CS 677: Homework Assignment 2
Due: February 15, 6:15pm

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.

Submission Format. Electronic submission on Canvas is mandatory. Submit a **zip** file containing a pdf file with your answers and code.

Problem 1. (60 points) For this problem, follow the steps below.

1. Download cs677_hw2.zip from `http://www.cs.stevens.edu/~mordohai/classes/cs677_s17/cs677s17_hw2.zip` and extract its contents into your SDK projects directory.

2. Edit the source files vector_reduction.cu and vector_reduction_kernel.cu to complete the functionality of the parallel addition reduction on the device. The size of the array is guaranteed to be equal to 512 elements for this assignment.

3. There are two modes of operation for the application.

   - No arguments: The application will create a randomly initialized array to process. After the device kernel is invoked, it will compute the correct solution value using the CPU, and compare that solution with the device-computed solution. If it matches (within a certain tolerance), if will print out "Test PASSED" to the screen before exiting.
   - One argument: The application will initialize the input array with the values found in the file provided as an argument.

In either case, the program will print out the final result of the CPU and GPU computations, and whether or not the comparison passed the test.

4. Answer the following questions:

   (a) How many times does your thread block synchronize to reduce the array of 512 elements to a single value? (2 points)

   (b) What is the minimum, maximum, and average number of "real" operations that a thread will perform? "Real" operations are those that directly contribute to the final reduction value. (4 point)

   (c) How would you modify the code to handle vectors of length less than 512? (No implementation required.) (4 points)

   (d) How would you modify the code to handle vectors of length more than 512? (No implementation required.) (5 points)

Problem 2. Kirk & Hwu problem 6.11. (40 points) The following scalar product code tests your understanding of the basic CUDA model. The code computes 1024 dot products, each of which is calculated from a pair of 256-element vectors. Assume that the code is executed on G80. Use the code to answer the following questions.

a) How many threads are there in total? (1 point)

b) How many threads are there in a Warp? (1 point)

c) How many threads are there in a Block? (1 point)

d) How many global memory loads and stores are done for each thread? (3 points)

e) How many accesses to shared memory are done for each block? (8 points)

f) List the source code lines, if any, that cause shared memory bank conflicts. (8 points)

g) How many iterations of the for loop (line 23) will have branch divergence? Show your derivation. (10 points)

h) Identify an opportunity to significantly reduce the bandwidth requirement on the global memory. How would you achieve this? How many accesses can you eliminate? (8 points)

```
1   #define VECTOR_N    1024
2   #define ELEMENT_N 256
3   const int DATA_N          = VECTOR_N * ELEMENT_N;
4   const int DATA_SZ    = DATA_N * sizeof(float);
5   const int RESULT_SZ = VECTOR_N  * sizeof(float);
...
6   float *d_A, *d_B, *d_C;
...
7   cudaMalloc((void **)&d_A, DATA_SZ);
8   cudaMalloc((void **)&d_B, DATA_SZ);
9   cudaMalloc((void **)&d_C, RESULT_SZ);
...
10  scalarProd<<<VECTOR_N, ELEMENT_N>>>(d_C, d_A, d_B, ELEMENT_N);
11
12  __global__ void
13  scalarProd(float *d_C, float *d_A, float *d_B, int ElementN)
14  {
15      __shared__ float accumResult[ELEMENT_N];
16      //Current vectors bases
17      float *A = d_A + ElementN * blockIdx.x;
18      float *B = d_B + ElementN * blockIdx.x;
19      int tx = threadIdx.x;
20
21      accumResult[tx] = A[tx] * B[tx];
22
23      for(int stride = ElementN /2; stride > 0; stride >>= 1)
24      {
25          __syncthreads();
26          if(tx < stride)
27              accumResult[tx] += accumResult[stride + tx];
28      }
30      d_C[blockIdx.x] = accumResult[0];
31  }
```