Machine Problem 1 Report

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

Nathan Goodman: 20228249
Date: April 4th, 2023

CUDA Code

For the three different kernel configurations requested, three different CUDA scripts were created:

1. Each thread only produces a single output element and 16x16 threadblocks.

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

//2. Write a kernel that has each thread producing one output matrix. Kernel config should be 16x16 thread blocks

/* MatrixAddition Kernel.
Parameters: pointer to output matrix C, two Pointers to input matricies A and B,
dimensions of matricies A and B (remember they're square matriceies so this can be single int)
*/
__global__ void matrixAddition_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;
    int idx = Row*sizeOfMatricies + Col;
    //0 1 2 3 4
    //5 6 7 8 9
    if (idx < sizeOfMatricies*sizeOfMatricies) { //Avoid accessing beyond end off matricies
        d_c[idx] = d_a[idx] + d_b[idx];
    }
}


/* Created a matrixAddition function. Should basically be the same as the kernel function.
This function will be used to check whether the kernel function created correct output (used to check)
*/
void matrixAddition(float *a, float *b, float *c, int sizeOfMatricies) { //Note acc
    for (int i = 0; i < sizeOfMatricies; i++) {
        for (int j = 0; j < sizeOfMatricies; j++) {
            *(c + i*sizeOfMatricies + j) = *(a + i*sizeOfMatricies + j) + *(b + i*sizeOfMatricies + j);
            //C[i][j] = A[i][j] + B[i][j];
        }
    }
}

//3b. Create function to check if two matricies are equal (to be used to compare outputs)
int correct_output(float *a, float *b, int sizeOfMatricies) {
    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;
}

//Additional function to help with debugging:
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");
    }
}

//Function for generating  randomly intitialized square matricies of a given length
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 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);
    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);

    /*
    1. Define two square input matricies A and B, and matching output matrix
    Note: they're floats
    */
    const int dimOfMatricies = 250; //Value determines size of matricies Ex. dimOfMatricies = 5 will result in 5x5 matricies

    float C[dimOfMatricies][dimOfMatricies];

    //1b. Calculate amount of memory they take:
    int nbytes = dimOfMatricies*dimOfMatricies*sizeof(float);

    //1c. Allocate host memory for matricies:
    float *a = 0;
    float *b = 0;
    float *c = 0;
    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);

    //printf("a = %x\n",&a);
    //printf("b = %x\n", &b);
    //printf("c = %x\n", &c);

    //1d. Store input matricies into memory

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

    //printf("%f\n", *b);
    //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; //block width and kength
    int threadsPerBlock = blckWidth*blckWidth;
    int threadsNeeded = dimOfMatricies*dimOfMatricies; //Because in this configuration one thread only produces one value
    int numBlocks = threadsNeeded / threadsPerBlock;
    if (numBlocks < 1) numBlocks++;

    dim3 dimBlocks = dim3(blckWidth, blckWidth);//asks for 16X16=256 thread blocks
                                                //256 threads per block layed out in 16x16
    dim3 dimGrid = dim3(numBlocks, numBlocks);
```

```
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaDeviceSynchronize();
    float gpu_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);

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

    //Copy outputs from device
    cudaMemcpyAsync(c, d_c, nbytes, cudaMemcpyDeviceToHost, 0);

    //2b. Load output matricies from memory
    for (int i = 0; i < dimOfMatricies; i++) {
        for (int j = 0; j < dimOfMatricies; j++) {
            C[i][j] = *(c + i*dimOfMatricies + j);
        }
    }

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop); //stop is updated here
    cudaEventElapsedTime(&gpu_time, start, stop);

    //print the GPU times
    printf("time spent executing by the GPU: %.2f\n", gpu_time);

    //Calculate matrixAddition using CPU:
    float D[dimOfMatricies][dimOfMatricies];
    float *d = 0;
    cudaMallocHost((void**)&d, nbytes);
    matrixAddition(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");
    }

    //printf("Matrix C\n");
    //printMatrix(c, dimOfMatricies);
    //printf("\n");
    //printMatrix(d, dimOfMatricies);

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

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

    return 0;

}
```

2. Each thread produces an output row and 16 threads per block.

Note: Due to scripts for the most part being identical, only changed parts are shown. Full file is found within attached .zip

```
/* 3. Create a matrixAddition Kernel where each thread calculates a row of output values.
Parameters: pointer to output matrix C, two Pointers to input matricies A and B,
dimensions of matricies A and B (remember they're square matriceies so this can be single int)
*/
__global__ void matrixAddition_kernel(float*d_a, float*d_b, float*d_c, int sizeOfMatricies) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;          //Assume block

    idx = idx*sizeOfMatricies; //To account for already computed indicies
                               //Ex. thread 1 (idx=0) will compute output elements 0 (5*0),1(5*0+1),2,3,4
                               //   thread 2 (idx=1) will compute output elements 5 (5*1),6 (5*1+1),7,8,9

    int n = sizeOfMatricies*sizeOfMatricies;
    if (idx < n) { //Avoid accessing beyond end off matricies
        for (int i = 0; i < sizeOfMatricies; i++) {
            //Each thread will calculate a row of output matrix:
            d_c[idx + i] = d_a[idx + i] + d_b[idx + i];
        }
    }
}
```

```
//Set kernel launch configuration
int blckWidth = 4; //block width and length (16 threads per block)
int threadsPerBlock = blckWidth*blckWidth;
int threadsNeeded = dimOfMatricies*dimOfMatricies;
int numBlocks = threadsNeeded / threadsPerBlock;
if (numBlocks < 1) numBlocks++;

dim3 dimBlocks = dim3(16, 1);                  // 16 threads per block
dim3 dimGrid = dim3(numBlocks, 1);
```

3. Each thread produces a column of output and 16 threads per block.

```
/* 4. Create a matrixAddition Kernel where each thread calculates a column of output values..
Parameters: pointer to output matrix C, two Pointers to input matricies A and B,
dimensions of matricies A and B (remember they're square matriceies so this can be single int)
*/
__global__ void matrixAddition_kernel(float*d_a, float*d_b, float*d_c, int sizeOfMatricies) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;          //Assume block
    int n = sizeOfMatricies*sizeOfMatricies;
    if (idx < n) { //Avoid accessing beyond end off matricies
        for (int i = 0; i < sizeOfMatricies; i++) {
            //Each thread will calculate a column of output matrix:
            d_c[idx + sizeOfMatricies*i] = d_a[idx + sizeOfMatricies*i] + d_b[idx + sizeOfMatricies*i];
        }
    }
}
```

```
//Set kernel launch configuration
int blckWidth = 4; //block width and length (16 threads per block)
int threadsPerBlock = blckWidth*blckWidth;
int threadsNeeded = dimOfMatricies*dimOfMatricies;
int numBlocks = threadsNeeded / threadsPerBlock;
if (numBlocks < 1) numBlocks++;

dim3 dimBlocks = dim3(16, 1);                  // 16 threads per block
dim3 dimGrid = dim3(numBlocks, 1);
```

Output

Following guidelines from the Machine Problem 2 Document, the dimOfMatricies variable (equivalent to BLOCK_WIDTH from the lecture slides) was varied and outputs were recorded. Unfortunately for whatever reason, when attempting to input a value of 500 or greater, the script failed to execute despite no compilation errors:

```
const int dimOfMatricies = 500; //Value determines size of matricies Ex. dimOfMatricies = 5 will
```

```
C:\Windows\system32\cmd.exe
Press any key to continue . . .
```

While I could see this being a possibility for the first set of launch configurations (each thread only produces a single output) as to calculate 500*500 output elements you'd need 500*500 threads. This issue continued to persist despite improved resource management in second and third launch configurations.

As such, instead of the values 500x500, 1000x1000, and 2000x2000; the values: 60x60, 175x175, 300x300 were added in their place. The outputs for these matrix sizes for each kernel configuration can be viewed below. The outputs from top to bottom are: 60x60, 125x125, 175x,175, 250x250, 300x300:

1. Each thread only produces a single output element and 16x16 threadblocks.

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 1899400
        Max Dimension of a Grid: 1899412
time spent executing by the GPU: 0.18
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 14481928
        Max Dimension of a Grid: 14481940
time spent executing by the GPU: 41.36
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 12777900
        Max Dimension of a Grid: 12777912
time spent executing by the GPU: 170.85
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 4847812
        Max Dimension of a Grid: 4847824
time spent executing by the GPU: 1.48
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 17823960
        Max Dimension of a Grid: 17823972
time spent executing by the GPU: 2.91
Test PASSED
Press any key to continue . . .
```

2. Each thread produces an output row and 16 threads per block.

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 15531332
        Max Dimension of a Grid: 15531344
time spent executing by the GPU: 0.24
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 9107908
        Max Dimension of a Grid: 9107920
time spent executing by the GPU: 0.34
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 6945344
        Max Dimension of a Grid: 6945356
time spent executing by the GPU: 0.51
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 3601924
        Max Dimension of a Grid: 3601936
time spent executing by the GPU: 0.93
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 16448768
        Max Dimension of a Grid: 16448780
time spent executing by the GPU: 0.77
Test PASSED
Press any key to continue . . .
```

3. Each thread produces a column of output and 16 threads per block.

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 6157800
        Max Dimension of a Grid: 6157812
time spent executing by the GPU: 0.19
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 13106548
        Max Dimension of a Grid: 13106560
time spent executing by the GPU: 0.87
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 8124572
        Max Dimension of a Grid: 8124584
time spent executing by the GPU: 2.30
Test PASSED
Press any key to continue . . .
```

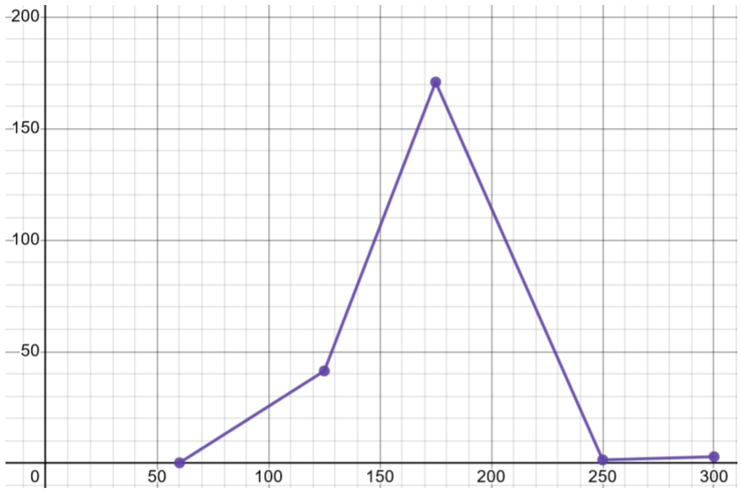
```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 17169232
        Max Dimension of a Grid: 17169244
time spent executing by the GPU: 6.11
Test PASSED
Press any key to continue . . .
```

```
CUDA device [Tesla C2075]
        Number of Mulitprocessors: 14
        Max Threads Per Block: 1024
        Max Dimension of a Block: 3930852
        Max Dimension of a Grid: 3930864
time spent executing by the GPU: 13.41
Test PASSED
Press any key to continue . . .
```

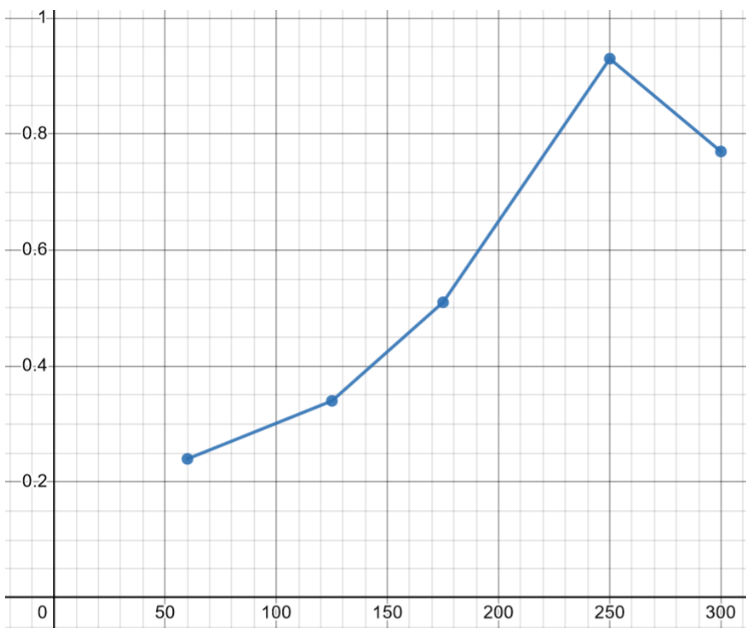
Analysis

The following graphs were generated for each of the launch configurations with the x-axis being the width of the matrix (in floats 4 bytes) and the y-axis being the GPU execution time (in ms):

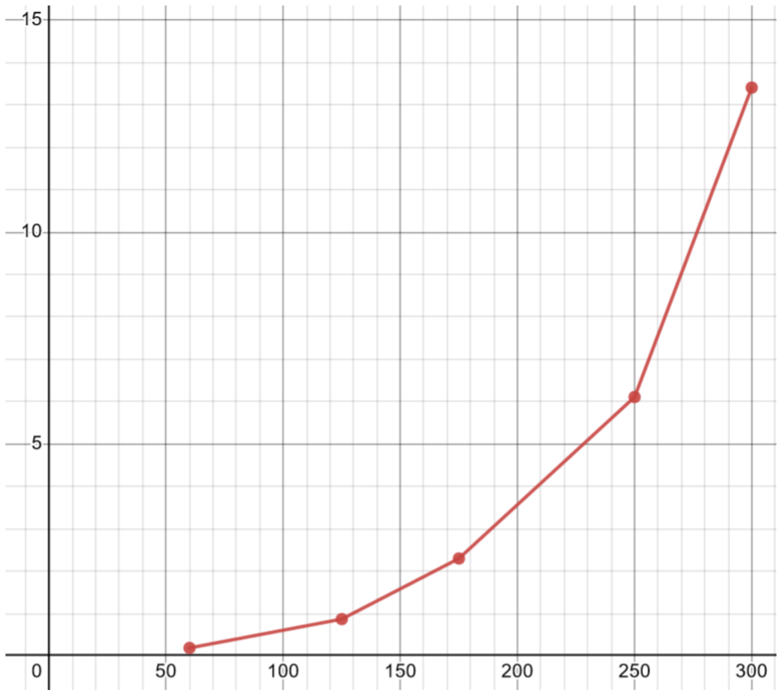
1. Each thread produces a single output element and 16x16 threadblocks:



2. Each thread produces a row of output and threads per block is 16:



3. Each thread produces a column of output and threads per block is 16:



As you can see above, (with the exception of the first launch configuration) as the width of the matrix grows, the execution time appears to increase exponentially. This makes sense after all the number of output elements or calculations performed is quadratically related to the width of the output matrix – elements to be calculated = matrix width * matrix width.

From the graphs above it is clear to see that the execution times for the 'row of output per thread and threads per block is 16' were the best, with them still being extremely low even as matrix width increased. It is expected that this outperformed the 'column of output' matrix addition as there were less operations being performed during each loop iteration (within kernel function).

On top of the quick times, both the column per thread and row per thread strategies save system resources as less threads have to be allocated for a task when compared to the element per thread algorithm.

As for the first launch configuration, I believe the unusual behavior and super long execution times were due to the measurements being recorded during a period of high traffic to the virtual machines (around 6pm the day before deadline). As mentioned previously, this methodology seeks to increase use of system resources to maximize the total number of tasks in parallel across all threads. It is unexpected then, that the times from this strategy would (1) be greater than the other two methods and (2) be so substantially greater than the other two.