH - MASK_WIDTH/2;
8.  int y_i = by * OUT_TILE_WIDTH - MASK_WIDTH/2;
9.  __shared__ float inputTile[IN_TILE_WIDTH][IN_TILE_WIDTH];
11. for(int i = 0; i < 3; i++){ // +1 for matching answer from a
12.   // Below are helper variables
15.   // idx is the linearized index of an element in the input tile
16.   int idx = ((i * by + ty) * bx + tx); // Must have some
dependence on i, ty, and tx to be considered valid.
17.   // xidx and yidx are the x and y positions of the corresponding
18.   // element in the tile
19.   int yidx = idx / IN_TILE_WIDTH; // yidx and xidx must each have
20.   int xidx = idx % IN_TILE_WIDTH; // a distinct calculation
21.   // xpos and ypos are the x and y positions of the corresponding
22.   // element in the input
23.   int ypos = y_i + yidx; // need to bring y_i and x_i to correctly
24.   int xpos = x_i + xidx; // calculate the element in input
25.   if(idx < IN_TILE_WIDTH * IN_TILE_WIDTH){
26.     if (xpos >= 0 && xpos < x_size && ypos >= 0 && ypos < y_size)
      // +1 for >=0 checks and +1 for size checks
27.       inputTile[yidx][xidx] = input [(ypos  * x_size) + xpos];
28.     else
39.       inputTile[yidx][xidx] = 0.0;
30.   }
31. }
    ….
32. }
```

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.

Short Answer: In strategy 1 has the advantage that all threads contribute to computation, while in strategy 2 many threads are turned off during computation.
This applies in general between strategy 1 and 2 since we have only changed how strategy 1 loads, not the overall algorithm.

Long Answer: When you instantiate the same number of threads in both strategies, only strategy 1 has all of the threads contributing towards calculations. In fact, you can always get more threads computing in strategy 1 than in strategy 2 due to the limits of the number of threads per block.
We also get some better global memory access to computation ratios (shared memory reuse esstially). Ideally we would want 1 global memory access for each computation. The worst case is if the ratio equals the mask size. Say we compute a 8x8x8 output in strategy 2 with a 3x3x3 mask. Then we have a 1000/512 ratio for memory access to compute = 1.95 global memory accesses per computation. If we consider strategy 1 it will have this same ratio on an output of the same size, but since strategy 1 uses less threads we can actually compute on larger output tiles and still be under the 1024 thread block limit, while strategy 2 cannot operate on larger tiles. Thus if we compare the 10x10x10 output strategy 1 can compute, we have a 1728/1000 = 1.728 ratio for memory accesses per computation.
This helps us see the trade off. We have longer latency loading stage in strategy 1, but we can always get better shared memory reusage.

Also accepting the answers of control divergence and more blocks per SM because fewer threads to compute the same output.

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.

Answer: 1, the only check that will diverge with the above assumptions is if(idx < inWidth * inWidth). Based on our math from part a, we have input tiles of size 256 and 100 threads. When we are on the third part of loading the input tile, we have 56 elements left among 100 threads, that means that the second warp will have 56 -32 = 24 threads loading elements and 8 threads not loading elements. This means warp 1 diverges (starting numbering from warp 0). All threads in the warp that precedes perform loads and the threads in the warps that follow do not load so they do not experience control divergence.

Partial credit awarded for answers that are correct for the 4 stage loading strategy up to 2 points. This method has 6 warps that diverge.