

ELEC 374

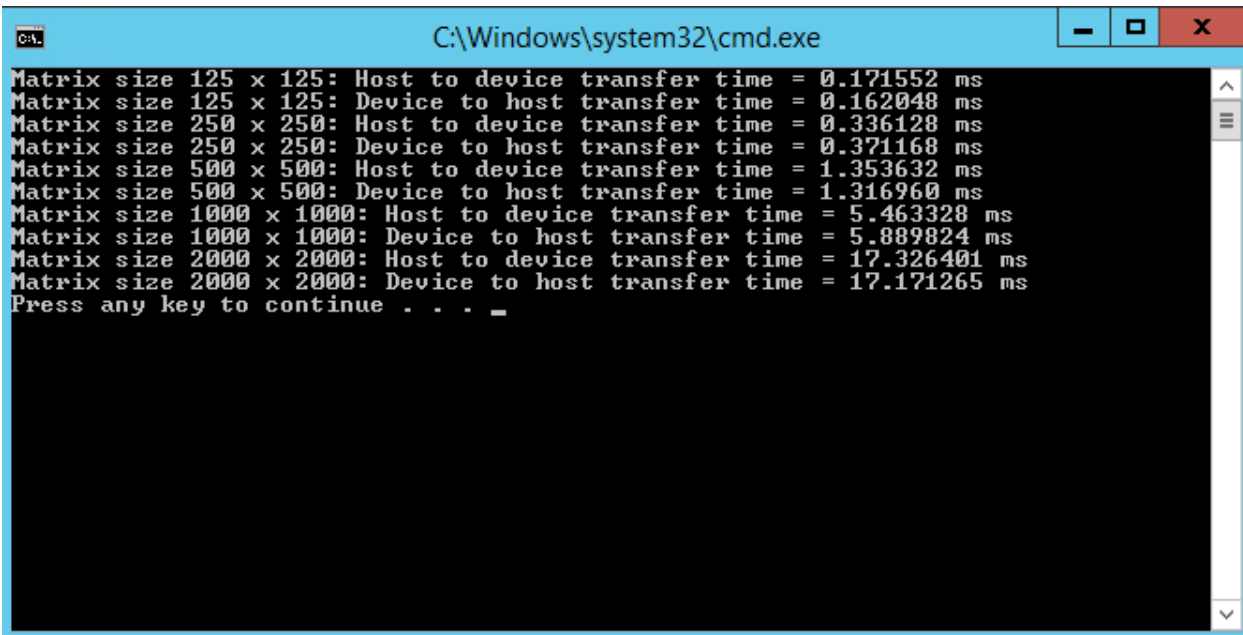
Machine Problem 3

Luka Gobovic

20215231

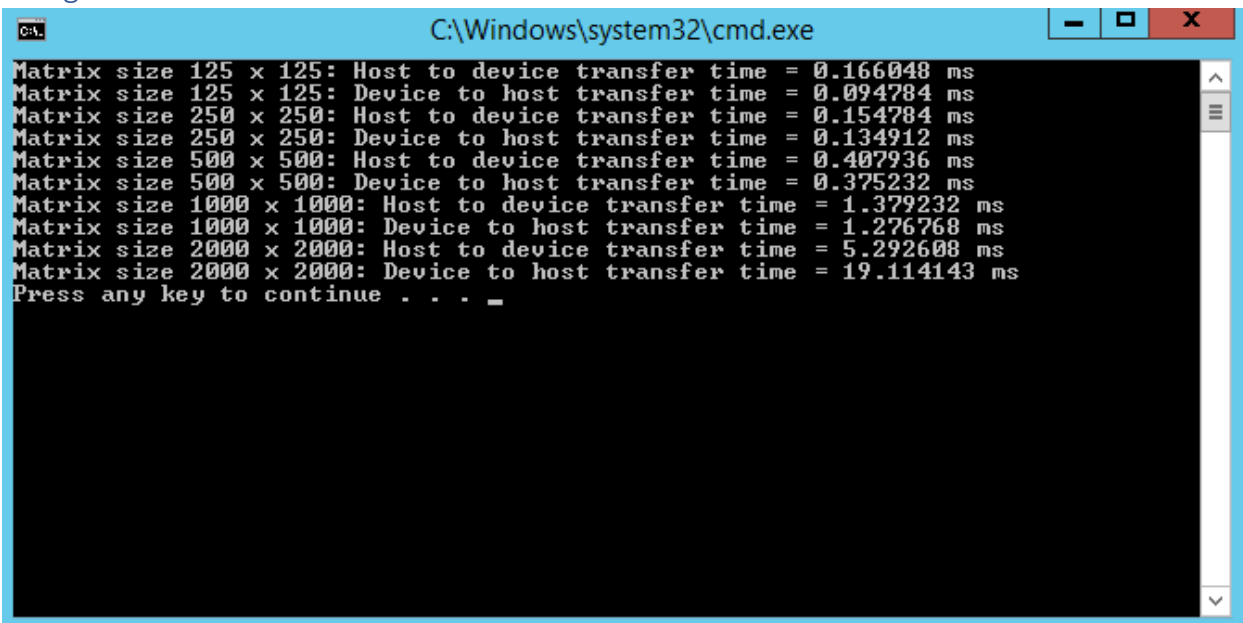
Output

Data transfer times using malloc:



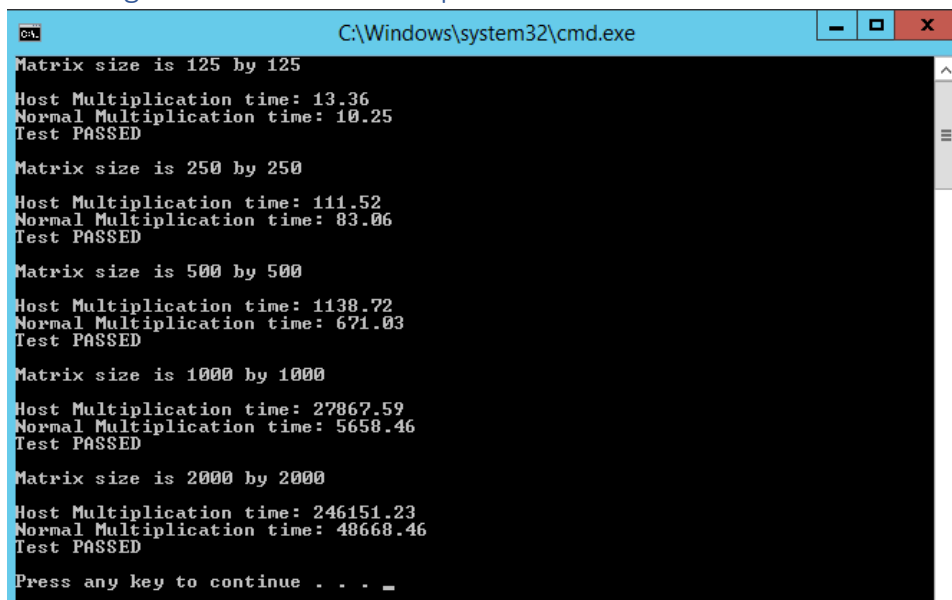
```
C:\Windows\system32\cmd.exe
Matrix size 125 x 125: Host to device transfer time = 0.171552 ms
Matrix size 125 x 125: Device to host transfer time = 0.162048 ms
Matrix size 250 x 250: Host to device transfer time = 0.336128 ms
Matrix size 250 x 250: Device to host transfer time = 0.371168 ms
Matrix size 500 x 500: Host to device transfer time = 1.353632 ms
Matrix size 500 x 500: Device to host transfer time = 1.316960 ms
Matrix size 1000 x 1000: Host to device transfer time = 5.463328 ms
Matrix size 1000 x 1000: Device to host transfer time = 5.889824 ms
Matrix size 2000 x 2000: Host to device transfer time = 17.326401 ms
Matrix size 2000 x 2000: Device to host transfer time = 17.171265 ms
Press any key to continue . . . _
```

Using cudaMallocHost:



```
C:\Windows\system32\cmd.exe
Matrix size 125 x 125: Host to device transfer time = 0.166048 ms
Matrix size 125 x 125: Device to host transfer time = 0.094784 ms
Matrix size 250 x 250: Host to device transfer time = 0.154784 ms
Matrix size 250 x 250: Device to host transfer time = 0.134912 ms
Matrix size 500 x 500: Host to device transfer time = 0.407936 ms
Matrix size 500 x 500: Device to host transfer time = 0.375232 ms
Matrix size 1000 x 1000: Host to device transfer time = 1.379232 ms
Matrix size 1000 x 1000: Device to host transfer time = 1.276768 ms
Matrix size 2000 x 2000: Host to device transfer time = 5.292608 ms
Matrix size 2000 x 2000: Device to host transfer time = 19.114143 ms
Press any key to continue . . . _
```

For a single block and 1 thread per block



```
cs. C:\Windows\system32\cmd.exe
Matrix size is 125 by 125
Host Multiplication time: 13.36
Normal Multiplication time: 10.25
Test PASSED

Matrix size is 250 by 250
Host Multiplication time: 111.52
Normal Multiplication time: 83.06
Test PASSED

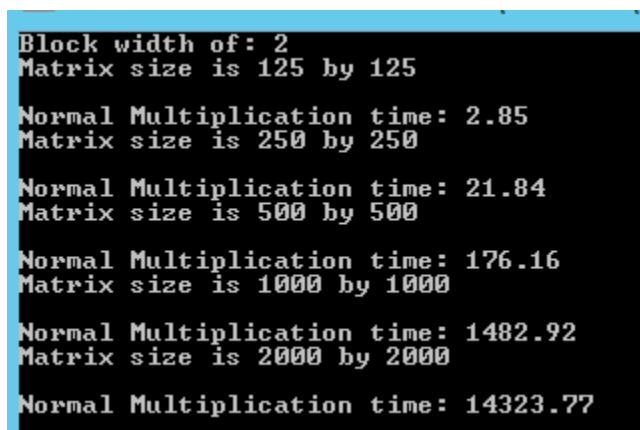
Matrix size is 500 by 500
Host Multiplication time: 1138.72
Normal Multiplication time: 671.03
Test PASSED

Matrix size is 1000 by 1000
Host Multiplication time: 27867.59
Normal Multiplication time: 5658.46
Test PASSED

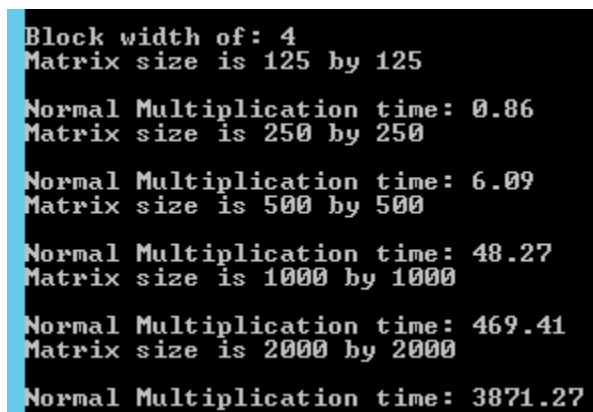
Matrix size is 2000 by 2000
Host Multiplication time: 246151.23
Normal Multiplication time: 48668.46
Test PASSED

Press any key to continue . . . _
```

No checks against the CPU were added due to the very long run time of the 2000x2000 matrix. It would take too long.



```
Block width of: 2
Matrix size is 125 by 125
Normal Multiplication time: 2.85
Matrix size is 250 by 250
Normal Multiplication time: 21.84
Matrix size is 500 by 500
Normal Multiplication time: 176.16
Matrix size is 1000 by 1000
Normal Multiplication time: 1482.92
Matrix size is 2000 by 2000
Normal Multiplication time: 14323.77
```



```
Block width of: 4
Matrix size is 125 by 125
Normal Multiplication time: 0.86
Matrix size is 250 by 250
Normal Multiplication time: 6.09
Matrix size is 500 by 500
Normal Multiplication time: 48.27
Matrix size is 1000 by 1000
Normal Multiplication time: 469.41
Matrix size is 2000 by 2000
Normal Multiplication time: 3871.27
```

```
Block width of: 10
Matrix size is 125 by 125

Normal Multiplication time: 0.21
Matrix size is 250 by 250

Normal Multiplication time: 1.17
Matrix size is 500 by 500

Normal Multiplication time: 9.04
Matrix size is 1000 by 1000

Normal Multiplication time: 74.88
Matrix size is 2000 by 2000

Normal Multiplication time: 592.89
```

```
Block width of: 20
Matrix size is 125 by 125

Normal Multiplication time: 0.23
Matrix size is 250 by 250

Normal Multiplication time: 0.98
Matrix size is 500 by 500

Normal Multiplication time: 7.71
Matrix size is 1000 by 1000

Normal Multiplication time: 62.59
Matrix size is 2000 by 2000

Normal Multiplication time: 497.89
```

```
Block width of: 25
Matrix size is 125 by 125

Normal Multiplication time: 0.16
Matrix size is 250 by 250

Normal Multiplication time: 1.02
Matrix size is 500 by 500

Normal Multiplication time: 8.34
Matrix size is 1000 by 1000

Normal Multiplication time: 65.11
Matrix size is 2000 by 2000

Normal Multiplication time: 518.73
```

Code Part 1

```
//Luka Sobovic
//20215231
//MP3 Part 1
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cuda.h>
#include <curand.h>
#include <time.h>
#include <device_launch_parameters.h>

int main()
{
    // Sizes of input matrices to test
    int sizes[] = { 125, 250, 500, 1000, 2000 };
    int num_sizes = sizeof(sizes) / sizeof(int);

    // Loop over matrix sizes
    for (int i = 0; i < num_sizes; i++)
    {
        int size = sizes[i];
        int num_elements = size * size;

        // Allocate memory for input matrices on host
        float *h_M, *h_N;
        cudaMallocHost(&h_M, num_elements * sizeof(float));
        cudaMallocHost(&h_N, num_elements * sizeof(float));

        srand(time(NULL));
        for (int i = 0; i < size * size; i++) {
            h_M[i] = (float)rand() / RAND_MAX;
            h_N[i] = (float)rand() / RAND_MAX;
        }

        // Allocate memory for input matrices on device
        float *d_M, *d_N;
        cudaMalloc(&d_M, num_elements * sizeof(float));
        cudaMalloc(&d_N, num_elements * sizeof(float));

        // Create events to measure time
        cudaEvent_t start1, stop1, start2, stop2;
        cudaEventCreate(&start1);
        cudaEventCreate(&stop1);
        cudaEventCreate(&start2);
        cudaEventCreate(&stop2);

        // Copy input matrices from host to device and measure time
        cudaEventRecord(start1);
        cudaMemcpy(d_M, h_M, num_elements * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_N, h_N, num_elements * sizeof(float), cudaMemcpyHostToDevice);
        cudaEventRecord(stop1);
        cudaEventSynchronize(stop1);
        float transfer_time = 0;
        cudaEventElapsedTime(&transfer_time, start1, stop1);
        printf("Matrix size %d x %d: Host to device transfer time = %f ms\n", size, size, transfer_time);

        // Copy input matrices from device to host and measure time
        cudaEventRecord(start1);
        cudaMemcpy(h_M, d_M, num_elements * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(h_N, d_N, num_elements * sizeof(float), cudaMemcpyDeviceToHost);
        cudaEventRecord(stop1);
        cudaEventSynchronize(stop1);
        transfer_time = 0;
        cudaEventElapsedTime(&transfer_time, start1, stop1);
        printf("Matrix size %d x %d: Device to host transfer time = %f ms\n", size, size, transfer_time);

        // Free memory
        cudaFreeHost(h_M);
        cudaFreeHost(h_N);
        cudaFree(d_M);
        cudaFree(d_N);
    }

    return 0;
}
```

Code Part 2

```
//Luka Gobovic
//20215231
//MP3 Part 3
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cuda.h>
#include <curand.h>
#include <time.h>
#include <device_launch_parameters.h>

__global__ void matrixMultiplication(float *M, float *N, float *P, int matrixSize) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < matrixSize && col < matrixSize) {
        float pValue = 0;
        for (int k = 0; k < matrixSize; ++k) {
            pValue += M[row * matrixSize + k] * N[k * matrixSize + col];
        }
        P[row * matrixSize + col] = pValue;
    }
}

// Function to perform matrix multiplication on CPU
void matMulCPU(float *M, float *N, float *P, int matrixSize) {
    for (int i = 0; i < matrixSize; i++) {
        for (int j = 0; j < matrixSize; j++) {
            float Pvalue = 0;
            for (int k = 0; k < matrixSize; k++) {
                Pvalue += M[j * matrixSize + k] * N[k * matrixSize + i];
            }
            P[j * matrixSize + i] = Pvalue;
        }
    }
}
```

```
int main()
{
    // Sizes of input matrices to test
    int sizes[] = { 125, 250, 500, 1000, 2000 };

    for (int i = 0; i < 5; i++)
    {
        int size = sizes[i];
        printf("Matrix size is %d by %d\n\n", size, size);
        size_t hostSize = size * size * sizeof(float);

        float gpu_time1 = 0.0f;
        float gpu_time2 = 0.0f;

        // Allocate memory for input matrices on host
        float* h_M = (float*)malloc(hostSize);
        float* h_N = (float*)malloc(hostSize);
        float* h_C_GPU = (float*)malloc(hostSize);
        float* h_C_CPU = (float*)malloc(hostSize);

        srand(time(NULL));
        for (int i = 0; i < size * size; i++) {
            h_M[i] = (float)rand() / RAND_MAX;
            h_N[i] = (float)rand() / RAND_MAX;
        }

        // Allocate memory for input matrices on device
        float *d_M, *d_N, *d_C;
        cudaMalloc(&d_M, hostSize);
        cudaMalloc(&d_N, hostSize);
        cudaMalloc(&d_C, hostSize);

        // Create events to measure time
        cudaEvent_t start1, stop1, start2, stop2;
        cudaEventCreate(&start1);
        cudaEventCreate(&stop1);
        cudaEventCreate(&start2);
        cudaEventCreate(&stop2);
```

```

//Host Multiplication
cudaEventRecord(start1, 0);
matMulCPU(h_M, h_N, h_C_CPU, size);
cudaEventRecord(stop1, 0);

cudaEventElapsedTime(&gpu_time1, start1, stop1);
printf("Host Multiplication time: %0.2f\n", gpu_time1);

cudaMemcpy(d_M, h_M, hostSize, cudaMemcpyHostToDevice);
cudaMemcpy(d_N, h_N, hostSize, cudaMemcpyHostToDevice);

//One thread per block and one total block
dim3 threadsPerBlock(1,1,1);
dim3 numberOfBlocks(ceil(size / (float)threadsPerBlock.x), ceil(size / (float)threadsPerBlock.y), 1);

// //Part 2 -----
cudaEventRecord(start2, 0);
matrixMultiplication << <numberOfBlocks, threadsPerBlock >> >(d_M, d_N, d_C, size);
cudaEventRecord(stop2, 0);
cudaEventSynchronize(stop2);
cudaEventElapsedTime(&gpu_time2, start2, stop2);
cudaMemcpy(h_C_GPU, d_C, hostSize, cudaMemcpyDeviceToHost);
printf("Normal Multiplication time: %0.2f\n", gpu_time2);

for (int i = 0; i < size * size; i++) {
    if (abs(h_C_CPU[i] - h_C_GPU[i]) > 0.00001) {
        printf("Test FAILED\n");
        break;
    }
}
printf("Test PASSED\n\n");

// Free memory
cudaFreeHost(h_M);
cudaFreeHost(h_N);
cudaFreeHost(h_C_CPU);
cudaFreeHost(h_C_GPU);
cudaFree(d_M);
cudaFree(d_N);
cudaFree(d_C);
}
return 0;
}

```

Code Part 3

```
//Luka Gobovic
//20215231
//MP3 Part 3
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cuda.h>
#include <curand.h>
#include <time.h>
#include <device_launch_parameters.h>

#define BLOCK_WIDTH 16

__global__ void matrixMultiplication(float *M, float *N, float *P, int matrixSize) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    if (row < matrixSize && col < matrixSize) {
        for (int k = 0; k < matrixSize; k++) {
            sum += M[row * matrixSize + k] * N[k * matrixSize + col];
        }
        P[row * matrixSize + col] = sum;
    }
}

// Function to perform matrix multiplication on CPU
void matMulCPU(float *M, float *N, float *P, int matrixSize) {
    for (int i = 0; i < matrixSize; i++) {
        for (int j = 0; j < matrixSize; j++) {
            float Pvalue = 0;
            for (int k = 0; k < matrixSize; k++) {
                Pvalue += M[j * matrixSize + k] * N[k * matrixSize + i];
            }
            P[j * matrixSize + i] = Pvalue;
        }
    }
}
```

```
int main()
{
    // Sizes of input matrices to test
    int sizeOfBlock = 1;
    int sizes[] = { 125, 250, 500, 1000, 2000 };
    int blockSizes[] = { 2,4,10,20,25 };

    for (int x = 0; x < 5; x++) {
        sizeOfBlock = blockSizes[x];
        printf("Block width of: %d\n", sizeOfBlock);
        // Loop over matrix sizes
        for (int i = 0; i < 5; i++)
        {
            int size = sizes[i];
            sizeOfBlock = blockSizes[x];
            printf("Matrix size is %d by %d\n\n", size, size);
            size_t hostSize = size * size * sizeof(float);

            float gpu_time1 = 0.0f;
            float gpu_time2 = 0.0f;

            // Allocate memory for input matrices on host
            float *h_M, *h_N, *h_C_CPU, *h_C_GPU;
            cudaMallocHost(&h_M, hostSize);
            cudaMallocHost(&h_N, hostSize);
            cudaMallocHost(&h_C_CPU, hostSize);
            cudaMallocHost(&h_C_GPU, hostSize);

            srand(time(NULL));
            for (int i = 0; i < size * size; i++) {
                h_M[i] = (float)rand() / RAND_MAX;
                h_N[i] = (float)rand() / RAND_MAX;
            }
        }
    }
}
```



```

// Allocate memory for input matrices on device
float *d_M, *d_N, *d_C;
cudaMalloc(&d_M, hostSize);
cudaMalloc(&d_N, hostSize);
cudaMalloc(&d_C, hostSize);

// Create events to measure time
cudaEvent_t start1, stop1, start2, stop2;
cudaEventCreate(&start1);
cudaEventCreate(&stop1);
cudaEventCreate(&start2);
cudaEventCreate(&stop2);

// Host Multiplication
cudaEventRecord(start1, 0);
matMulCPU(h_M, h_N, h_C_CPU, size);
cudaEventRecord(stop1, 0);

cudaEventElapsedTime(&gpu_time1, start1, stop1);
printf("Host Multiplication time: %0.2f\n", gpu_time1);
//-----

cudaMemcpy(d_M, h_M, hostSize, cudaMemcpyHostToDevice);
cudaMemcpy(d_N, h_N, hostSize, cudaMemcpyHostToDevice);

int NumBlocks = size / sizeofBlock;
if (size % sizeofBlock) NumBlocks++;

dim3 numberOfBlocks(NumBlocks, NumBlocks);
dim3 threadsPerBlock(sizeofBlock, sizeofBlock);

cudaEventRecord(start2, 0);
matrixMultiplication << <numberOfBlocks, threadsPerBlock >> >(d_M, d_N, d_C, size);
cudaEventRecord(stop2, 0);
cudaEventSynchronize(stop2);
cudaEventElapsedTime(&gpu_time2, start2, stop2);
cudaMemcpy(h_C_GPU, d_C, hostSize, cudaMemcpyDeviceToHost);
printf("Normal Multiplication time: %0.2f\n", gpu_time2);

```

```

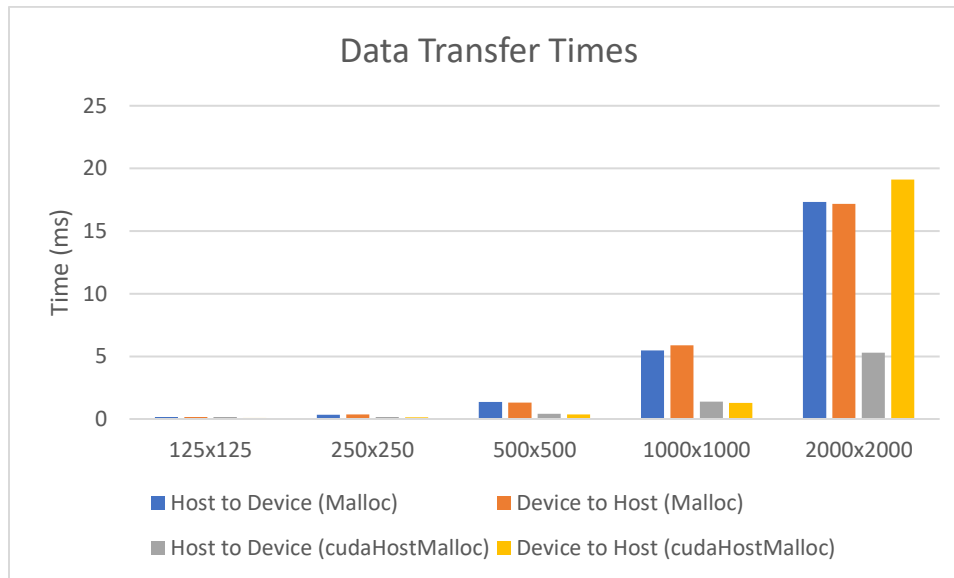
//Test to see if matrices are the same
for (int i = 0; i < size * size; i++) {
    if (abs(h_C_CPU[i] - h_C_GPU[i]) > 0.00001) {
        printf("Test FAILED\n");
        break;
    }
}
printf("Test PASSED\n\n");

// Free memory
cudaFreeHost(h_M);
cudaFreeHost(h_N);
cudaFreeHost(h_C_CPU);
cudaFreeHost(h_C_GPU);
cudaFree(d_M);
cudaFree(d_N);
cudaFree(d_C);
}
printf("\n\n");
}
return 0;
}

```

Analysis

Part 1:



When using both methods, the transfer times seem to be relatively similar for to the device and back from the device. However, when using cudaHostMalloc to allocate memory on the host, the transfer time from the host to the device is around 1/4th of the transfer time back from the device to the host. Generally, as the matrix sizes increase, it can be seen that using cudaHostMalloc results in overall lower transfer times. cudaMallocHost() is a CUDA function that allocates pinned memory. Pinned memory is faster for data transfer between the host and the device because the operating system can avoid copying the data to the pagefile on disk. The pinned memory is stored in page-locked physical memory, which can be directly accessed by the GPU.

Part 2:

When considering data transport time, offloading matrix multiplication to the device (GPU) is not always advantageous. The size of the matrices and the amount of available bandwidth between the host and the device determine whether to offload the computation to the GPU.

The computation time may be dominated by the data transfer time for smaller matrix sizes, making offloading to the device slower. The processing time on the GPU, on the other hand, might be substantially faster than the CPU for higher matrix sizes, balancing the data transmission time and producing a net speedup.

Part 3:

a) Since each element of the input matrix is loaded once into the GPU memory for each multiplication, the total number of loads per element is equal to the total number of multiplications performed. The total number of multiplications needed to compute the output is MATRIX_WIDTH^3 . So if we have a width of 250, the total number of loads would be $250 \times 250 \times 250 = 15,625,000$

b) For each matrix multiplication, each element requires one multiplication and addition. Therefore the number of floating-point operations is equal to 2 multiplied by the number of elements in the output matrix. The number of memory access is equal to the number of elements in both input matrices.

CGMA can be calculated as follows:

$$\text{CGMA} = (\text{number of floating-point operations}) / (\text{number of memory accesses})$$

$$\text{Number of floating-point operations} = 2 * (\text{matrix size})^3$$

$$\text{Number of bytes accessed} = 2 * (\text{matrix size})^2 * \text{sizeof(float)}$$

Therefore, the CGMA in FLOPs/B is:

$$\text{CGMA} = (2 * (\text{matrix size})^3) / (2 * (\text{matrix size})^2 * \text{sizeof(float)})$$

$$= (\text{matrix size}) / \text{sizeof(float)}$$

For a 250x250 matrix of floats (each float taking 4 bytes), the CGMA is:

$$\text{CGMA} = 250 / 4 = 62.5 \text{ FLOPs/B}$$

Generally, for matrix multiplication, the CGMA is 0.25 FLOP/B.