ELEC 374

Machine Problem 2

Luka Gobovic

20215231

Academic Integrity

"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".

Code

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

#define BLOCK_SIZE 16

//Regular Host Matrix Addition
void hostAddition(float * C, const float *A, const float *B, int N)
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i*N + j] = A[i*N + j] + B[i*N + j];
        }
    }
}

//Matrix Additon
__global__ void matrix_addition(float* C, const float* A, const float* B, int N)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    int idx = i + j * N;

    if (i < N && j < N) {
        C[idx] = A[idx] + B[idx];
    }
}

//Matrix Additon via Columns
__global__ void matrixAddColKernel(float* C, const float* A, const float* B, int N) {
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    if (col < N) {
        for (int i = 0; i < N; i++) {
            C[i * N + col] = A[i * N + col] + B[i * N + col];
        }
    }
}

//Matrix Additon via Row
__global__ void matrixAddRowKernel(float* C, float* A, float* B, int N)
{
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    if (row < N) {
        for (int j = 0; j < N; j++) {
            C[row * N + j] = A[row * N + j] + B[row * N + j];
        }
    }
}
```

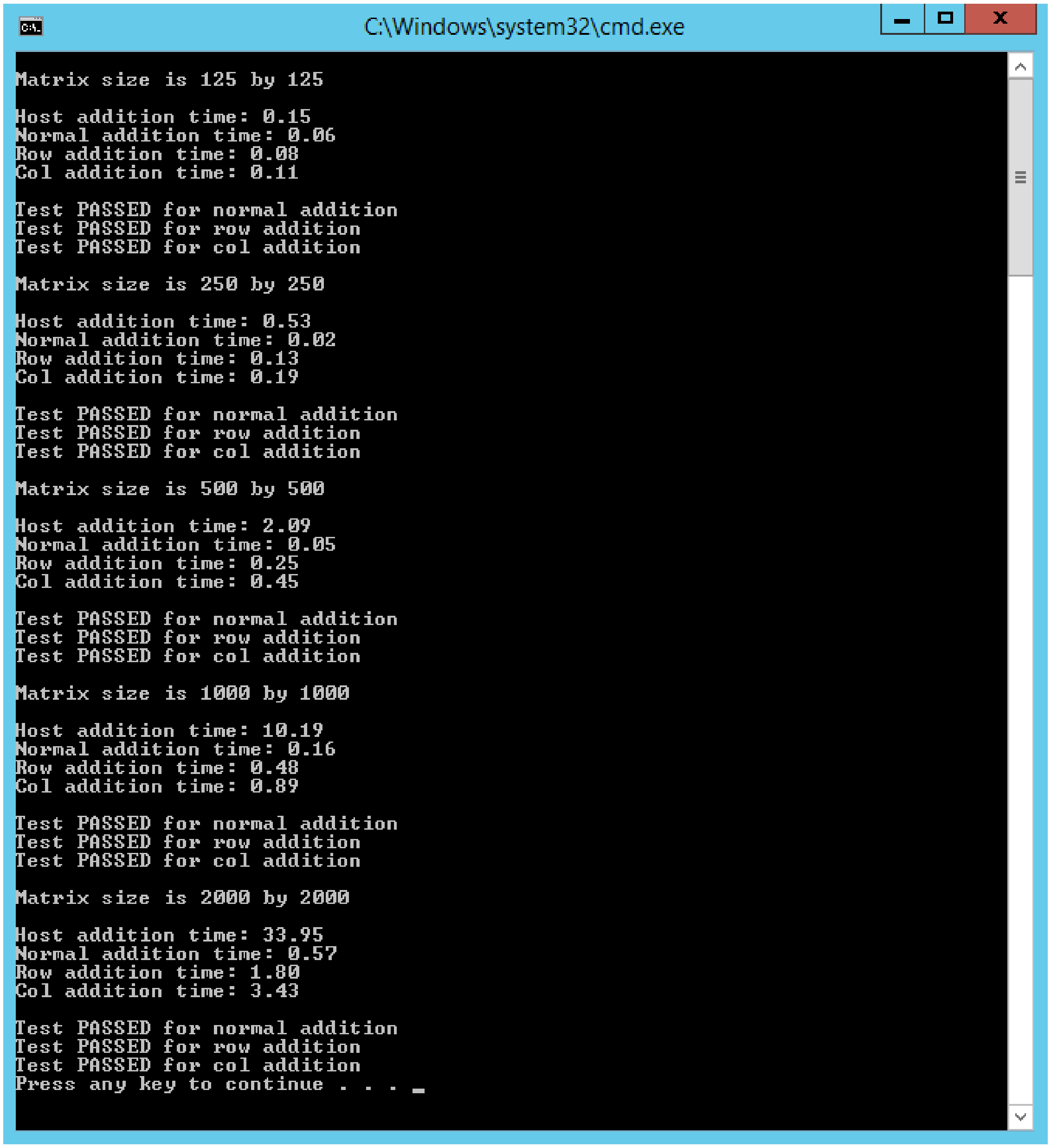
```
int main()
{
    int sizes[5] = { 125,250,500,1000,2000 };
    int n;
    for (int x = 0; x < 5; x++) {

        printf("\n");
        n = sizes[x];
        printf("Matrix size is %d by %d\n\n", n, n);
        size_t bytes = n * n * sizeof(float);

        time_t t;
        cudaEvent_t startHost, stopHost, start1, stop1, start2, stop2, start3, stop3;

        //Events for start timers
        cudaEventCreate(&startHost);
        cudaEventCreate(&start1);
        cudaEventCreate(&start2);
        cudaEventCreate(&start3);

        //Events for stop timers
        cudaEventCreate(&stopHost);
        cudaEventCreate(&stop1);
        cudaEventCreate(&stop2);
        cudaEventCreate(&stop3);

        float gpu_time = 0.0f, gpu_time1 = 0.0f, gpu_time2 = 0.0f, gpu_time3 = 0.0f;

        // Allocate host memory
        float* h_A = (float*)malloc(bytes);
        float* h_B = (float*)malloc(bytes);
        float* h_C_reg = (float*)malloc(bytes);
        float* h_C_row = (float*)malloc(bytes);
        float* h_C_col = (float*)malloc(bytes);
        float* h_C_host = (float*)malloc(bytes);

        // Initialize matrices A and B with random values
        srand(time(NULL));
        for (int i = 0; i < n * n; i++) {
            h_A[i] = (float)rand() / RAND_MAX;
            h_B[i] = (float)rand() / RAND_MAX;
        }

        // Allocate device memory
        float* d_A, *d_B, *d_C;
        cudaMalloc(&d_A, bytes);
        cudaMalloc(&d_B, bytes);
        cudaMalloc(&d_C, bytes);
```

```
//HOST ADDITION PORTION
//--------------------------------------------------
cudaEventRecord(startHost, 0);
hostAddition(h_C_host, h_A, h_B, n);
cudaEventRecord(stopHost, 0);

cudaEventElapsedTime(&gpu_time, startHost, stopHost);
printf("Host addition time: %0.2f\n", gpu_time);
//--------------------------------------------------

// Copy data from host to device
cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration
dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 numberOfBlocks(ceil((n + threadsPerBlock.x - 1) / threadsPerBlock.x), ceil((n + threadsPerBlock.y - 1) / threadsPerBlock.y));

//Individual Threads
//--------------------------------------------------
cudaEventRecord(start1, 0);
matrix_addition << < numberOfBlocks, threadsPerBlock >> >(d_C, d_A, d_B, n);
cudaEventRecord(stop1, 0);
cudaEventSynchronize(stop1);

// Copy data from device to host
cudaMemcpy(h_C_reg, d_C, bytes, cudaMemcpyDeviceToHost);
//Get time
cudaEventElapsedTime(&gpu_time1, start1, stop1);
printf("Normal addition time: %0.2f\n", gpu_time1);
//--------------------------------------------------

//Row Threads
//--------------------------------------------------
cudaEventRecord(start2, 0);
matrixAddRowKernel << < ceil(n / BLOCK_SIZE), BLOCK_SIZE >> >(d_C, d_A, d_B, n);
cudaEventRecord(stop2, 0);
cudaEventSynchronize(stop2);

// Copy data from device to host
cudaMemcpy(h_C_row, d_C, bytes, cudaMemcpyDeviceToHost);
//Get time
cudaEventElapsedTime(&gpu_time2, start2, stop2);
printf("Row addition time: %0.2f\n", gpu_time2);
//--------------------------------------------------
```

```
//Col Threads
//-----------------------------------------------------
cudaEventRecord(start3, 0);
matrixAddColKernel << < ceil(n / BLOCK_SIZE), BLOCK_SIZE >> >(d_C, d_A, d_B, n);
cudaEventRecord(stop3, 0);
cudaEventSynchronize(stop3);

// Copy data from device to host
cudaMemcpy(h_C_col, d_C, bytes, cudaMemcpyDeviceToHost);
//Get time
cudaEventElapsedTime(&gpu_time3, start3, stop3);
printf("Col addition time: %0.2f\n", gpu_time3);
//-----------------------------------------------------

printf("\n");
// Compare results for normal addition
for (int i = 0; i < n * n; i++) {
    if (abs(h_C_host[i] - h_C_reg[i]) > 0.00001) {
        printf("Test FAILED\n");
        break;
    }
}
printf("Test PASSED for normal addition\n");

// Compare results for row addition
for (int i = 0; i < n * n; i++) {
    if (abs(h_C_host[i] - h_C_row[i]) > 0.00001) {
        printf("Test FAILED\n");
        break;
    }
}
printf("Test PASSED for row addition\n");

// Compare results for col addition
for (int i = 0; i < n * n; i++) {
    if (abs(h_C_host[i] - h_C_col[i]) > 0.00001) {
        printf("Test FAILED\n");
        break;
    }
}
printf("Test PASSED for col addition\n");
```

```
        // Free memory
        free(h_A);
        free(h_B);
        free(h_C_reg);
        free(h_C_row);
        free(h_C_col);
        free(h_C_host);
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);

    }
    return 0;
}
```

Output Screenshots



Results

Table 1: Table of the run times of each method for each matrix size, over an average of 4 runs.

	125x125	250x250	500x500	1000x1000	2000x2000
Host Time	0.153	0.595	2.408	9.52	39.863
Normal GPU Matrix Addition	0.068	0.02	0.05	0.16	0.583
Row Addition	0.08	0.133	0.245	0.47	1.79
Column Addition	0.108	0.195	0.455	0.888	3.44

Table 2: Each of the 4 runs that was used to find the averages above.

Run 1

0.14	0.52	2.48	9.87	39.05
0.07	0.02	0.05	0.16	0.58
0.08	0.13	0.24	0.47	1.79
0.11	0.2	0.45	0.89	3.43

Run 2

0.16	0.61	2.12	8.58	40.43
0.05	0.02	0.05	0.16	0.58
0.08	0.13	0.24	0.46	1.79
0.11	0.19	0.45	0.88	3.44

Run 3

0.17	0.61	2.46	9.8	39.76
0.08	0.02	0.05	0.16	0.59
0.08	0.14	0.25	0.48	1.8
0.11	0.2	0.46	0.89	3.44

Run 4

0.14	0.64	2.57	9.81	40.21
0.07	0.02	0.05	0.16	0.58
0.08	0.13	0.25	0.47	1.78
0.1	0.19	0.46	0.89	3.43

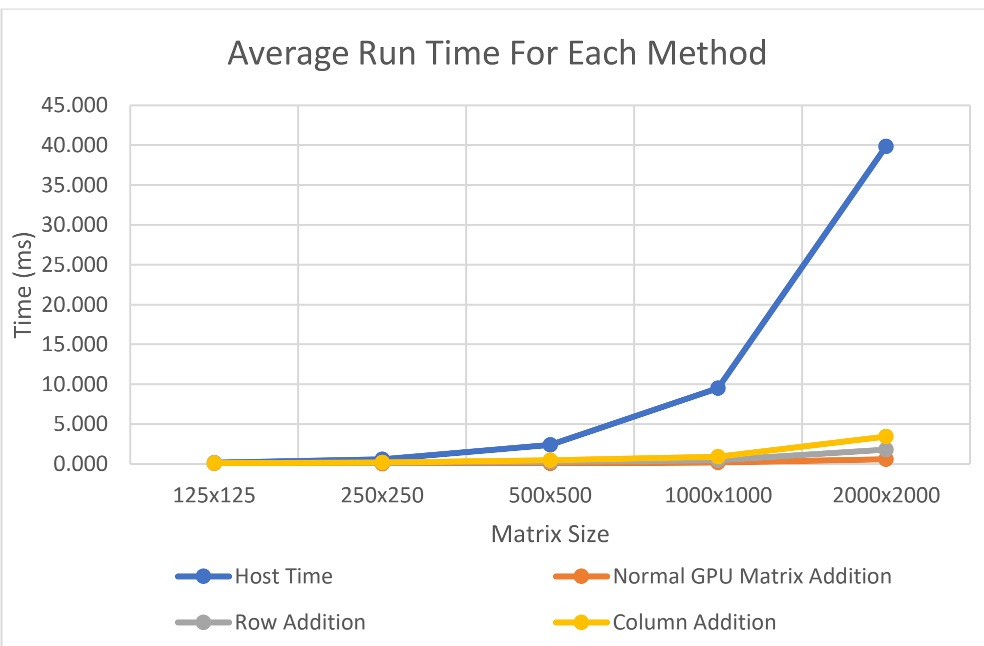


Figure 1: Average run time of all 4 matrix addition methods.

Analysis

It appears that for smaller size matrices, such as 125x125, the host is able to compute matrix addition fairly quickly, keeping up with the GPU. However, when the sizes increase, the host simply lacks the computational power to keep up with the GPU. Based on results above, it can be seen that for any matrices above 125x125, the GPU should be used to avoid unnecessarily long computation times.

As for the three different methods of performing GPU matrix addition, performing it by individual thread appears to be the quickest and most stable method when compared against row and column addition. The row addition appears to be slightly quicker than column addition, but as sizes increase, both methods take longer that normal GPU matrix addition. Row addition tends to be faster because consecutive threads access consecutive elements in memory, leading to coalesced memory access, which is overall faster. Despite this, all three methods passed the tests, and the values were transferred correctly.