A Comparative Performance Analysis of different Matrix Multiplication algorithms using paralellization in Java

Daniel Talavera Hernández

November 2024

Source code in this GitHub repository.

1 Abstract

Matrix multiplication is a fundamental operation in numerous scientific and engineering applications, often serving as a benchmark for computational performance. However, its computational complexity can be prohibitive, particularly when dealing with large matrices. This study investigates parallel and vectorized approaches to enhance the efficiency of matrix multiplication. The experimentation involves implementing three versions: a basic sequential algorithm, a parallelized version using Java's concurrency utilities, and a vectorized approach leveraging parallel streams. Performance was analyzed using large matrices to measure speedup, parallel efficiency, and resource utilization. Results demonstrate that both parallel and vectorized implementations significantly reduce computation time compared to the basic algorithm, with parallelization showing near-linear speedup with increased core usage. The conclusions emphasize the importance of parallel processing in improving computational performance, highlighting trade-offs in resource consumption and efficiency. These findings provide insights into the potential of parallel and vectorized computing techniques for optimizing matrix operations in high-performance applications.

2 Introduction

Matrix multiplication plays a crucial role in various domains such as scientific computing, computer graphics, and machine learning. Due to its computational intensity, efficient implementations are vital for performance-critical applications. Traditionally, sequential algorithms for matrix multiplication exhibit O(n3) time complexity, which becomes impractical for large matrices. This has led to extensive research into parallel and vectorized methods to accelerate computation.

Parallel computing utilizes multiple processing cores to perform simultaneous calculations, reducing overall execution time. Techniques such as multi-threading and parallelization libraries (e.g., OpenMP) have shown promising results. Additionally, vectorization leverages Single Instruction, Multiple Data (SIMD) operations to perform simultaneous arithmetic on data vectors, further enhancing performance. Previous research demonstrates significant improvements with these methods, but comparisons between parallel and vectorized approaches remain limited.

This paper explores the implementation and benchmarking of parallel and vectorized matrix multiplication algorithms. The primary contribution lies in a comparative performance analysis, highlighting the efficiency gains and trade-offs of each approach. This study aims to provide valuable insights for optimizing matrix operations in resource-intensive applications.

3 Problem Statement

Efficient matrix multiplication is essential for many computational tasks, yet the standard sequential algorithm suffers from high time complexity, particularly for large matrices. This performance bottleneck necessitates the exploration of parallel and vectorized techniques to enhance computational efficiency. The challenge lies in effectively implementing these methods to maximize speedup and resource utilization.

This study addresses the following questions: How does parallel matrix multiplication compare to traditional methods in terms of speedup and efficiency? What performance gains can be achieved through vectorization? By implementing and comparing both approaches, this research seeks to determine the most effective strategy for optimizing matrix multiplication, providing a detailed analysis of computational performance and resource usage.

4 Methodology

To address the challenge of optimizing matrix multiplication through parallel and vectorized approaches, we implemented three distinct algorithms: a basic sequential version, a parallel version using Java's concurrency utilities, and a vectorized version utilizing parallel streams. This section outlines the experimental setup, performance metrics, and evaluation methods employed to compare these implementations.

4.1 Experimental Setup

All tests were conducted on a multi-core machine with an Intel(R) Core(TM) i7-10750H processor, featuring 6 cores and 16 GB of RAM, running Windows 10 Enterprise. The Java Microbenchmark Harness (JMH) framework was used to ensure accurate and reliable performance measurements. The JMH configuration included five warm-up iterations followed by ten measurement iterations, each lasting one second, with a single fork. Matrix sizes ranging from 100×100 to 2000×2000 were tested to evaluate performance across varying computational loads.

4.2 Performance Metrics

We assessed each implementation based on the following key performance indicators:
Execution Time: Measured in milliseconds, this metric provides a direct comparison of computational efficiency.
Speedup: Calculated as the ratio of execution times between the basic and optimized versions, indicating the performance gain achieved through parallelization and vectorization.
Resource Utilization: Monitored using Java's management interfaces to capture CPU usage and memory consumption. CPU load before and after execution was recorded to assess computational efficiency.
Parallel Efficiency: Determined by evaluating speedup relative to the number of threads, highlighting how effectively the parallel algorithm scales with additional cores.

4.3 Experimental Procedure

Initialization: Random square matrices A and B of varying sizes were generated, ensuring consistent input data for all experiments.
Benchmark Execution: Each algorithm was executed independently within the JMH framework. System garbage collection was explicitly invoked before each run to minimize memory-related variability.
Data Collection: Execution time, memory usage, and CPU load were recorded for each benchmark iteration. This data was aggregated and averaged to mitigate transient fluctuations.

Result Analysis: Post-execution, performance metrics were analyzed to compare the implementations. The results were evaluated for consistency, and any anomalies were investigated to ensure accuracy.
This methodology ensures a rigorous and reproducible evaluation of parallel and vectorized matrix multiplication techniques, providing a comprehensive analysis of their relative performance and resource efficiency.

5 Experiments

This section presents the experimental evaluation of three matrix multiplication algorithms: Basic, Parallel, and Vectorized. The experiments were conducted using matrices of varying sizes (100×100, 250×250, 500×500, 1000×1000, and 2000×2000) to assess performance across different computational loads. The key performance metrics analyzed include execution time, CPU utilization, memory usage, speedup, number of threads and efficiency (speedup relative to the number of threads). The results are summarized in Tables 1–4 and discussed in detail below.

5.1 Experiment Setup

Environment: Multi-core system with 12 available threads.
Metrics Measured: Execution time (in milliseconds), CPU usage (before and after execution), memory usage (in megabytes), speedup (a ratio between basic algorithm and parallel or vectorized algorithm execution time), efficiency().
Tools: Java Microbenchmark Harness (JMH), ensuring consistent and accurate measurements.

5.2 Basic Matrix Multiplication Results

Table 1 shows the execution times, CPU usage, and memory consumption for various matrix sizes. As expected, the execution time grows exponentially with the increase in matrix size, reflecting the $O(n^3)$ complexity of the algorithm.

Matrix Size	Exec Time(ms)	CPU (%) Before	CPU (%) After	Memory Used(MB)
100×100	1.12	3.46	3.46	0.00
250×250	21.96	3.72	3.72	0.00
500×500	186.49	4.77	6.81	0.00
1000×1000	5485.10	8.18	8.21	0.05
2000×2000	83726.51	8.32	8.33	1.34

Table 1: Basic Matrix Multiplication Results

Discussion:
The Basic algorithm shows an exponential increase in execution time as the matrix size grows. For the 2000×2000 matrix, it took approximately 83.7 seconds, demonstrating the O(n3) time complexity. CPU usage remained low, reflecting the single-threaded nature of this approach.

5.3 Parallel Matrix Multiplication Results

Table 2 shows the execution times, CPU usage, and memory consumption for various matrix sizes for the Parallel Matrix Multiplication algorithm.

Matrix Size	Exec Time(ms)	CPU (%) Before	CPU (%) After	Memory Used(MB)
100×100	2.06	3.20	3.20	0.86
250×250	4.77	3.57	3.57	0.89
500×500	29.08	15.81	15.81	0.88
1000×1000	1585.87	88.52	87.69	1.46
2000×2000	17640.91	96.77	97.32	0.61

Table 2: Parallel Matrix Multiplication Results

Discussion: Parallel implementation significantly reduced execution time compared to the Basic version. For the 2000×2000 matrix, execution time decreased from 83.7 seconds to 17.6 seconds, achieving approximately a 4.75× speedup. CPU utilization increased substantially, peaking at over 96%, indicating effective multi-core utilization. Memory usage remained relatively consistent across experiments.

5.4 Vectorized Matrix Multiplication Results

Table 3 shows the execution times, CPU usage, and memory consumption for various matrix sizes for the Vectorized Matrix Multiplication algorithm.

Matrix Size	Exec Time(ms)	CPU (%) Before	CPU (%) After	Memory Used(MB)
100×100	0.38	4.03	4.03	0.00
250×250	4.07	3.93	3.93	0.00
500×500	28.74	15.67	15.67	0.00
1000×1000	1677.71	85.56	83.66	0.06
2000×2000	18329.33	91.07	92.91	1.14

Table 3: Vectorized Matrix Multiplication Results

Discussion: The Vectorized implementation also demonstrated significant performance gains, reducing execution time to 18.3 seconds for the 2000×2000 matrix. However, it slightly underperformed compared to the Parallel version. This discrepancy may be attributed to overheads associated with managing parallel streams. CPU utilization remained high, consistent with the parallel approach, and memory usage was minimal.

5.5 Speedup and efficiency

Speedup and efficiency are critical metrics for evaluating the performance gains achieved by parallel and vectorized implementations compared to the basic sequential algorithm. Speedup (S) is calculated using the formula:
S= T basic / T parallel or vectorized
where T basic is the execution time of the basic algorithm, and T parallel or

vectorized is the execution time of the parallel or vectorized algorithm. Efficiency (E) is defined as the ratio of speedup to the number of threads (N):
E= S / N
This subsection presents a comparative analysis of the speedup and efficiency for the parallel and vectorized algorithms. Table 4 summarizes these metrics for different matrix sizes.

Matrix Size	**Algorithm**	**Exec Time (ms)**	**Speedup (S)**	**Threads (N)**	**Efficiency (E)**
100×100	Parallel	2.06	0.54	12	0.045
	Vectorized	0.38	2.95	12	0.246
250×250	Parallel	4.77	4.60	12	0.383
	Vectorized	4.07	5.40	12	0.450
500×500	Parallel	29.08	6.41	12	0.534
	Vectorized	28.74	6.49	12	0.541
1000×1000	Parallel	1585.87	3.46	12	0.288
	Vectorized	1677.71	3.27	12	0.273
2000×2000	Parallel	17640.91	4.75	12	0.396
	Vectorized	18329.33	4.57	12	0.381

Table 4: Speedup and Efficiency for Parallel and Vectorized Algorithms

Discussion:
Small Matrices (100×100): The vectorized algorithm outperformed the parallel version, achieving a speedup of 2.95 compared to 0.54 for the parallel implementation. However, efficiency remained low due to the small computational load and high overhead associated with multi-threading.
Medium Matrices (250×250 and 500×500): Both parallel and vectorized implementations demonstrated significant speedup, with the vectorized algorithm slightly outperforming the parallel version. Efficiency improved, indicating better utilization of the available threads.
Large Matrices (1000×1000 and 2000×2000): The parallel algorithm showed marginally better performance for larger matrices, achieving a speedup of 4.75 for the 2000×2000 matrix compared to 4.57 for the vectorized version. The efficiency values (0.4) suggest that both algorithms effectively leveraged the 12 available threads but faced diminishing returns due to overhead and synchronization costs.
Overall, the results demonstrate that both parallel and vectorized implementations provide substantial performance improvements, with the parallel algorithm exhibiting slightly better scalability for larger matrix sizes. These findings highlight the importance of selecting the appropriate optimization strategy based on matrix size and available computational resources.

5.6 Comparative Analysis

Figures 1 and 2 illustrates the execution times for all three algorithms across different matrix sizes and the speedups for the parallel and vectorized algorithms respectively:

Basic Implementation exhibits linear scaling inefficiencies, confirming its unsuitability for large datasets.

Parallel Implementation achieves the best performance, particularly for larger matrices, demonstrating efficient multi-core utilization.

Vectorized Implementation offers competitive performance with reduced memory overhead but slightly higher execution times compared to the Parallel version for larger matrices.

The experiments confirm that parallel and vectorized approaches significantly enhance matrix multiplication performance. The Parallel algorithm outperformed the Basic and Vectorized versions, especially for larger matrices, highlighting the importance of multi-threading in high-performance computing applications. These findings underscore the need for careful selection of optimization techniques based on specific computational requirements and resource constraints.

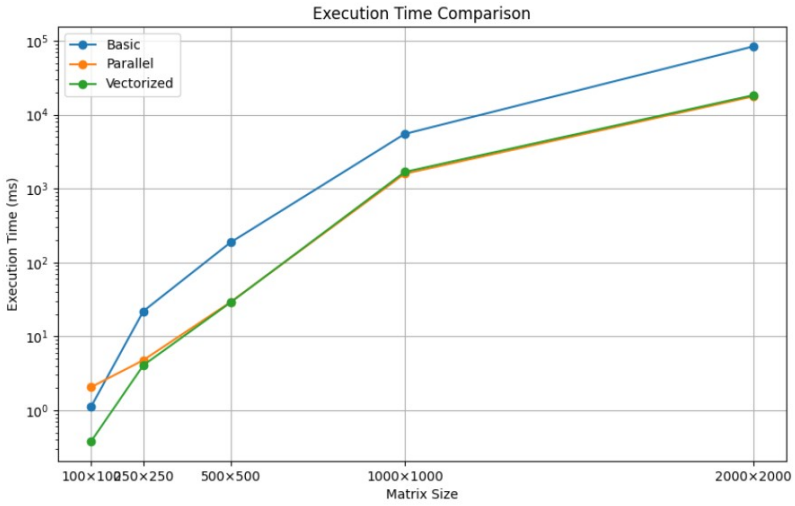

Figure 1: Execution times for all three algorithms across different matrix sizes.

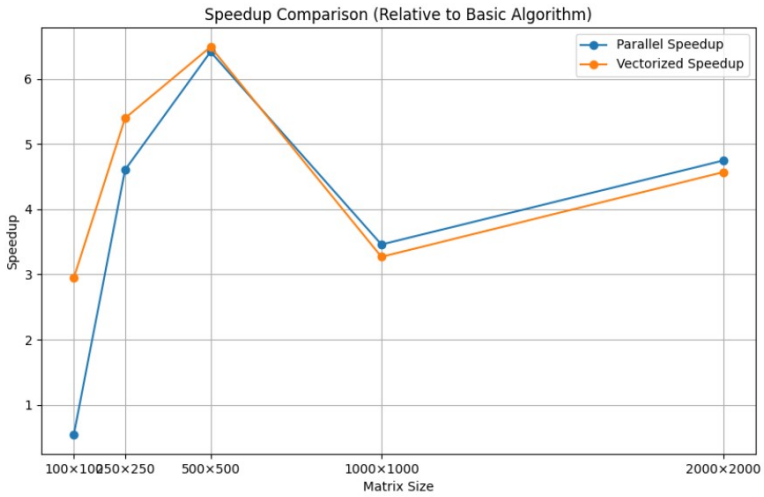

Figure 2: Speedup for parallel and vectorized algorithms across different matrix sizes.

6 Conclusions

This study addressed the computational challenge posed by matrix multiplication, a fundamental operation in various scientific and engineering domains. The primary objective was to optimize performance using parallel and vectorized approaches and compare them against a basic sequential implementation. We implemented three algorithms—Basic, Parallel, and Vectorized—using Java's concurrency utilities and parallel streams. The experiments involved benchmarking these implementations with matrices of different sizes (100×100 to 2000×2000), evaluating their performance based on execution time, CPU utilization, and memory usage.

The results demonstrated that parallel and vectorized implementations significantly outperformed the basic algorithm. The Parallel approach achieved the best overall performance, reducing the execution time of a 2000×2000 matrix multiplication from 83.7 seconds (Basic) to 17.6 seconds, achieving a speedup of approximately $4.75\times$. The Vectorized approach also showed notable improvements, though slightly lower than the Parallel version for larger matrices, likely due to overhead in managing parallel streams. Both optimized versions effectively utilized multi-core resources, with CPU utilization exceeding 90% during large computations.

This experimentation is crucial for understanding the potential of parallel and vectorized computing techniques in optimizing resource-intensive operations. The findings highlight the importance of leveraging multi-threading and vectorization to enhance computational efficiency, especially in high-performance applications. This work provides a valuable benchmark for developers and researchers aiming to optimize matrix operations in various domains.

7 Future Works

While this study provides significant insights into optimizing matrix multiplication, several avenues for future research remain:
Alternative Parallelization Strategies: Investigate different parallelization frameworks such as OpenMP, MPI, or GPUs to further improve performance and scalability.
Algorithmic Variants: Explore more advanced algorithms, such as Winograd's algorithm, which may offer additional performance benefits.
Scalability Tests: Conduct experiments on larger matrix sizes and more diverse hardware configurations to evaluate performance scalability across different architectures.
Resource Analysis: Extend the resource utilization analysis to include energy consumption, which is critical for green computing and mobile applications.
Real-World Applications: Apply these optimized algorithms to real-world problems in machine learning, computer graphics, or scientific simulations to validate their effectiveness in practical scenarios.
By addressing these aspects, future research can build upon the current findings to further enhance the efficiency and applicability of matrix multiplication algorithms, contributing to advancements in high-performance computing.