

ELEC 374

Machine Problem 4

Luka Gobovic

20215231

## Results



```
Tile width of: 2
Matrix size is 125 by 125
Host Multiplication time: 0.00
Normal Multiplication time: 5.43
Test FAILED

Matrix size is 250 by 250
Host Multiplication time: 124.28
Normal Multiplication time: 43.17
Test PASSED

Matrix size is 500 by 500
Host Multiplication time: 899.20
Normal Multiplication time: 345.43
Test PASSED

Matrix size is 1000 by 1000
Host Multiplication time: 19363.02
Normal Multiplication time: 2833.60
Test PASSED

Matrix size is 2000 by 2000
Host Multiplication time: 160210.19
Normal Multiplication time: 23907.50
Test PASSED

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

The first test with 125x125 fails because the matrix size is not divisible by the tile width of 2, and this, without proper checks in place, causes the matrix multiplication to fail. I will however attempt to remedy this in the bonus section.

For the purposes of speeding up the testing, I removed the host multiplication portion for the remaining tests since it would take on average around 2-3 minutes. The tests should pass except for when the size is not divisible evenly by the tile width, so 125 and 10, 125 and 20, and 250 and 20. That is why the above screenshot will look different than the rest.

**SIDE NOTE, ALL OF THE RESULTS SHOULD SAY TILED MATRIX MUL NOT NORMAL**

**This was a mistake on my part.**

```
CA. C:\Windows\system32\cmd.exe
File width of: 5
Matrix size is 125 by 125
Host Multiplication time: 0.00
Normal Multiplication time: 0.67
Test PASSED

Matrix size is 250 by 250
Host Multiplication time: 119.25
Normal Multiplication time: 4.47
Test PASSED

Matrix size is 500 by 500
Host Multiplication time: 918.69
Normal Multiplication time: 35.73
Test PASSED

Matrix size is 1000 by 1000
Host Multiplication time: 24168.86
Normal Multiplication time: 300.44
Test PASSED

Matrix size is 2000 by 2000
Host Multiplication time: 196670.31
Normal Multiplication time: 2333.67
Test PASSED
```

```
CA. C:\Windows\system32\cmd.exe
File width of: 10
Matrix size is 125 by 125
Normal Multiplication time: 0.30
Matrix size is 250 by 250
Normal Multiplication time: 1.71
Matrix size is 500 by 500
Normal Multiplication time: 13.46
Matrix size is 1000 by 1000
Normal Multiplication time: 106.50
Matrix size is 2000 by 2000
Normal Multiplication time: 849.76
Press any key to continue . . . _
```

```
C:\Windows\system32\cmd.exe

Tile width of: 20
Matrix size is 125 by 125
Normal Multiplication time: 0.24
Matrix size is 250 by 250
Normal Multiplication time: 1.13
Matrix size is 500 by 500
Normal Multiplication time: 9.54
Matrix size is 1000 by 1000
Normal Multiplication time: 75.50
Matrix size is 2000 by 2000
Normal Multiplication time: 602.43
Press any key to continue . . . _
```

```
C:\Windows\system32\cmd.exe

Tile width of: 25
Matrix size is 125 by 125
Normal Multiplication time: 0.22
Matrix size is 250 by 250
Normal Multiplication time: 1.24
Matrix size is 500 by 500
Normal Multiplication time: 8.94
Matrix size is 1000 by 1000
Normal Multiplication time: 70.50
Matrix size is 2000 by 2000
Normal Multiplication time: 560.21
Press any key to continue . . . _
```

## Code

Tile width parameter at the top was changed every run.

```
//Luka Gobovic
//20215231
//MP4 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>

#define TILE_WIDTH 2

// 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;
        }
    }
}

//Tiled Matrix Multiplication
__global__ void tiled_matrix_multiply(float *A, float *B, float *C, int n)
{
    __shared__ float tileA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float tileB[TILE_WIDTH][TILE_WIDTH];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;
```

```
float sum = 0;

for (int i = 0; i < n / TILE_WIDTH; i++) {
    tileA[ty][tx] = A[row * n + (i * TILE_WIDTH + tx)];
    tileB[ty][tx] = B[(i * TILE_WIDTH + ty) * n + col];
    __syncthreads();

    for (int j = 0; j < TILE_WIDTH; j++) {
        sum += tileA[ty][j] * tileB[j][tx];
    }
    __syncthreads();
}

C[row * n + col] = sum;
}

int main()
{
    // Sizes of input matrices to test
    int sizes[] = { 125, 250, 500, 1000, 2000 };
    printf("Tile width of: %d\n", TILE_WIDTH);

    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);

dim3 threadsPerBlock(TILE_WIDTH, TILE_WIDTH);
dim3 numberOfBlocks(size / TILE_WIDTH, size / TILE_WIDTH);

//Tiled Mul -----
cudaEventRecord(start2, 0);
tiled_matrix_multiply << << 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]) > 0.1) {
            printf("Test FAILED\n");
            break;
        }
    }
    printf("Test PASSED\n\n");

    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;
}

```

## BONUS CODE

```
//Luka Gobovic
//20215231
//MP4 Part 2 (BONUS)
#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
#define TILE_WIDTH 8

//CPU Multiply
void matrix_multiply(float *A, float *B, float *C, int m, int n, int p) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            float sum = 0;
            for (int k = 0; k < n; k++) {
                sum += A[i * n + k] * B[k * p + j];
            }
            C[i * p + j] = sum;
        }
    }
}

__global__ void tiled_matrix_multiply(float *A, float *B, float *C, int m, int n, int p, int tile_size_x, int tile_size_y) {
    // calculate the row and column indices of the current thread
    int row = blockIdx.y * tile_size_y + threadIdx.y;
    int col = blockIdx.x * tile_size_x + threadIdx.x;

    // allocate shared memory for the tile of A and B
    extern __shared__ float tile[];
    float *tile_A = &tile[0];
    float *tile_B = &tile[tile_size_x * tile_size_y];

    // initialize the tile of C to zero
    float tile_C = 0;

    // iterate over tiles of A and B
    for (int i = 0; i < (n + tile_size_x - 1) / tile_size_x; ++i) {
        // check if the current thread is within the bounds of A and B
        if (row < m && i * tile_size_x + threadIdx.x < n) {
            tile_A[threadIdx.y * tile_size_x + threadIdx.x] = A[row * n + i * tile_size_x + threadIdx.x];
        }
        else {
            tile_A[threadIdx.y * tile_size_x + threadIdx.x] = 0;
        }
        if (col < p && i * tile_size_x + threadIdx.y < n) {
            tile_B[threadIdx.y * tile_size_x + threadIdx.x] = B[(i * tile_size_x + threadIdx.y) * p + col];
        }
        else {
            tile_B[threadIdx.y * tile_size_x + threadIdx.x] = 0;
        }
        // synchronize threads to ensure tiles have been loaded
        __syncthreads();
        // perform the matrix multiplication for the current tile
        for (int j = 0; j < tile_size_x; ++j) {
            tile_C += tile_A[threadIdx.y * tile_size_x + j] * tile_B[j * tile_size_x + threadIdx.x];
        }
        // synchronize threads to ensure previous calculation has completed
        __syncthreads();
    }
    // write the result to global memory if within the bounds of C
    if (row < m && col < p) {
        C[row * p + col] = tile_C;
    }
}
```

```

__global__ void tiled_matrix_multiply_noBoundaries(float *A, float *B, float *C, int n)
{
    __shared__ float tileA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float tileB[TILE_WIDTH][TILE_WIDTH];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;

    float sum = 0;

    for (int i = 0; i < n / TILE_WIDTH; i++) {
        tileA[ty][tx] = A[row * n + (i * TILE_WIDTH + tx)];
        tileB[ty][tx] = B[(i * TILE_WIDTH + ty) * n + col];
        __syncthreads();

        for (int j = 0; j < TILE_WIDTH; j++) {
            sum += tileA[ty][j] * tileB[j][tx];
        }
        __syncthreads();
    }

    C[row * n + col] = sum;
}

```

```

int main()
{
    const int M_rows = 350;
    const int M_cols = 400;
    const int N_rows = 400;
    const int N_cols = 500;

    printf("Matrix M size is %d by %d\n\n", M_rows, M_cols);
    printf("Matrix N size is %d by %d\n\n", N_rows, N_cols);

    // Allocate memory for matrices on host
    float* M = (float*)malloc(M_rows * M_cols * sizeof(float));
    float* N = (float*)malloc(N_rows * N_cols * sizeof(float));
    float* P = (float*)malloc(M_rows * N_cols * sizeof(float));
    float* P_CPU = (float*)malloc(M_rows * N_cols * sizeof(float));

    // Initialize matrices with random values
    srand(time(NULL));
    for (int i = 0; i < M_rows; i++) {
        for (int j = 0; j < M_cols; j++) {
            M[i * M_cols + j] = (float)rand() / RAND_MAX;
        }
    }
    for (int i = 0; i < N_rows; i++) {
        for (int j = 0; j < N_cols; j++) {
            N[i * N_cols + j] = (float)rand() / RAND_MAX;
        }
    }

    // Matrix dimensions on device
    const int M_rows_d = M_rows;
    const int M_cols_d = M_cols;
    const int N_rows_d = N_rows;
    const int N_cols_d = N_cols;
    const int P_rows_d = M_rows_d;
    const int P_cols_d = N_cols_d;
}

```



```

// Allocate memory for matrices on device
float* M_d, *N_d, *P_d;
cudaMalloc(&M_d, M_rows_d * M_cols_d * sizeof(float));
cudaMalloc(&N_d, N_rows_d * N_cols_d * sizeof(float));
cudaMalloc(&P_d, P_rows_d * P_cols_d * sizeof(float));

// Copy matrices from host to device
cudaMemcpy(M_d, M, M_rows_d * M_cols_d * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(N_d, N, N_rows_d * N_cols_d * sizeof(float), cudaMemcpyHostToDevice);

// Kernel configuration
const int tile_width = 8;
const int tile_height = 15;
dim3 gridDim((N_cols + tile_width - 1) / tile_width, (M_rows + tile_height - 1) / tile_height, 1);
dim3 blockDim(tile_width, tile_height, 1);
int sharedMemSize = 2 * tile_width * tile_height * sizeof(float);

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

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

//HOST TEST
cudaEventRecord(start1, 0);
matrix_multiply(M, N, P_CPU, M_rows, M_cols, N_cols);
cudaEventRecord(stop1, 0);

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

```

```

//-----
cudaEventRecord(start2, 0);
tiled_matrix_multiply << <gridDim, blockDim, sharedMemSize >> > (M_d, N_d, P_d, M_rows_d, M_cols_d, N_cols_d, tile_width, tile_height);
cudaDeviceSynchronize();
cudaEventRecord(stop2, 0);
cudaEventSynchronize(stop2);
cudaEventElapsedTime(&gpu_time2, start2, stop2);
cudaMemcpy(P, P_d, P_rows_d * P_cols_d * sizeof(float), cudaMemcpyDeviceToHost);
printf("Tiled Boundary Multiplication time: %0.2f\n", gpu_time2);

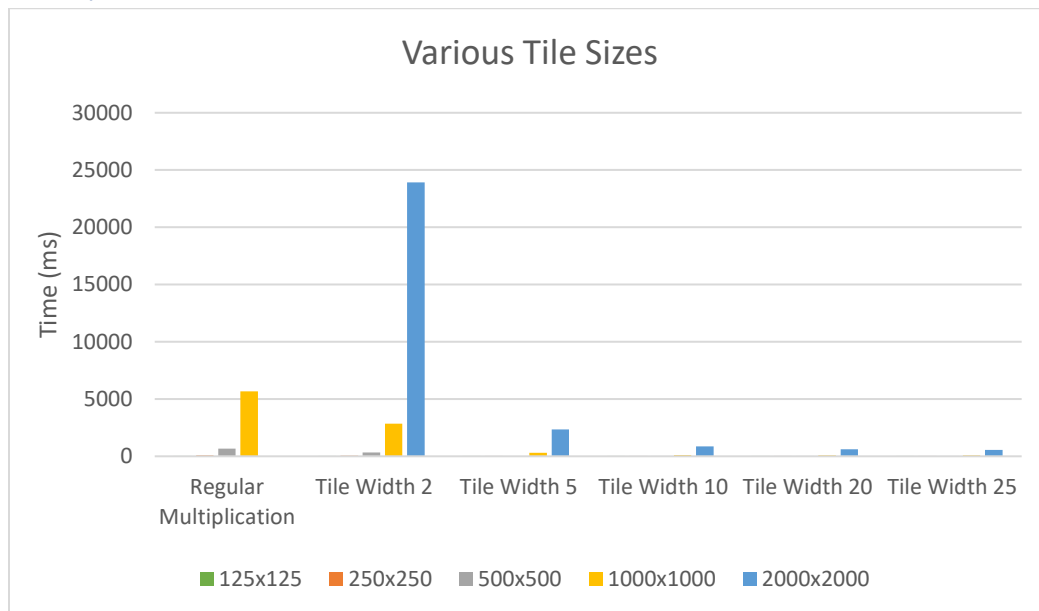
int passedFlag = 1;
for (int i = 0; i < M_rows * N_cols; i++) {
    if (abs(P[i] - P_CPU[i]) > 0.001) {
        passedFlag = 0;
        break;
    }
}
if (passedFlag) {
    printf("Test PASSED\n\n");
}
else {
    printf("Test FAILED\n\n");
}

// Free memory
free(M);
free(N);
free(P);
cudaFree(M_d);
cudaFree(N_d);
cudaFree(P_d);

return 0;

```

## Analysis



The 2000x2000 result for the regular matrix multiplication was removed from the chart above to allow the rest of the results to scale better. The result for that test was 23,907.50 ms

It can be seen that when comparing tiled multiplication to single thread single block multiplication, it is significantly faster, and tends to decrease as tile width is increased, especially for the larger matrices. However, when comparing the results to the second portion of MP3, where we experimented with different block widths, the results tend to be much more closely matched. For example, when using a block width of 20, the results for the 2000x2000 matrix was 497.89 ms, marginally faster than the tiled multiplication using a tile width of 25.

## Questions

1. When using the Tesla C2075, which has 14 streaming multiprocessors, the maximum number of threads that can be simultaneously executing on the device is  $14 \times 2048 = 28,672$  threads.

Maximum number of threads per block: 1024

Maximum number of blocks per SM: 8 (for compute capability 2.x)

Total number of blocks:  $14 \times 8 = 112$

Total number of threads:  $112 \times 1024 = 114688$

114688 threads can be simultaneously scheduled on the GPU.

2. All of these values are using a tile width of 25: This was found using the `nvcc--ptxas-options=-v` options when compiling.

Number of registers used by matrixMultiplication kernel: 11

Number of registers used by tiled\_matrix\_multiply kernel: 17

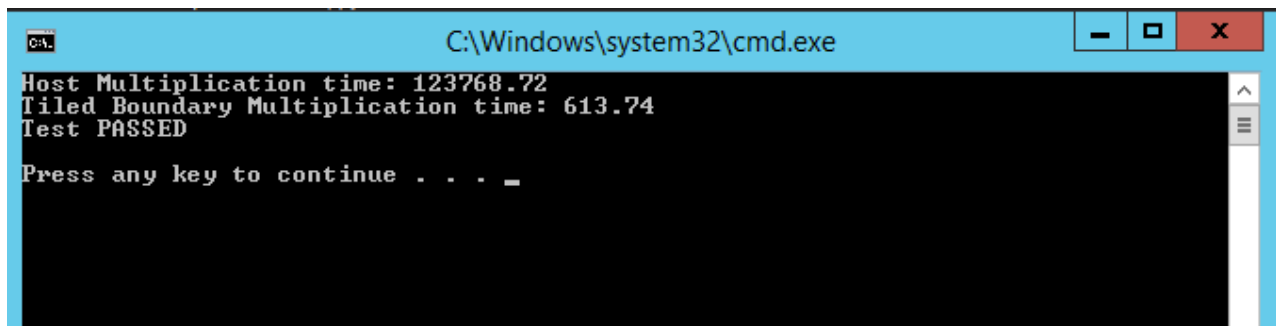
Size of shared memory used by tiled\_matrix\_multiply kernel: 5000 bytes

From above, the number of blocks per streaming multiprocessor is 8, and the maximum total threads simultaneously scheduled is 114,688, with 28,672 executing.

If however, I use a tile width of 2, the size of shared memory decreases to 32 bytes. From this, it can be seen that the size of both of the shared matrices in the kernel is equal to  $(\text{tile\_width})^2 * 2(\text{size of float})$ . So in this case, one of the share matrices is  $4 * 4 \text{ bytes} = 16 \text{ bytes}$ , equating to 32 total bytes for both.

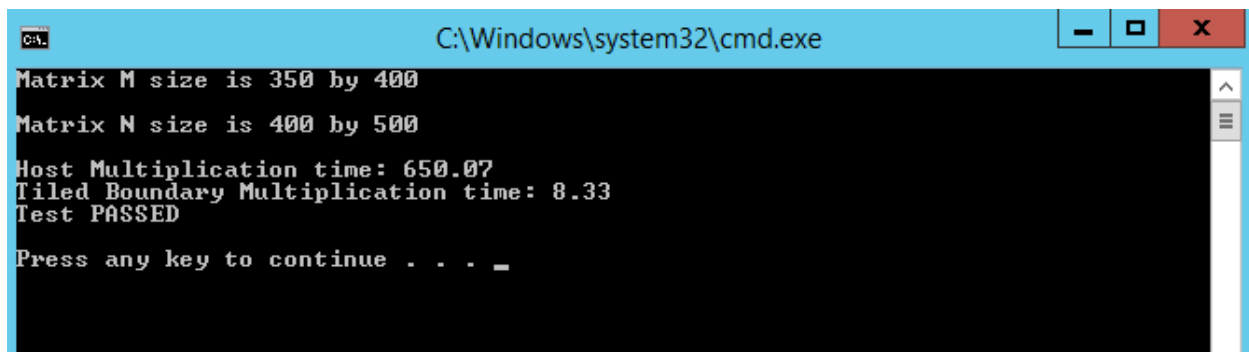
## BONUS

Tiled Multiplication for 1900x1600 and 1600x1300, tile width of 8 by 15



```
C:\Windows\system32\cmd.exe
Host Multiplication time: 123768.72
Tiled Boundary Multiplication time: 613.74
Test PASSED
Press any key to continue . . . _
```

Tiled Multiplication for 350x400 and 400x500, tile width of 8 by 15



```
C:\Windows\system32\cmd.exe
Matrix M size is 350 by 400
Matrix N size is 400 by 500
Host Multiplication time: 650.07
Tiled Boundary Multiplication time: 8.33
Test PASSED
Press any key to continue . . . _
```

## Analysis

The tests above passed because I ran the host multiplication on non-square matrices as well, simply by changing around some of the parameters being passed in along with the for loops.

For a 400x400 matrix, using a tile size of 8, it took roughly 6.5ms for the computation. Seeing as the output matrix of the boundary check matrix has 15,000 more elements, that could explain the discrepancy. However, it is difficult to exactly compare the results of the two different methods. While it may take longer to execute the calculations due to additional calculations and checks that are necessary to avoid out of bounds issues, the revised kernel allows us to perform operations on non-square matrices and where the dimensions are not a multiple of the dimensions of the tiles.