Name: **17.11.2022**
Student ID:

CENG443 Midterm Exam - Fall 2022
(90 minutes)

Q.1 (20 Points) If the SM of a CUDA device can take up to 1536 threads and up to 4 thread blocks. Which of the following block configuration would result in the largest number of threads in the SM? Explain the reason for each configuration.

A. 128 threads per block
B. 256 threads per block
C. 512 threads per block
D. 1024 threads per block

A. 128 x 4 = 512
B. 256 x 4= 1024
C. 512 x 4 = 2048 Not possible!
512 x 3 = 1536 ****LARGEST
D. 1024 x2 = 2048 Not possible!
1024 x1 = 1024

Q.2 (20 Points) Explain the benefit of having more threads than the available processing units/cores for a parallel program? Give an example scenario.

When there is a thread with long-latency instruction that is not utilizing cores, it can be descheduled and the instructions can be executed by other available threads.

Q.3 (30 Points)

```
__global__ void vecAddKernel(float* A, float* B, float* C, int n){
        int i = threadIdx.x+blockDim.x*blockIdx.x;
        if(i<n) C[i] = A[i] + B[i];
}
```

For the vector addition kernel given above, assume that the vector length is 4000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements.

a) How many threads will be in the grid? Write the kernel invocation statement (Assume we have A, B, and C variables already before the invocation).

1024 x 4 = 4096 threads, 4 blocks

vecAddKernel<<<4, 1024>>> vecAddKernel(A, B, C, 4000);

b) How many floating operations are being performed in the vector addition kernel? Explain your answer.

4000 (for each vector element)

c) How many global memory reads are being performed by your kernel? Explain your answer.

4000 x 2 memory reads (A and B)

d) How many global memory writes are being performed by your kernel? Explain your answer.

4000 x 1 memory reads (C)

e) Modify the kernel code such that each thread will compute two output elements, with the same number of blocks and the half number of threads. After each thread finishes its work at the current index, you need to increment each of them by the total number of threads running in the grid (If we have total **m** threads, the first thread will be responsible for the computation of **C[0]** and **C[m]**). By considering the number of threads necessary for the modified version, write the kernel invocation statement.

```
__global__ void vecAddKernel(float* A, float* B, float* C, int n){
        int i = threadIdx.x+blockDim.x*blockIdx.x;
        while(i<n){
                C[i] = A[i] + B[i];
                i += blockDim.x * gridDim.x; // i + total threads
        }
}
```

vecAddKernel<<<4, 512>>> vecAddKernel(A, B, C, 4000);

Q.4 (30 Points) We have the following **compute** kernel function for 1D finite difference method, where each thread is responsible for one element computation of the output vector (**out**):

```
__global__ void compute (float *out, float *out_prev, int n, float dx, float dt, float c, int blocksize){
        int i = threadIdx.x+blocksize*blockIdx.x;
        if(i<n) out[i] = out_prev[i] – c * (dt/dx) * (out_prev[i] – out_prev[i-1]);
}
```

Modify the kernel function such that it will utilize the GPU shared memory. Discuss the advantage and disadvantage of the shared memory usage for this code by providing example execution scenarios. You can give numeric values to support your discussion.

```
__global__ void compute (float *out, float *out_prev, int n, float dx, float dt, float c, int blocksize){
        int i = threadIdx.x+blocksize*blockIdx.x;
        __shared__ float out_shared [blocksize];
        if(i<n) out_shared[i] = out_prev[i];
        __syncthreads();
         if(i<n) out[i] = out_shared[i] – c * (dt/dx) * (out_shared[i] – out_shared[i-1]);
}
```

Not so memory-bound, just two elements in common. May not be so helpful.