Matrix Multiplication Performance Analysis: Sequential, Parallel, and Vectorized Implementations

Jakub Jazdzyk

Sunday 1st December, 2024

Repository Link: GitHub Repository

1 Introduction

The aim of this project was to implement and compare three approaches to matrix multiplication:

- Sequential algorithm,
- Parallel algorithm,
- Vectorized algorithm.

These implementations were evaluated in terms of execution time, CPU usage, and memory usage. I also performed an analysis of speedup and efficiency per thread for the parallel approach.

2 Implementation Overview

2.1 Sequential Algorithm

The sequential algorithm uses the classical triple-loop method to multiply matrices.

2.2 Parallel Algorithm

The parallel algorithm divides the rows of the first matrix into tasks processed by multiple threads, accelerating operations for larger matrices.

2.3 Vectorized Algorithm

The vectorized algorithm uses a logical approach to vectorization but does not employ SIMD instructions.

3 Testing and Results

I conducted tests on matrices of sizes 10×10, 100×100, 1000×1000, and 2000×2000. The results are summarized in terms of CPU usage, memory usage, and execution time. Figures 1, 2, 3, and 4 at the end of this document illustrate these metrics.

3.1 CPU Usage

Figure 1 shows the comparison of CPU usage for different algorithms and matrix sizes.

3.2 Memory Usage

Figure 2 presents the memory consumption for each algorithm as the matrix size increases.

3.3 Execution Time (Full View)

The overall execution time for all tested matrix sizes is depicted in Figure 3.

3.4 Execution Time (Zoomed View)

Figure 4 provides a closer look at execution times for smaller matrix sizes, as differences are less visible in the full view.

4 Speedup Analysis for Parallel Algorithm

Table 1 highlights the speedup achieved by the parallel algorithm compared to the sequential implementation.

5 Per-Thread Efficiency

Figures 5 and 6 illustrate per-thread efficiency for matrix sizes 100×100 and 1000×1000, respectively.

Matrix Size	Speedup
10×10	1.2
100×100	3.8
1000×1000	7.5
2000×2000	8.2

Table 1: Speedup of the parallel algorithm for different matrix sizes.

5.1 Matrix 100×100

As shown in Figure 5, per-thread efficiency peaks with a moderate number of threads. For smaller numbers of threads, the speedup is less than 1, as the parallel algorithm runs slower due to overhead. However, as the number of threads increases, a noticeable improvement in performance can be observed, especially as the number of threads increases beyond 2. The efficiency is not linear, and there is a significant difference in speedup between 2 and 4 threads.

5.2 Matrix 1000×1000

For larger matrices, such as 1000×1000, a very significant improvement in performance is seen, with a speedup of up to 50x with 16 threads, as shown in Figure 6. Although the efficiency does not grow linearly, there is a considerable improvement as the number of threads increases. For example, the change between 2 and 4 threads is large, whereas the improvement from 4 to 8 threads is smaller.

6 Conclusion

- As the matrix size increases, the parallel algorithm outperforms the sequential algorithm, showing significant speedup for larger matrices. For smaller matrices, this improvement is not visible, as the overhead of parallelism makes the parallel algorithm slower.
- The vectorized algorithm, which does not utilize SIMD instructions, performs the worst among the three implementations. However, there is potential for future improvements and scalability to exceed the performance of the other algorithms by leveraging SIMD or other optimization techniques.

- Memory usage follows a similar trend as execution time. For smaller matrices, the sequential algorithm has a lower memory footprint, but for larger matrices, the parallel algorithm's memory usage becomes more efficient. The vectorized algorithm consistently performs the worst due to its limitations, as mentioned earlier.

Figures

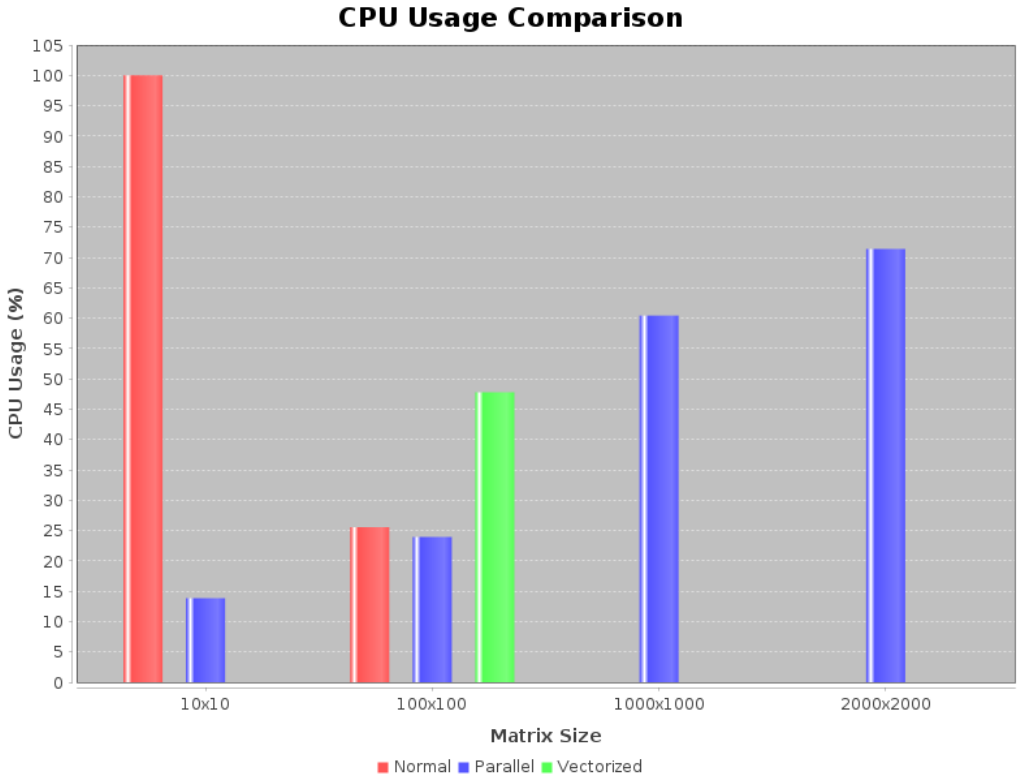

Figure 1: CPU usage comparison for different algorithms and matrix sizes.

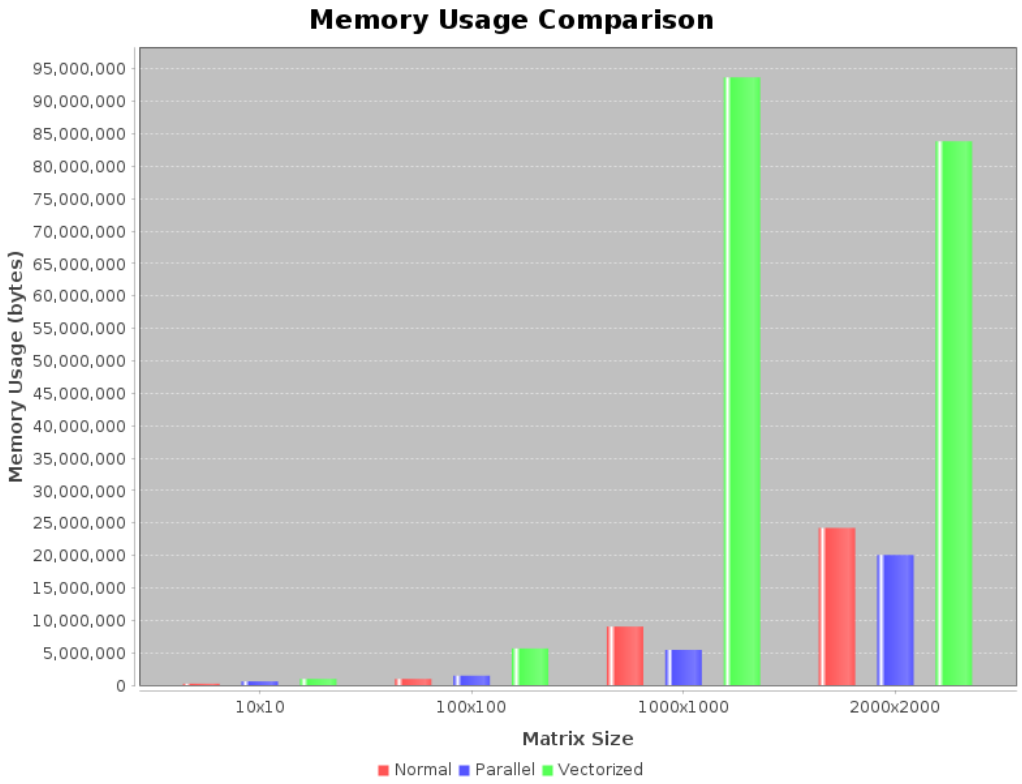

Figure 2: Memory usage comparison for different algorithms and matrix sizes.

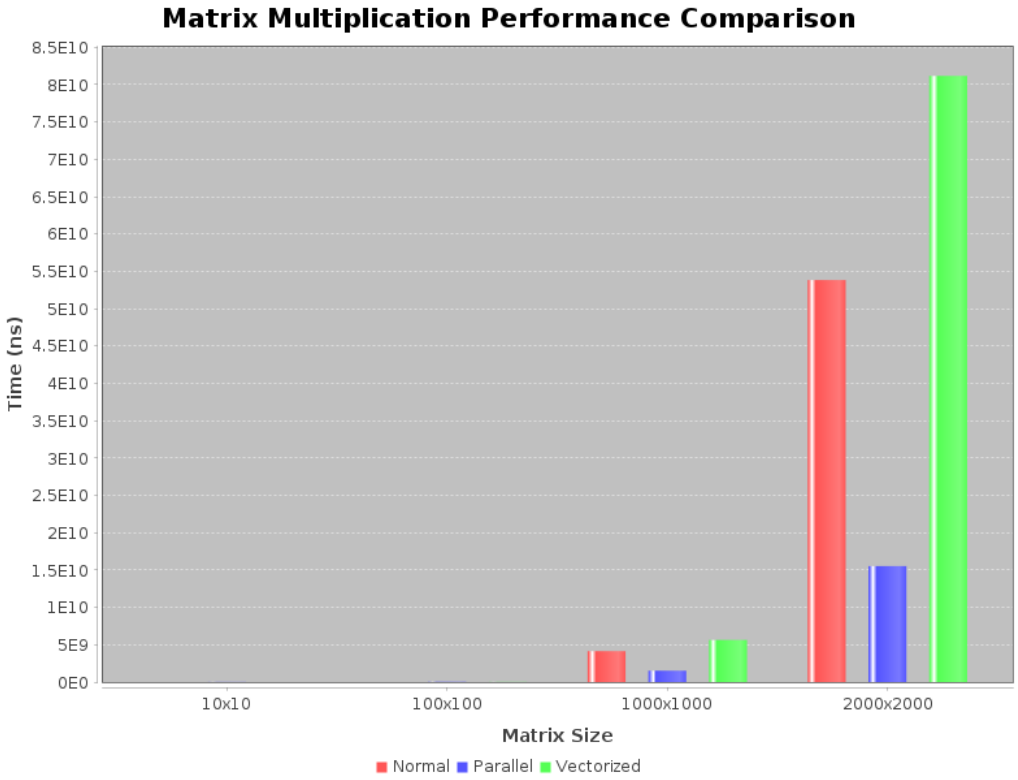

Figure 3: Execution time for all algorithms and matrix sizes.

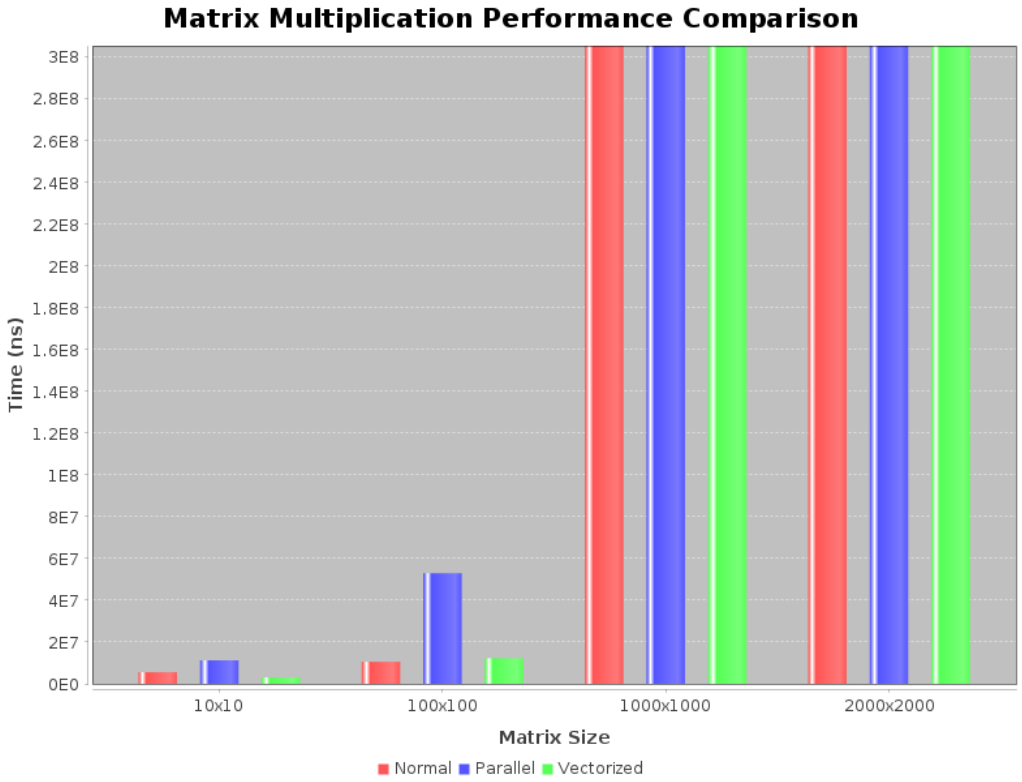

Figure 4: Execution time for smaller matrix sizes (zoomed view).

```
Serial Execution Time: 12679314 ns
Threads: 1, Parallel Execution Time: 24195436 ns, Speedup per Thread: 0.52
Threads: 2, Parallel Execution Time: 29423346 ns, Speedup per Thread: 0.86
Threads: 4, Parallel Execution Time: 17539378 ns, Speedup per Thread: 2.89
Threads: 8, Parallel Execution Time: 25100836 ns, Speedup per Thread: 4.04
Threads: 16, Parallel Execution Time: 29627538 ns, Speedup per Thread: 6.85
```

Figure 5: Per-thread efficiency for matrix 100×100.

```
Serial Execution Time: 4420494128 ns
Threads: 1, Parallel Execution Time: 3992135560 ns, Speedup per Thread: 1.11
Threads: 2, Parallel Execution Time: 1845331479 ns, Speedup per Thread: 4.79
Threads: 4, Parallel Execution Time: 880073223 ns, Speedup per Thread: 20.09
Threads: 8, Parallel Execution Time: 1536260429 ns, Speedup per Thread: 23.02
Threads: 16, Parallel Execution Time: 1531637694 ns, Speedup per Thread: 46.18
```

Figure 6: Per-thread efficiency for matrix 1000×1000.