CS 677: Homework Assignment 1
Due: February 17, 6:00pm

Philippos Mordohai
Department of Computer Science
Stevens Institute of Technology
Philippos.Mordohai@stevens.edu

Collaboration Policy. Homeworks will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems. Use of the Internet is allowed, but should not include searching for previous solutions or answers to the specific questions of the assignment. I will assume that, as participants in a graduate course, you will be taking the responsibility of making sure that you personally understand the solution to any work arising from collaboration.

Late Policy. No late submissions will be allowed without consent from the instructor. If urgent or unusual circumstances prevent you from submitting a homework assignment in time, please e-mail me explaining the situation.

Deliverable. Submit on Canvas a zip file containing:

1. source code for the first three problems,
2. a pdf file describing your approach to Problems 1 and 3, and containing your answers to Problems 4-6.

Problem 1. (20 points) Download vecAdd.cu from `https://github.com/olcf/vector_addition_tutorials/blob/master/CUDA/vecAdd.cu`. Compile and execute the code. This can be done using:

```
    nvcc -o vecAdd  vecAdd.cu
```

or the makefile on the repository above.

Now, modify the code to perform addition of two 1024×1024 matrices generated in a similar way as the vectors a and b in the example. In your report, show the top-left 5×5 submatrices of the input and output. **Submit the source code for this part without changing the filename.**

Write three kernel functions:

(a) One that has each thread producing one element of the output matrix.

(b) One that has each thread producing one row of the output matrix.

(c) One that has each thread producing one column of the output matrix.

There is no need to submit printouts for all three kernels. They should produce the same answer.

Problem 2. (20 points) Create a new source file named `MatAddArb.cu`. Modify the above kernels so that they can add matrices of arbitrary dimensions (width and height) that are less or equal to 1024. Obviously, the two matrices must have the same dimensions. **Submit the source code for this part using the above filename. No need to discuss it in your report.**

Problem 3. (20 points) Create a new source file named `MaxPerRow.cu`. Write a program that finds the maximum in each row of a matrix of arbitrary dimensions (width and height) that are less or equal to 1024 and stores the detected maxima in global memory. **Submit the source code for this part using the above filename. No need to discuss it in your report.**

Problem 4. (10 points) For a vector addition, assume that the vector length is 2000, each thread calculates one output element, the thread block size is 512 threads and the entire computation is carried out by one grid. How many threads will be in the grid? How many warps do you expect to have divergence due to the boundary check on the vector length?

Problem 5. (10 points) Consider matrix addition where each element of the output matrix is the sum of the corresponding elements of the two input matrices. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: analyze the elements accessed by each thread and see if there is any commonality between threads.

Problem 6. (20 points) Consider the following program that transposes the *tiles* of a large matrix. The size of the tiles is `BLOCK_WIDTH`×`BLOCK_WIDTH` and the dimensions of the matrix A are guaranteed to be multiples of `BLOCK_WIDTH`, which is known at compile time. `BLOCK_WIDTH` is in the range from 1 to 20.

```
dim3 blockDim(BLOCK_WIDTH, BLOCK_WIDTH);
dim3 gridDim(A_width/BLOCK_WIDTH, A_height/BLOCK_WIDTH);

BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);

__global__ void BlockTranspose( float* A, int A_width, int A_height)
{
  __shared__ float blockA[BLOCK_WIDTH][BLOCK_WIDTH];
  int baseIdx = blockId.x * BLOCK_WIDTH + threadId.x;
  baseIdx += (blockId.y * BLOCK_WIDTH + threadId.y) * A_width;
  blockA[threadId.y][threadId.x] = A[baseIdx];
  A[baseIdx] = blockA[threadId.x][threadId.y];
}
```

(a) For which values of `BLOCK_WIDTH` in the range from 1 to 20 will this kernel function execute correctly on the device?
(b) If the code does not execute correctly for all values of `BLOCK_WIDTH`, suggest a correction to make the code work for all values of `BLOCK_WIDTH`.