CS 461

Lab Assignment 8

Name: Gandhi Dhruv Vipulkumar

Institute ID: 202151053

Date: 22-10-2024

**NOTE**: Due to unavailability of NVIDIA GPU in the local machine the following code is run on google colab.

Link: <https://colab.research.google.com/drive/1ztJ8tsSUh5BT5raDCKg8Tmfa3-HYcn_m?usp=sharing>

**Q. Implement Matrix Multiplication using CUDA**

!apt-get install nvidia-cuda-toolkit g++

%%writefile matrix\_mul.cu  
#include <stdio.h>  
#include <stdlib.h>  
#include <assert.h>  
#include <cuda\_runtime.h>  
#include <omp.h>  
  
#define BLOCK\_SIZE 16  
#define PRINT\_LIMIT 10 // Limit to print elements of large matrices  
  
// Function to print a matrix (with limits for large matrices)  
void print\_matrix(int\* matrix, int rows, int cols, const char\* name)   
{  
    printf("Matrix %s (%d x %d):\n", name, rows, cols);  
    for (int i = 0; i < rows && i < PRINT\_LIMIT; ++i)   
    {  
        for (int j = 0; j < cols && j < PRINT\_LIMIT; ++j)   
        {  
            printf("%4d ", matrix[i \* cols + j]);  
        }  
        if (cols > PRINT\_LIMIT)   
        {  
            printf("... "); // Print ellipsis if there are more columns  
        }  
        printf("\n");  
    }  
    if (rows > PRINT\_LIMIT)   
    {  
        printf("... \n"); // Print ellipsis if there are more rows  
    }  
    printf("\n");  
}  
  
// CUDA kernel for general matrix multiplication  
\_\_global\_\_ void gpu\_matrix\_mult(int\* a, int\* b, int\* c, int m, int n, int k)   
{  
    int row = blockIdx.y \* blockDim.y + threadIdx.y;  
    int col = blockIdx.x \* blockDim.x + threadIdx.x;  
    int sum = 0;  
    if (col < k && row < m)   
    {  
        for (int i = 0; i < n; i++)   
        {  
            sum += a[row \* n + i] \* b[i \* k + col];  
        }  
        c[row \* k + col] = sum;  
    }  
}  
  
// CUDA kernel for square matrix multiplication  
\_\_global\_\_ void gpu\_square\_matrix\_mult(int\* d\_a, int\* d\_b, int\* d\_result, int n)   
{  
    \_\_shared\_\_ int tile\_a[BLOCK\_SIZE][BLOCK\_SIZE];  
    \_\_shared\_\_ int tile\_b[BLOCK\_SIZE][BLOCK\_SIZE];  
  
    int row = blockIdx.y \* BLOCK\_SIZE + threadIdx.y;  
    int col = blockIdx.x \* BLOCK\_SIZE + threadIdx.x;  
    int tmp = 0;  
    int idx;  
  
    for (int sub = 0; sub < gridDim.x; ++sub)   
    {  
        idx = row \* n + sub \* BLOCK\_SIZE + threadIdx.x;  
        tile\_a[threadIdx.y][threadIdx.x] = (idx < n \* n) ? d\_a[idx] : 0;  
  
        idx = (sub \* BLOCK\_SIZE + threadIdx.y) \* n + col;  
        tile\_b[threadIdx.y][threadIdx.x] = (idx < n \* n) ? d\_b[idx] : 0;  
  
        \_\_syncthreads();  
  
        for (int k = 0; k < BLOCK\_SIZE; ++k)   
        {  
            tmp += tile\_a[threadIdx.y][k] \* tile\_b[k][threadIdx.x];  
        }  
        \_\_syncthreads();  
    }  
  
    if (row < n && col < n)   
    {  
        d\_result[row \* n + col] = tmp;  
    }  
}  
  
// OpenMP function for matrix multiplication (parallelized)  
void openmp\_matrix\_mult(int\* h\_a, int\* h\_b, int\* h\_c, int m, int n, int k)   
{  
#pragma omp parallel for collapse(2)  
    for (int i = 0; i < m; ++i)   
    {  
        for (int j = 0; j < k; ++j)   
        {  
            int tmp = 0;  
            for (int h = 0; h < n; ++h)   
            {  
                tmp += h\_a[i \* n + h] \* h\_b[h \* k + j];  
            }  
            h\_c[i \* k + j] = tmp;  
        }  
    }  
}  
  
// Normal (sequential) matrix multiplication function  
void cpu\_matrix\_mult(int\* h\_a, int\* h\_b, int\* h\_result, int m, int n, int k)   
{  
    for (int i = 0; i < m; ++i)   
    {  
        for (int j = 0; j < k; ++j)   
        {  
            int tmp = 0;  
            for (int h = 0; h < n; ++h)   
            {  
                tmp += h\_a[i \* n + h] \* h\_b[h \* k + j];  
            }  
            h\_result[i \* k + j] = tmp;  
        }  
    }  
}  
  
// Main function  
int main(int argc, char const\* argv[])   
{  
    int m, n, k;  
    srand(3333); // Fixed seed  
    printf("Please type in m, n, and k: ");  
    scanf("%d %d %d", &m, &n, &k);  
  
    // Allocate memory in host RAM  
    int\* h\_a, \* h\_b, \* h\_c, \* h\_cc;  
    cudaMallocHost((void\*\*)&h\_a, sizeof(int) \* m \* n);  
    cudaMallocHost((void\*\*)&h\_b, sizeof(int) \* n \* k);  
    cudaMallocHost((void\*\*)&h\_c, sizeof(int) \* m \* k);  
    cudaMallocHost((void\*\*)&h\_cc, sizeof(int) \* m \* k);  
  
    // Random initialize matrix A  
    for (int i = 0; i < m; ++i)   
    {  
        for (int j = 0; j < n; ++j)   
        {  
            h\_a[i \* n + j] = rand() % 1024;  
        }  
    }  
  
    // Random initialize matrix B  
    for (int i = 0; i < n; ++i)   
    {  
        for (int j = 0; j < k; ++j)   
        {  
            h\_b[i \* k + j] = rand() % 1024;  
        }  
    }  
  
    // Print matrices A and B  
    print\_matrix(h\_a, m, n, "A");  
    print\_matrix(h\_b, n, k, "B");  
  
    float gpu\_elapsed\_time\_ms, cpu\_elapsed\_time\_ms, normal\_elapsed\_time\_ms;  
  
    // Start measuring GPU execution time  
    cudaEvent\_t start, stop;  
    cudaEventCreate(&start);  
    cudaEventCreate(&stop);  
    cudaEventRecord(start, 0);  
  
    // Allocate memory space on the device   
    int\* d\_a, \* d\_b, \* d\_c;  
    cudaMalloc((void\*\*)&d\_a, sizeof(int) \* m \* n);  
    cudaMalloc((void\*\*)&d\_b, sizeof(int) \* n \* k);  
    cudaMalloc((void\*\*)&d\_c, sizeof(int) \* m \* k);  
  
    // Copy matrix A and B from host to device memory  
    cudaMemcpy(d\_a, h\_a, sizeof(int) \* m \* n, cudaMemcpyHostToDevice);  
    cudaMemcpy(d\_b, h\_b, sizeof(int) \* n \* k, cudaMemcpyHostToDevice);  
  
    unsigned int grid\_rows = (m + BLOCK\_SIZE - 1) / BLOCK\_SIZE;  
    unsigned int grid\_cols = (k + BLOCK\_SIZE - 1) / BLOCK\_SIZE;  
    dim3 dimGrid(grid\_cols, grid\_rows);  
    dim3 dimBlock(BLOCK\_SIZE, BLOCK\_SIZE);  
  
    // Launch the appropriate kernel  
    if (m == n && n == k)   
    {  
        gpu\_square\_matrix\_mult << <dimGrid, dimBlock >> > (d\_a, d\_b, d\_c, n);  
    }  
    else   
    {  
        gpu\_matrix\_mult << <dimGrid, dimBlock >> > (d\_a, d\_b, d\_c, m, n, k);  
    }  
  
    // Transfer results from device to host   
    cudaMemcpy(h\_c, d\_c, sizeof(int) \* m \* k, cudaMemcpyDeviceToHost);  
    cudaDeviceSynchronize(); // Wait for GPU to finish  
    cudaEventRecord(stop, 0);  
    cudaEventSynchronize(stop);  
  
    // Compute time elapsed on GPU computing  
    cudaEventElapsedTime(&gpu\_elapsed\_time\_ms, start, stop);  
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on GPU: %f ms.\n", m, n, n, k, gpu\_elapsed\_time\_ms);  
  
    // Print result matrix C (GPU result)  
    print\_matrix(h\_c, m, k, "C (GPU Result)");  
  
    // Start measuring normal (sequential) execution time  
    double start\_time = omp\_get\_wtime();  
    cpu\_matrix\_mult(h\_a, h\_b, h\_cc, m, n, k);  
    double end\_time = omp\_get\_wtime();  
    normal\_elapsed\_time\_ms = (end\_time - start\_time) \* 1000.0; // Convert to milliseconds  
    printf("Time elapsed on normal matrix multiplication of %dx%d . %dx%d on CPU: %f ms.\n", m, n, n, k, normal\_elapsed\_time\_ms);  
  
    // Print result matrix C (CPU result)  
    print\_matrix(h\_cc, m, k, "C (CPU Result)");  
  
    // Start measuring CPU execution time using OpenMP  
    start\_time = omp\_get\_wtime();  
    openmp\_matrix\_mult(h\_a, h\_b, h\_cc, m, n, k);  
    end\_time = omp\_get\_wtime();  
    cpu\_elapsed\_time\_ms = (end\_time - start\_time) \* 1000.0; // Convert to milliseconds  
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on CPU (OpenMP): %f ms.\n", m, n, n, k, cpu\_elapsed\_time\_ms);  
  
    // Compare the results  
    int all\_ok = 1;  
    for (int i = 0; i < m; i++)   
    {  
        for (int j = 0; j < k; j++)   
        {  
            if (h\_cc[i \* k + j] != h\_c[i \* k + j])   
            {  
                all\_ok = 0;  
                printf("Mismatch at [%d][%d]: GPU=%d, CPU=%d\n", i, j, h\_c[i \* k + j], h\_cc[i \* k + j]);  
                break;  
            }  
        }  
        if (!all\_ok) break;  
    }  
  
    printf("Matrix multiplication %s\n", all\_ok ? "successful!" : "failed.");  
  
    // Free GPU memory  
    cudaFree(d\_a);  
    cudaFree(d\_b);  
    cudaFree(d\_c);  
  
    // Free CPU memory  
    cudaFreeHost(h\_a);  
    cudaFreeHost(h\_b);  
    cudaFreeHost(h\_c);  
    cudaFreeHost(h\_cc);  
  
    return 0;  
}

**Code Explanation:**

**1.** **Includes and Defines**:

* Includes standard libraries for input/output, memory management, and CUDA runtime.
* Defines BLOCK\_SIZE, which sets the dimensions of the blocks used in the CUDA kernel.

**2.** **CUDA Kernels**:

* **gpu\_matrix\_mult**: A kernel for general matrix multiplication. It computes the value for each element of the result matrix C based on matrices A and B.
* **gpu\_square\_matrix\_mult**: An optimized kernel for square matrices that uses shared memory for better performance. It loads sub-matrices (tiles) into shared memory, reducing global memory accesses.

**3.** **Matrix Multiplication Functions**:

* **openmp\_matrix\_mult**: Uses OpenMP for parallel matrix multiplication on the CPU. It employs nested loops with #pragma omp parallel for collapse(2) to parallelize both outer loops.
* **cpu\_matrix\_mult**: A normal sequential implementation for matrix multiplication on the CPU.

**4.** **Main Function**:

* Reads the dimensions of the matrices from the user.
* Allocates memory for matrices A, B, and results C on both host (CPU) and device (GPU).
* Initializes matrices A and B with random values.
* Measures and prints the time taken for matrix multiplication on the GPU and CPU.
* Compares the results from GPU and CPU computations for correctness.
* Frees the allocated memory.

**Key Features:**

1. **CUDA Implementation**:

* The code utilizes CUDA for GPU acceleration, enabling efficient handling of matrix multiplication tasks, especially for larger matrices.

2. **Optimized Memory Usage**:

* Uses shared memory in the gpu\_square\_matrix\_mult kernel to speed up memory access times by reducing global memory accesses.

3. **Parallelization**:

* The code showcases different levels of parallelization: GPU-based with CUDA and CPU-based using OpenMP.

4. **Performance Measurement**:

* It measures the execution time for GPU and CPU matrix multiplication, allowing for performance comparisons.

5. **Validation**:

* The code checks the results of the matrix multiplication between GPU and CPU to ensure correctness, printing mismatches if found.

6. **Dynamic Input**:

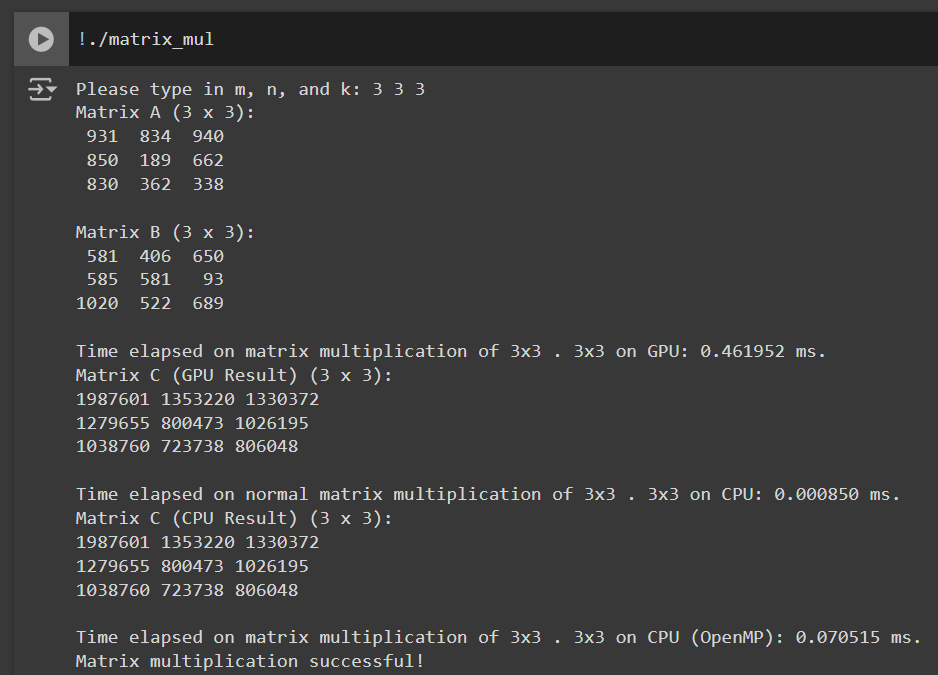
* The dimensions of the matrices are provided by the user at runtime, making the program flexible for different sizes of matrices.

7. **User-Friendly Output**:

* It prints the matrices before multiplication and shows the resulting matrices for both GPU and CPU, making it easy to verify results.

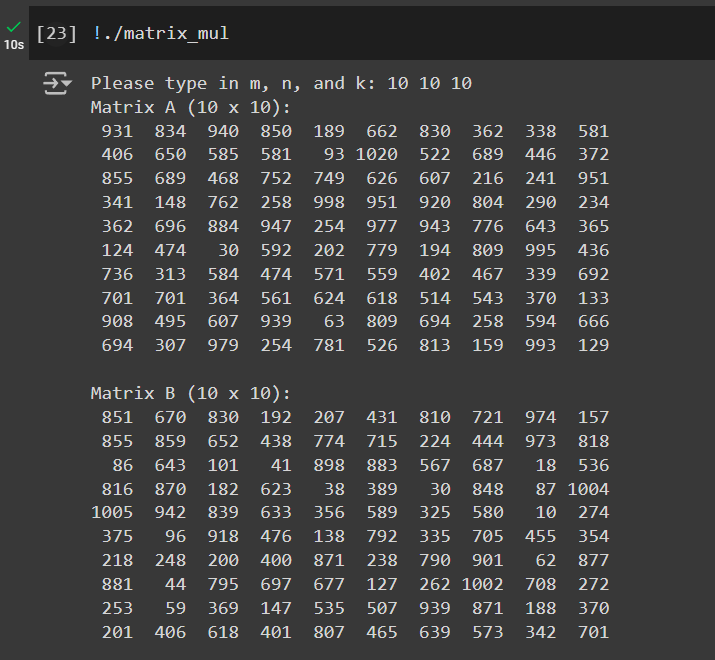
**Testing Phase:**

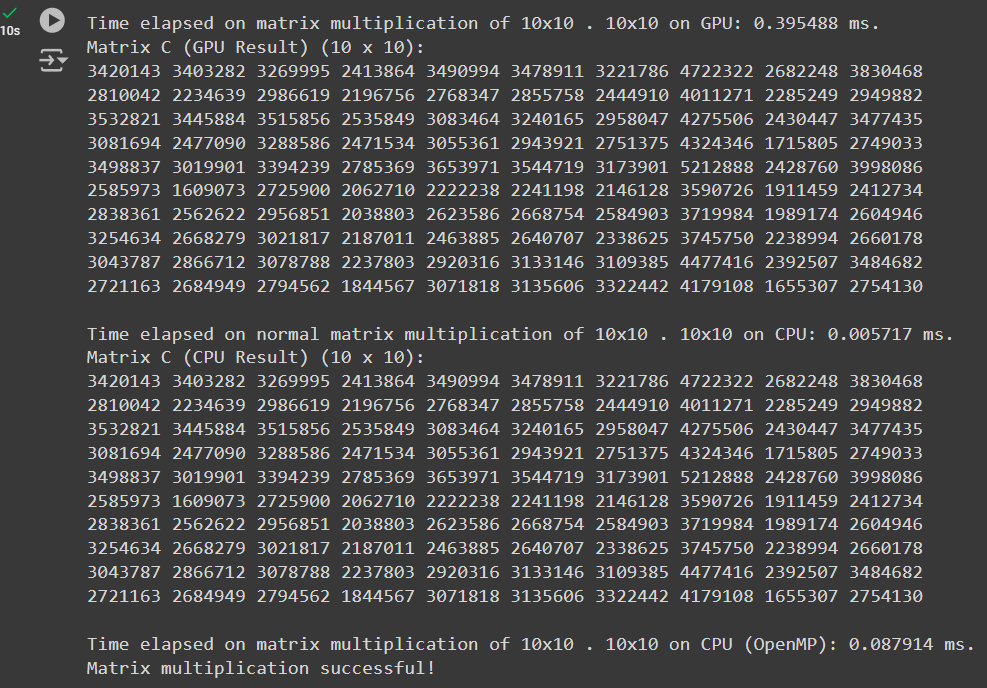
**1) multiplication for very small matrices(3x3):**

****

* Time elapsed on matrix multiplication of 3x3 . 3x3 on GPU**: 0.461952 ms.**
* Time elapsed on normal matrix multiplication of 3x3 . 3x3 on CPU: **0.000850 ms.**
* Time elapsed on matrix multiplication of 3x3 . 3x3 on CPU (OpenMP): **0.070515 ms.**

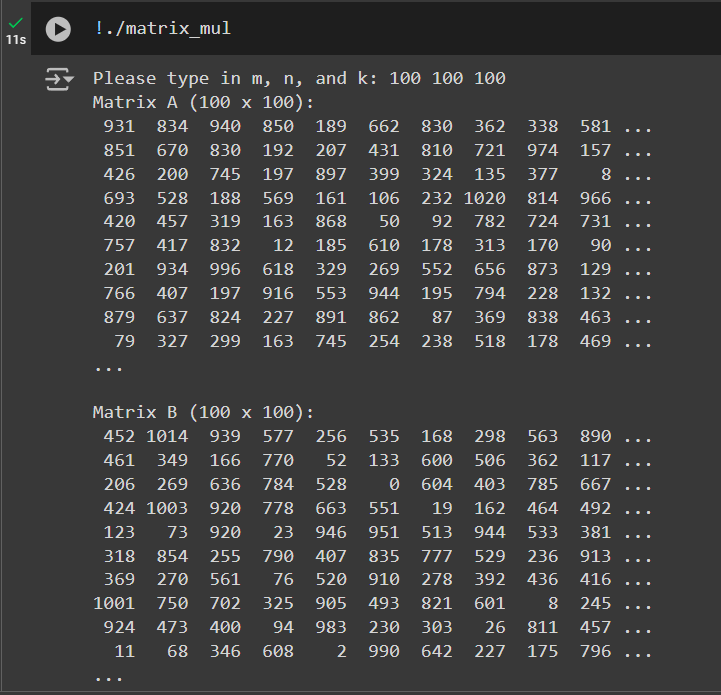
**2) Multiplication for small matrices(10x10):**

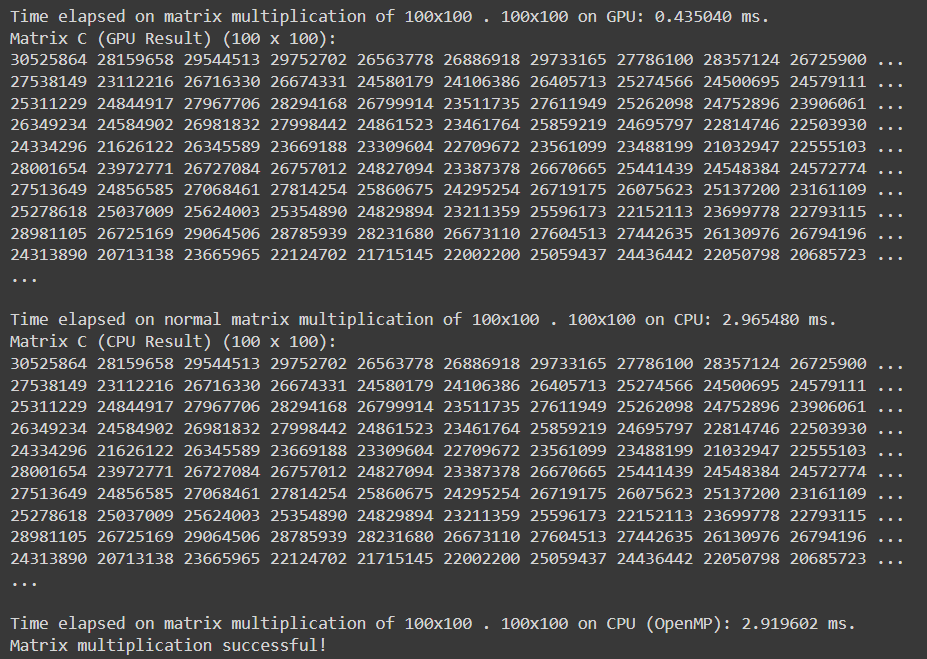
****

****

* Time elapsed on matrix multiplication of 10x10 . 10x10 on GPU: **0.395488 ms.**
* Time elapsed on normal matrix multiplication of 10x10 . 10x10 on CPU: **0.005717 ms.**
* Time elapsed on matrix multiplication of 10x10 . 10x10 on CPU (OpenMP): **0.087914 ms.**

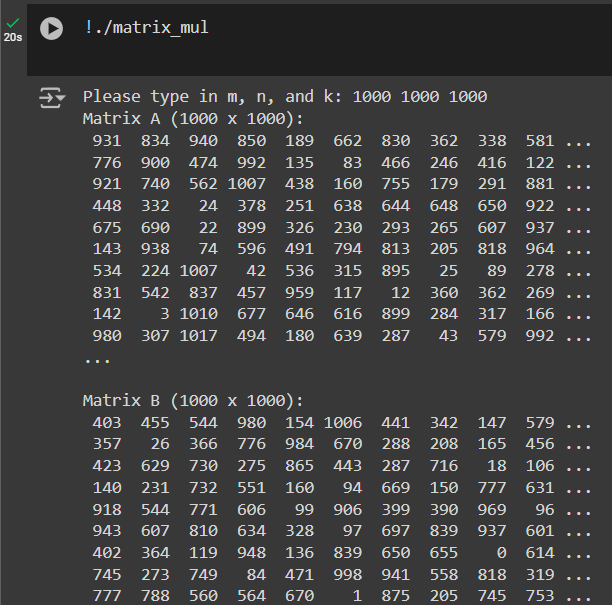
**3) Increase the size of matrices(100x100):**

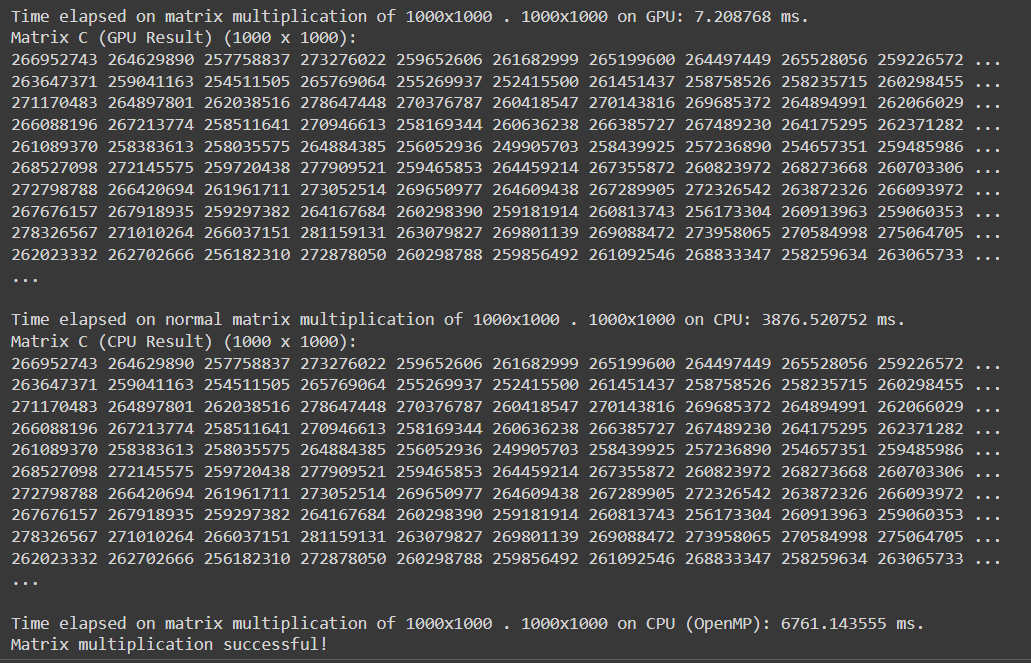
****

****

* Time elapsed on matrix multiplication of 100x100 . 100x100 on GPU: **0.435040 ms.**
* Time elapsed on normal matrix multiplication of 100x100 . 100x100 on CPU: **2.965480 ms.**
* Time elapsed on matrix multiplication of 100x100 . 100x100 on CPU (OpenMP): **2.919602 ms.**

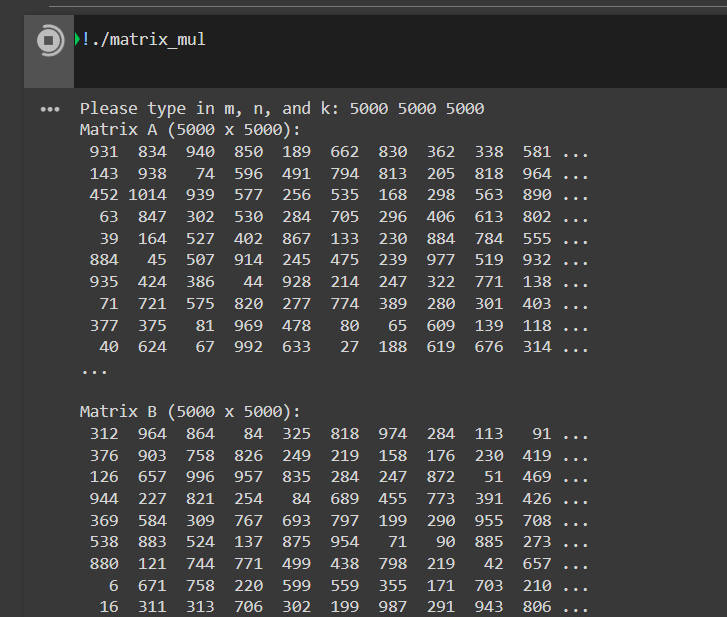
**4) Matrix multiplication for large matrices(1000x1000):**

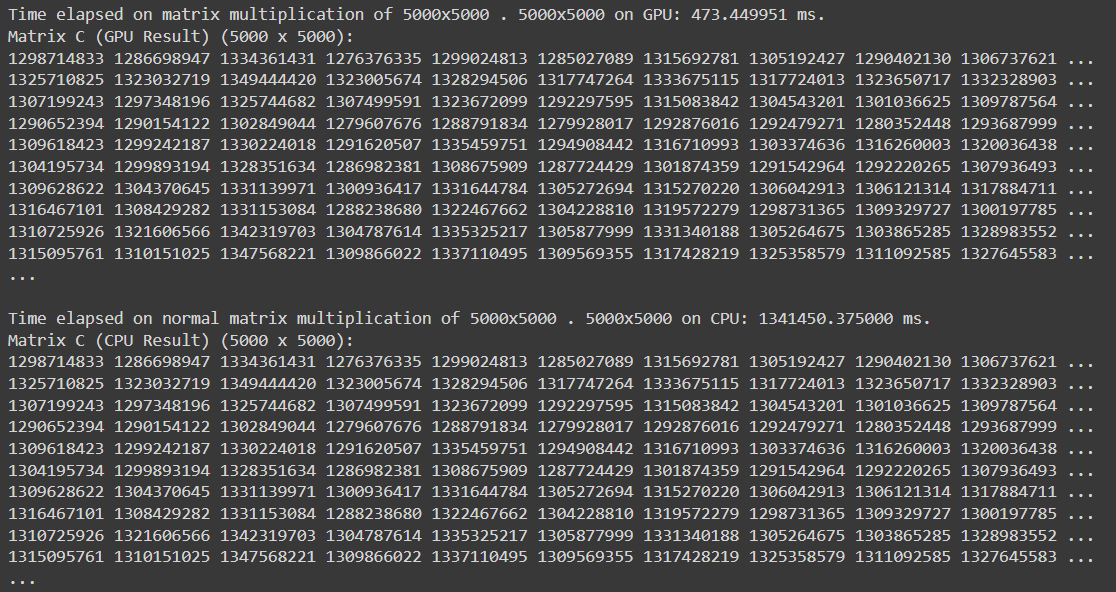
****

****

* Time elapsed on matrix multiplication of 1000x1000 . 1000x1000 on GPU: **7.208768 ms.**
* Time elapsed on normal matrix multiplication of 1000x1000 . 1000x1000 on CPU: **3876.520752 ms.**
* Time elapsed on matrix multiplication of 1000x1000 . 1000x1000 on CPU (OpenMP): **6761.143555 ms.**

**5) Matrix Multiplication for very large matrices(5000x5000):**

****

****

* Time elapsed on matrix multiplication of 5000x5000 . 5000x5000 on GPU: **473.449951 ms.**
* Time elapsed on normal matrix multiplication of 5000x5000 . 5000x5000 on CPU: **1341450.375000 ms.**

**Conclusion:** Successfully computed Matrix Multiplication using CUDA and we can clearly observe that for large matrices, the matrix multiplication on GPU is much faster than CPU.