ME 766: Assignment 3
Matrix multiplication using CUDA

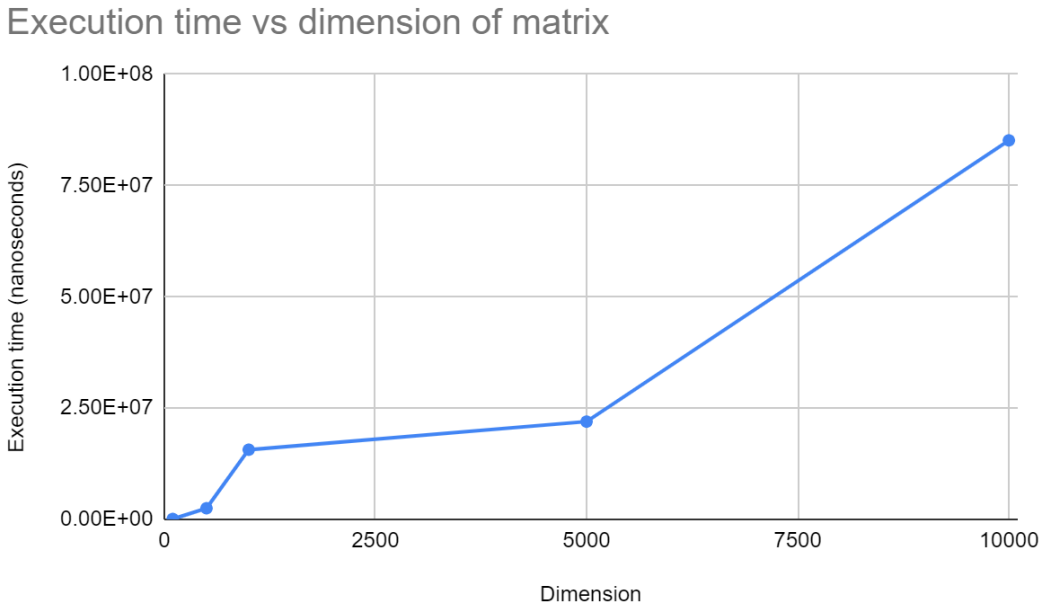
Abhijeet Prasad Bodas
190100004

Machine specifications

Environment: Google colab notebook, using [nvcc4jupyter]() plugin.
CUDA v11.2
GPU: Nvidia Tesla T4

Timing study

N	Run 1	Run 2	Run 3	Average (nanoseconds)
100	88534	84869	87864	8.71E+04
500	2490419	2483295	2508935	2.49E+06
1000	15629685	15703947	15617744	1.57E+07
5000	21546761	22386895	21953473	2.20E+07
10000	84661015	86524166	84290646	8.52E+07



Code

```
/*
  Optimized matrix multiplication using CUDA.
  Authored by: Abhijeet Prasad Bodas <190100004@iitb.ac.in>
*/

#include <bits/stdc++.h>
using namespace std;

__global__ void multiplyMatrices(const int *a, const int *b, int *c, int N)
{
    // Kernel function
    int block = blockIdx.x;
    int thread = threadIdx.x;

    c[block * N + thread] = 0;
    for (int i = 0; i < N; i++)
    {
        // Iterate over a row of A and a column of B and multiply corresponding
        // elements.
        // Because of how we have mapped the matrix to memory locations,
        // these elements will not need to be accessed by any other thread.
        c[block * N + thread] = c[block * N + thread] + a[block * N + i] * b[i * N + thread];
    }
}

void test(vector<int> &A, vector<int> &B, vector<int> &C, int N)
{
    // Compare the result found using CUDA and assert that it is
    // the same as that calculated serially.

    // We do not include this function in time measurements.

    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            int serially_calculated_element = 0;
            for (int k = 0; k < N; k++)
            {
                serially_calculated_element += A[i * N + k] * B[k * N + j];
            }

            // Check that the serial and CUDA calculated elements are equal!
            assert(serially_calculated_element == C[i * N + j]);
        }
    }
```

```
}

int main()
{
    int N = 1 << 5; // Dimension of the matrices
    size_t size_in_bytes = N * N * sizeof(int);

    // Host vectors
    vector<int> A_host(N * N);
    vector<int> B_host(N * N);
    vector<int> C_host(N * N);

    // Randomly generate elements of A and B.
    generate(A_host.begin(), A_host.end(), []() { return rand() % 10; });
    generate(B_host.begin(), B_host.end(), []() { return rand() % 10; });

    // Allocate device memory
    int *A_device, *B_device, *C_device;
    cudaMalloc(&A_device, size_in_bytes);
    cudaMalloc(&B_device, size_in_bytes);
    cudaMalloc(&C_device, size_in_bytes);

    // Copy data from host to  device
    cudaMemcpy(A_device, A_host.data(), size_in_bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(B_device, B_host.data(), size_in_bytes, cudaMemcpyHostToDevice);

    auto start_time = std::chrono::high_resolution_clock::now();

    // Launch kernel
    multiplyMatrices<<<N, N>>>(A_device, B_device, C_device, N);

    // Copy data back from device back to the host
    cudaMemcpy(C_host.data(), C_device, size_in_bytes, cudaMemcpyDeviceToHost);

    auto end_time = std::chrono::high_resolution_clock::now();
    auto time_delta = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time -
start_time);
    cout << "Time taken for multiplication (nanoseconds): " << time_delta.count() << "\n";

    // Verify that multiplication was done correctly.
    test(A_host, B_host, C_host, N);

    cout << "Run succeeded";

    // Free memory on device
    cudaFree(A_device);
    cudaFree(B_device);
    cudaFree(C_device);

    return 0;
}
```