Be sure that your exam booklet has NINE pages.
Write your name and Net ID on the first page.
Do not tear the exam apart.
This is a closed book exam. You may not use a calculator.
You are allowed one handwritten 8.5×11-inch sheet of notes (both sides).
Absolutely no interaction between students is allowed.
Show all work, and clearly indicate any assumptions that you make.
Don’t panic, and good luck!

Problem 1 30 points _______________________________
Problem 2 24 points _______________________________
Problem 3 20 points _______________________________
Problem 4 26 points _______________________________

Total 100 points _______________________________
Problem 1 (30 points): Short Answer Questions

1. (10 points) For each statement below, circle either T for true or F for false.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

T F The __syncthreads() function synchronizes all threads in the grid.

T F The __global__ function qualifier indicates that the function can be executed on the host and on the device (GPU). The function can be called from the host or from the device.

T F The __device__ function qualifier indicates that the function is executed on the device. The function can be called from the host.

T F A variable marked with a __shared__ type qualifier is shared among all threads in a thread block, while a variable with a __constant__ type qualifier is shared among all threads in the grid.

T F Shared memory has higher throughput and lower latency compared to global memory.

2. You decide to parallelize a C program that takes 20 seconds to run on a CPU. You implement a CUDA kernel to replace a C function that consumes 80% of the CPU runtime and find that the total runtime drops to 12 seconds.

a. (2 points) How much time was spent in the new code? \[ 8 = 12 - 20 \times (1 - 0.8) \] seconds

b. (2 points) You measure the execution time for the GPU kernel and find that it finishes in 2 seconds. What other activity can account for the remaining time spent in the new code? USE NO MORE THAN 10 WORDS FOR YOUR ANSWER.

- copying memory [ between device and host ]

3. Relative to a fully connected layer of perceptrons, explain TWO benefits of using a convolution layer when computing the forward path of a deep neural network. USE NO MORE THAN 15 WORDS FOR EACH EXPLANATION. [ many possible answers – five listed below ]

a. (2 points) (1) fewer weights per perceptron (2) can change input size w/o retraining

(3) same weights used for all perceptrons

b. (2 points) (4) maintains spatial relationships from input in output

(5) dimensionality of input reflected in network structure

4. (4 points) USING NO MORE THAN 15 WORDS, explain one advantage of using a softmax function rather than an argmax function when training a deep neural network for classification.

- softmax has smooth, non-zero derivatives [ argmax derivative is 0 wherever it is defined ]
Problem 1, continued:

5. Your friend notices that the kernel shown below doesn’t always work and asks you to help debug it. Each block has a shared flag (cleared in each iteration), and any thread that triggers some condition sets the flag. The kernel should terminate when no thread triggers the condition, but sometimes some of the threads terminate early.

```c
__global__ void iterate(void* data)
{
    __shared__ int blockContinueFlag;
    do {
        if (threadIdx.x == 0) {
            blockContinueFlag = 0;
        }
        __syncthreads();
        //do some work...
        if (some condition) {
            // If >= 1 thread(s) execute this, flag will be written to 1
            blockContinueFlag = 1;
        }
        __syncthreads();
    } while (blockContinueFlag);
}
```

a. (4 points) USING NO MORE THAN 30 WORDS, explain what’s going wrong.

Thread 0 is able to reset the flag to 0 before other threads (other warps) evaluate the flag for the loop test.

b. (4 points) USING NO MORE THAN 15 WORDS, explain how to fix the bug.

Add __syncthreads at the start of the loop.
**Problem 2** (24 points): Adventures in One Dimension

The GPU kernel below performs a bitwise XOR operation between two vectors. Arrays \( A \) and \( B \) are the two input vectors, and array \( C \) is the output. All arrays consist of 8-bit unsigned integers, \( \text{uint8}_t \). The length of all three arrays in bytes is given by the \( \text{size\_in\_bytes} \) parameter.

```c
__global__ void vecXORKernel(uint8_t *A, uint8_t *B, uint8_t *C, int size_in_bytes)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < size_in_bytes) { // The size of uint8_t is 1 byte.
        C[i] = A[i] ^ B[i];
    }
}
```

1. (4 points) If the GPU device supports a maximum of 32 warps per SM and a maximum of 16 blocks per SM, how many threads per block are needed to fully utilize both resources? Show your work and write your answer here: 

   \[ \frac{32 \text{ warps/SM}}{16 \text{ blocks/SM}} = 2 \text{ warps/block}, \text{ or } 64 \text{ threads/block} \]

2. (4 points) Assume that \( \text{size\_in\_bytes} \) is 2080 and that your answer to **Problem 2.1** is used for the thread block size. How many warps experience control divergence when the kernel executes? Show your work and write your answer here: 

   \[ 2080 = 2048 + 32, \text{ so none of the warps has threads that both execute and do not execute the if statement} \]

3. (4 points) Assume that DRAM burst size is 128 bytes. Does changing the data type of arrays \( A, B, \) and \( C \) to \( \text{uint32}_t \) (32-bit unsigned integers) as shown below improve use of memory bandwidth with long arrays? Explain your answer.

   Yes: with the shorter data type, a warp reads only 32 bytes, and hence \( \frac{3}{4} \) of each DRAM burst may be wasted. With the longer data type, a warp reads 128 bytes (a full burst).

```c
__global__ void vecXORKernel(uint32_t *A, uint32_t *B, uint32_t *C, int size_in_bytes)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if ((i*4) < size_in_bytes) { // The size of uint32_t is 4 bytes.
        C[i] = A[i] ^ B[i];
    }
}
```
Problem 2, continued:

```c
__global__ void PictureKernel(unsigned char* d_Pin, unsigned char* d_Pout,
                               int n, int m, int thresh) {
  // 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)) {
    unsigned char local = d_Pin[Row * n + Col];
    if (thresh >= local) {
      local = 0;
    }
    d_Pout[Row * n + Col] = local;
  }
}
```

4. (6 points) Consider the CUDA kernel shown above executing on an input image (d_Pin) with 408 rows and 500 columns (m = 408, n = 500). The launch uses enough 16×16 thread blocks to cover all pixels in the input image. How many warps experience control divergence? Show your work and write your answer here: ___204____

[Note: (408 / 2) * ceil (500 / 16) = 204 * 32 = 6528 warps is also acceptable due to the inner if.]

Each warp consists of two rows of some 16×16 thread block.

In the vertical direction, 500 is even, so none of the warps has divergence.

In the horizontal direction, all warps in the last column of blocks (but not off the bottom of the image) have divergence. So 408 / 2 = 204 warps.

5. (6 points) Your friend rewrote the kernel as shown below to use 1D thread blocks and grids. She claims that her kernel improves memory coalescing and reduces control divergence. Do you agree? Explain your answer USING NO MORE THAN 30 WORDS.

___Yes, now right border of image gives no branch divergence and no partial DRAM bursts.___

```
__global__ void PictureKernel(unsigned char* d_Pin, unsigned char* d_Pout,
                               int n, int m, int thresh) {
  // Calculate the row and column # of d_Pin and d_Pout element to process
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int Row = idx / n;
  int Col = idx % n;
  // each thread computes one element of d_Pout if in range
  if ((Row < m) && (Col < n)) {
    unsigned char local = d_Pin[Row * n + Col];
    if (thresh >= local) {
      local = 0;
    }
    d_Pout[Row * n + Col] = local;
  }
}
```
Problem 3 (20 points): Supertiles to the Rescue

Recognizing that larger tile sizes improve dense matrix multiply performance, you try increasing thread block size to 64×64, but your kernel doesn’t launch, as CUDA allows at most 1,024 threads per block. You decide to set block size to 32×32 but use a tile size of 64×64. Each thread in a block must thus load four elements of each tile and compute four elements of the output matrix.

1. (12 points) Fill in the blanks in the partial kernel shown below to read a tile from matrix A into the shared array Mds in a way that maximizes memory coalescing. Assume that values outside of A can be safely ignored.

```c
#define TW 64    // tile width    [Note: many ways work, and at least two
#define BW 32    // block width    ways give full credit. ]

__global__ void matrixMultiplyShared(float *A, float *B, float *C,
    int numARows, int numACols,
    int numBRows, int numBCols,
    int numCRows, int numCCols) {

    __shared__ float Mds[TW][TW];
    __shared__ float Nds[TW][TW];
    int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y;
    int Row = by * TW + ty;
    int Col = bx * TW + tx;

    for (int p = 0; p < ceil(((float) numACols)/TW); ++p){
        if(( Row __________________________ < numARows) &&
            (p * TW + tx ______________________ < numACols)){
            Mds[_________ ty _______________][________ tx ________________] =
                A[ Row * numACols + p * TW + tx ____________________];
        }
        if(( Row __________________________ < numARows) &&
            (p * TW + BW + tx ______________________ < numACols)){
            Mds[_________ ty _______________][________ tx + BW ________________] =
                A[ Row * numACols + p * TW + BW + tx ______________________];
        }
        if(( Row + BW __________________________ < numARows) &&
            (p * TW + tx ______________________ < numACols)){
            Mds[_________ ty + BW ____________][________ tx ________________] =
                A[ (Row + BW) * numACols + p * TW + tx ___________________];
        }
        if(( Row + BW __________________________ < numARows) &&
            (p * TW + BW + tx ______________________ < numACols)){
            Mds[_________ ty + BW ____________][________ tx + BW ________________] =
                A[ (Row + BW) * numACols + p * TW + BW + tx _____________________];
        }
    }
    // read tile of N, compute, and so forth...
```
Problem 3, continued:

2. **(4 points)** In the new code, how many threads use each element of the shared memory when computing the output elements?
   
   
   ______ 32 __________ [ answer depends on 3.1 strategy ]

3. **(4 points)** After observing shorter execution times with the of 64×64 tile size, you modify the code to use even larger tiles, of size 256×256. Sadly, the kernel fails to launch. **USING NO MORE THAN 15 WORDS**, explain the problem.
   
   __ not enough shared memory to hold larger tiles ________________________________
Problem 4 (26 points): Convoluted Code

At your first internship, you are asked to complete a former co-worker’s 3D convolution code, shown on the next page. The code convolves a 3D mask array, mask, with a 3D input array, input, to produce a 3D output array, output. In this application, the mask can be smaller than $\text{MASK\_WIDTH}$. The parameter $\text{real\_mask\_width}$ gives the mask width to be used in the convolution, which you may assume is always odd and is never greater than $\text{MASK\_WIDTH}$. The input array has dimension $\text{Height} \times \text{Rows} \times \text{Cols}$, but the output is smaller, and contains only those elements for which the mask fits fully within the input array.

The code uses the approach in which all threads load an input tile to shared memory, and a subset of threads compute the output.

1. **(6 points)** Fill in the blanks in the code.

2. **(4 points)** How big is an output tile? ___ $\text{TILE\_WIDTH} \times \text{TILE\_WIDTH} \times \text{TILE\_WIDTH}$ ___

3. **(4 points)** What should the $\text{BLOCK\_WIDTH}$ be? _____ 8 _____ (a number)

4. **(4 points)** Using the variable names from the code, specify the values of $\text{dimGrid.z}$ and $\text{dimBlock.z}$ needed for launching the kernel.

   \[ \text{dimGrid.z} = \underbrace{\text{ceil} \left( \frac{\text{Height} - \text{real\_mask\_width} + 1}{1.0 \times \text{TILE\_WIDTH}} \right)}; \]

   \[ \text{dimBlock.z} = \underbrace{\text{BLOCK\_WIDTH}}; \]

5. **(4 points)** How many calls to $\text{__syncthreads()}$ are needed in the code? ____ 1 ____

   Add $\text{__syncthreads()}$ to the code on the next page at each necessary location.

6. **(4 points)** Can the code work properly? Explain your answer USING NO MORE THAN 20 WORDS.

   __ No: kernel cannot write to constant memory. __
Problem 4, continued:

#define MASK_WIDTH 5
#define TILE_WIDTH 4
#define BLOCK_WIDTH xxx // Defined by you in Problem 4.3.

__constant__ float mask[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];

__global__ void conv(float *input, float *output,
    int Height, int Rows, int Cols, int real_mask_width)
{
    __shared__ float subTile[BLOCK_WIDTH][BLOCK_WIDTH][BLOCK_WIDTH];
    int tx = threadIdx.x, ty = threadIdx.y, tz = threadIdx.z;
    float Pvalue = 0.0f;

    if (real_mask_width < MASK_WIDTH) {
        // Set out-of-bounds mask values to 0.
        if (tx >= real_mask_width && tx < MASK_WIDTH && ty >= real_mask_width &&
            ty < MASK_WIDTH && tz >= real_mask_width && tz < MASK_WIDTH) {
            mask[ty][tx][tz] = 0.0f;
        }
    }

    int col = blockIdx.x * TILE_WIDTH + tx; // input column index
    int row = blockIdx.y * TILE_WIDTH + ty; // input row index
    int height = blockIdx.z * TILE_WIDTH + tz; // input height index

    int Height_output = Height - real_mask_width + 1; // output height
    int Rows_output = Rows - real_mask_width + 1; // output number of rows
    int Cols_output = Cols - real_mask_width + 1; // output number of columns

    if (height < Height && col < Cols && row < Rows) { // load to shared memory
        subTile[tz][ty][tx] = input[(height * Rows + row) * Cols + col];
    }

    __syncthreads();

    if (tx < TILE_WIDTH && ty < TILE_WIDTH && tz < TILE_WIDTH) { // calc. output
        for (int k = 0; k < real_mask_width; ++k) {
            for (int i = 0; i < real_mask_width; ++i) {
                for (int j = 0; j < real_mask_width; ++j) {
                    Pvalue += mask[k][i][j] * subTile[tz+k][ty+i][tx+j];
                }
            }
        }
    }

    // Copy output to global memory.
    if (height < Height_output && row < Rows_output && col < Cols_output) {
        output[(height * Rows_output + row) * Cols_output + col] = Pvalue;
    }
}

// kernel launch--you must define the grid parameters in Problem 4.4
conv<<<dimGrid, dimBlock>>>(input, output, Height, Rows, Cols, real_mask_width);