Assignment 3

Aryaman Srivastava 2210110206

Prime Number Counting Using Multithreading with CUDA

1. Introduction

Counting prime numbers within a large range is a computationally intensive task that can benefit significantly from parallel processing. This report evaluates three different multithreading approaches to prime number counting and compares their performance based on execution time. The primary goal is to determine the most efficient method for parallelizing the prime counting operation using threads.

2. Problem Description

Given an input number n, count all prime numbers between 1 and 10^n, where a prime number is defined as:

- A number greater than 1
- Only divisible by 1 and itself
- Can be verified by checking divisibility up to its square root

For a number x, the primality check can be expressed as:

- x is prime if no number in range [2, √x] divides x perfectly as:

3. Approaches

Three different multithreading approaches were implemented:

Approach 1: Static Load Balancing (Equal Range Division)

In this approach, the range [1, 10^n] is divided equally among threads, with each thread processing a continuous block of numbers.

Advantages:

- Simple implementation with minimal synchronization needs
- Good cache locality due to continuous memory access
- Predictable workload distribution

Disadvantages:

- May lead to workload imbalance due to varying computational complexity of different ranges
- Higher ranges typically contain fewer primes but require more divisibility checks

Approach 2: Static Load Balancing (Cyclic Distribution)

This method assigns numbers to threads in a round-robin fashion, where each thread processes numbers with steps of thread count.

Advantages:

- More balanced distribution of computational load
- Better handling of uneven prime distribution across ranges

Disadvantages:

- Poor cache utilization due to non-continuous memory access
- Higher complexity in number assignment and tracking
- Increased memory access patterns may lead to cache misses

Approach 3: Dynamic Load Balancing

Uses a shared counter to dynamically assign numbers to threads as they become available for processing

Advantages:

- Optimal thread utilization with minimal idle time
- Naturally adapts to varying computational complexity
- Better handling of workload imbalances

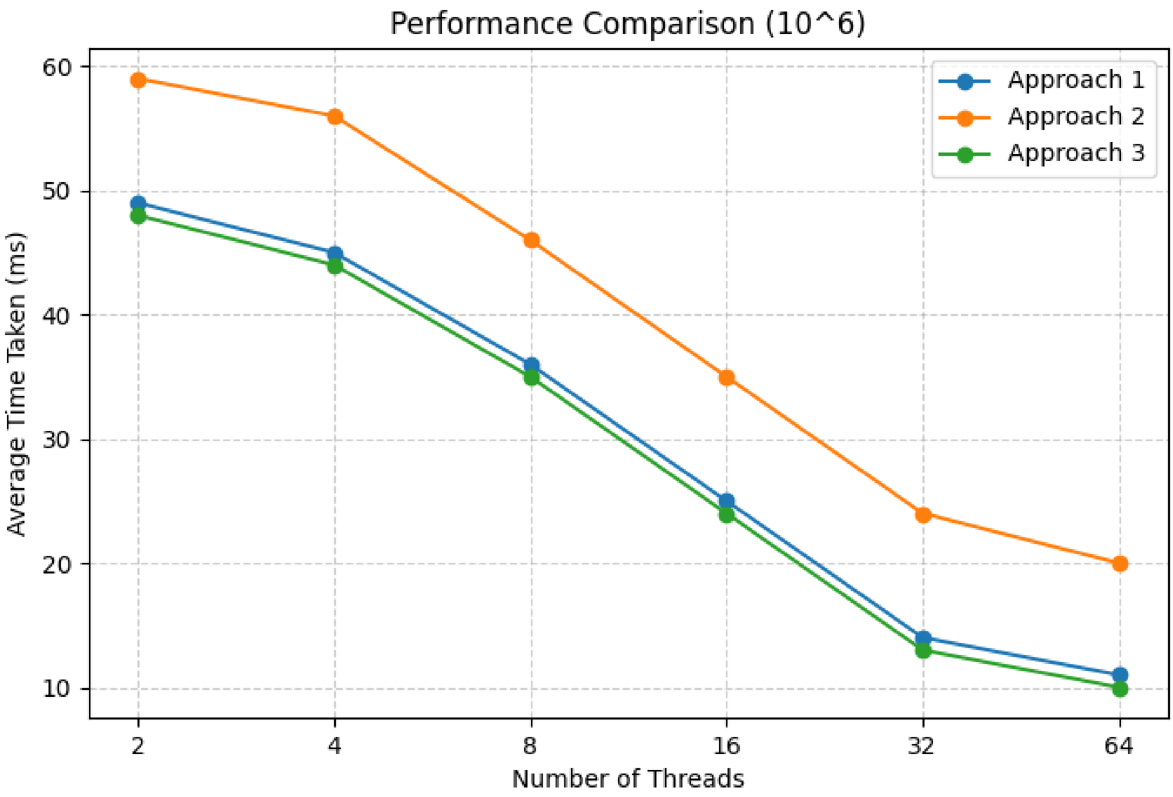
Disadvantages:

- Requires synchronization for counter access
- Slight overhead from dynamic work distribution
- Potential contention at the shared counter

4. Performance Evaluation

4.1 Varying Number of Threads

We evaluated execution time using **2, 4, 8, 16, 32, and 64 threads** with input size fixed at **10^6**.

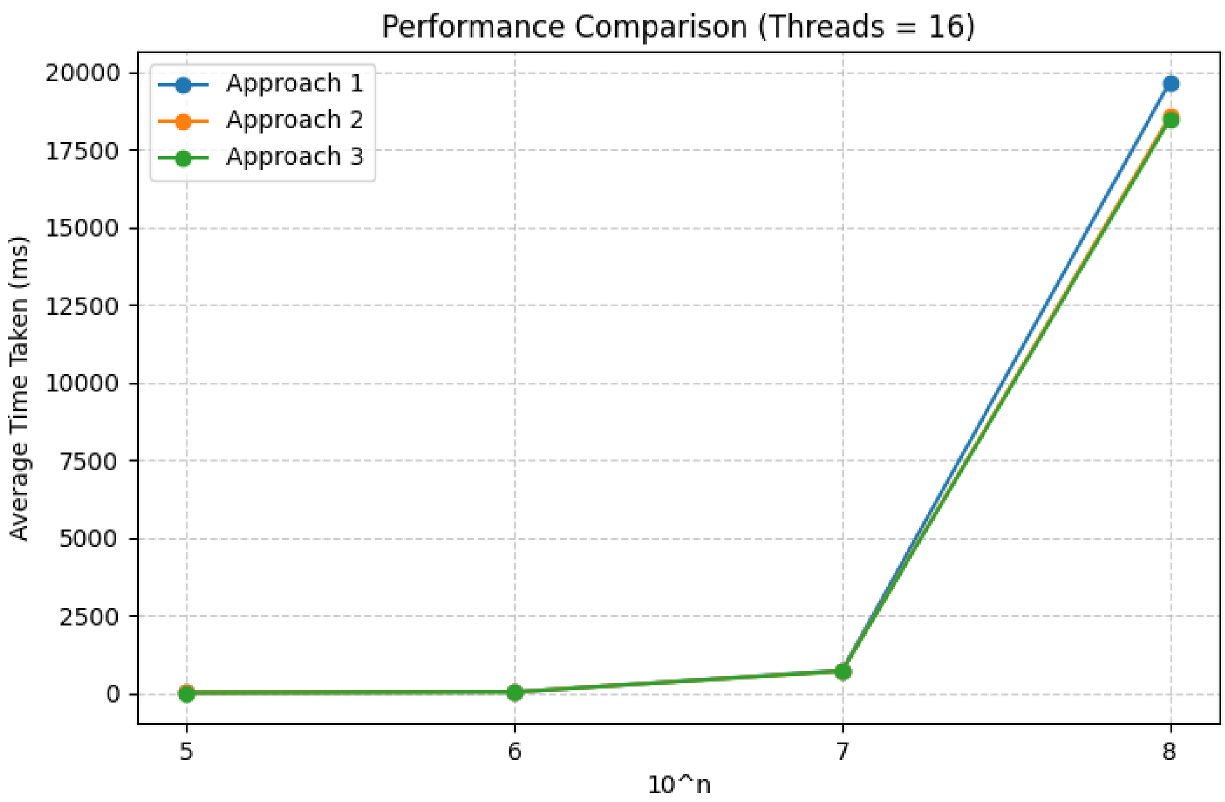


Observation:

- Approach 3 maintains the best performance across thread counts

4.2 Varying Matrix Size

For a fixed **16-thread** configuration, we tested with input sizes from **10^5 to 10^8**.



Observations:

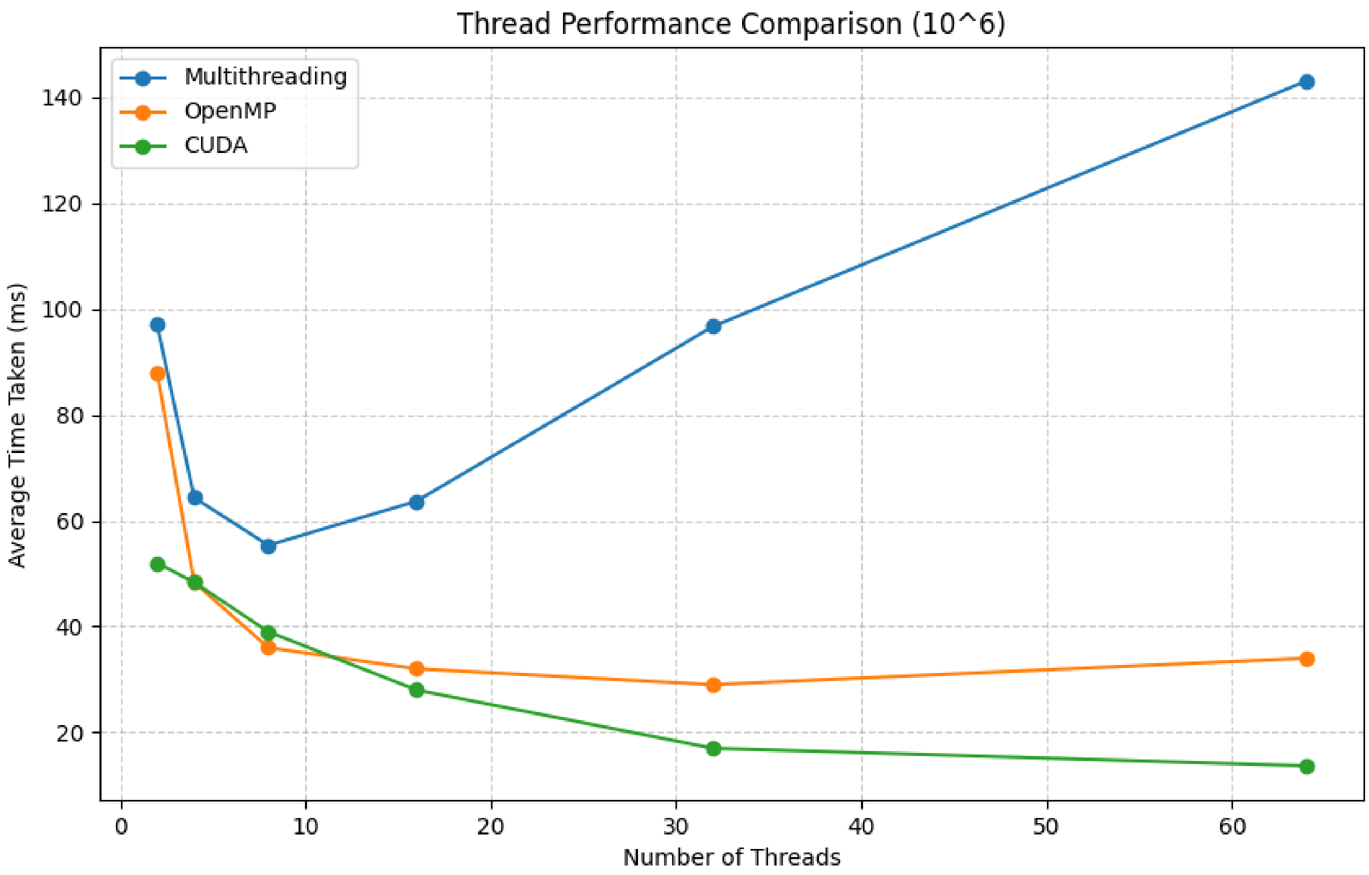
- All approaches show similar performance up to 10^7
- Exponential increase in execution time beyond 10^7
- Approach 3 maintains slight advantage at larger input sizes

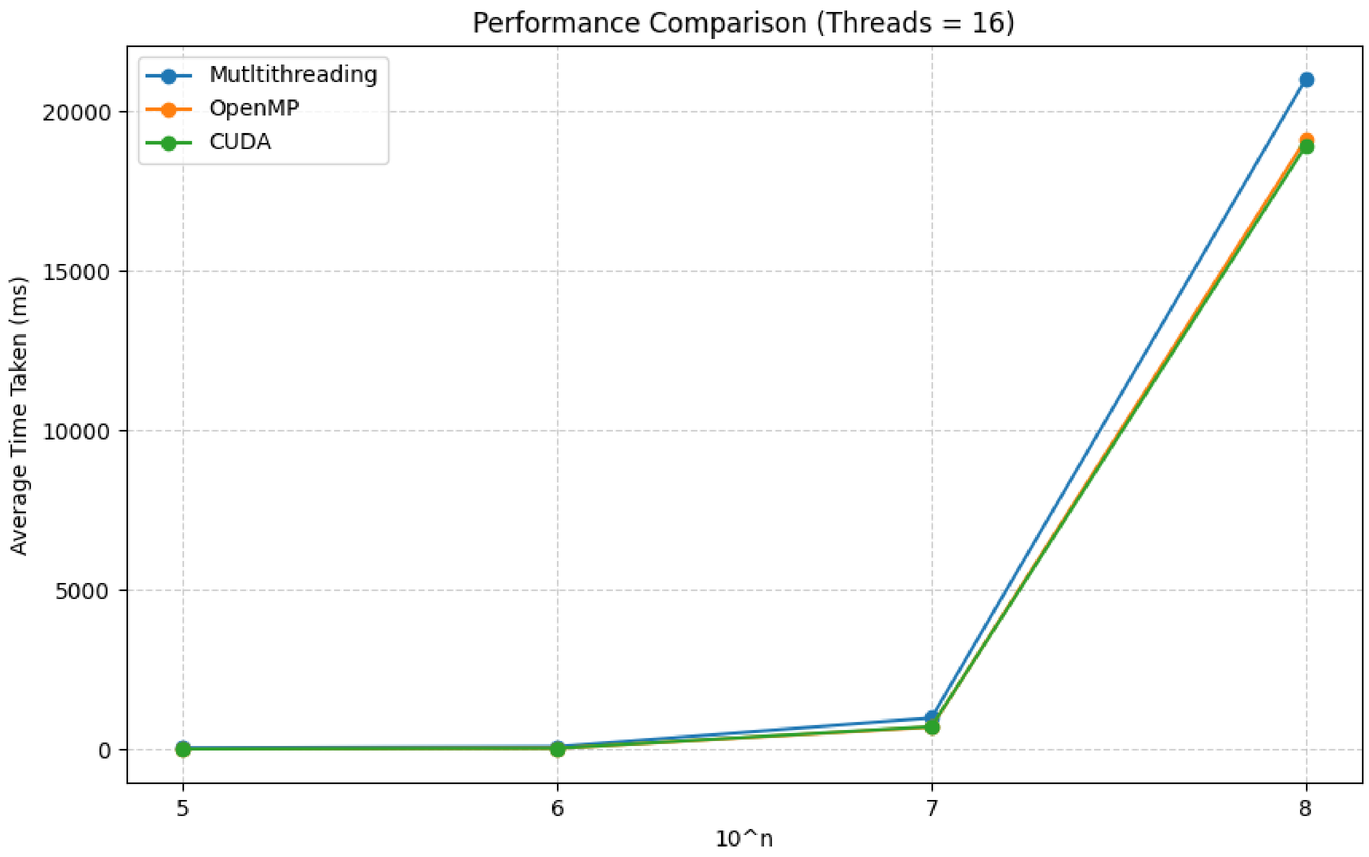
5. Conclusion

- **Approach 3 (Dynamic Load Balancing)** provides the best overall performance, particularly for high thread counts and input sizes
- **Approach 1 (Equal Range Division)** offers consistent but slightly slower performance
- **Approach 2 (Cyclic Distribution)** shows poor scaling
- For very large inputs (>10^7), algorithmic optimization might be more beneficial than threading optimization

6. Comparisons

By taking the averages of all three approaches with same upper limit 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.