# Machine Problem 1

Jacob Badali ~ 20290739 ~ March 13th, 2024

# Part 1

Code was written to print out the proper parameters, the output is in Figure 1. The full code can be found in the Appendix.

Microsoft Visual Studio Debug Console

```
Device 0 Information:
Name: NVIDIA RTX A2000 12GB
Compute Capability: 8.6
Clock Rate: 1200000 kHz
Number of SM (Streaming Multiprocessors): 26
Number of Cores per SM: 0
Warp Size: 32
Global Memory Size: 12878086144 bytes
Constant Memory Size: 65536 bytes
Shared Memory Size per Block: 49152 bytes
Registers per Block: 65536
Max Threads per Block: 1024
Max Size of Each Dimension of a Block: (1024, 1024, 64)
Max Size of Each Dimension of a Grid: (2147483647, 65535, 65535)


C:\Users\20jasb1\source\repos\MP1_Part1\x64\Debug\MP1_Part1.exe (process 25396) exited with code 0.
Press any key to close this window . . .
```

*Figure 1 Output Part 1*

# Part 2

## Section 1

Code was written to transfer the matrices from the host to device. The results for different matrix sizes can be found below. Plotting these results can be found below the table of results. Note care was taken to remove outliers from these tables.

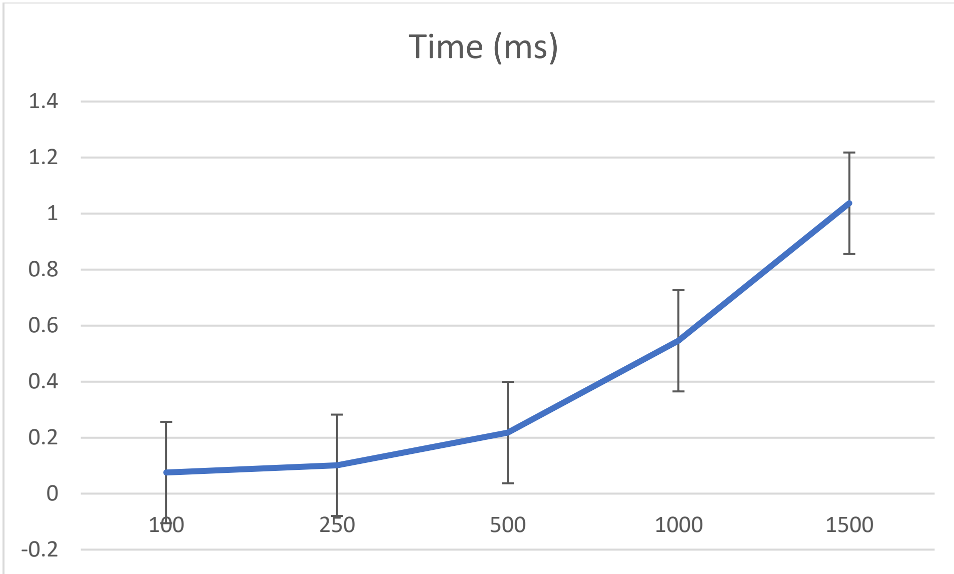
Example of output is seen below:

```
Transfer Host to Device, size[100]: 0.075392 ms || Transfer Device to Host, size[100]: 0.026592 ms

C:\Users\20jasb1\source\repos\MP1_Part1\x64\Debug\MP1_Part1.exe (process 25576) exited with code 0.
Press any key to close this window . . .
```

| **Matrix Size** | **Time (ms)** |
|---|---|
| 100 | 0.075392 |
| 250 | 0.101120 |
| 500 | 0.217952 |
| 1000 | 0.545792 |
| 1500 | 1.036960 |

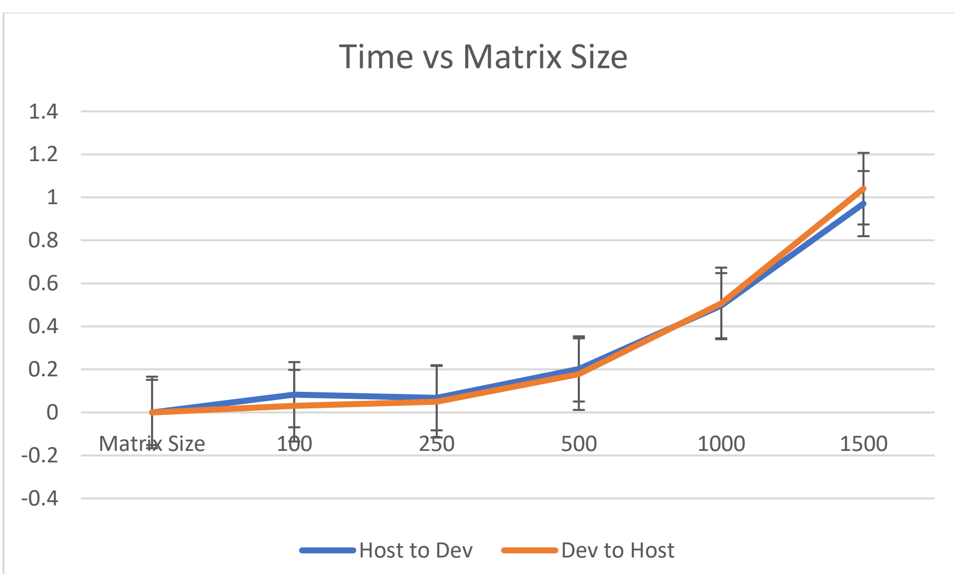
A plot can be found below:




The code was modified to also transfer data back from device to host, results seen below. Care was taken to remove outliers from these results.

| Matrix Size | Time (ms) Host to Device | Time (ms) Device to Host |
|---|---|---|
| 100 | 0.082720 | 0.031360 |
| 250 | 0.068384 | 0.049920 |
| 500 | 0.202464 | 0.178208 |
| 1000 | 0.496544 | 0.507104 |
| 1500 | 0.971104 | 1.040672 |

A plot can be seen below.

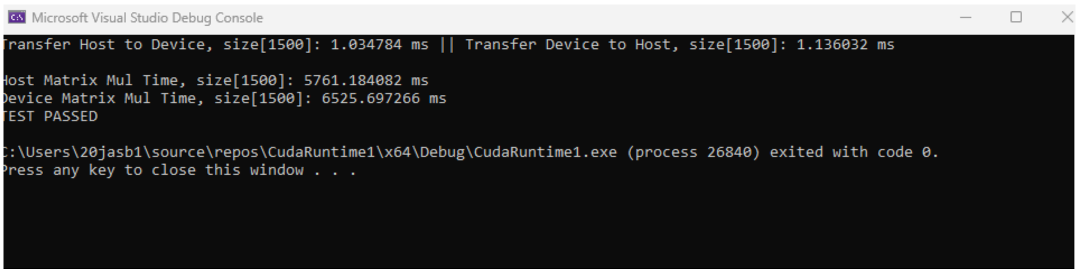


You can see that there is a small difference between transfer times comparing host to device and device to host. This makes sense, as the PCIe transfer bus do not operate in a traditionally parallel mode. For this reason, since the CPU is better at handling series operations, it would be faster.

Code for Part 2 Section 1 can be found in the Appendix.

## Section 2

In section 2, code needed to be modified to perform matrix multiplication on the CPU and GPU. At first, a BLOCK_WIDTH of 1 (1 thread) was used, along with the number of blocks being 1. Sample output seen in. Table containing results is below.



```
Microsoft Visual Studio Debug Console
ransfer Host to Device, size[1500]: 1.034784 ms || Transfer Device to Host, size[1500]: 1.136032 ms

ost Matrix Mul Time, size[1500]: 5761.184082 ms
evice Matrix Mul Time, size[1500]: 6525.697266 ms
EST PASSED

:\Users\20jasb1\source\repos\CudaRuntime1\x64\Debug\CudaRuntime1.exe (process 26840) exited with code 0.
ress any key to close this window . . .
```

Note: Time in ms.

| Matrix Size | GPU Time (mul) | CPU Time (mul) |
|---|---|---|
| 100 | 3.063584 | 1.478560 |
| 250 | 42.656063 | 23.614529 |
| 500 | 303.493439 | 191.966278 |
| 1000 | 2063.967773 | 1572.421021 |
| 1500 | 6525.697266 | 5761.184082 |

If you add the transfer time on top of this, you can see it would not make a significant difference, as it is fractions of milliseconds. With only one thread, it is not beneficial ever to offload to GPU. (This should change as more threads are added).

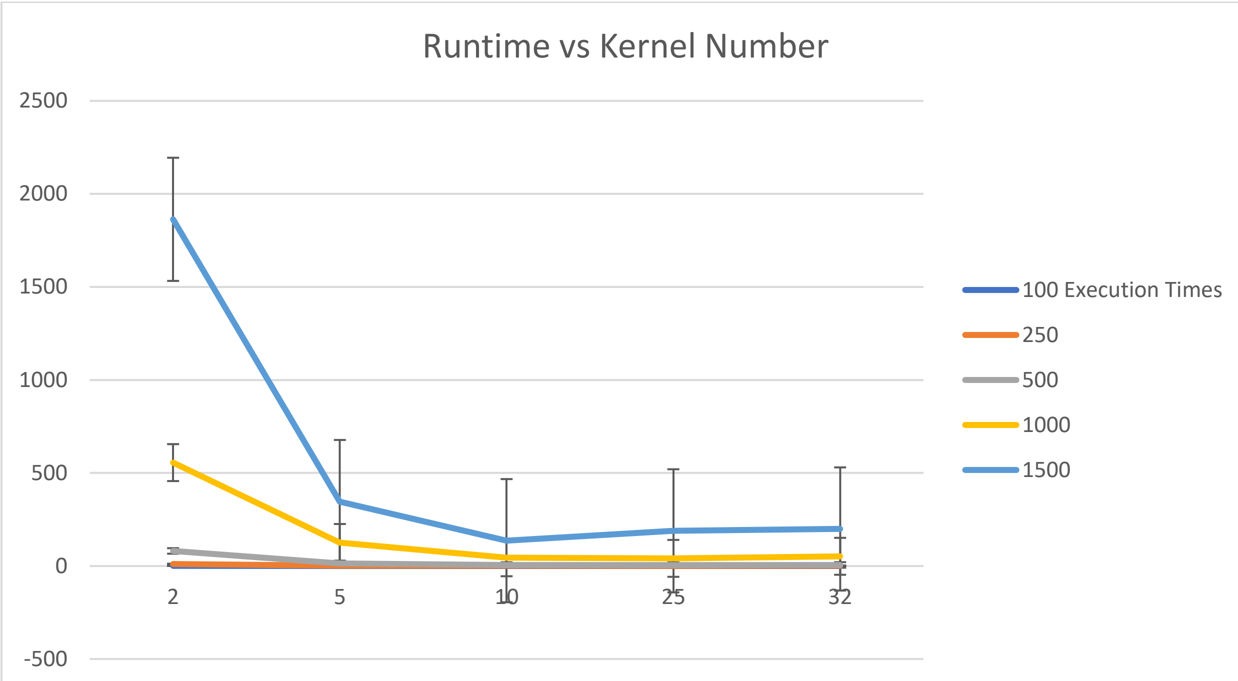
## Section 3

The block width (number of kernels) was changed, and a sample of the following results were found:

| Time, 1000x1000 | | | | | |
|---|---|---|---|---|---|
| | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| CPU | 1605.687 | 1562.348 | 1572.901 | 1546.797 | 1571.933 |
| GPU Threads | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| 2 | 586.8982 | 530.8599 | 566.4705 | 539.0612 | 555.8224 |
| 5 | 127.2612 | 128.7012 | 125.0887 | 124.5066 | 126.3894 |
| 10 | 43.92358 | 45.28125 | 43.44951 | 44.33088 | 44.2463 |
| 25 | 41.75322 | 40.19728 | 41.18055 | 41.1024 | 41.05836 |
| 32 | 52.33053 | 51.75632 | 52.17632 | 52.30147 | 52.14116 |

You can see full tables in the appendix.

These were plotted for each matrix size and plotted. Resulting in:




As you can see, there is exponential decline at the start, then it evens out as you get into 10 to 32 threads. To answer the two questions:

a) For each element in each matrix M and N, the CUDA grid is responsible for computing one element of the output matrix P. For just one element of matrix P, each element of input matrix M (one row) and each element of input matrix N (one column) needs to be loaded.
b) CGMA Ratio is calculated by Amount of FLOPs/Number of Global Memory Accesses. In the matrix multiplication kernel, the ratio is given below.

Legend:

Blue = Global memory access

Red = Floating point multiplication

Green = Floating point addition

```
//Multiplication kernel function
__global__ void mulKernel(float* M, float* N, float* P, int size) {
    int rows = blockIdx.y * blockDim.y + threadIdx.y;
    int cols = blockIdx.x * blockDim.x + threadIdx.x;


    if (rows < size && cols < size) {
        float temp_sum = 0.0;
        for (int i = 0; i < size; i++) {
            temp_sum += M[rows * size + i] * N[i * size + cols];
        }
        P[rows * size + cols] = temp_sum;
    }
}
```

**This gives us a CGMA Ratio of 2/3**.

# Appendix

## Part 1 Code

```
#include <cuda_runtime.h>
#include <stdio.h>

// Helper function to convert compute capability to the number of cores
int ConvertSMVer2Cores(int major, int minor) {
    int cores;

    switch ((major << 4) + minor) {
    case 0x10:
        cores = 8;
        break;
    case 0x11:
    case 0x12:
        cores = 8;
        break;
    case 0x13:
        cores = 32;
        break;
    case 0x20:
        cores = 32;
        break;
    default:
        cores = 0;
        break;
    }

    return cores;
}


int main() {
    cudaSetDevice(0); // Set the device to GPU 0 (or the appropriate GPU index)

    int num_devices;
    cudaGetDeviceCount(&num_devices);

    if (num_devices == 0) {
        fprintf(stderr, "No CUDA devices found.\n");
        return 1;
    }

    for (int device_id = 0; device_id < num_devices; ++device_id) {
        cudaDeviceProp device_prop;
        cudaError_t cuda_status = cudaGetDeviceProperties(&device_prop, device_id);

        if (cuda_status != cudaSuccess) {
            fprintf(stderr, "Error: cudaGetDeviceProperties failed with error code 
%d\n", cuda_status);
            return 1;
        }

        printf("Device %d Information:\n", device_id);
        printf("Name: %s\n", device_prop.name);
        printf("Compute Capability: %d.%d\n", device_prop.major, device_prop.minor);
        printf("Clock Rate: %d kHz\n", device_prop.clockRate);
```

```
        printf("Number of SM (Streaming Multiprocessors): %d\n",
device_prop.multiProcessorCount);
        printf("Number of Cores per SM: %d\n", ConvertSMVer2Cores(device_prop.major,
device_prop.minor) * device_prop.multiProcessorCount);
        printf("Warp Size: %d\n", device_prop.warpSize);
        printf("Global Memory Size: %zu bytes\n", device_prop.totalGlobalMem);
        printf("Constant Memory Size: %zu bytes\n", device_prop.totalConstMem);
        printf("Shared Memory Size per Block: %zu bytes\n",
device_prop.sharedMemPerBlock);
        printf("Registers per Block: %d\n", device_prop.regsPerBlock);
        printf("Max Threads per Block: %d\n", device_prop.maxThreadsPerBlock);
        printf("Max Size of Each Dimension of a Block: (%d, %d, %d)\n",
            device_prop.maxThreadsDim[0], device_prop.maxThreadsDim[1],
device_prop.maxThreadsDim[2]);
        printf("Max Size of Each Dimension of a Grid: (%d, %d, %d)\n",
            device_prop.maxGridSize[0], device_prop.maxGridSize[1],
device_prop.maxGridSize[2]);

        printf("\n");
    }

    return 0;
}
```

## Part 2 Section 1 Code

```
// Jacob Badali 20290739
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define Threads_per_block 128

#define WIDTH (1500) //Not sure what this size means, will find out later. wondering
why it isn't like [][]...

#define BLOCK_WIDTH 32



//Multiplication kernel function
__global__ void mulKernel(int* M, int* N, int* P, int size) {
    int rows = blockIdx.y * blockDim.y + threadIdx.y;
    int cols = blockIdx.x * blockDim.x + threadIdx.x;


    if (rows < size && cols < size) {
        float temp_sum = 0.0;
        for (int i = 0; i < size; i++) {
            temp_sum += M[rows * size + i] * N[i * size + cols];
        }
        P[rows * size + cols] = temp_sum;
    }
}
```

```
void mulDevice(float* A, float* B, float* C, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            float temp = 0;
            for (int k = 0; k < size; ++k) {
                temp += A[i * size + k] * B[k * size + j];
            }
            C[i * size + j] = temp;
        }
    }
}


int main()
{

    float* d_M = 0;
    float* d_N = 0;
    float* d_P = 0;

    float* h_M;
    float* h_N;
    float* h_P;



    //int WIDTH[5] = { 100, 250, 500, 1000, 2500 }; // Initialize WIDTH array here


    //for (int i = 0; i < 5; i++) {

    int size = WIDTH * WIDTH * sizeof(float);

    cudaMallocHost((void**)&h_M, size);
    cudaMallocHost((void**)&h_N, size);
    cudaMallocHost((void**)&h_P, size);

    int NumBlocks = WIDTH / BLOCK_WIDTH;
    if (WIDTH % BLOCK_WIDTH) NumBlocks++;

    dim3 dimGrid(NumBlocks, NumBlocks);
    dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

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

    float elapsedTime_DevToHost;
    float elapsedTime_HostToDev;

    //Allocate appropriate memory size for each array
    cudaMalloc((void**)&d_M, size);
    cudaMalloc((void**)&d_N, size);
    cudaMalloc((void**)&d_P, size);


    //fill host matrices
    for (int k = 0; k < WIDTH; k++) {
```

```
        for (int j = 0; j < WIDTH; j++) {
            h_M[k * WIDTH + j] = ((float)rand() / RAND_MAX) * 100.0; // fill with
rand values from 0-100
            h_N[k * WIDTH + j] = ((float)rand() / RAND_MAX) * 100.0;
            h_P[k * WIDTH + j] = 0;
        }
    }
    /*
        //Host matrix multiplication
        for (int i = 0; i < num_row; i++) {
            for (int j = 0; j < num_col; j++) {
                h_P[i][j] = 0;
                for (int k = 0; k < num_row; k++) {
                    h_P[i][j] += h_M[i][k] * h_N[k][j];
                }
            }
        }
    */


    //Cpy to dev, timer
    cudaEventRecord(start, 0);
    cudaMemcpy(d_M, h_M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_N, h_N, size, cudaMemcpyHostToDevice);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime_HostToDev, start, stop);
    printf("Transfer Host to Device, size[%d]: %f ms |", WIDTH,
elapsedTime_HostToDev);

    cudaEventRecord(start, 0);
    cudaMemcpy(h_M, d_M, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_N, d_N, size, cudaMemcpyDeviceToHost);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime_DevToHost, start, stop);
    printf("| Transfer Device to Host, size[%d]: %f ms\n", WIDTH,
elapsedTime_DevToHost);

    //Device Matrix multiplication
    //mulKernel << <dimBlock, dimGrid, 0, 0 >> > (d_M, d_N, d_P, size);

    //Cpy to dev, timer

    cudaFree(d_M);
    cudaFree(d_N);
    cudaFree(d_P);
}
//}
```

## Part 2 Section 2 Code

```
// Jacob Badali 20290739
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
```

```
#include <math.h>
#include <stdlib.h>


#define WIDTH (1500) //CHANGE THIS!!!

#define BLOCK_WIDTH 1



//Multiplication kernel function
__global__ void mulKernel(float* M, float* N, float* P, int size) {
    int rows = blockIdx.y * blockDim.y + threadIdx.y;
    int cols = blockIdx.x * blockDim.x + threadIdx.x;


    if (rows < size && cols < size) {
        float temp_sum = 0.0;
        for (int i = 0; i < size; i++) {
            temp_sum += M[rows * size + i] * N[i * size + cols];
        }
        P[rows * size + cols] = temp_sum;
    }
}


int main()
{

    float* d_M = 0;
    float* d_N = 0;
    float* d_P = 0;

    float* h_M;
    float* h_N;
    float* h_P;
    float* h_Pcheck;


    int size = WIDTH * WIDTH * sizeof(float);

    cudaMallocHost((void**)&h_M, size);
    cudaMallocHost((void**)&h_N, size);
    cudaMallocHost((void**)&h_P, size);
    cudaMallocHost((void**)&h_Pcheck, size);

    int NumBlocks = WIDTH / BLOCK_WIDTH;
    if (WIDTH % BLOCK_WIDTH) NumBlocks++;

    dim3 dimGrid(NumBlocks, NumBlocks);
    dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

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

    float elapsedTime_DevToHost;
    float elapsedTime_HostToDev;
```

```
    float elapsedTime_MatrixMulHost;
    float elapsedTime_MatrixMulDev;

    //Allocate appropriate memory size for each array
    cudaMalloc((void**)&d_M, size);
    cudaMalloc((void**)&d_N, size);
    cudaMalloc((void**)&d_P, size);

    //fill host matrices
    for (int k = 0; k < WIDTH; k++) {
        for (int j = 0; j < WIDTH; j++) {
            h_M[k * WIDTH + j] = ((float)rand() / RAND_MAX) * 100.0f; // fill with
rand values from 0-100
            h_N[k * WIDTH + j] = ((float)rand() / RAND_MAX) * 100.0f;
            h_P[k * WIDTH + j] = 0.0;
            h_Pcheck[k * WIDTH + j] = 0.0;
        }
    }



    //Cpy to dev, timer
    cudaEventRecord(start, 0);
    cudaMemcpy(d_M, h_M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_N, h_N, size, cudaMemcpyHostToDevice);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime_HostToDev, start, stop);
    printf("Transfer Host to Device, size[%d]: %f ms |", WIDTH,
elapsedTime_HostToDev);

    cudaEventRecord(start, 0);
    cudaMemcpy(h_M, d_M, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_N, d_N, size, cudaMemcpyDeviceToHost);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime_DevToHost, start, stop);
    printf("| Transfer Device to Host, size[%d]: %f ms\n", WIDTH,
elapsedTime_DevToHost);

    printf("\n");

    //Host Matrix Multiplication
    cudaEventRecord(start, 0);


    for (int i = 0; i < WIDTH; i++) {
        for (int j = 0; j < WIDTH; j++) {
            for (int k = 0; k < WIDTH; k++) {
                h_Pcheck[i * WIDTH + j] += h_M[i * WIDTH + k] * h_N[k * WIDTH + j];
            }
        }
    }

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
```

```
    cudaEventElapsedTime(&elapsedTime_MatrixMulHost, start, stop);
    printf("Host Matrix Mul Time, size[%d]: %f ms\n", WIDTH,
elapsedTime_MatrixMulHost);

    //Device Matrix Multiplication
    cudaEventRecord(start, 0);
    cudaMemcpy(d_M, h_M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_N, h_N, size, cudaMemcpyHostToDevice);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventRecord(start, 0);

    mulKernel << <dimGrid, dimBlock, 0, 0>> > (d_M, d_N, d_P, WIDTH);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime_MatrixMulDev, start, stop);
    printf("Device Matrix Mul Time, size[%d]: %f ms\n", WIDTH,
elapsedTime_MatrixMulDev);

    cudaMemcpy(h_M, d_M, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_N, d_N, size, cudaMemcpyDeviceToHost);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaMemcpy(h_P, d_P, size, cudaMemcpyDeviceToHost);

    int check = 0;
    for (int i = 0; i < WIDTH; i++) {
        for (int j = 0; j < WIDTH; j++) {
            if (abs(h_P[i * WIDTH + j] - h_Pcheck[i * WIDTH + j]) > 1) {
                check = 1;
            }
        }
    }

    if (check == 0) {
        printf("TEST PASSED\n");
    }
    else {
        printf("TEST FAILED\n");
    }

    cudaFree(d_M);
    cudaFree(d_N);
    cudaFree(d_P);
    cudaFree(h_M);
    cudaFree(h_N);
    cudaFree(h_P);
    cudaFree(h_Pcheck);
}
```

## Part 3 Code

```
// Jacob Badali 20290739
#include "cuda_runtime.h"
```

```
#include "device_launch_parameters.h"

#include <stdio.h>
#include <math.h>
#include <stdlib.h>


#define WIDTH (1500) //CHANGE THIS!!!

#define BLOCK_WIDTH 2 //CHANGE THIS!!!



//Multiplication kernel function
__global__ void mulKernel(float* M, float* N, float* P, int size) {
    int rows = blockIdx.y * blockDim.y + threadIdx.y;
    int cols = blockIdx.x * blockDim.x + threadIdx.x;


    if (rows < size && cols < size) {
        float temp_sum = 0.0;
        for (int i = 0; i < size; i++) {
            temp_sum += M[rows * size + i] * N[i * size + cols];
        }
        P[rows * size + cols] = temp_sum;
    }
}


int main()
{

    float* d_M = 0;
    float* d_N = 0;
    float* d_P = 0;

    float* h_M;
    float* h_N;
    float* h_P;
    float* h_Pcheck;


    int size = WIDTH * WIDTH * sizeof(float);

    cudaMallocHost((void**)&h_M, size);
    cudaMallocHost((void**)&h_N, size);
    cudaMallocHost((void**)&h_P, size);
    cudaMallocHost((void**)&h_Pcheck, size);

    int NumBlocks = WIDTH / BLOCK_WIDTH;
    if (WIDTH % BLOCK_WIDTH) NumBlocks++;

    dim3 dimGrid(NumBlocks, NumBlocks);
    dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

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

```
    float elapsedTime_DevToHost;
    float elapsedTime_HostToDev;
    float elapsedTime_MatrixMulHost;
    float elapsedTime_MatrixMulDev;

    //Allocate appropriate memory size for each array
    cudaMalloc((void**)&d_M, size);
    cudaMalloc((void**)&d_N, size);
    cudaMalloc((void**)&d_P, size);

    //fill host matrices
    for (int k = 0; k < WIDTH; k++) {
        for (int j = 0; j < WIDTH; j++) {
            h_M[k * WIDTH + j] = ((float)rand() / RAND_MAX) * 100.0f; // fill with
rand values from 0-100
            h_N[k * WIDTH + j] = ((float)rand() / RAND_MAX) * 100.0f;
            h_P[k * WIDTH + j] = 0.0;
            h_Pcheck[k * WIDTH + j] = 0.0;
        }
    }



    //Cpy to dev, timer
    cudaEventRecord(start, 0);
    cudaMemcpy(d_M, h_M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_N, h_N, size, cudaMemcpyHostToDevice);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime_HostToDev, start, stop);
    printf("Transfer Host to Device, size[%d]: %f ms |", WIDTH,
elapsedTime_HostToDev);

    cudaEventRecord(start, 0);
    cudaMemcpy(h_M, d_M, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_N, d_N, size, cudaMemcpyDeviceToHost);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime_DevToHost, start, stop);
    printf("| Transfer Device to Host, size[%d]: %f ms\n", WIDTH,
elapsedTime_DevToHost);

    printf("\n");

    //Host Matrix Multiplication
    cudaEventRecord(start, 0);


    //for (int i = 0; i < WIDTH; i++) {
    //    for (int j = 0; j < WIDTH; j++) {
    //        for (int k = 0; k < WIDTH; k++) {
    //            h_Pcheck[i * WIDTH + j] += h_M[i * WIDTH + k] * h_N[k * WIDTH +
j];
    //        }
    //    }
```

```
    //}

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime_MatrixMulHost, start, stop);
    printf("Host Matrix Mul Time, size[%d]: %f ms\n", WIDTH,
elapsedTime_MatrixMulHost);

    //Device Matrix Multiplication
    cudaEventRecord(start, 0);
    cudaMemcpy(d_M, h_M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_N, h_N, size, cudaMemcpyHostToDevice);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventRecord(start, 0);

    mulKernel << <dimGrid, dimBlock, 0, 0>> > (d_M, d_N, d_P, WIDTH);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime_MatrixMulDev, start, stop);
    printf("Device Matrix Mul Time, size[%d]: %f ms\n", WIDTH,
elapsedTime_MatrixMulDev);

    cudaMemcpy(h_M, d_M, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_N, d_N, size, cudaMemcpyDeviceToHost);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaMemcpy(h_P, d_P, size, cudaMemcpyDeviceToHost);

    int check = 0;
    for (int i = 0; i < WIDTH; i++) {
        for (int j = 0; j < WIDTH; j++) {
            if (abs(h_P[i * WIDTH + j] - h_Pcheck[i * WIDTH + j]) > 1) {
                check = 1;
            }
        }
    }

    if (check == 0) {
        printf("TEST PASSED\n");
    }
    else {
        printf("TEST FAILED\n");
    }

    cudaFree(d_M);
    cudaFree(d_N);
    cudaFree(d_P);
    cudaFree(h_M);
    cudaFree(h_N);
    cudaFree(h_P);
    cudaFree(h_Pcheck);
}
```

## Part 3 Tables

| Time, 100x100 | | | | | |
|---|---|---|---|---|---|
| | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| CPU | 1.46848 | 1.508928 | 1.510336 | 1.506688 | 1.498608 |
| GPU Threads | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| 2 | 0.673952 | 0.810688 | 0.809248 | 0.672992 | 0.74172 |
| 5 | 0.269376 | 0.143168 | 0.284352 | 0.268352 | 0.241312 |
| 10 | 0.261056 | 0.255904 | 0.264896 | 0.133216 | 0.228768 |
| 25 | 0.259808 | 0.14176 | 0.136384 | 0.143232 | 0.170296 |
| 32 | 0.266816 | 0.138176 | 0.262944 | 0.135968 | 0.200976 |

| Time, 250x250 | | | | | |
|---|---|---|---|---|---|
| | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| CPU | 23.85578 | 23.37235 | 23.77354 | 23.7249 | 23.68164 |
| GPU Threads | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| 2 | 11.0863 | 10.75923 | 9.906176 | 10.74902 | 10.62518 |
| 5 | 2.025504 | 2.923424 | 2.559392 | 2.040992 | 2.387328 |
| 10 | 0.960256 | 0.964096 | 1.50528 | 0.961824 | 1.097864 |
| 25 | 0.733792 | 0.739936 | 0.739424 | 0.754752 | 0.741976 |
| 32 | 0.987136 | 0.985664 | 1.348832 | 0.990656 | 1.078072 |

| Time, 500x500 | | | | | |
|---|---|---|---|---|---|
| | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| CPU | 190.2036 | 188.0452 | 191.2605 | 189.5066 | 189.754 |
| GPU Threads | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| 2 | 81.72301 | 80.31344 | 80.74394 | 81.16781 | 80.98705 |
| 5 | 14.10966 | 14.23398 | 14.00234 | 13.67123 | 14.0043 |
| 10 | 4.955936 | 4.997568 | 4.990336 | 5.004512 | 4.987088 |
| 25 | 4.699136 | 5.460512 | 4.631296 | 5.145408 | 4.984088 |
| 32 | 5.593728 | 5.576704 | 6.132736 | 5.628352 | 5.73288 |

| Time, 1000x1000 | | | | | |
|---|---|---|---|---|---|
| | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| CPU | 1605.687 | 1562.348 | 1572.901 | 1546.797 | 1571.933 |
| GPU Threads | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| 2 | 586.8982 | 530.8599 | 566.4705 | 539.0612 | 555.8224 |
| 5 | 127.2612 | 128.7012 | 125.0887 | 124.5066 | 126.3894 |
| 10 | 43.92358 | 45.28125 | 43.44951 | 44.33088 | 44.2463 |
| 25 | 41.75322 | 40.19728 | 41.18055 | 41.1024 | 41.05836 |
| 32 | 52.33053 | 51.75632 | 52.17632 | 52.30147 | 52.14116 |

| Time, 1500x1500 | | | | | |
|---|---|---|---|---|---|
| | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| CPU | 5783.762 | 5508.285 | 5579.558 | 5561.682 | 5608.322 |
| GPU Threads | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Average |
| 2 | 1900.626 | 1861.793 | 1845.312 | 1845.587 | 1863.33 |
| 5 | 340.6214 | 358.3709 | 340.5679 | 346.5012 | 346.5153 |
| 10 | 144.9095 | 130.8658 | 132.0595 | 137.0765 | 136.2278 |
| 25 | 335.7727 | 137.9879 | 145.0468 | 137.8224 | 189.1575 |
| 32 | 164.8822 | 164.9286 | 155.4491 | 311.1648 | 199.1062 |