# Assignment 3

Aryaman Srivastava 2210110206

## Matrix Multiplication Using Multithreading with CUDA

### 1. Introduction

Matrix multiplication is a fundamental operation in scientific computing, often requiring significant computational resources. This report evaluates three different multithreading approaches to matrix multiplication and compares their performance based on execution time. The primary goal is to determine the most efficient method for parallelizing matrix multiplication using threads.

### 2. Problem Description

Given two matrices:

- **Matrix A** of size $m \times n$
- **Matrix B** of size $n \times p$

The resulting matrix **C** is of size $m \times p$, where each element **c[i][j]** is computed as:

$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot b[k][j]$$

### 3. Approaches

Three different multithreading approaches were implemented:

#### Approach 1: Static Load Balancing (Row-wise)

In this approach, each thread is assigned a fixed set of contiguous rows of **Matrix A**. Each thread computes all elements in its assigned rows of **Matrix C** independently.

**Advantages:**

- Easy to implement and requires minimal synchronization.
- Efficient when workload distribution among rows is even.

**Disadvantages:**

- May lead to workload imbalance if some rows require more computation than others.
- Threads assigned to computationally heavier rows may finish later, leading to idle threads

#### Approach 2: Static Load Balancing (Cyclic)

This method assigns rows to threads in a round-robin manner. Each thread computes non-contiguous rows, ensuring a more even distribution of computational load.

**Advantages:**

- More balanced workload distribution compared to row-wise allocation.
- Reduces the chances of idle threads as computation is spread more evenly.

**Disadvantages:**

- Increased complexity in indexing rows due to non-contiguous assignments.
- Can lead to cache inefficiencies since threads access non-consecutive memory locations.

### Approach 3: Dynamic Load Balancing

Instead of pre-allocating rows, a shared counter is used to dynamically assign rows to threads as they become available. Each thread fetches the next available row until all rows are processed.

**Advantages:**

- Maximizes thread utilization, ensuring minimal idle time.
- Adapts well to uneven computation loads among rows.

**Disadvantages:**

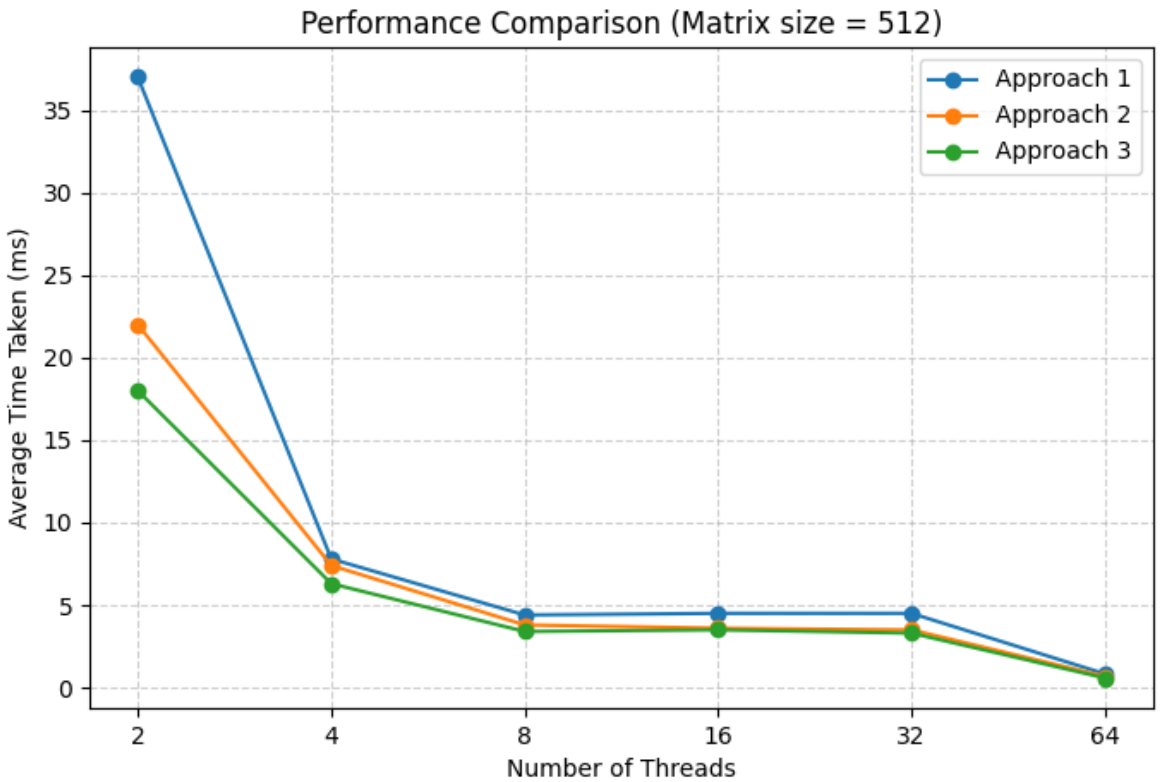
- Requires synchronization mechanisms like atomic operations or mutex locks to prevent race conditions.
- Can introduce overhead due to thread coordination.

## 4. Performance Evaluation

To analyze performance, we conducted two sets of tests:

#### 4.1 Varying Number of Threads

We evaluated execution time by using **2, 4, 8, 16, 32, and 64 threads** keeping the matrix size fixed at **512 × 512**.

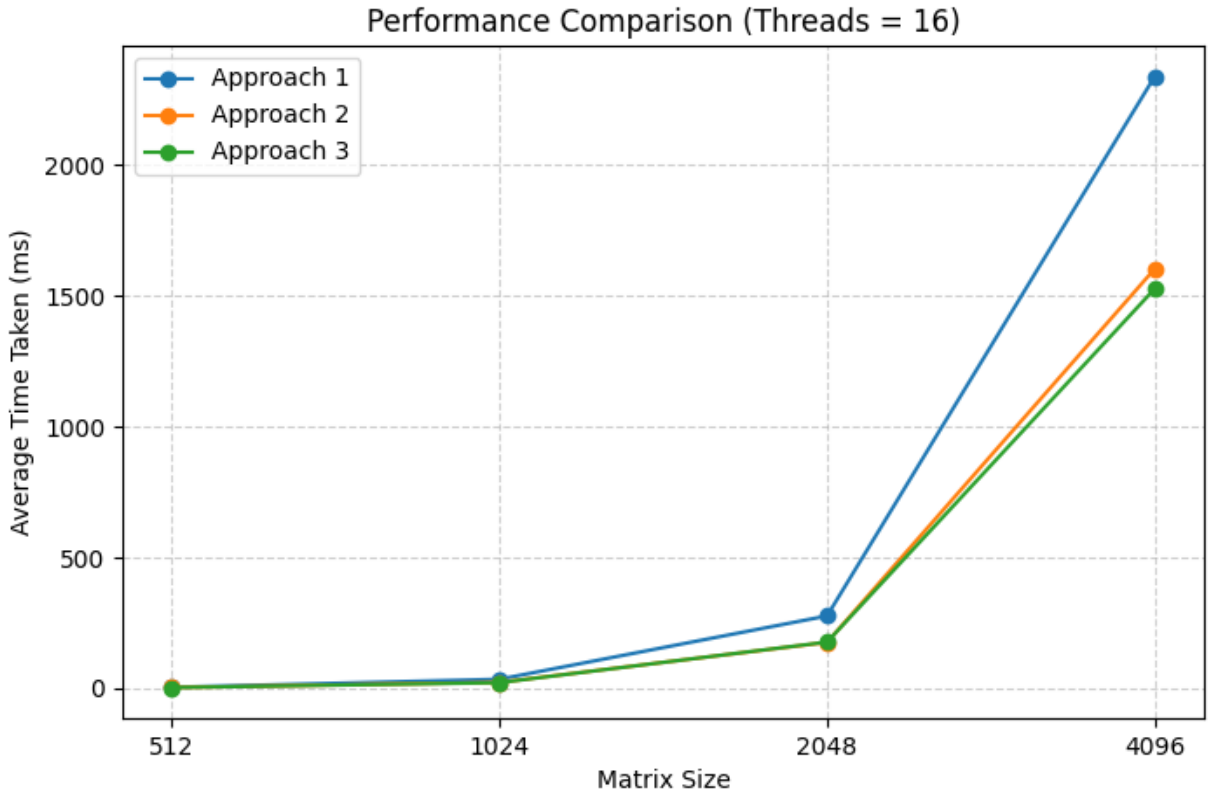


**Observations:**

- Increasing thread count reduces execution time
- Approach 3 consistently performs better as it dynamically distributes work

### 4.2 Varying Matrix Size

For a fixed **16-thread** configuration, we varied matrix sizes (**512 × 512**, **1024 × 1024**, **2048 × 2048**, **4096 × 4096**).




**Observations:**

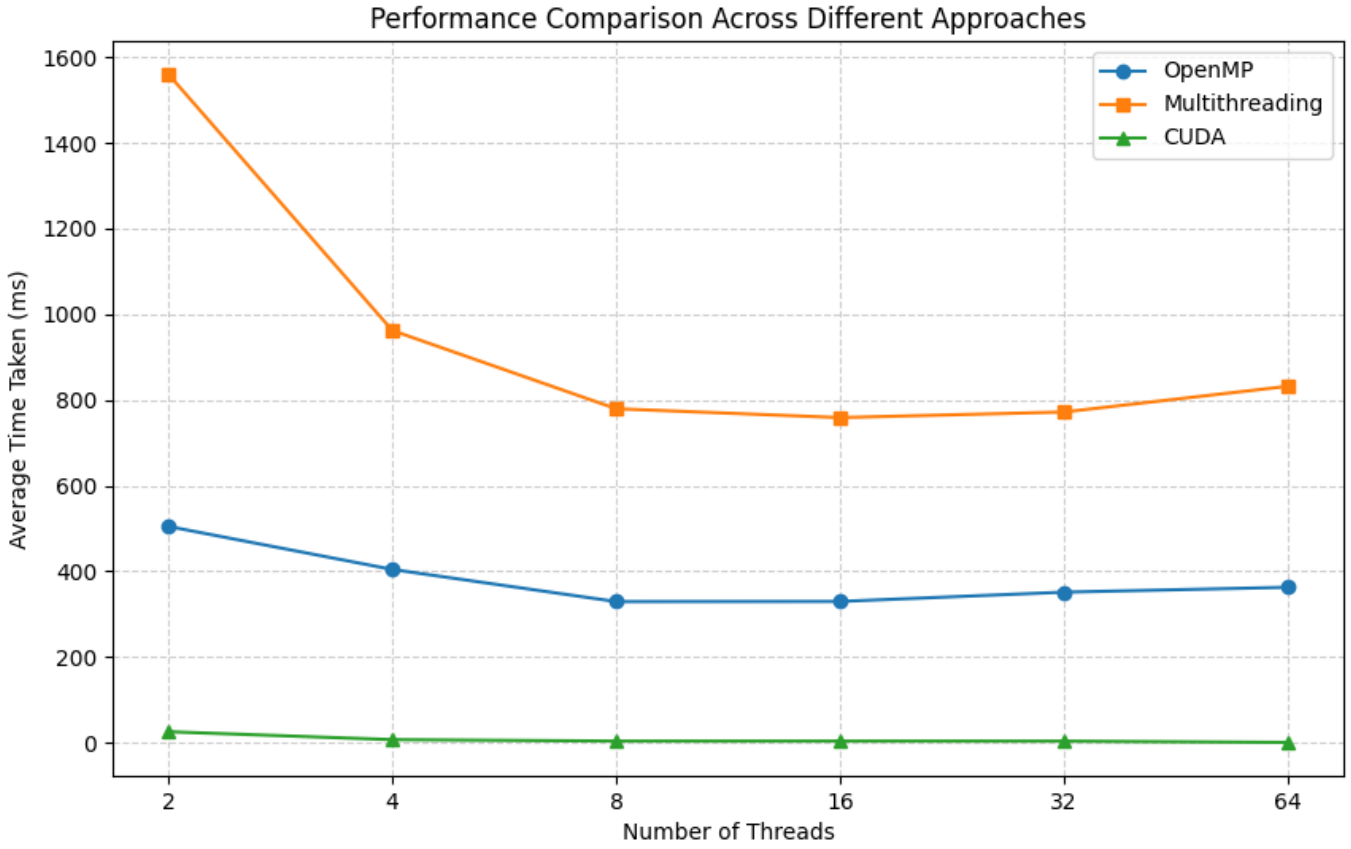
- Execution time increases exponentially with matrix size.
- Approach 3 scales better than the other two due to dynamic task allocation.
- Approach 1 exhibits poor scalability as static row allocation leads to uneven workload distribution.

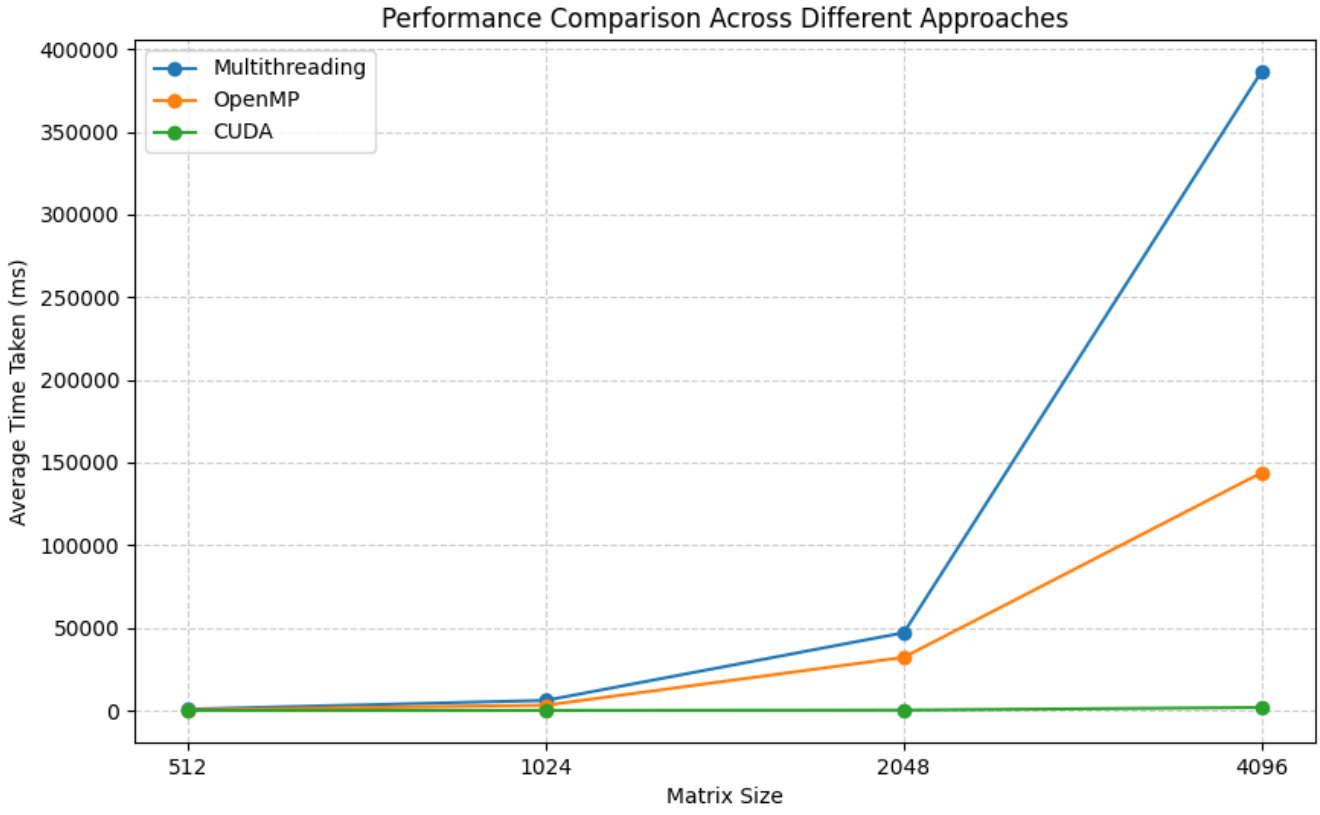
## 5. Conclusion

- **Approach 3 (Dynamic Load Balancing)** is the most efficient, especially for larger matrix sizes and higher thread counts.
- **Approach 1 (Static Row-wise)** performs the worst due to poor workload distribution.
- **Approach 2 (Static Cyclic)** improves load balancing but cannot fully mitigate idle time.

## 6. Comparisons

By taking the average of all three approaches with same matrix size and varying the number of threads, we get the following results:




By these graphs, we can conclude that using CUDA for multithreading is the most beneficial as it takes significantly lesser time, followed by OpenMP, and multithreading without these performs the worst.