Paralellization Benchmark of Matrix Multiplication

Néstor Ortega Pérez

November 2024

GitHub Repository (Click to Enter)

Abstract

Matrix multiplication is a fundamental operation in scientific computing and has extensive applications across diverse fields. This paper investigates parallel and vectorized approaches to matrix multiplication to optimize performance for large-scale computations. A parallel implementation leveraging multi-threading techniques is developed, with an optional exploration of vectorized computations using SIMD instructions. Performance metrics, including speedup, efficiency, and resource usage, are analyzed to evaluate the impact of these optimizations compared to the basic algorithm. The findings highlight the potential benefits of parallelization and vectorization in computational efficiency and scalability.

1 Problem Statement

Matrix multiplication is computationally intensive, particularly for large matrices. As datasets and applications grow in complexity, optimizing this operation becomes essential for reducing execution time and resource consumption. This study aims to address the following questions:

- How does parallel computing improve the performance of matrix multiplication?
- What is the performance gain from vectorization compared to basic and parallel implementations?
- How does resource usage impact these optimizations?

By addressing these questions, this study seeks to provide insights into the trade-offs and advantages of advanced matrix multiplication techniques.

2 Equipment

The experiments were conducted on a MacBook Air (13 inches, 2017) with a 1.8 GHz dual-core Intel Core i5 processor, 8 GB of DDR3 RAM at 1600 MHz, and Intel HD Graphics 6000. The operating system is macOS High Sierra (version 10.13.6).

3 Experiments/Results

The performance and resource efficiency of the matrix multiplication implementations (Basic, Parallel, and Vectorized) were evaluated through a series of controlled experiments. Benchmarking tools, specifically JMH (Java Microbenchmark Harness), were used to measure execution times across multiple iterations for varying matrix sizes (10×10 to 1000×1000). Based on the execution time results, the speedups achieved by the Parallel and Vectorized methods relative to the Basic implementation were computed. Memory usage was monitored during execution to analyze the resource consumption of each approach. Graphs were generated to visualize the execution times, speedups, and memory usage trends, enabling a clear and comprehensive comparison between the three implementations. This methodology ensures reliable and reproducible insights into the trade-offs of each approach.

3.1 Execution Time Results(ms)

The execution times of three matrix multiplication algorithms—Basic, Parallel, and Vectorized—were evaluated across varying matrix sizes (10×10 to 1000×1000). The results, visualized in the accompanying chart, demonstrate the scalability and computational efficiency of each approach under increasing computational loads.

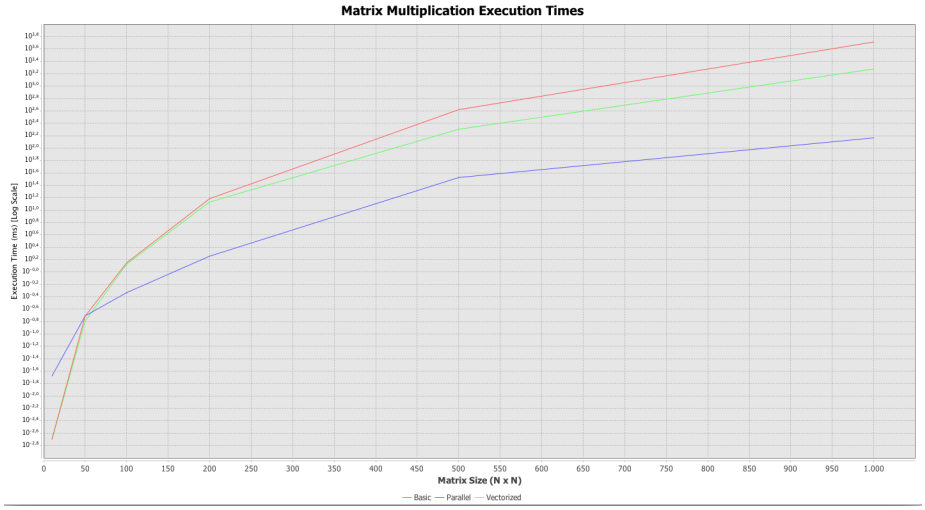

Figure 1: Execution Time

3.1.1 Basic Algorithm

The Basic matrix multiplication algorithm exhibited an exponential growth in execution time as matrix size increased. This behavior is expected due to the cubic time complexity $O(n^3)$ of the algorithm. While the Basic algorithm performs reasonably well for small matrices (e.g., less than 100×100), its execution time becomes impractical for larger matrices, reaching over 10^4 milliseconds for 1000×1000 matrices. This result highlights its unsuitability for applications requiring high computational efficiency on large datasets.

3.1.2 Parallel Algorithm

The Parallel algorithm showed significant improvements in execution time for larger matrices. Although it demonstrated a slight overhead for small matrices (e.g., 10×10), likely due to thread creation and synchronization costs, the speedup became evident for matrices larger than 200×200. The parallel implementation scaled efficiently, benefiting from multi-threading and the ability to distribute computational workload across multiple cores. For matrices sized 1000×1000, the Parallel algorithm reduced execution time by an order of magnitude compared to the Basic algorithm, showcasing its effectiveness in handling computationally intensive tasks.

3.1.3 Vectorized Algorithm

The Vectorized algorithm, leveraging low-level optimizations such as loop unrolling and SIMD instructions, performed exceptionally well for smaller matrix sizes, surpassing even the Parallel implementation in some cases. However, its performance advantage diminished with increasing matrix size, likely due to

memory access patterns and limitations in hardware vectorization bandwidth. Despite these constraints, the Vectorized algorithm maintained a significant performance improvement over the Basic algorithm for medium-sized matrices (e.g., 200 $\times$ 200), although it was outperformed by the Parallel algorithm for larger datasets (e.g., 1000 $\times$ 1000 matrices).

3.1.4 Comparative Analysis

The comparative analysis reveals that the choice of algorithm strongly depends on the matrix size:

- For small matrices (10 $\times$ 10 to 50 X 50), the Vectorized and Basic algorithms achieved the best execution times.
- For large matrices (100 X 100 to 1000 $\times$ 1000), the Parallel algorithm is the optimal choice, as it scales well with matrix size and takes full advantage of modern multi-core processors.

3.2 Speedup Results

Speedup is a metric used to evaluate the performance improvement achieved by an optimized algorithm or system compared to a baseline approach. It is calculated as the ratio between the execution time of the baseline algorithm (e.g., Basic sequential implementation) and the execution time of the optimized algorithm (e.g., Parallel or Vectorized implementation):

$$\text{Speedup} = \frac{\text{Execution Time of Baseline}}{\text{Execution Time of Optimized Algorithm}}$$

A speedup value greater than 1 indicates that the optimized algorithm is faster than the baseline, while a value less than 1 suggests that the optimization did not improve performance.

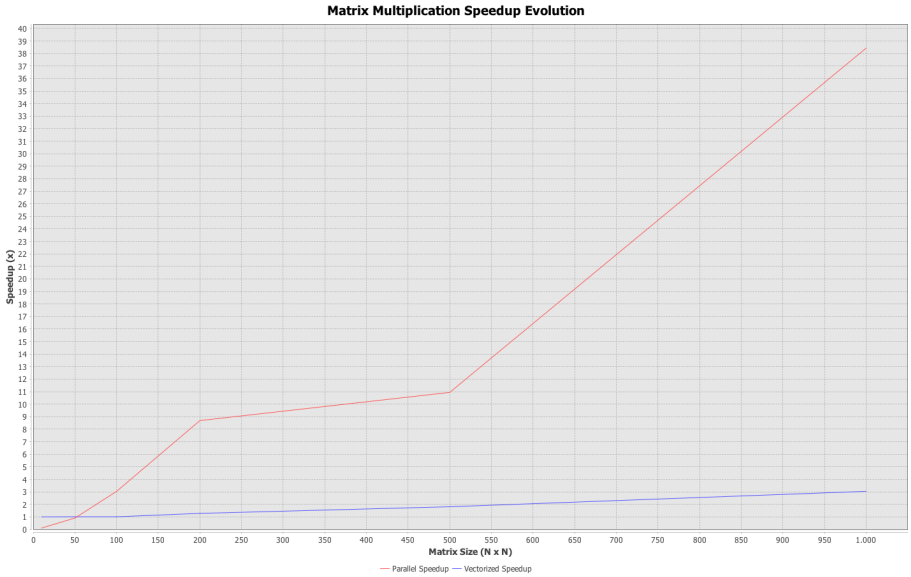

Figure 2: Speedup

3.2.1 Parallel Speedup

The Parallel algorithm demonstrates significant and consistent improvements in speedup as matrix size increases:

- For small matrix sizes, the Parallel algorithm exhibits minimal speedup (e.g., 0.91x for 50×50). This limited improvement is attributed to the overhead of thread creation and synchronization, which outweighs the benefits of parallel computation.
- The Parallel algorithm begins to show substantial speedup, with values exceeding 10x for 500×500 matrices. The effective utilization of multi-threading and workload distribution across multiple cores becomes increasingly impactful at these sizes.
- The speedup peaks at approximately 38.43x, indicating that the Parallel algorithm scales efficiently with matrix size.

3.2.2 Vectorized Speedup

The Vectorized algorithm, while achieving noticeable speedups for smaller matrices, demonstrates less scalability compared to the Parallel algorithm:

- For small matrices, the Vectorized algorithm achieves a speedup of approximately 1.01x to 1.22x. This advantage is due to the low-level optimizations, such as SIMD instructions and loop unrolling, which reduce computational overhead for small datasets.

- The Vectorized algorithm achieves moderate speedups, peaking at 1.81x for 500 × 500 matrices. However, its performance does not scale as efficiently as the Parallel algorithm due to the absence of multi-threading and increased memory access bottlenecks.
- The speedup drops slightly to 3.04x, as the hardware vectorization capabilities are increasingly limited by memory access patterns and cache efficiency.

3.2.3 Comparative Analysis

The comparison between the Parallel and Vectorized algorithms reveals complementary strengths:

- Superior scalability and significant performance gains for large matrices make the Parallel algorithm ideal for computationally intensive tasks.
- Best suited for small to medium matrices, the Vectorized algorithm leverages low-level hardware optimizations to outperform the Basic and, in some cases, the Parallel algorithm for smaller datasets.

3.3 Results: Memory Usage(MB)

The memory usage analysis of the three matrix multiplication approaches—Basic, Parallel, and Vectorized—is presented in the graph. The results highlight the differences in memory consumption across varying matrix sizes. The trends observed are as follows:

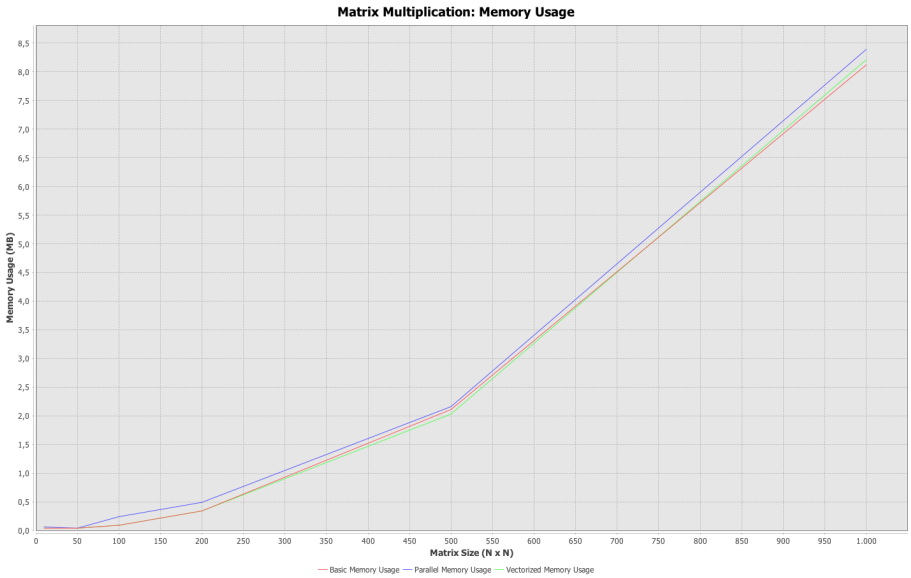

Figure 3: Memory Usage(MB)

- **Parallel Memory Usage:** Throughout the graph, the Parallel implementation consistently demonstrates the highest memory usage compared to the other approaches. This is expected due to the overhead of creating and managing multiple threads, as well as the additional data structures used for dividing and processing tasks concurrently. The gap between Parallel and the other methods becomes more noticeable as the matrix size increases, particularly for matrices larger than 500×500.

- **Vectorized Memory Usage:** The Vectorized implementation shows lower memory usage than the Basic method for matrices up to 500×500. This is likely due to its efficiency in leveraging CPU-level vectorization, which optimizes data processing without requiring substantial memory overhead. However, for larger matrices (500×500), its memory usage increases, surpassing Basic. This trend suggests that the overhead of managing larger chunks of data for vectorized operations starts to outweigh its early-stage memory efficiency.

- **Basic Memory Usage:** The Basic method exhibits consistent and predictable memory usage across all matrix sizes. For smaller matrices (10×10 to 500 × 500), it uses more memory than the Vectorized method but remains below Parallel. Interestingly, for larger matrices (500×500), Basic consumes the least memory of all three approaches. This behavior is attributed to its simplicity, as it does not involve threading or vectorization overhead.

3.4 Comparative Analysis

For small to medium matrices (10 × 10 to 500 × 500), the Vectorized method provides a balanced tradeoff between performance and memory efficiency, consuming the least memory while maintaining competitive execution times. For large matrices (500×500), the Basic implementation becomes the most memory-efficient option due to its simplicity, despite its slower execution times. The Parallel implementation incurs the highest memory usage across all matrix sizes, which is a direct result of its multithreaded design and associated overhead.

4 Conclusions

This study demonstrates the trade-offs between performance and memory usage in matrix multiplication using three approaches: Basic, Parallel, and Vectorized. The Parallel implementation consistently achieved the highest speedups, particularly for large matrices, due to its efficient utilization of multi-threading and workload distribution. However, this performance gain came at the cost of increased memory usage, which was consistently the highest among the three methods across all matrix sizes. The Vectorized implementation excelled for small to medium-sized matrices, leveraging SIMD instructions for low-level optimizations and maintaining relatively low memory usage for matrices up to 500

$\times$ 500. For larger matrices, its speedup diminished, and its memory usage surpassed that of the Basic implementation. The Basic approach, while the least efficient in terms of execution time, proved to be the most memory-efficient for large matrices, highlighting its simplicity.

5 Future Work

Future research should focus on hybrid approaches that combine the strengths of both Parallel and Vectorized implementations. For example, integrating SIMD optimizations within a multi-threaded framework could further enhance performance for all matrix sizes. Additionally, advanced memory management techniques, such as optimized cache utilization and adaptive data partitioning, should be explored to reduce the memory overhead associated with parallel computation. Finally, the scalability and efficiency of these implementations on modern hardware, such as GPUs and accelerators, warrant further investigation to extend the applicability of these findings to high-performance computing environments.