

Machine Problem 1

Matteo Van der Plaat

20287556

March 13, 2024

ELEC374

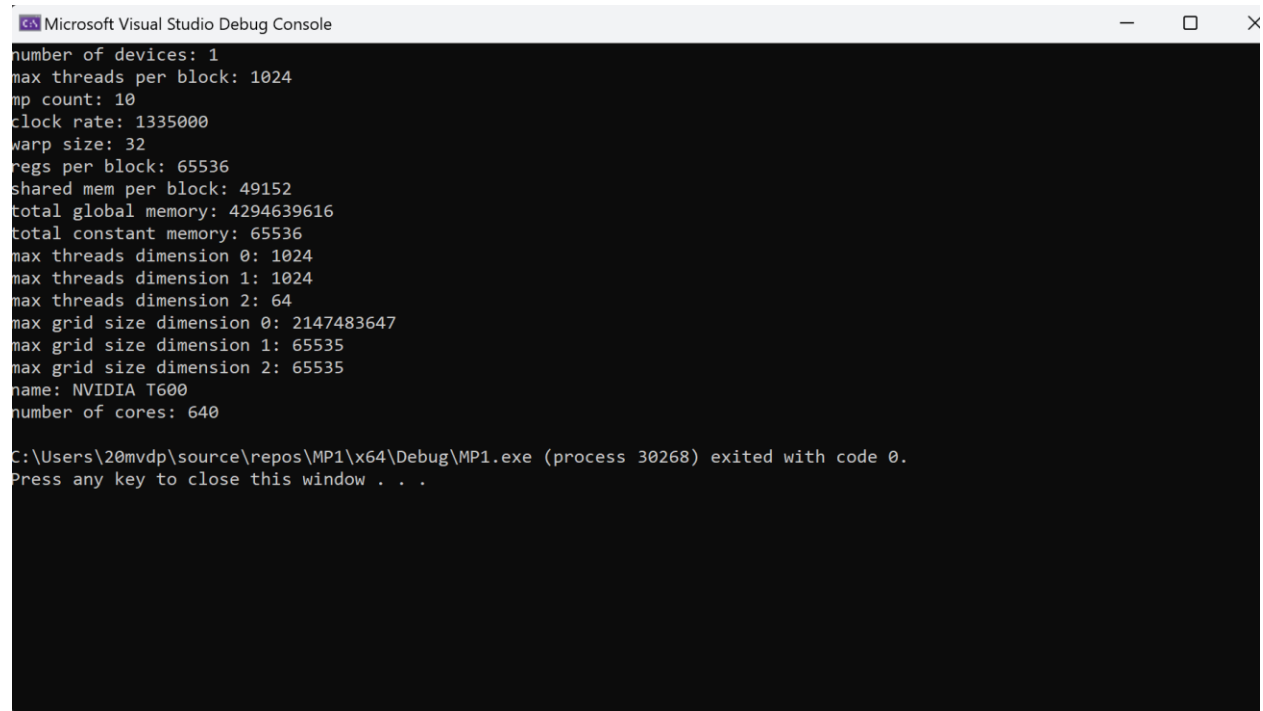
I do hereby verify that this machine problem submission is my own work and contains my own original ideas, concepts, and designs. No portion of this report or code has been copied in whole or in part from another source, with the possible exception of properly referenced material

Analysis

Please note that full code for all parts can be found in the Appendix. The numbers displayed in the graphs below are also available in the zip file submission.

Part 1

Part 1 was completed to get the number of devices and properties for each device in the system. The output of this code can be found below outlining information such as number of devices, name of devices, clock rate, etc.



```
Microsoft Visual Studio Debug Console
number of devices: 1
max threads per block: 1024
mp count: 10
clock rate: 1335000
warp size: 32
regs per block: 65536
shared mem per block: 49152
total global memory: 4294639616
total constant memory: 65536
max threads dimension 0: 1024
max threads dimension 1: 1024
max threads dimension 2: 64
max grid size dimension 0: 2147483647
max grid size dimension 1: 65535
max grid size dimension 2: 65535
name: NVIDIA T600
number of cores: 640

C:\Users\20mvdv\source\repos\MP1\x64\Debug\MP1.exe (process 30268) exited with code 0.
Press any key to close this window . . .
```

Part 2.1

Part 2.1 was completed to analyze the difference in timing between transferring data to the host from a device and transferring data to a device from the host. Data from the analysis can be seen below. Note that each trial was done 5 times to ensure a useable average was attained and outliers were excluded.

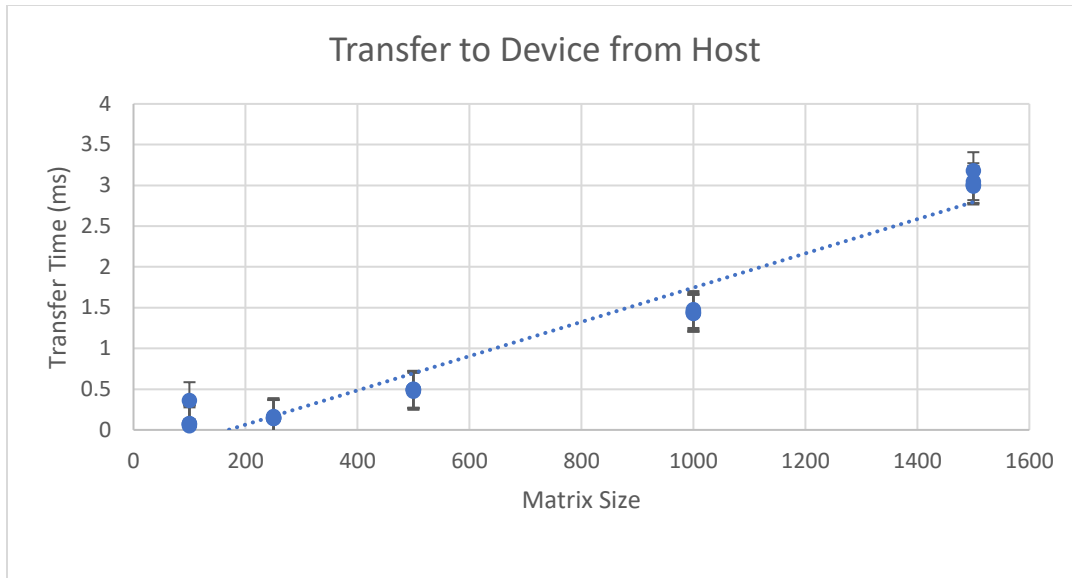


Figure 1: Graph of transfer time when transferring data from host to device.

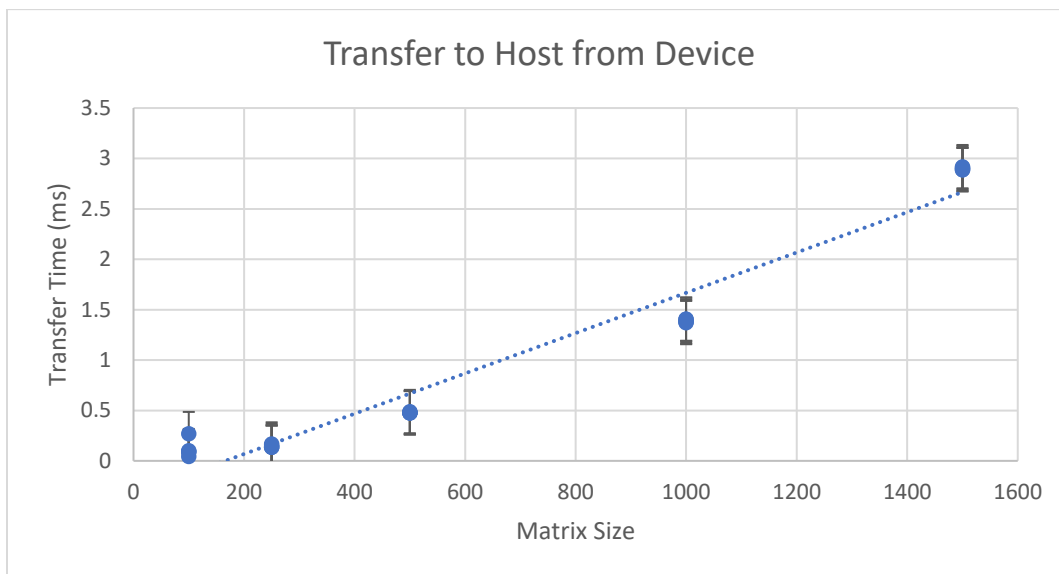


Figure 2: Graph of transfer time when transferring data from host to device.

Looking at these graphs, it can be seen that the transferring time when the device is sending information to the host takes less time than when the host is sending information to the device. This is due to the device being able to use parallel processing while copying memory to the host.

```

Time to send matrices to device from host: 0.186240
Time to send matrices to host from device: 0.159168
Time to send matrices to device from host: 0.158208
Time to send matrices to host from device: 0.138400
Time to send matrices to device from host: 0.150432
Time to send matrices to host from device: 0.142528
Time to send matrices to device from host: 0.152416
Time to send matrices to host from device: 0.139168
Time to send matrices to device from host: 0.151936
Time to send matrices to host from device: 0.147584

```

Figure 3: A snippet of the output of the code written for Part 2.1.

Part 2.2

Part 2.2 was completed by implementing matrix multiplication using both the host and device and comparing the differences in their running times. This was repeated twice, once considering memory transfer times and once without. Each trial was repeated 5 times to ensure correctness. Graphs illustrating these differences can be found below.

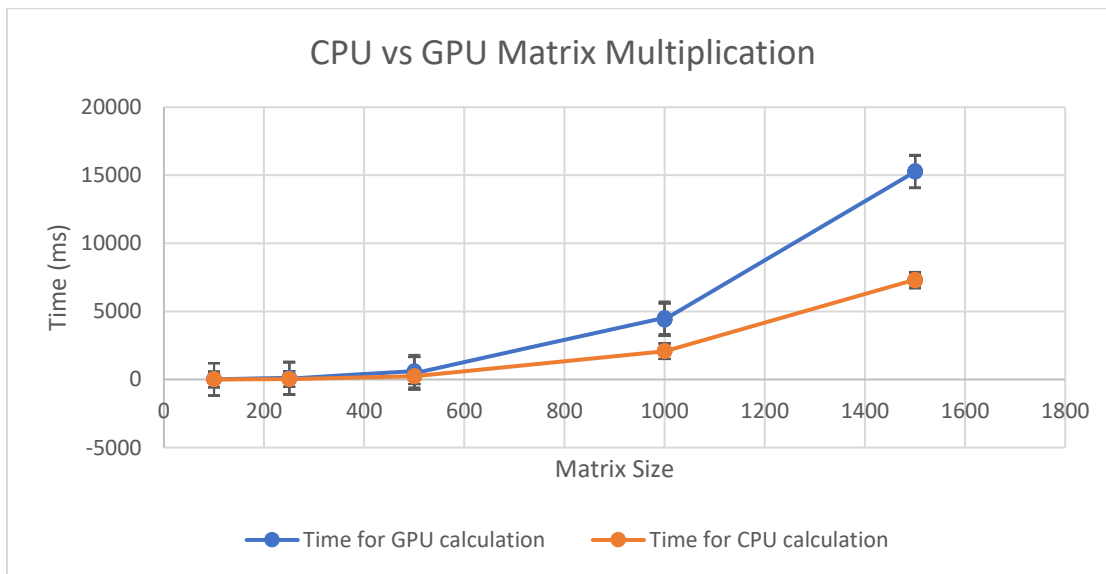


Figure 4: Graph comparing time required for matrix multiplication on both CPU and GPU

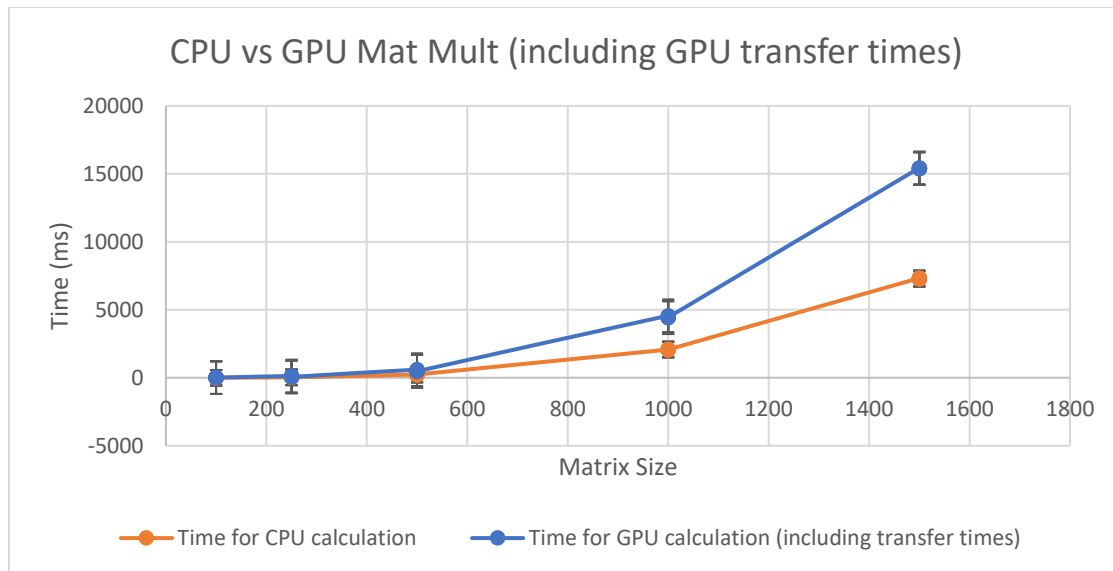


Figure 5: Graph comparing time required for matrix multiplication on both CPU and GPU including memory transfer times

Analyzing the graphs above, it is not always beneficial to offload matrix multiplication to the device. This is due to the low block width removing the advantage the device has when those numbers are higher. As seen above, the host is better at computing matrix multiplication sequentially – analysis taking parallelism into account will be conducted later on.

```
Time for GPU matrix multiplication: 152.069382
TEST PASSED
Time for GPU matrix multiplication: 74.441696
TEST PASSED
Time for GPU matrix multiplication: 71.171585
TEST PASSED
Time for GPU matrix multiplication: 61.510399
TEST PASSED
Time for GPU matrix multiplication: 61.898689
TEST PASSED
Time for CPU matrix multiplication: 30.463488
Time for CPU matrix multiplication: 30.127359
Time for CPU matrix multiplication: 30.658112
Time for CPU matrix multiplication: 30.856256
Time for CPU matrix multiplication: 30.293407
```

Figure 6: A snippet of the output of the code written for Part 2.2.

Part 2.3

Part 2.3 was completed with the goal of determining how block widths affect the system performance when computing matrix multiplication. Analysis can be found below. Note that each trial was completed 5 times for each block width and matrix size for correctness and to minimize the effect of outliers.

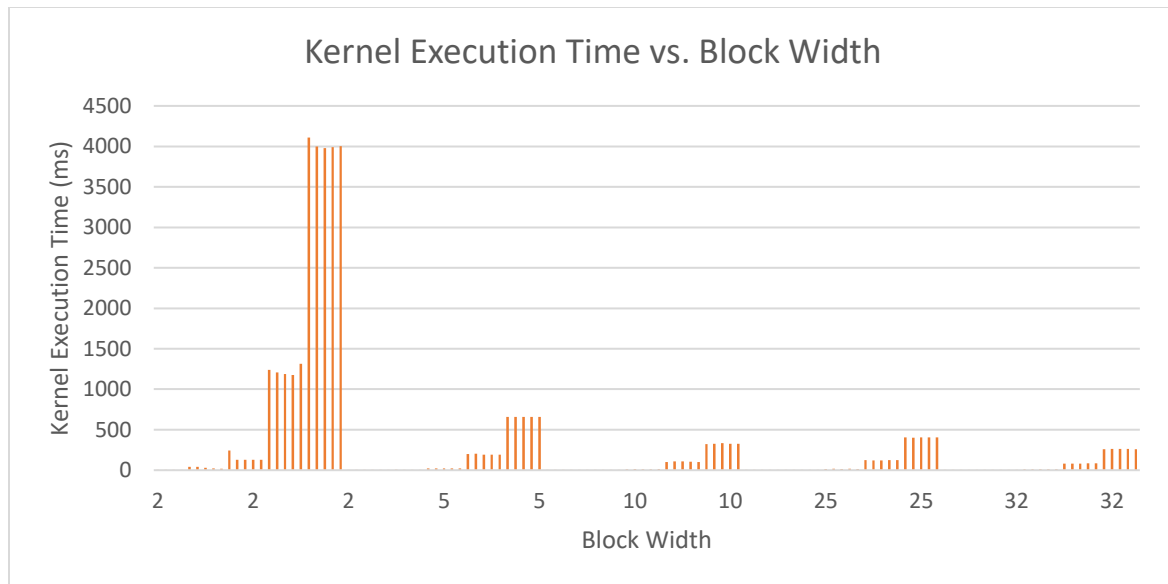


Figure 7: Kernel execution time vs. block width (not considering transfer times)

In this graph it can be seen that increasing the block width beyond one or two drastically decreases the kernel execution time. An optimal point for decreasing kernel execution time is reached with a block width of 32. This graph also displays that it is beneficial to offload matrix multiplication to the device when using a block width of two or more.

Further analysis questions:

- a.) Each element of each input matrix is loaded once during the execution of the kernel. This is to ensure peak memory efficiency in the system.
- b.) The CGMA ratio in each thread is 0.25 FLOP/Byte. This is due to there being one global memory access for each of the elements being multiplied (one from M, one from N), and each fetching four bytes equaling 8 bytes total. Therefore, the CGMA ratio is 2 FLOP to 8 bytes or 0.25 FLOP/Byte.

```
Time for GPU matrix multiplication: 31.693407
TEST PASSED
Time for GPU matrix multiplication: 31.645376
TEST PASSED
Time for GPU matrix multiplication: 27.194847
TEST PASSED
Time for GPU matrix multiplication: 30.194208
TEST PASSED
Time for GPU matrix multiplication: 18.555904
TEST PASSED
Time for GPU matrix multiplication: 4.548064
TEST PASSED
Time for GPU matrix multiplication: 3.744288
TEST PASSED
Time for GPU matrix multiplication: 3.435616
TEST PASSED
Time for GPU matrix multiplication: 3.394048
TEST PASSED
Time for GPU matrix multiplication: 3.342432
TEST PASSED
Time for GPU matrix multiplication: 1.880032
TEST PASSED
Time for GPU matrix multiplication: 1.937312
TEST PASSED
Time for GPU matrix multiplication: 1.975072
TEST PASSED
Time for GPU matrix multiplication: 1.885760
TEST PASSED
Time for GPU matrix multiplication: 1.828640
TEST PASSED
Time for GPU matrix multiplication: 2.412288
TEST PASSED
Time for GPU matrix multiplication: 2.124672
TEST PASSED
Time for GPU matrix multiplication: 2.570336
TEST PASSED
Time for GPU matrix multiplication: 2.394656
TEST PASSED
Time for GPU matrix multiplication: 2.112544
TEST PASSED
Time for GPU matrix multiplication: 1.635424
TEST PASSED
Time for GPU matrix multiplication: 3.148768
TEST PASSED
Time for GPU matrix multiplication: 1.803136
TEST PASSED
Time for GPU matrix multiplication: 2.090752
TEST PASSED
Time for GPU matrix multiplication: 1.761408
TEST PASSED
```

Figure 8: A snippet of the output of the code written for Part 2.3.

Appendix

Code for Part 1

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

//use attributes major and minor to determine number of cores per device
int getNumCores(cudaDeviceProp devprop)
{
```

```

int mp = devprop.multiProcessorCount;

switch (devprop.major)
{
case 2:
    return (devprop.minor == 1) ? (mp * 48) : (mp * 32);
case 3:
    return mp * 192;
case 5:
    return mp * 128;
case 6:
    if (devprop.minor == 1 || devprop.minor == 2) return mp * 128;
    else if (devprop.minor == 0) return mp * 64;
    else return -1;
case 7:
    if (devprop.minor == 0 || devprop.minor == 5) return mp * 64;
    else return -1;
default:
    return -1;
}
}

int main()
{
    int dev_count;
    cudaGetDeviceCount(&dev_count); //determine number of devices
    printf("number of devices: %d\n", dev_count);

    cudaDeviceProp dev_prop;
    for (int i = 0; i < dev_count; i++) {
        cudaGetDeviceProperties(&dev_prop, i);
        printf("max threads per block: %d\n", dev_prop.maxThreadsPerBlock);
        printf("mp count: %d\n", dev_prop.multiProcessorCount);
        printf("clock rate: %d\n", dev_prop.clockRate);
        printf("warp size: %d\n", dev_prop.warpSize);
        printf("regs per block: %d\n", dev_prop.regsPerBlock);
        printf("shared mem per block: %lu\n", dev_prop.sharedMemPerBlock);
        printf("total global memory: %lu\n", dev_prop.totalGlobalMem);
        printf("total constant memory: %lu\n", dev_prop.totalConstMem);
        printf("max threads dimension 0: %d\n", dev_prop.maxThreadsDim[0]);
        printf("max threads dimension 1: %d\n", dev_prop.maxThreadsDim[1]);
        printf("max threads dimension 2: %d\n", dev_prop.maxThreadsDim[2]);
        printf("max grid size dimension 0: %d\n", dev_prop.maxGridSize[0]);
        printf("max grid size dimension 1: %d\n", dev_prop.maxGridSize[1]);
        printf("max grid size dimension 2: %d\n", dev_prop.maxGridSize[2]);
        printf("name: %s\n", dev_prop.name);
        printf("number of cores: %d\n", getNumCores(dev_prop));
    }
}

```

Code for Part 2.1

```

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdlib.h>

```



```

#include <stdio.h>
#include <time.h>

#define MATRIX_WIDTH 100 //dimensions of matrices
#define MATRIX_SIZE (MATRIX_WIDTH * MATRIX_WIDTH) //total number of elements in
matrices
#define NBYTES (MATRIX_SIZE * sizeof(float))

int BLOCK_WIDTH = 1;
//int BLOCK_WIDTH[] = { 2, 5, 10, 25, 32 };

float M[MATRIX_SIZE];
float N[MATRIX_SIZE];
float P[MATRIX_SIZE];

//functions to be tested
void cudaTransferTest();
void cudaMatMult(float* M, float* N, float* P, int WIDTH);

//matrix multiplication kernel, called by cudaMatMult function
__global__ void matMultKernel(float* M, float* N, float* P, int WIDTH)
{
    // calculate row, col index
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < WIDTH && col < WIDTH)
    {
        float Pvalue = 0;
        //each thread computes one element of the block sub-matrix
        for (int k = 0; k < WIDTH; k++) {
            Pvalue += M[row * WIDTH + k] * N[k * WIDTH + col];
        }
        P[row * WIDTH + col] = Pvalue;
    }
}

//function used to check validity of value outputted by GPU Mat Mult function
void checkGPUanswer(float* M, float* N, float* GPU_P, int WIDTH)
{
    bool passed;
    float check;

    //calculate correct values in CPU and compare against GPU value
    for (int i = 0; i < WIDTH; i++)
    {
        for (int j = 0; j < WIDTH; j++)
        {
            check = 0;

            for (int k = 0; k < WIDTH; k++)
            {
                check += M[i * WIDTH + k] * N[k * WIDTH + j];
                if (GPU_P[i * WIDTH + j] != check) passed = 0;
            }
        }
    }
    passed = 1; //if all values match up, test passed
}

```

```

        if (passed) printf("TEST PASSED\n");
        else        printf("TEST FAILED\n");
    }

//standard matrix multiplication, computed using CPU
void CPUmatMult(float* M, float* N, float* P, int WIDTH)
{
    //initialization values used for timing
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    for (int l = 0; l < 5; l++)
    {
        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        for (int i = 0; i < WIDTH; i++)
        {
            for (int j = 0; j < WIDTH; j++)
            {
                for (int k = 0; k < WIDTH; k++)
                {
                    P[i * WIDTH + j] += M[i * WIDTH + k] * N[k * WIDTH
+ j];
                }
            }

            cudaEventRecord(stop, 0); // end timer
            cudaEventSynchronize(stop);
            cudaEventElapsedTime(&gpu_time, start, stop);
            printf("Time for CPU matrix multiplication: %f\n", gpu_time); //display
results
        }

    }

int main()
{
    srand(time(NULL)); //seed random function

    //allocate memory in host
    cudaMallocHost((void**)&M, NBYTES);
    cudaMallocHost((void**)&N, NBYTES);
    cudaMallocHost((void**)&P, NBYTES);

    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        // fill matrices M and N with random values for testing
        M[i] = rand() % 100 / (float)10.0;
        N[i] = rand() % 100 / (float)10.0;
        P[i] = 0.0;
    }
}

```

```

    //function used for testing transferring data between host and device
    cudaTransferTest();

    //functions used for testing matrix multiplication using GPUs and comparing
    against CPUs
    //cudaMatMult(M, N, P, MATRIX_WIDTH); // GPU/Cuda matrix multiplication
    //CPUMatMult(M, N, P, MATRIX_WIDTH); // CPU matrix multiplication

    //free host memory
    cudaFreeHost(M);
    cudaFreeHost(N);
    cudaFreeHost(P);

    return 0;
}

void cudaTransferTest()
{
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    float* dM, * dN, * dP;

    //allocate memory for matrices on device
    cudaMalloc((void**)&dM, NBYTES);
    cudaMalloc((void**)&dN, NBYTES);
    cudaMalloc((void**)&dP, NBYTES);

    //check memory allocation was successful
    err = cudaGetLastError();
    if (err != cudaSuccess)
        printf("Error allocating memory in device");

    //repeat 5 times to ensure correctness
    for (int i = 0; i < 5; i++)
    {
        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        //copy information from host to device
        cudaMemcpy(dM, M, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dN, N, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time to send matrices to device from host: %f\n", gpu_time);
    }

    //display results

    cudaEventRecord(start, 0); // start timer
    cudaDeviceSynchronize();

    //copy information from device to host

```

```

        cudaMemcpy(M, dM, MATRIX_SIZE * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(N, dN, MATRIX_SIZE * sizeof(float), cudaMemcpyDeviceToHost);

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time to send matrices to host from device: %f\n", gpu_time);
//display results

    }
    //free device memory
    cudaFree(dM);
    cudaFree(dN);
    cudaFree(dP);
}

void cudaMatMult(float* M, float* N, float* P, int WIDTH)
{
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    float* dM, * dN, * dP;

    //allocate memory for matrices on device
    cudaMalloc((void**)&dM, NBYTES);
    cudaMalloc((void**)&dN, NBYTES);
    cudaMalloc((void**)&dP, NBYTES);

    //check memory allocation was successful
    err = cudaGetLastError();
    if (err != cudaSuccess)
        printf("Error allocating memory in device");

    //copy memory from host to device
    cudaMemcpy(dM, M, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dN, N, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);

    //testing for part 3, cycling through block widths of 2, 5, 10, 25, 32
    //for (int i = 0; i < 5; i++)
    //{
        int NUM_BLOCKS = WIDTH / BLOCK_WIDTH;

        //int NUM_BLOCKS = WIDTH / BLOCK_WIDTH[i];
        //if (WIDTH % BLOCK_WIDTH[i]) NUM_BLOCKS++;

        //define dimensions of grid and blocks
        dim3 dimGrid(NUM_BLOCKS, NUM_BLOCKS);
        dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

        for (int i = 0; i < 5; i++)
        {
            cudaEventRecord(start, 0); // start timer

```

```

        cudaDeviceSynchronize();

        //calculate matrix multiplication using Cuda and GPUs,, enabling
synchronization
        matMultKernel <<<dimGrid, dimBlock >>> (dM, dN, dP, WIDTH);

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time for GPU matrix multiplication: %f\n", gpu_time);
//display results
        checkGPUanswer(M, N, P, MATRIX_WIDTH); //make sure answers are
correct by comparing against CPU values

    }
    //}

    //free device memory
    cudaFree(dM);
    cudaFree(dN);
    cudaFree(dP);
}

```

Code for Part 2.2 without transfer times

```

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define MATRIX_WIDTH 100 //dimensions of matrices
#define MATRIX_SIZE (MATRIX_WIDTH * MATRIX_WIDTH) //total number of elements in
matrices
#define NBYTES (MATRIX_SIZE * sizeof(float))

int BLOCK_WIDTH = 1;
//int BLOCK_WIDTH[] = { 2, 5, 10, 25, 32 };

float M[MATRIX_SIZE];
float N[MATRIX_SIZE];
float P[MATRIX_SIZE];

//functions to be tested
void cudaTransferTest();
void cudaMatMult(float* M, float* N, float* P, int WIDTH);

//matrix multiplication kernel, called by cudaMatMult function
__global__ void matMultKernel(float* M, float* N, float* P, int WIDTH)
{
    // calculate row, col index
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < WIDTH && col < WIDTH)
    {

```

```

        float Pvalue = 0;
        //each thread computes one element of the block sub-matrix
        for (int k = 0; k < WIDTH; k++) {
            Pvalue += M[row * WIDTH + k] * N[k * WIDTH + col];
        }
        P[row * WIDTH + col] = Pvalue;
    }
}

//function used to check validity of value outputted by GPU Mat Mult function
void checkGPUanswer(float* M, float* N, float* GPU_P, int WIDTH)
{
    bool passed;
    float check;

    //calculate correct values in CPU and compare against GPU value
    for (int i = 0; i < WIDTH; i++)
    {
        for (int j = 0; j < WIDTH; j++)
        {
            check = 0;

            for (int k = 0; k < WIDTH; k++)
            {
                check += M[i * WIDTH + k] * N[k * WIDTH + j];
                if (GPU_P[i * WIDTH + j] != check) passed = 0;
            }
        }
    }
    passed = 1; //if all values match up, test passed

    if (passed) printf("TEST PASSED\n");
    else printf("TEST FAILED\n");
}

//standard matrix multiplication, computed using CPU
void CPumatMult(float* M, float* N, float* P, int WIDTH)
{
    //initialization values used for timing
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    for (int l = 0; l < 5; l++)
    {
        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        for (int i = 0; i < WIDTH; i++)
        {
            for (int j = 0; j < WIDTH; j++)
            {
                for (int k = 0; k < WIDTH; k++)
                {

```

```

        P[i * WIDTH + j] += M[i * WIDTH + k] * N[k * WIDTH
+ j];
    }
}

    cudaEventRecord(stop, 0); // end timer
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&gpu_time, start, stop);
    printf("Time for CPU matrix multiplication: %f\n", gpu_time); //display
results
}

}

int main()
{
    srand(time(NULL)); //seed random function

    //allocate memory in host
    cudaMallocHost((void**)&M, NBYTES);
    cudaMallocHost((void**)&N, NBYTES);
    cudaMallocHost((void**)&P, NBYTES);

    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        // fill matrices M and N with random values for testing
        M[i] = rand() % 100 / (float)10.0;
        N[i] = rand() % 100 / (float)10.0;
        P[i] = 0.0;
    }

    //function used for testing transferring data between host and device
    //cudaTransferTest();

    //functions used for testing matrix multiplication using GPUs and comparing
against CPUs
    cudaMatMult(M, N, P, MATRIX_WIDTH); // GPU/Cuda matrix multiplication
    CPUmatMult(M, N, P, MATRIX_WIDTH); // CPU matrix multiplication

    //free host memory
    cudaFreeHost(M);
    cudaFreeHost(N);
    cudaFreeHost(P);

    return 0;
}

void cudaTransferTest()
{
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    float* dM, * dN, * dP;

```

```

//allocate memory for matrices on device
cudaMalloc((void**)&dM, NBYTES);
cudaMalloc((void**)&dN, NBYTES);
cudaMalloc((void**)&dP, NBYTES);

//check memory allocation was successful
err = cudaGetLastError();
if (err != cudaSuccess)
    printf("Error allocating memory in device");

//repeat 5 times to ensure correctness
for (int i = 0; i < 5; i++)
{
    cudaEventRecord(start, 0); // start timer
    cudaDeviceSynchronize();

    //copy information from host to device
    cudaMemcpy(dM, M, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dN, N, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);

    cudaEventRecord(stop, 0); // end timer
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&gpu_time, start, stop);
    printf("Time to send matrices to device from host: %f\n", gpu_time);
//display results

    cudaEventRecord(start, 0); // start timer
    cudaDeviceSynchronize();

    //copy information from device to host
    cudaMemcpy(M, dM, MATRIX_SIZE * sizeof(float), cudaMemcpyDeviceToHost);
    cudaMemcpy(N, dN, MATRIX_SIZE * sizeof(float), cudaMemcpyDeviceToHost);

    cudaEventRecord(stop, 0); // end timer
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&gpu_time, start, stop);
    printf("Time to send matrices to host from device: %f\n", gpu_time);
//display results

}
//free device memory
cudaFree(dM);
cudaFree(dN);
cudaFree(dP);
}

void cudaMatMult(float* M, float* N, float* P, int WIDTH)
{
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

```



```

float* dM, * dN, * dP;

//allocate memory for matrices on device
cudaMalloc((void**)&dM, NBYTES);
cudaMalloc((void**)&dN, NBYTES);
cudaMalloc((void**)&dP, NBYTES);

//check memory allocation was successful
err = cudaGetLastError();
if (err != cudaSuccess)
    printf("Error allocating memory in device");

//copy memory from host to device
cudaMemcpy(dM, M, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(dN, N, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);

//testing for part 3, cycling through block widths of 2, 5, 10, 25, 32
//for (int i = 0; i < 5; i++)
//{
    int NUM_BLOCKS = WIDTH / BLOCK_WIDTH;

    //int NUM_BLOCKS = WIDTH / BLOCK_WIDTH[i];
    if (WIDTH % BLOCK_WIDTH[i]) NUM_BLOCKS++;

    //define dimensions of grid and blocks
    dim3 dimGrid(NUM_BLOCKS, NUM_BLOCKS);
    dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

    for (int i = 0; i < 5; i++)
    {
        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        //calculate matrix multiplication using Cuda and GPUs,, enabling
synchronization    matMultKernel <<<dimGrid, dimBlock >>> (dM, dN, dP, WIDTH);

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time for GPU matrix multiplication: %f\n", gpu_time);
//display results    checkGPUanswer(M, N, P, MATRIX_WIDTH); //make sure answers are
correct by comparing against CPU values

    }
//}

//free device memory
cudaFree(dM);
cudaFree(dN);
cudaFree(dP);
}

```

Code for Part 2.2 including transfer times

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

#define MATRIX_WIDTH 250 //dimensions of matrices
#define MATRIX_SIZE (MATRIX_WIDTH * MATRIX_WIDTH) //total number of elements in
matrices
#define NBYTES (MATRIX_SIZE * sizeof(float))

int BLOCK_WIDTH = 1;
//int BLOCK_WIDTH[] = { 2, 5, 10, 25, 32 };

float M[MATRIX_SIZE];
float N[MATRIX_SIZE];
float P[MATRIX_SIZE];

//functions to be tested
void cudaTransferTest();
void cudaMatMult(float* M, float* N, float* P, int WIDTH);

//matrix multiplication kernel, called by cudaMatMult function
__global__ void matMultKernel(float* M, float* N, float* P, int WIDTH)
{
    // calculate row, col index
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < WIDTH && col < WIDTH)
    {
        float Pvalue = 0;
        //each thread computes one element of the block sub-matrix
        for (int k = 0; k < WIDTH; k++) {
            Pvalue += M[row * WIDTH + k] * N[k * WIDTH + col];
        }
        P[row * WIDTH + col] = Pvalue;
    }
}

//function used to check validity of value outputted by GPU Mat Mult function
void checkGPUanswer(float* M, float* N, float* GPU_P, int WIDTH)
{
    bool passed;
    float check;

    //calculate correct values in CPU and compare against GPU value
    for (int i = 0; i < WIDTH; i++)
    {
        for (int j = 0; j < WIDTH; j++)
        {
            check = 0;

            for (int k = 0; k < WIDTH; k++)
            {
                check += M[i * WIDTH + k] * N[k * WIDTH + j];
            }
        }
    }
}
```

```

        if (GPU_P[i * WIDTH + j] != check) passed = 0;
    }
}
passed = 1; //if all values match up, test passed

if (passed) printf("TEST PASSED\n");
else printf("TEST FAILED\n");
}

//standard matrix multiplication, computed using CPU
void CPUmatMult(float* M, float* N, float* P, int WIDTH)
{
    //initialization values used for timing
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    for (int l = 0; l < 5; l++)
    {
        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        for (int i = 0; i < WIDTH; i++)
        {
            for (int j = 0; j < WIDTH; j++)
            {
                for (int k = 0; k < WIDTH; k++)
                {
                    P[i * WIDTH + j] += M[i * WIDTH + k] * N[k * WIDTH
+ j];
                }
            }
        }

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time for CPU matrix multiplication: %f\n", gpu_time); //display
results
    }

}

int main()
{
    srand(time(NULL)); //seed random function

    //allocate memory in host
    cudaMallocHost((void**)&M, NBYTES);
    cudaMallocHost((void**)&N, NBYTES);
    cudaMallocHost((void**)&P, NBYTES);

    for (int i = 0; i < MATRIX_SIZE; i++)

```

```

{
    // fill matrices M and N with random values for testing
    M[i] = rand() % 100 / (float)10.0;
    N[i] = rand() % 100 / (float)10.0;
    P[i] = 0.0;
}

//function used for testing transferring data between host and device
//cudaTransferTest();

//functions used for testing matrix multiplication using GPUs and comparing
against CPUs
cudaMatMult(M, N, P, MATRIX_WIDTH); // GPU/Cuda matrix multiplication
//CPUMatMult(M, N, P, MATRIX_WIDTH); // CPU matrix multiplication

//free host memory
cudaFreeHost(M);
cudaFreeHost(N);
cudaFreeHost(P);

return 0;
}

void cudaTransferTest()
{
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    float* dM, * dN, * dP;

    //allocate memory for matrices on device
    cudaMalloc((void**)&dM, NBYTES);
    cudaMalloc((void**)&dN, NBYTES);
    cudaMalloc((void**)&dP, NBYTES);

    //check memory allocation was successful
    err = cudaGetLastError();
    if (err != cudaSuccess)
        printf("Error allocating memory in device");

    //repeat 5 times to ensure correctness
    for (int i = 0; i < 5; i++)
    {
        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        //copy information from host to device
        cudaMemcpy(dM, M, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dN, N, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time to send matrices to device from host: %f\n", gpu_time);
    }
    //display results
}

```

```

        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        //copy information from device to host
        cudaMemcpy(M, dM, MATRIX_SIZE * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(N, dN, MATRIX_SIZE * sizeof(float), cudaMemcpyDeviceToHost);

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time to send matrices to host from device: %f\n", gpu_time);
//display results

    }
    //free device memory
    cudaFree(dM);
    cudaFree(dN);
    cudaFree(dP);
}

void cudaMatMult(float* M, float* N, float* P, int WIDTH)
{
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    float* dM, * dN, * dP;

    //allocate memory for matrices on device
    cudaMalloc((void**)&dM, NBYTES);
    cudaMalloc((void**)&dN, NBYTES);
    cudaMalloc((void**)&dP, NBYTES);

    //check memory allocation was successful
    err = cudaGetLastError();
    if (err != cudaSuccess)
        printf("Error allocating memory in device");

    //testing for part 3, cycling through block widths of 2, 5, 10, 25, 32
    //for (int i = 0; i < 5; i++)
    //{
        int NUM_BLOCKS = WIDTH / BLOCK_WIDTH;
        if (WIDTH % BLOCK_WIDTH) NUM_BLOCKS++;

        //define dimensions of grid and blocks
        dim3 dimGrid(NUM_BLOCKS, NUM_BLOCKS);
        dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

        for (int i = 0; i < 5; i++)
        {
            cudaEventRecord(start, 0); // start timer
            cudaDeviceSynchronize();

```

```

        //copy memory from host to device
        cudaMemcpy(dM, M, MATRIX_SIZE * sizeof(float),
cudaMemcpyHostToDevice);
        cudaMemcpy(dN, N, MATRIX_SIZE * sizeof(float),
cudaMemcpyHostToDevice);

        //calculate matrix multiplication using Cuda and GPUs,, enabling
synchronization
        matMultKernel << <dimGrid, dimBlock >> > (dM, dN, dP, WIDTH);

        cudaMemcpy(M, dM, MATRIX_SIZE * sizeof(float),
cudaMemcpyDeviceToHost);
        cudaMemcpy(N, dN, MATRIX_SIZE * sizeof(float),
cudaMemcpyDeviceToHost);

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time for GPU matrix multiplication: %f\n", gpu_time);
//display results
        //checkGPUanswer(M, N, P, MATRIX_WIDTH); //make sure answers are
correct by comparing against CPU values

    }
    //}

    //free device memory
    cudaFree(dM);
    cudaFree(dN);
    cudaFree(dP);
}

```

Code for Part 2.3

```

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define MATRIX_WIDTH 100 //dimensions of matrices
#define MATRIX_SIZE (MATRIX_WIDTH * MATRIX_WIDTH) //total number of elements in
matrices
#define NBYTES (MATRIX_SIZE * sizeof(float))

//int BLOCK_WIDTH = 1;
int BLOCK_WIDTH[] = { 2, 5, 10, 25, 32 };

float M[MATRIX_SIZE];
float N[MATRIX_SIZE];
float P[MATRIX_SIZE];

//functions to be tested
void cudaTransferTest();
void cudaMatMult(float* M, float* N, float* P, int WIDTH);

```

```

//matrix multiplication kernel, called by cudaMatMult function
__global__ void matMultKernel(float* M, float* N, float* P, int WIDTH)
{
    // calculate row, col index
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < WIDTH && col < WIDTH)
    {
        float Pvalue = 0;
        //each thread computes one element of the block sub-matrix
        for (int k = 0; k < WIDTH; k++) {
            Pvalue += M[row * WIDTH + k] * N[k * WIDTH + col];
        }
        P[row * WIDTH + col] = Pvalue;
    }
}

//function used to check validity of value outputted by GPU Mat Mult function
void checkGPUanswer(float* M, float* N, float* GPU_P, int WIDTH)
{
    bool passed;
    float check;

    //calculate correct values in CPU and compare against GPU value
    for (int i = 0; i < WIDTH; i++)
    {
        for (int j = 0; j < WIDTH; j++)
        {
            check = 0;

            for (int k = 0; k < WIDTH; k++)
            {
                check += M[i * WIDTH + k] * N[k * WIDTH + j];
                if (GPU_P[i * WIDTH + j] != check) passed = 0;
            }
        }
    }
    passed = 1; //if all values match up, test passed

    if (passed) printf("TEST PASSED\n");
    else printf("TEST FAILED\n");
}

//standard matrix multiplication, computed using CPU
void CPumatMult(float* M, float* N, float* P, int WIDTH)
{
    //initialization values used for timing
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    for (int l = 0; l < 5; l++)
    {

```

```

        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        for (int i = 0; i < WIDTH; i++)
        {
            for (int j = 0; j < WIDTH; j++)
            {
                for (int k = 0; k < WIDTH; k++)
                {
                    P[i * WIDTH + j] += M[i * WIDTH + k] * N[k * WIDTH
+ j];
                }
            }
        }

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time for CPU matrix multiplication: %f\n", gpu_time); //display
results
    }

}

int main()
{
    srand(time(NULL)); //seed random function

    //allocate memory in host
    cudaMallocHost((void**)&M, NBYTES);
    cudaMallocHost((void**)&N, NBYTES);
    cudaMallocHost((void**)&P, NBYTES);

    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        // fill matrices M and N with random values for testing
        M[i] = rand() % 100 / (float)10.0;
        N[i] = rand() % 100 / (float)10.0;
        P[i] = 0.0;
    }

    //function used for testing transferring data between host and device
    //cudaTransferTest();

    //functions used for testing matrix multiplication using GPUs and comparing
    against CPUs
    cudaMatMult(M, N, P, MATRIX_WIDTH); // GPU/Cuda matrix multiplication
    //CPUMatMult(M, N, P, MATRIX_WIDTH); // CPU matrix multiplication

    //free host memory
    cudaFreeHost(M);
    cudaFreeHost(N);
    cudaFreeHost(P);

    return 0;
}

void cudaTransferTest()

```



```

{
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    float* dM, * dN, * dP;

    //allocate memory for matrices on device
    cudaMalloc((void**)&dM, NBYTES);
    cudaMalloc((void**)&dN, NBYTES);
    cudaMalloc((void**)&dP, NBYTES);

    //check memory allocation was successful
    err = cudaGetLastError();
    if (err != cudaSuccess)
        printf("Error allocating memory in device");

    //repeat 5 times to ensure correctness
    for (int i = 0; i < 5; i++)
    {
        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        //copy information from host to device
        cudaMemcpy(dM, M, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dN, N, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time to send matrices to device from host: %f\n", gpu_time);
    //display results

        cudaEventRecord(start, 0); // start timer
        cudaDeviceSynchronize();

        //copy information from device to host
        cudaMemcpy(M, dM, MATRIX_SIZE * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(N, dN, MATRIX_SIZE * sizeof(float), cudaMemcpyDeviceToHost);

        cudaEventRecord(stop, 0); // end timer
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpu_time, start, stop);
        printf("Time to send matrices to host from device: %f\n", gpu_time);
    //display results

    }
    //free device memory
    cudaFree(dM);
    cudaFree(dN);
    cudaFree(dP);
}

```

```

void cudaMatMult(float* M, float* N, float* P, int WIDTH)
{
    cudaEvent_t start, stop;
    float gpu_time = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaError_t err;

    float* dM, * dN, * dP;

    //allocate memory for matrices on device
    cudaMalloc((void**)&dM, NBYTES);
    cudaMalloc((void**)&dN, NBYTES);
    cudaMalloc((void**)&dP, NBYTES);

    //check memory allocation was successful
    err = cudaGetLastError();
    if (err != cudaSuccess)
        printf("Error allocating memory in device");

    //copy memory from host to device
    cudaMemcpy(dM, M, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dN, N, MATRIX_SIZE * sizeof(float), cudaMemcpyHostToDevice);

    //testing for part 3, cycling through block widths of 2, 5, 10, 25, 32
    for (int i = 0; i < 5; i++)
    {
        int NUM_BLOCKS = WIDTH / BLOCK_WIDTH[i];
        if (WIDTH % BLOCK_WIDTH[i]) NUM_BLOCKS++;

        //define dimensions of grid and blocks
        dim3 dimGrid(NUM_BLOCKS, NUM_BLOCKS);
        dim3 dimBlock(BLOCK_WIDTH[i], BLOCK_WIDTH[i]);

        for (int i = 0; i < 5; i++)
        {
            cudaEventRecord(start, 0); // start timer
            cudaDeviceSynchronize();

            //calculate matrix multiplication using Cuda and GPUs,, enabling
synchronization
            matMultKernel << <dimGrid, dimBlock >> > (dM, dN, dP, WIDTH);

            cudaEventRecord(stop, 0); // end timer
            cudaEventSynchronize(stop);
            cudaEventElapsedTime(&gpu_time, start, stop);
            printf("Time for GPU matrix multiplication: %f\n", gpu_time);

        //display results
            checkGPUanswer(M, N, P, MATRIX_WIDTH); //make sure answers are
correct by comparing against CPU values

        }
    }

    //free device memory
    cudaFree(dM);
    cudaFree(dN);
}

```

```
        cudaFree(dP);  
    }
```