**Problem definition:**

This problem consists of 3 parts.

First one is calculating the vector product of 2 vectors. These vectors need to be same length. Parallelization would decrease computation time for all cases that have large vector size, overall program time would increase for cases that have small vector size because of memory copy operations from and to the CUDA device.

Second one is summation of 2 matrices. These matrices need to have same number of rows and columns. Parallelization would decrease computation time for all cases that have large row and-or column size, overall program time would increase for cases that have small row and column size because of memory copy operations from and to the CUDA device.

Third one is multiplying a vector and a matrix. The vector must have the same size as the rows of the matrix, the resulting vector will have the same size as the columns of the matrix. Parallelization would decrease the computation time for all cases that have large rows and-or column size, overall program time would increase for most cases because of memory copy operation from and to the CUDA device and waiting for the summation after multiplication of each column (race condition at summation).

**Algorithm Description:**

*(all the algorithms check for CUDA errors after each CUDA operation)*

First algorithm creates 2 pointers for each of the arrays (one or CPU one for CUDA device). Then it allocates memory for the arrays on the CPU, sets the array values. Then allocates memory on CUDA device, sets block size to a value between 0 and 1024 and number of blocks to array size / block size (adds 1 if array size % block size is not 0 to prevent creating less threads than desired). Kernel call to array multiplication is done with pointers to 2 arrays on CUDA memory and size of an array as parameters. In the kernel call a unique ID is created for each thread and if that ID is smaller than array size ID’th indexes of the arrays are multiplied and stored in the first array to use less memory. The first array on CUDA device then gets copied to the first array on CPU (to save space). As the last step contents of the first array gets summed up and printed, then used memory is freed.

Second algorithm does everything the same as first algorithm until the kernel call (albeit with 2 matrices instead of 2 arrays). In kernel call to Matrix Summation method 2 pointers to the matrices on CUDA device and size of a matrix is passed as a parameter, which then sums each element that are on the same position of the matrices. The same (as the first algorithm) fashion of unique ID creation-size checking is used here and result is saved to the first matrix to save space again. After that the first matrix on CUDA device is copied to the first matrix on CPU and printed using Print Matrix method with first matrix (on CPU), row size, column size as parameters. Finally, all memory is freed.

Third algorithm does everything the same as first algorithm until the kernel call (albeit with 1 matrix and 1 vector instead of 2 arrays). In kernel call to Matrix Vector Multiplication method pointers to the matrix and vector on CUDA device and size of the matrix is passed as a parameter, which then multiplies each element of the column of the matrix with the same index on array (stores the result on the matrix to save space). The same (as the first algorithm) fashion of unique ID creation-size checking used here. After that first matrix on CUDA device is copied to the first matrix on CPU. Then each column of the matrix is summed, and the result is stored in a new vector that has size of the matrix’s column count. The resulting vector is printed, and all memory is freed.

**Benchmarking:**

*(benchmark results are included as hw1\_benchmark.txt file)*

First exercise is good for medium-large arrays since when we go from 800 elements to 800.000 elements(1000x) we see much less(30x) time increase and when we go from 800.000 elements to 8.000.000 elements(10x) we see equal(10x) time increase. This means that for small array size memory copy decreases performance and may not be desirable.

Second exercise is good for medium-large matrices since when we go from 150 elements to 1.500.000 elements(10.000x) we see much less(20x-40x) time increase and when we go from 1.500.000 elements to 15.000.000 elements(10x) we see equal time increase(10x). This means for small matrix size memory copy decreases performance and may not be desirable. Row or column size alone does not affect computation time since when we go from 15.000 rows to 1.500 rows(1/10x) and 100 columns to 10.000 columns(100x) we see 10x time increase.

Third exercise is good for medium matrices and even better for large matrices since when we go from 35 elements to 3.500.000 elements(100.000x) we see much less(100x) time increase and when we go from 3.500.000 elements to 350.000.000(100x) we see about equal(85x-90x) time increase. Increasing column size 100x in medium or large cases gives expected(85x-90x) time increase as well as increasing row size 10x in medium or large cases gives expected(9x) time increase.

**Pros-cons:**

All three cases utilize CUDA device’s core count for parallel computation. But they also use memory on CUDA device.

In first exercise we sum the contents of the resulted array in CPU because of race condition. Using mutexes(etc.) may or may not decrease performance but it also would decrease memory usage and cudaMemcpy’s execution time on CPU if we use mutexes(etc.) on CUDA device.

In third exercise we sum each column of the resulted matrix in CPU because of race condition. Using mutexes(etc.) for individual columns would increase performance proportional to column count and decrease memory usage and cudaMemcpy’s execution time on CPU if we use mutexes(etc.) on CUDA device.

**Discussion:**

This part is done at benchmarking and Pros-cons sections. But in short we use memory on CUDA device in addition to CPU, we lose time because of race conditions on first and third exercises, we lose time because of memory copy operations on all three cases, we gain time with the computation on medium-large sized arrays/matrices.

**Environment:**

NVIDIA GeForce GTX 1060 3GB (compute capability 6.1)

Intel Core-I3 7100 @3.9GHz \w 8GB DDR4 RAM

Windows 10 Education Version 1809 (OS Build 17763.55)

Nvcc V10.0.130 compiler / Visual studio 2017 Community

**NOTE:**

Each exercise has its own main function/method, beware when compiling.