Machine Problem 3 Report

Presented to Prof. Ahmad Afsahi
ELEC 374: Digital Systems Engineering
Faculty of Applied Science
Queen’s University

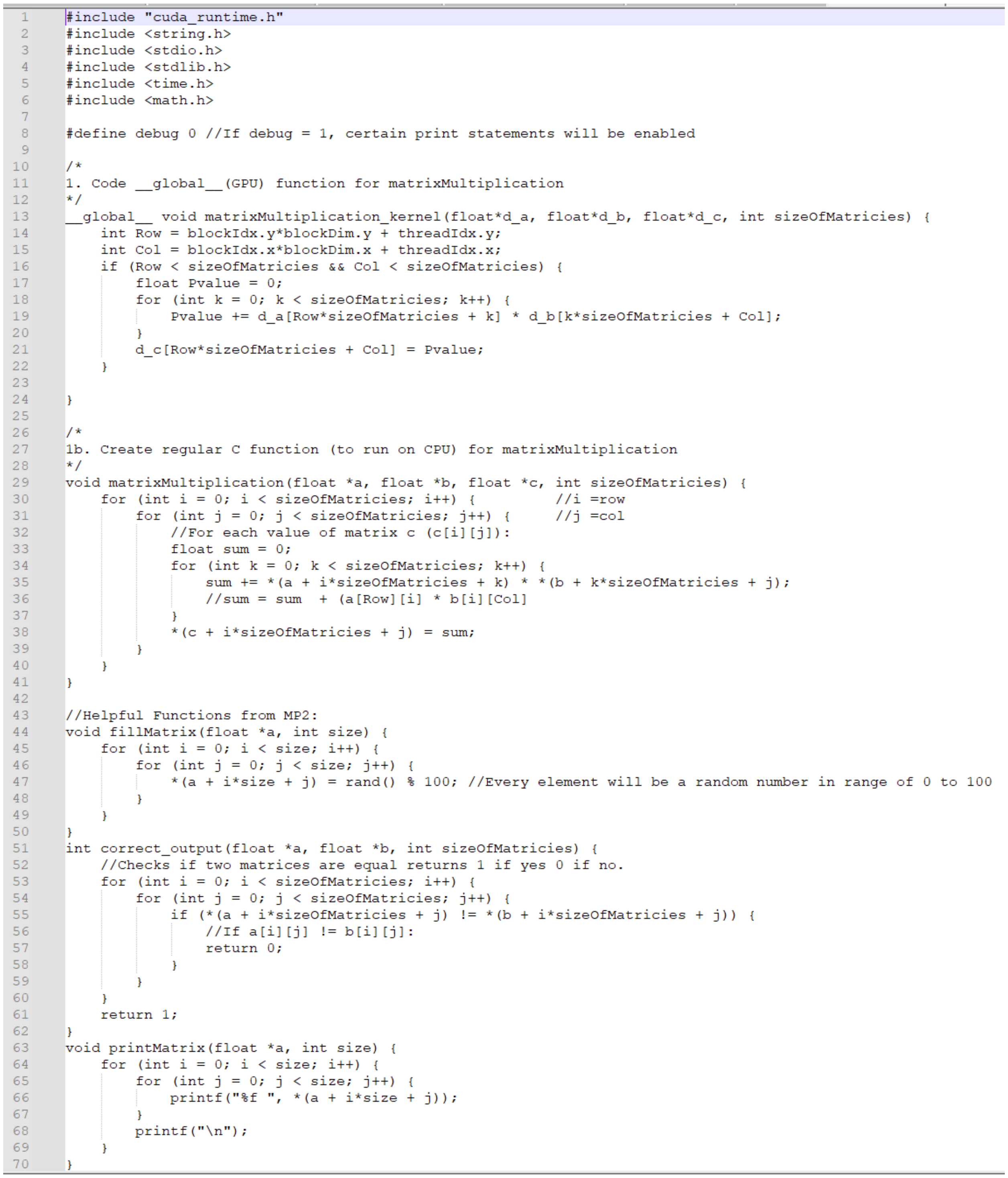
Nathan Goodman: 20228249
Date: April 5th, 2023

CUDA Code

For the three different requested outputs, three different CUDA scripts were created:

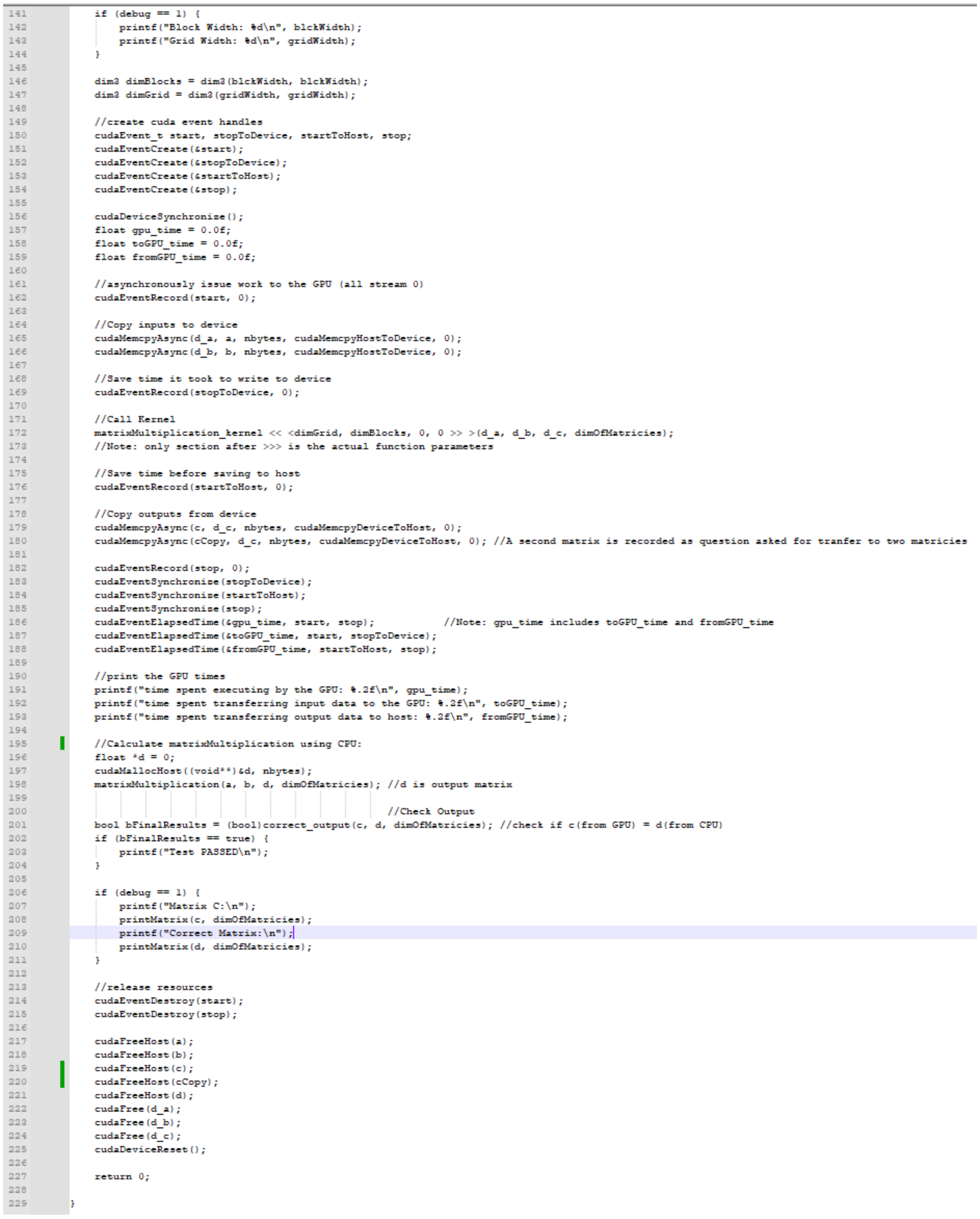
1. Find the time it takes to transfer two matrices worth of data to and from device:

Note: Block width is equal to 16 and gpu execution time includes transfer times.



```
#include "cuda_runtime.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define debug 0 //If debug = 1, certain print statements will be enabled

/*
1. Code __global__(GPU) function for matrixMultiplication
*/
__global__ void matrixMultiplication_kernel(float*d_a, float*d_b, float*d_c, int sizeOfMatricies) {
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    if (Row < sizeOfMatricies && Col < sizeOfMatricies) {
        float Pvalue = 0;
        for (int k = 0; k < sizeOfMatricies; k++) {
            Pvalue += d_a[Row*sizeOfMatricies + k] * d_b[k*sizeOfMatricies + Col];
        }
        d_c[Row*sizeOfMatricies + Col] = Pvalue;
    }

}

/*
1b. Create regular C function (to run on CPU) for matrixMultiplication
*/
void matrixMultiplication(float *a, float *b, float *c, int sizeOfMatricies) {
    for (int i = 0; i < sizeOfMatricies; i++) {          //i =row
        for (int j = 0; j < sizeOfMatricies; j++) {      //j =col
            //For each value of matrix c (c[i][j]):
            float sum = 0;
            for (int k = 0; k < sizeOfMatricies; k++) {
                sum += *(a + i*sizeOfMatricies + k) * *(b + k*sizeOfMatricies + j);
                //sum = sum  + (a[Row][i] * b[i][Col]
            }
            *(c + i*sizeOfMatricies + j) = sum;
        }
    }
}

//Helpful Functions from MP2:
void fillMatrix(float *a, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            *(a + i*size + j) = rand() % 100; //Every element will be a random number in range of 0 to 100
        }
    }
}
int correct_output(float *a, float *b, int sizeOfMatricies) {
    //Checks if two matrices are equal returns 1 if yes 0 if no.
    for (int i = 0; i < sizeOfMatricies; i++) {
        for (int j = 0; j < sizeOfMatricies; j++) {
            if (*(a + i*sizeOfMatricies + j) != *(b + i*sizeOfMatricies + j)) {
                //If a[i][j] != b[i][j]:
                return 0;
            }
        }
    }
    return 1;
}
void printMatrix(float *a, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf("%f ", *(a + i*size + j));
        }
        printf("\n");
    }
}
```

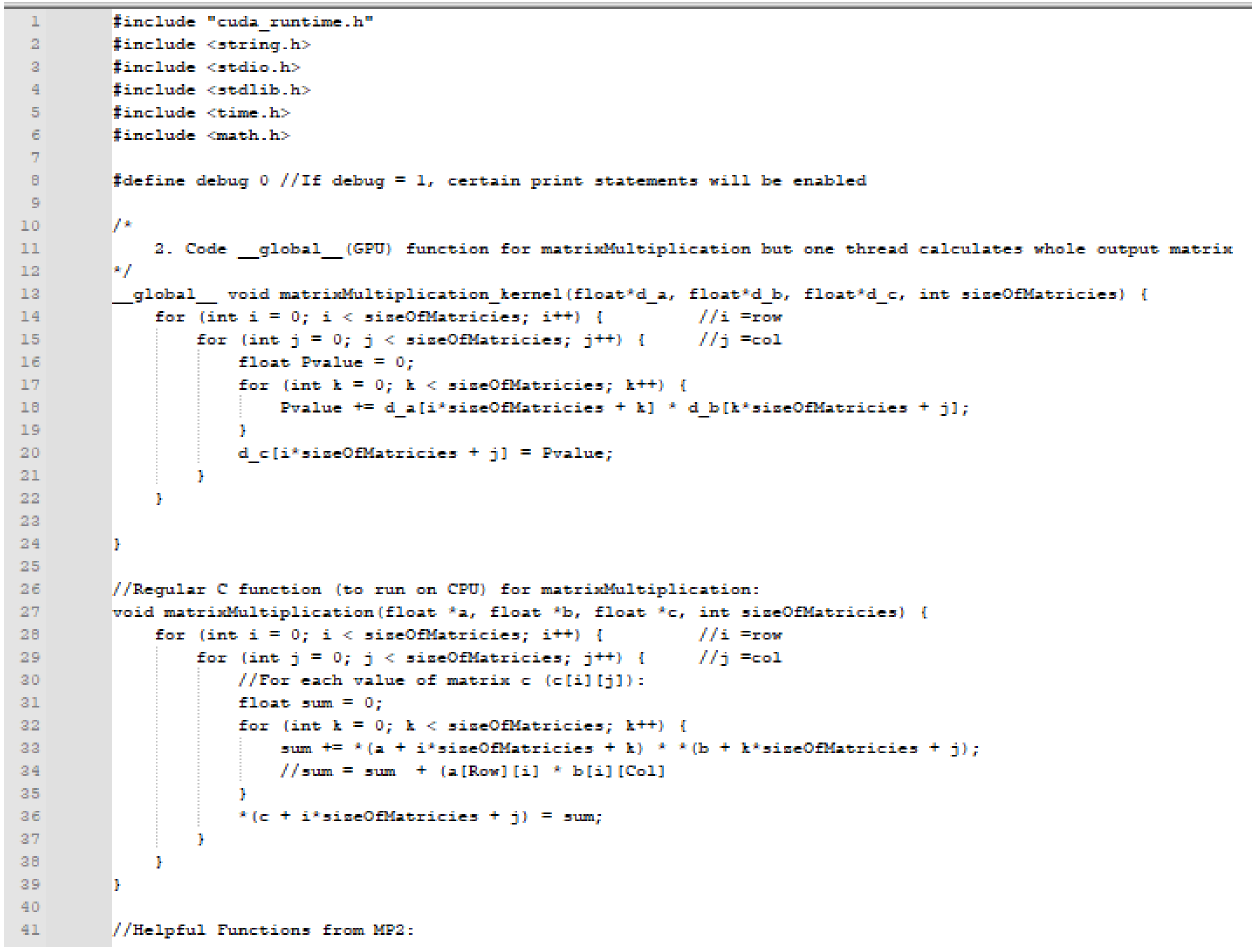
```c
int main(int argc, char *argv[]) {
    cudaDeviceProp deviceProps;

    //Get Device Name (They did this in tutorial so I'll do it here)
    cudaGetDeviceProperties(&deviceProps, 0);
    printf("CUDA device [%s]\n", deviceProps.name);
    if (debug == 1) {
        printf("\tNumber of Mulitprocessors: %d\n", deviceProps.multiProcessorCount);
        printf("\tMax Threads Per Block: %d\n", deviceProps.maxThreadsPerBlock);
        printf("\tMax Dimension of a Block: %d\n", deviceProps.maxThreadsDim);
        printf("\tMax Dimension of a Grid: %d\n", deviceProps.maxGridSize);
    }

    //Set Size of Matricies:
    const int dimOfMatricies = 2000; //Value determines size of matricies Ex. dimOfMatricies = 5 will result in 5x5 matricies
    printf("Dimensions of Matricies: %dx%d\n", dimOfMatricies, dimOfMatricies);

    //Calculate amount of memory they take:
    int nbytes = dimOfMatricies*dimOfMatricies*sizeof(float);

    //Allocate host memory for matricies:
    float *a = 0;
    float *b = 0;
    float *c = 0;
    float *cCopy = 0;                   //Pointer to a copy of second output matrix, to be used to observe GPU to host tranfer time
    cudaMallocHost((void**)&a, nbytes); //Allocates host memory for matrix A, and points pointer a to first value.
    cudaMallocHost((void**)&b, nbytes);
    cudaMallocHost((void**)&c, nbytes);
    cudaMallocHost((void**)&cCopy, nbytes);

    if (debug == 1) {
        printf("Memory Locations of matricies:\n");
        printf("\ta = %x\n",&a);
        printf("\tb = %x\n", &b);
        printf("\tc = %x\n", &c);
    }

    //Generate input matricies into memory

    srand(time(NULL));
    fillMatrix(a, dimOfMatricies);
    fillMatrix(b, dimOfMatricies);

    if (debug == 1) {
        printf("Matrix A:\n");
        printMatrix(a, dimOfMatricies);
        printf("Matrix B:\n");
        printMatrix(b, dimOfMatricies);
    }

    //1e. Allocate device memory for matricies:
    float *d_a = 0;
    float *d_b = 0;
    float *d_c = 0;
    cudaMalloc((void**)&d_a, nbytes); //Allocates memory for matrix A, and points pointer a to first value.
    cudaMemset(d_a, 255, nbytes);     //Sets all allocated bytes to 255 (they did this in tutorial so i did it here)
    cudaMalloc((void**)&d_b, nbytes);
    cudaMemset(d_b, 255, nbytes);
    cudaMalloc((void**)&d_c, nbytes);
    cudaMemset(d_c, 255, nbytes);

    //Set kernel launch configuration
    int blckWidth = 16;
    int threadsPerBlock = blckWidth*blckWidth;
    int threadsNeeded = ceil(dimOfMatricies*dimOfMatricies); //Because in this configuration one thread only produces one value
    int blocksNeeded = ceil(threadsNeeded / threadsPerBlock);
    if (blocksNeeded < 1) blocksNeeded++;
    int gridWidth = ceil(sqrt(blocksNeeded)); //Note grid will be square
```



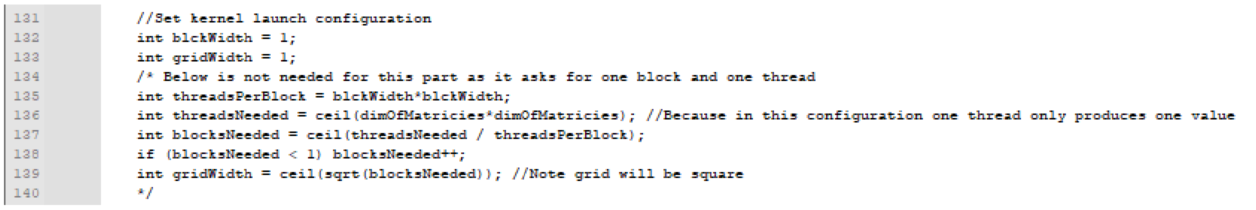
```
    if (debug == 1) {
        printf("Block Width: %d\n", blckWidth);
        printf("Grid Width: %d\n", gridWidth);
    }

    dim3 dimBlocks = dim3(blckWidth, blckWidth);
    dim3 dimGrid = dim3(gridWidth, gridWidth);

    //create cuda event handles
    cudaEvent_t start, stopToDevice, startToHost, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stopToDevice);
    cudaEventCreate(&startToHost);
    cudaEventCreate(&stop);

    cudaDeviceSynchronize();
    float gpu_time = 0.0f;
    float toGPU_time = 0.0f;
    float fromGPU_time = 0.0f;

    //asynchronously issue work to the GPU (all stream 0)
    cudaEventRecord(start, 0);

    //Copy inputs to device
    cudaMemcpyAsync(d_a, a, nbytes, cudaMemcpyHostToDevice, 0);
    cudaMemcpyAsync(d_b, b, nbytes, cudaMemcpyHostToDevice, 0);

    //Save time it took to write to device
    cudaEventRecord(stopToDevice, 0);

    //Call Kernel
    matrixMultiplication_kernel << <dimGrid, dimBlocks, 0, 0 >> >(d_a, d_b, d_c, dimOfMatricies);
    //Note: only section after >>> is the actual function parameters

    //Save time before saving to host
    cudaEventRecord(startToHost, 0);

    //Copy outputs from device
    cudaMemcpyAsync(c, d_c, nbytes, cudaMemcpyDeviceToHost, 0);
    cudaMemcpyAsync(cCopy, d_c, nbytes, cudaMemcpyDeviceToHost, 0); //A second matrix is recorded as question asked for tranfer to two matricies

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stopToDevice);
    cudaEventSynchronize(startToHost);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&gpu_time, start, stop);           //Note: gpu_time includes toGPU_time and fromGPU_time
    cudaEventElapsedTime(&toGPU_time, start, stopToDevice);
    cudaEventElapsedTime(&fromGPU_time, startToHost, stop);

    //print the GPU times
    printf("time spent executing by the GPU: %.2f\n", gpu_time);
    printf("time spent transferring input data to the GPU: %.2f\n", toGPU_time);
    printf("time spent transferring output data to host: %.2f\n", fromGPU_time);

    //Calculate matrixMultiplication using CPU:
    float *d = 0;
    cudaMallocHost((void**)&d, nbytes);
    matrixMultiplication(a, b, d, dimOfMatricies); //d is output matrix

                                                //Check Output
    bool bFinalResults = (bool)correct_output(c, d, dimOfMatricies); //check if c(from GPU) = d(from CPU)
    if (bFinalResults == true) {
        printf("Test PASSED\n");
    }

    if (debug == 1) {
        printf("Matrix C:\n");
        printMatrix(c, dimOfMatricies);
        printf("Correct Matrix:\n");
        printMatrix(d, dimOfMatricies);
    }

    //release resources
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    cudaFreeHost(a);
    cudaFreeHost(b);
    cudaFreeHost(c);
    cudaFreeHost(cCopy);
    cudaFreeHost(d);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaDeviceReset();

    return 0;

}
```

2. Find time it takes to execute matrix on GPU with one block and one thread vs CPU

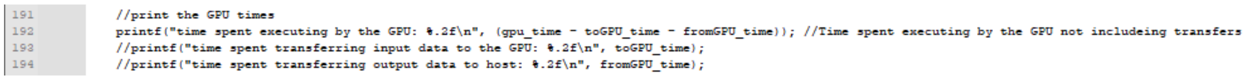
Note: block width is equal to 16. gpu execution time does NOT include transfer times.



```c
#include "cuda_runtime.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define debug 0 //If debug = 1, certain print statements will be enabled

/*
    2. Code __global__(GPU) function for matrixMultiplication but one thread calculates whole output matrix
*/
__global__ void matrixMultiplication_kernel(float*d_a, float*d_b, float*d_c, int sizeOfMatricies) {
    for (int i = 0; i < sizeOfMatricies; i++) {          //i =row
        for (int j = 0; j < sizeOfMatricies; j++) {      //j =col
            float Pvalue = 0;
            for (int k = 0; k < sizeOfMatricies; k++) {
                Pvalue += d_a[i*sizeOfMatricies + k] * d_b[k*sizeOfMatricies + j];
            }
            d_c[i*sizeOfMatricies + j] = Pvalue;
        }
    }

}

//Regular C function (to run on CPU) for matrixMultiplication:
void matrixMultiplication(float *a, float *b, float *c, int sizeOfMatricies) {
    for (int i = 0; i < sizeOfMatricies; i++) {          //i =row
        for (int j = 0; j < sizeOfMatricies; j++) {      //j =col
            //For each value of matrix c (c[i][j]):
            float sum = 0;
            for (int k = 0; k < sizeOfMatricies; k++) {
                sum += *(a + i*sizeOfMatricies + k) * *(b + k*sizeOfMatricies + j);
                //sum = sum  + (a[Row][i] * b[i][Col]
            }
            *(c + i*sizeOfMatricies + j) = sum;
        }
    }
}

//Helpful Functions from MP2:
```



```c
    //Set kernel launch configuration
    int blckWidth = 1;
    int gridWidth = 1;
    /* Below is not needed for this part as it asks for one block and one thread
    int threadsPerBlock = blckWidth*blckWidth;
    int threadsNeeded = ceil(dimOfMatricies*dimOfMatricies); //Because in this configuration one thread only produces one value
    int blocksNeeded = ceil(threadsNeeded / threadsPerBlock);
    if (blocksNeeded < 1) blocksNeeded++;
    int gridWidth = ceil(sqrt(blocksNeeded)); //Note grid will be square
    */
```

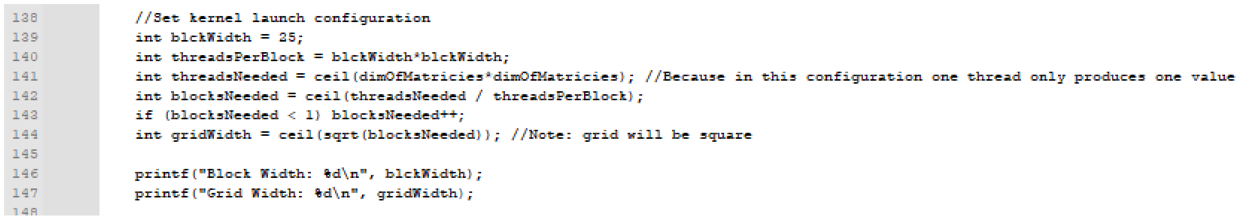


```c
    //print the GPU times
    printf("time spent executing by the GPU: %.2f\n", (gpu_time - toGPU_time - fromGPU_time)); //Time spent executing by the GPU not includeing transfers
    //printf("time spent transferring input data to the GPU: %.2f\n", toGPU_time);
    //printf("time spent transferring output data to host: %.2f\n", fromGPU_time);
```

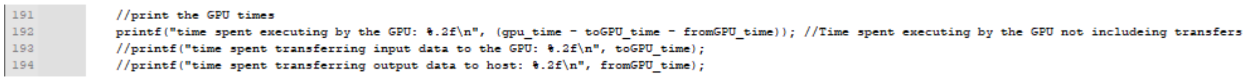
Note: All parts not shown were identical to the parts from the first configuration. Full script can be viewed from attached .zip

3. Increase block_width as matrix size increases.

Note: Gpu execution time does NOT include transfer times.



```c
    //Set kernel launch configuration
    int blckWidth = 25;
    int threadsPerBlock = blckWidth*blckWidth;
    int threadsNeeded = ceil(dimOfMatricies*dimOfMatricies); //Because in this configuration one thread only produces one value
    int blocksNeeded = ceil(threadsNeeded / threadsPerBlock);
    if (blocksNeeded < 1) blocksNeeded++;
    int gridWidth = ceil(sqrt(blocksNeeded)); //Note: grid will be square

    printf("Block Width: %d\n", blckWidth);
    printf("Grid Width: %d\n", gridWidth);
```



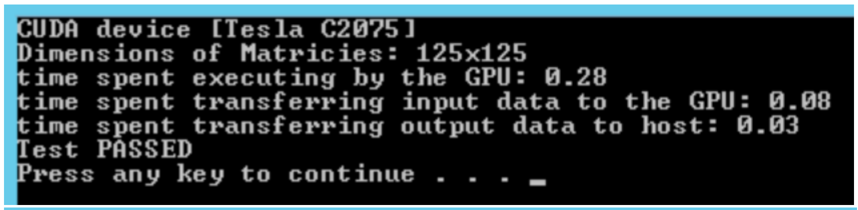
```
    //print the GPU times
    printf("time spent executing by the GPU: %.2f\n", (gpu_time - toGPU_time - fromGPU_time)); //Time spent executing by the GPU not includeing transfers
    //printf("time spent transferring input data to the GPU: %.2f\n", toGPU_time);
    //printf("time spent transferring output data to host: %.2f\n", fromGPU_time);
```

Note: All parts not shown were identical to the parts from the first configuration. Full scripts can be viewed from attached .zip

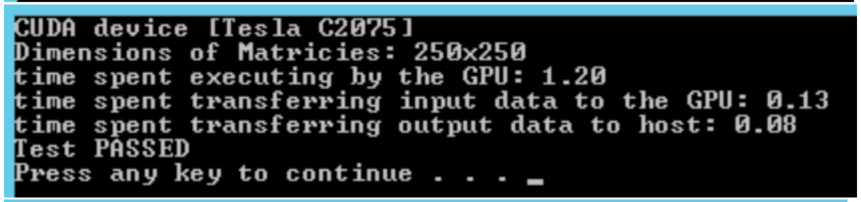
Output

For each question, outputs were recorded for the following matrix sizes: 125x125, 250x250, 500x500, 1000x1000, 2000x2000; with each configuration from above.

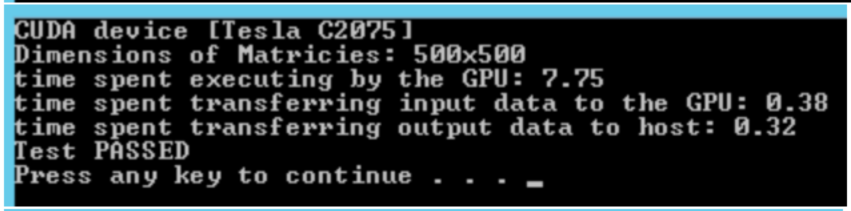
1. Find the time it takes to transfer two matrices worth of data to and from device:



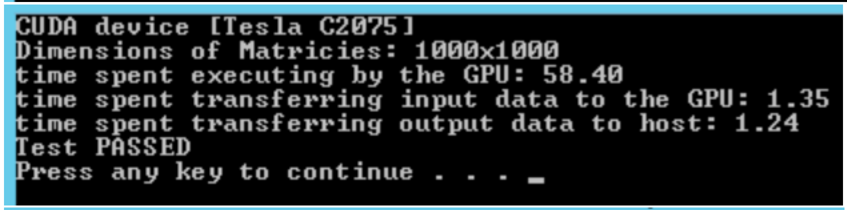
```
CUDA device [Tesla C2075]
Dimensions of Matricies: 125x125
time spent executing by the GPU: 0.28
time spent transferring input data to the GPU: 0.08
time spent transferring output data to host: 0.03
Test PASSED
Press any key to continue . . .
```



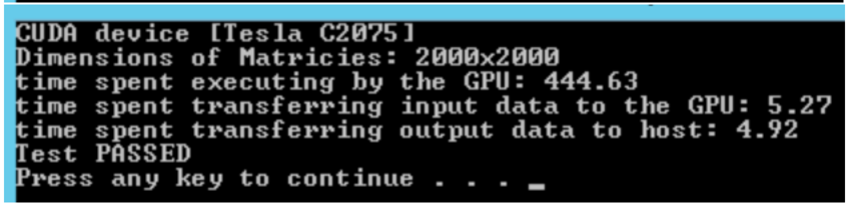
```
CUDA device [Tesla C2075]
Dimensions of Matricies: 250x250
time spent executing by the GPU: 1.20
time spent transferring input data to the GPU: 0.13
time spent transferring output data to host: 0.08
Test PASSED
Press any key to continue . . .
```



```
CUDA device [Tesla C2075]
Dimensions of Matricies: 500x500
time spent executing by the GPU: 7.75
time spent transferring input data to the GPU: 0.38
time spent transferring output data to host: 0.32
Test PASSED
Press any key to continue . . .
```

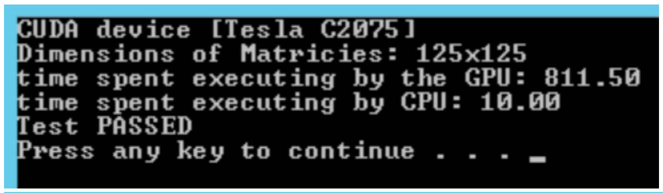


```
CUDA device [Tesla C2075]
Dimensions of Matricies: 1000x1000
time spent executing by the GPU: 58.40
time spent transferring input data to the GPU: 1.35
time spent transferring output data to host: 1.24
Test PASSED
Press any key to continue . . .
```

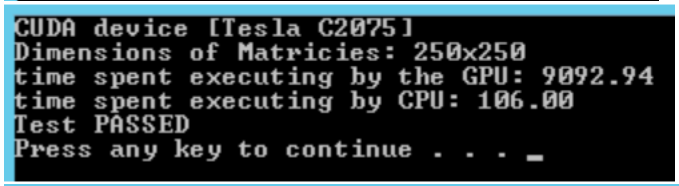


```
CUDA device [Tesla C2075]
Dimensions of Matricies: 2000x2000
time spent executing by the GPU: 444.63
time spent transferring input data to the GPU: 5.27
time spent transferring output data to host: 4.92
Test PASSED
Press any key to continue . . .
```

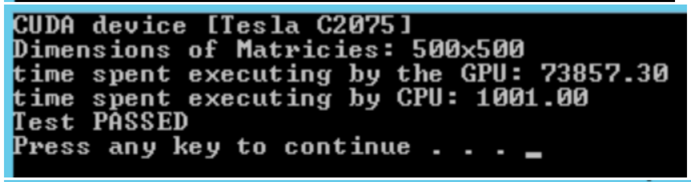
2. Find time it takes to execute matrix on GPU with one block and one thread vs CPU



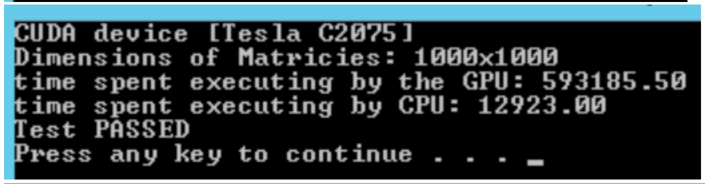
```
CUDA device [Tesla C2075]
Dimensions of Matricies: 125x125
time spent executing by the GPU: 811.50
time spent executing by CPU: 10.00
Test PASSED
Press any key to continue . . . _
```



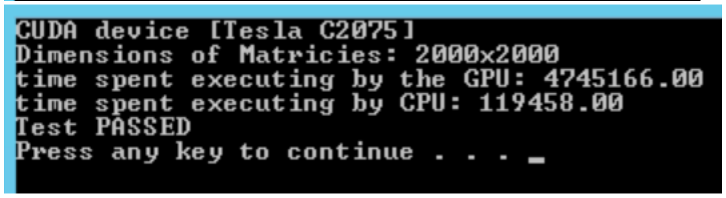
```
CUDA device [Tesla C2075]
Dimensions of Matricies: 250x250
time spent executing by the GPU: 9092.94
time spent executing by CPU: 106.00
Test PASSED
Press any key to continue . . . _
```



```
CUDA device [Tesla C2075]
Dimensions of Matricies: 500x500
time spent executing by the GPU: 73857.30
time spent executing by CPU: 1001.00
Test PASSED
Press any key to continue . . . _
```

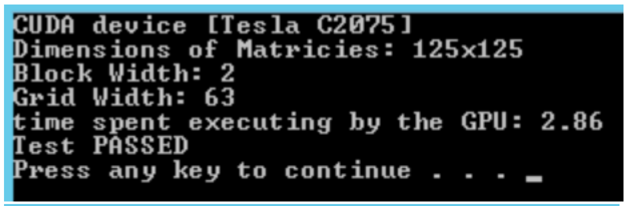


```
CUDA device [Tesla C2075]
Dimensions of Matricies: 1000x1000
time spent executing by the GPU: 593185.50
time spent executing by CPU: 12923.00
Test PASSED
Press any key to continue . . . _
```

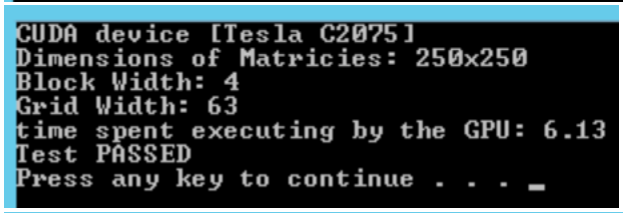


```
CUDA device [Tesla C2075]
Dimensions of Matricies: 2000x2000
time spent executing by the GPU: 4745166.00
time spent executing by CPU: 119458.00
Test PASSED
Press any key to continue . . . _
```

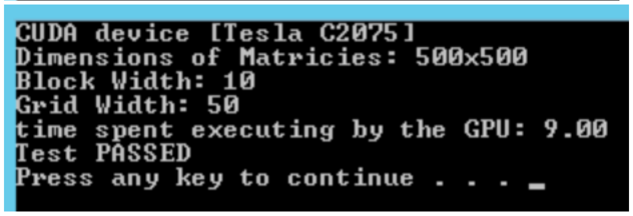
3. Increase block_width as matrix size increases.



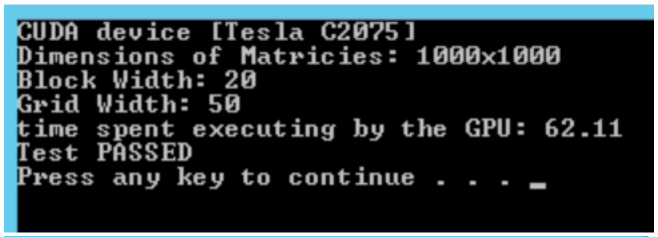
```
CUDA device [Tesla C2075]
Dimensions of Matricies: 125x125
Block Width: 2
Grid Width: 63
time spent executing by the GPU: 2.86
Test PASSED
Press any key to continue . . . _
```



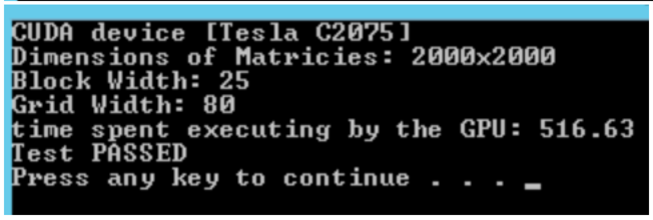
```
CUDA device [Tesla C2075]
Dimensions of Matricies: 250x250
Block Width: 4
Grid Width: 63
time spent executing by the GPU: 6.13
Test PASSED
Press any key to continue . . . _
```



```
CUDA device [Tesla C2075]
Dimensions of Matricies: 500x500
Block Width: 10
Grid Width: 50
time spent executing by the GPU: 9.00
Test PASSED
Press any key to continue . . . _
```



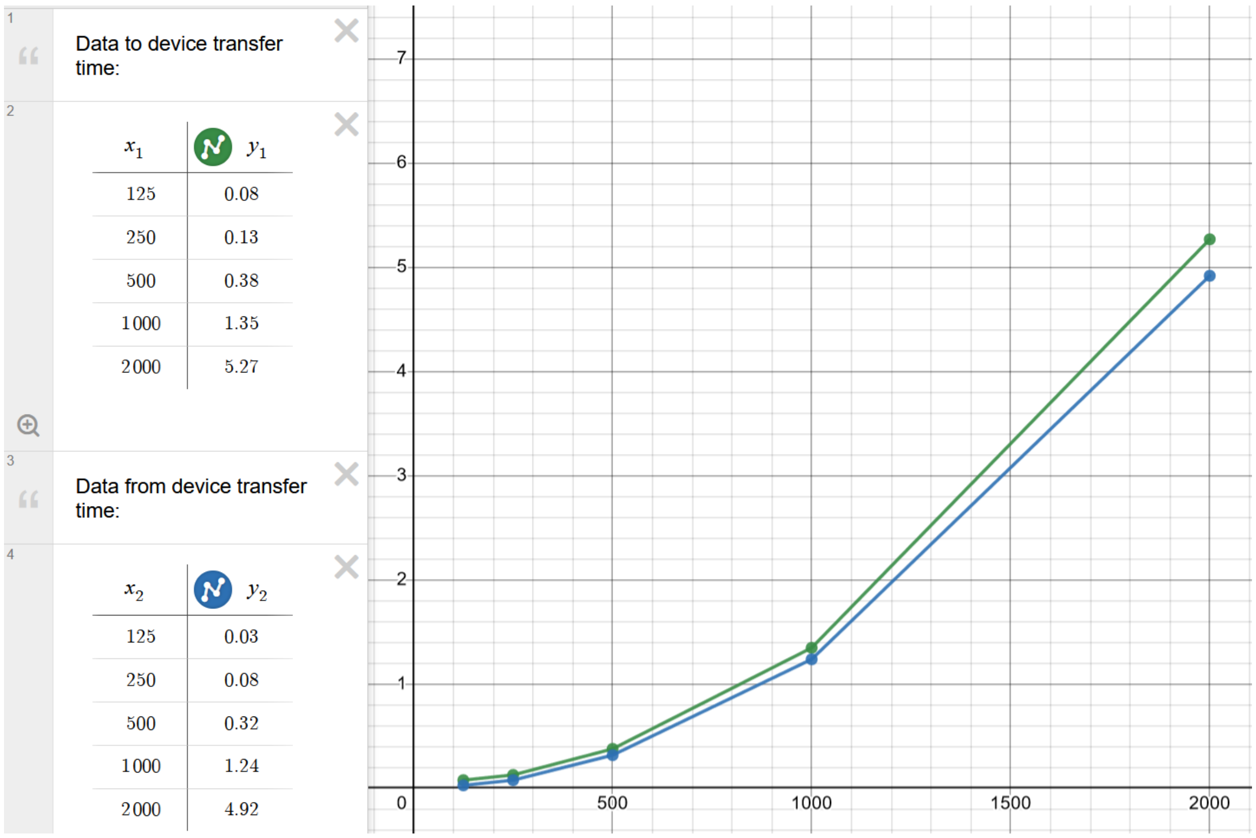
```
CUDA device [Tesla C2075]
Dimensions of Matricies: 1000x1000
Block Width: 20
Grid Width: 50
time spent executing by the GPU: 62.11
Test PASSED
Press any key to continue . . . _
```



```
CUDA device [Tesla C2075]
Dimensions of Matricies: 2000x2000
Block Width: 25
Grid Width: 80
time spent executing by the GPU: 516.63
Test PASSED
Press any key to continue . . . _
```

Analysis

1. For the first configuration, the following graph showing transfer times to and from device versus matrix size, were generated from the outputs above. The x-axis is block width and the y-axis is transfer time (in milliseconds):



Data to device transfer time:

x_1	y_1
125	0.08
250	0.13
500	0.38
1000	1.35
2000	5.27

Data from device transfer time:

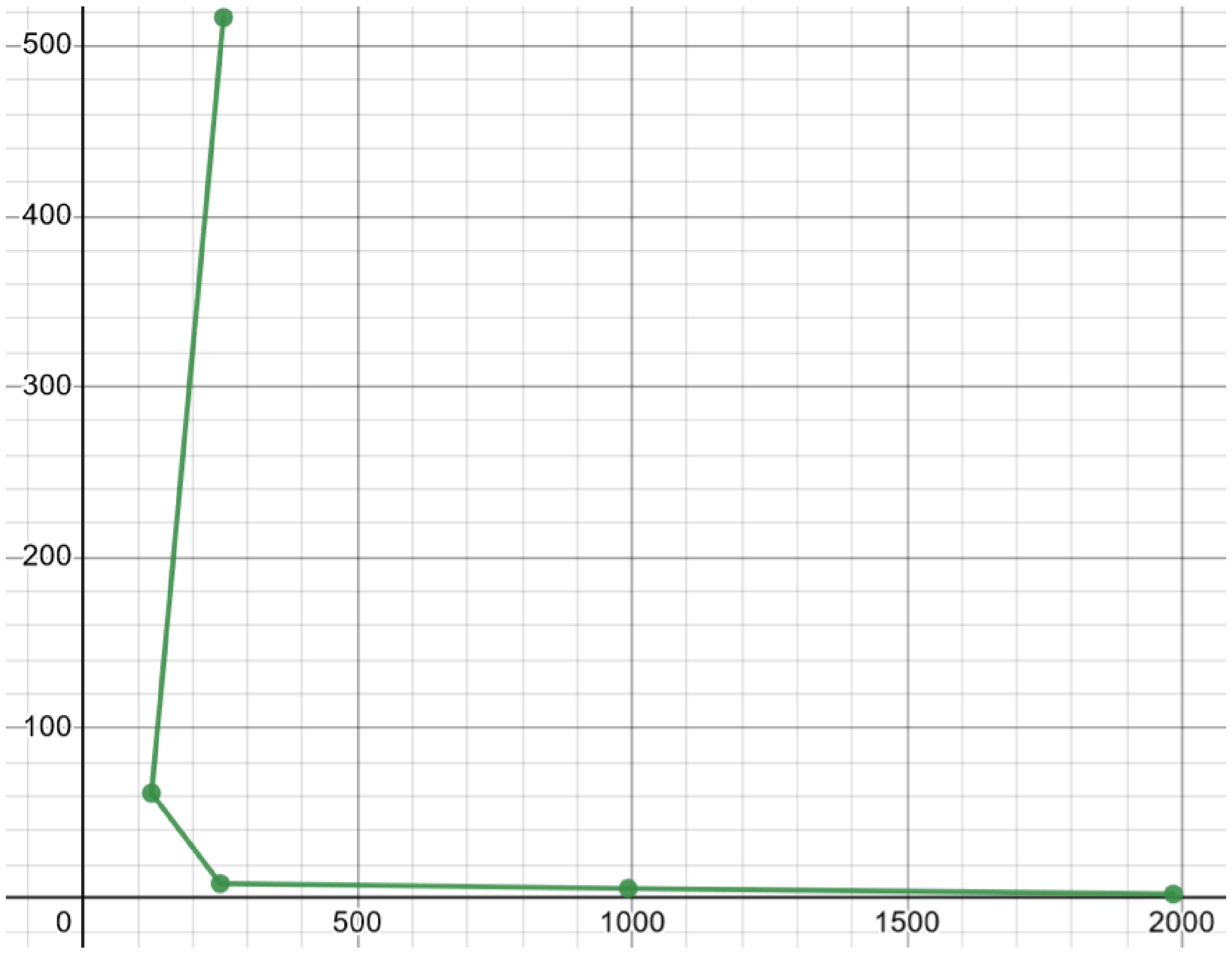
x_2	y_2
125	0.03
250	0.08
500	0.32
1000	1.24
2000	4.92

There is for the most part no difference in transfer speeds to and from the gpu device, with only a slight—seemingly proportional difference being seen (difference < 0.4 ms for 2000x2000). This small difference is likely due to a difference in memory speeds between the device and host.

2. Observing the outputs from the second configuration (above in Outputs), it becomes obvious that setting the grid width and block width to one had a detrimental impact on the GPU's performance. When comparing this performance to the performance of the matrix multiplication on the CPU, its clear that it is not always beneficial to offload your matrix multiplication to the GPU; it is only optimal to do so if the kernel configuration is optimized correctly.

3. The outputs for the third configuration were generated from the following matrixWidth and blockWidth pairs: (125, 2), (250, 4), (500, 10), (1000,20), (2000,25).

The following graph of kernel execution time vs (numBlocks/blockWidth) were generated from the outputs above (Note: numBlocks = gridWidth^2). The x-axis is numBlocks/blockWidth and the y-axis is execution time (in milliseconds):



This clearly shows that when there are more blocks with less threads per block, the execution time will be faster than when there is less blocks with more threads per block.

a) Using our knowledge of the matrix multiplication algorithm, we know that to calculate an element in output matrix C, we must access every element from input matrix A sharing a row with the output element, and we must access every element from input matrix B sharing a column with the output element. This means for every element in matrix C we are accessing 2*matrixWidth number of input elements:

number of total accesses = amount of output matrix elements * 2*matrixWidth

Since C is a square matrix ‘matrixWidth’ wide, the number of output elements will be: matrixWidth^2

If we are trying to find the amount of times each input element is accessed:

average number of times an input element is accessed = $\frac{number\ of\ total\ accesses}{number\ of\ input\ elements}$

The number of input elements is equal to all the elements in A plus all the elements from B. Since both of these square matrices are a ‘matrixWidth’ wide:

Number of input elements = 2(matrixWidth^2)

Combining all of these into one equation:

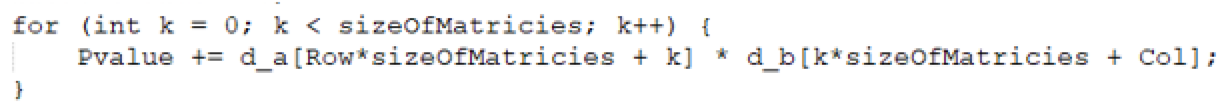
$$avgInputElementAccesses = \frac{totalAccesses}{numberOfInputElements}$$

$$avgInputElementAccesses = \frac{matrixWidth^2 * (2 * matrixWidth)}{2 * matrixWidth^2}$$

$$avgInputElementAccesses = matrixWidth$$

Therefore, for this configuration where the output matrixWidth is equal to the input matrices matrixWidth, the number of times a given input matrix element will be accessed is equal to the matrix width. This makes sense as a given element in A or B needs to be accessed once for each element in C, in the same row/column (=matrixWidth).

b) CGMA ratio is the number of operations a thread performs over the number of memory fetches. Looking at our matrix multiplication kernel for the third configuration (identical to the one for the first configuration), for each loop iteration within the kernel:



```
for (int k = 0; k < sizeOfMatricies; k++) {
    Pvalue += d_a[Row*sizeOfMatricies + k] * d_b[k*sizeOfMatricies + Col];
}
```

There’s two global memory accesses d_a[…] and d_b[…] for one floating-point multiplication and one floating-point addition operation. Giving a CGMA ratio of floating-point operations to bytes accessed from global memory of 0.25 FLOP/B.