Parallel and vectorized matrix multiplication

DIEGO JESUS TORREJÓN CABRERA

November 2024

GitHub link: Parallel-and-vectorized-matrix-multiplication

1 Abstract

This work addresses the challenge of optimizing matrix multiplication, a core operation in fields like machine learning and scientific computing. The computational complexity of this task increases significantly as the size of the matrices grows, making it crucial to explore methods that reduce execution time and resource consumption. To tackle this, three approaches were implemented and evaluated: sequential matrix multiplication, parallel matrix multiplication, and vectorized matrix multiplication. The experiments were carried out using matrices of sizes ranging from 784 x 784 to 2024 x 2024, with performance measured in terms of execution time, memory usage, speedup, and efficiency.

The results demonstrated that the parallel implementation achieved substantial speedup, with improvements exceeding a factor of 6, particularly for larger matrices. The vectorized approach also showed notable performance gains compared to the sequential method but with increased memory usage. These findings emphasize the trade-off between computational efficiency and memory usage, particularly for real-time or large-scale data processing tasks. The experiments provide valuable insights into optimizing matrix multiplication, serving as a foundation for future research into distributed computing and hardware-specific optimizations.

2 Introduction

Matrix multiplication is a fundamental operation in many fields, including scientific computing, computer graphics, and machine learning. The process involves multiplying two matrices to produce a third, requiring O(n^3) operations in its naive implementation for square matrices of size n x n. As the size of matrices grows, so does the computational cost, making optimization a key focus in both academia and industry.

Over the years, researchers have developed various strategies to optimize matrix multiplication. Parallel computing techniques, such as multithreading, have been employed to distribute the workload across multiple processors, thereby reducing execution time. Additionally, vectorization approaches, which leverage Single Instruction Multiple Data (SIMD) capabilities of modern CPUs, enable simultaneous operations on multiple data points, further enhancing performance. For instance, studies have shown that SIMD-based implementations can significantly outperform traditional sequential algorithms by efficiently utilizing CPU registers.

In this paper, we explore and compare three implementations of matrix multiplication: a naive sequential approach, a parallel approach utilizing multithreading, and a vectorized approach employing SIMD instructions. The added value of this work lies in its comprehensive evaluation of these methods, focusing on speedup, efficiency, and resource utilization. By analyzing the performance trade-offs, this study provides practical insights for selecting the most suitable method based on the computational context.

3 Methodology

To address the problem of optimizing matrix multiplication, three algorithms have been implemented and evaluated using matrices of increasing sizes: 784 x 784, 1024 x 1024, 1764 x 1764, and 2024 x 2024. The experiments were designed to measure and compare both execution time and memory usage for each approach, with the goal of understanding their efficiency and scalability as the matrix size grows.

First, a benchmark has been implemented to compare the performance of three dense matrix multiplication methods:

- Naive Sequential Algorithm: This method uses a basic triple-nested loop to compute the product, serving as a reference for comparison.

- Parallel Algorithm: This method leverages multithreading to accelerate dense matrix multiplication by dividing the computation across multiple threads using Java's ExecutorService. Each thread computes a single row of the result matrix C, with tasks submitted to a fixed thread pool. Synchronization is managed via Future objects to ensure all threads complete before proceeding.

- SIMD Vectorized Algorithm: This implementation leverages the jdk.incubator.vector library to process data in parallel using CPU vector instructions, optimizing instruction-level parallelism. In order to use this library, a plugin was added to Maven and the execution configuration was modified, adding the **–add-modules jdk.incubator.vector** module in the VM options section.

Each of these implementations is executed and evaluated on matrices of the specified sizes, recording execution time and memory usage for each case. These metrics provide insight into the computational cost and scalability of each algorithm.

The benchmarks were conducted using the Java Microbenchmark Harness (JMH), ensuring reliable and reproducible measurements. For all tests, 5 warm-up iterations and 10 measurement iterations were used. Moreover, for the parallel matrix multiplication method, a pool of 11 threads was used. This configuration allows the JVM to apply Just-In-Time (JIT) optimizations during the warm-up phase, resulting in more accurate performance measurements.

The experiments were performed on a machine equipped with an M3 Pro chip with 11 cpu cores, 18 GB of unified memory, running MacOS Sequoia 15.1. To ensure fairness and reproducibility:

- Garbage collection was explicitly invoked before each benchmark invocation to minimize memory overhead.

- Each implementation was executed in isolation, with no other major processes running on the machine.

The execution time was measured in seconds, representing the time required to complete the multiplication. The memory usage was tracked by calculating the difference between the JVM's total memory and free memory after each iteration. These metrics provide a detailed comparison of the computational efficiency and resource utilization of each method.

Additionally, the speedup achieved by the parallel and SIMD implementations compared to the naive approach was computed. This metric highlights the effectiveness of these optimizations in reducing computation time.

4 Experiments

This section describes the experiments performed to evaluate the performance of the three matrix multiplication methods: sequential multiplication, parallel multiplication, and vectorized multiplication. The results are presented in terms of execution time, memory usage, and measures of speedup and efficiency, providing a detailed overview of the advantages and limitations of each approach.

4.1 Execution Time

Size (n x n)	**Matrix Multiplication(s)**	**Parallel Matrix Multiplication(s)**	**Vectorized Matrix Multiplication(s)**
784	0.359	0.062	0.167
1024	1.504	0.226	0.378
1764	8.770	1.336	1.986
2024	14.491	2.358	2.984

Table 1: Comparison of times for different matrix multiplication methods

The measured execution time for the three methods is presented in Table 1. It is observed that the parallel implementation achieves significantly lower times compared to the sequential one, while the vectorized one also improves the times, although to a lesser extent.

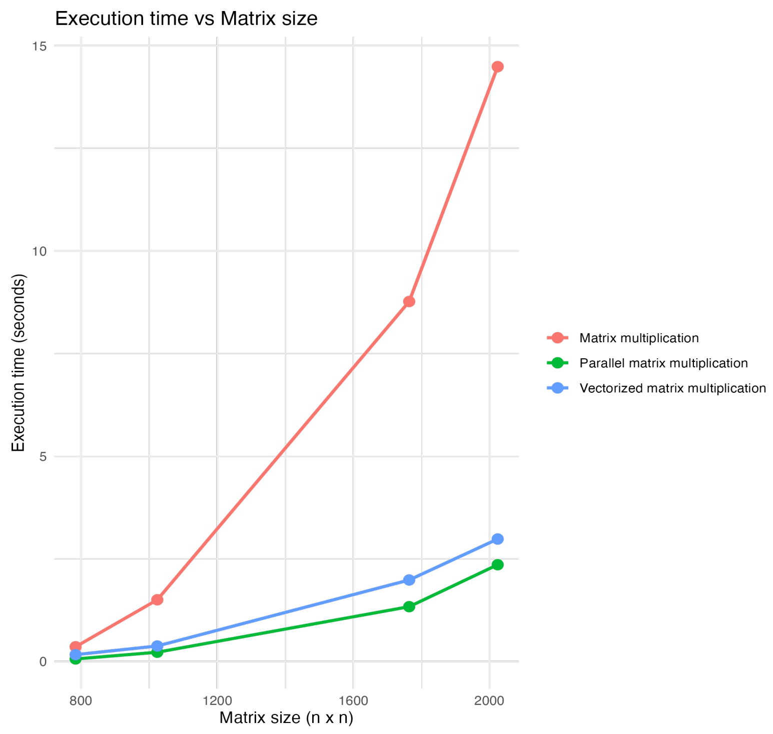

Figure 1: Graph that shows the execution time for different matrix multiplication methods

Figure 1 graphically represents the execution time for each method. It is evident that the times increase non-linearly with the size of the matrices, especially in the case of sequential implementation. Here it is clear how there is a significant improvement in execution times using multithreading and vectorized versions.

4.2 Memory Usage

Size (n x n)	Matrix Multiplication(MB)	Parallel Matrix Multiplication(MB)	Vectorized Matrix Multiplication(MB)
784	19.669	20.462	24.539
1024	29.667	30.447	38.371
1764	78.488	81.933	104.589
2024	101.717	106.067	135.999

Table 2: Comparison of memory usage for different matrix multiplication methods

The average memory usage for the three methods is detailed in Table 2. The vectorized implementation has the highest memory consumption due to the additional overhead associated with handling vector registers, while the parallel implementation maintains moderate memory consumption.

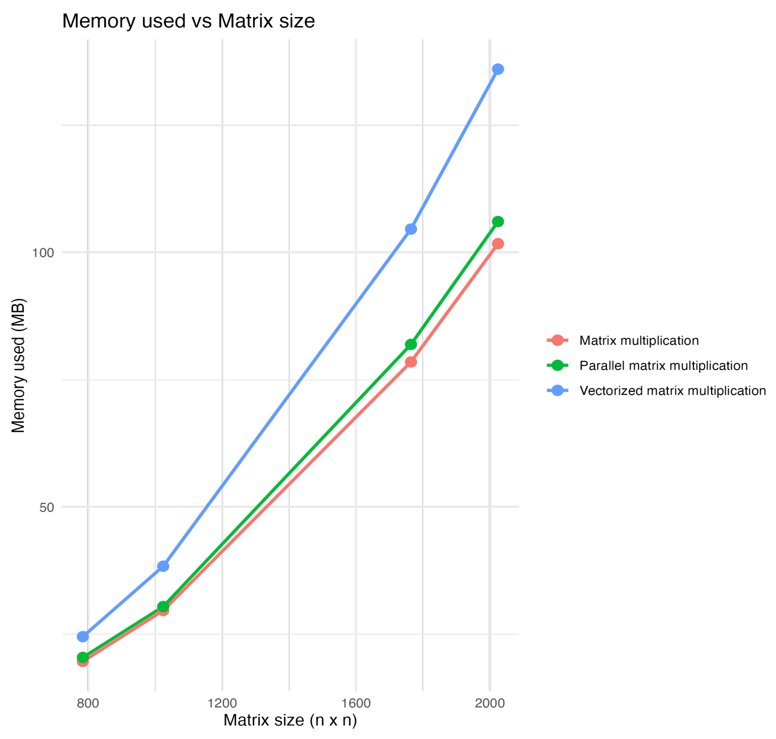

Figure 2: Graph that shows the memory used for different matrix multiplication methods

In Figure 2, it can be seen that memory usage increases with the size of the arrays in all methods, although the sequential implementation remains the most efficient in terms of resource consumption.

4.3 Speedup an Efficiency

4.3.1 Speedup

Speedup measures the acceleration achieved by parallel and vectorized methods compared to sequential. The acceleration is calculated as:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{method}}}$$

Size (n x n)	Parallel Speedup	Vectorized Speedup
784	5.79	2.15
1024	6.65	3.98
1764	6.56	4.41
2024	6.14	4.86

Table 3: Acceleration achieved by parallel and vectorized methods

Table 3 shows that the parallel method achieves a higher speedup at all sizes, indicating a higher efficiency in time reduction.

4.3.2 Efficiency

Efficiency measures how well resources are used, considering the number of threads in the parallel method:

$$\text{Efficiency} = \frac{T_{\text{Speedup}}}{T_{\text{N threads (11)}}}$$

Size (n x n)	Parallel Efficiency
784	0.53
1024	0.61
1764	0.60
2024	0.56

Table 4: Efficiency of parallel method

Table 4 highlights that efficiency decreases slightly as the size of the arrays grows, due to the overhead associated with thread coordination and vector register handling.

5 Conclusions

This work addressed the challenge of optimizing matrix multiplication, a fundamental operation in various fields. The computational complexity associated with this problem, especially when working with large matrices, necessitates exploring techniques that improve both execution time and resource utilization.

To tackle this challenge, three different approaches were implemented and evaluated: sequential matrix multiplication, parallel matrix multiplication, and vectorized matrix multiplication. A reproducible methodology based on JMH was designed to consistently measure execution time, memory usage, speedup, and efficiency. The experiments were conducted with matrices of increasing sizes, from 784 x 784 to 2024 x 2024, to assess how each technique performed in terms of scalability and resource usage.

The experimental results show that the parallel implementation achieved the greatest speedup, surpassing a factor of 6 in all cases. This highlights the effectiveness of dividing the computational load across multiple threads, particularly for larger matrices. On the other hand, the vectorized implementation also showed significant improvements compared to the sequential method, although at the cost of higher memory usage due to the handling of vector registers.

The memory usage analysis revealed that while parallel and vectorized methods are faster, they come with a higher resource consumption. However, this tradeoff is reasonable in scenarios where execution time is critical, such as real-time applications or large-scale data processing.

This experiment not only validates the effectiveness of parallel and vectorization optimizations but also underscores the importance of selecting the appropriate technique based on the problem's requirements. The proposed methodology allows for an objective comparison of these implementations, providing a useful framework for future research in optimization of algebraic operations and high-performance computing.

The significance of this experimentation lies in its provision of quantitative evidence of the impact of parallelism and vectorization techniques on matrix multiplication efficiency, laying the groundwork for more advanced explorations such as distributed methods or adaptations for specialized hardware like GPUs or heterogeneous architectures. With this contribution, we aim to further advance the optimization of matrix computations and promote their efficient application in real-world critical tasks.

6 Future Works

While the current study focuses on optimizing matrix multiplication through parallelism and vectorization on a single machine, an important direction for future research is the exploration of distributed matrix multiplication. As the size of data grows, matrices that exceed the memory capacity of a single machine become a significant challenge. Distributed computing offers a promising solution by dividing large matrices into smaller sub-matrices that can be processed in parallel across multiple machines, leveraging the combined memory and processing power of a distributed system.

In a distributed setting, matrix multiplication can be performed by decomposing the matrices into blocks and distributing these blocks across different nodes in a cluster. Each node computes a partial product, and the results are then aggregated to produce the final output. This approach requires careful handling of data distribution, load balancing, and network communication, as the overhead from data transfer can often reduce the performance gains achieved from parallel processing.

The scalability of distributed matrix multiplication will be a key area of investigation. It is essential to evaluate how performance scales as more machines are added to the system, and how the system handles the increasing complexity of managing data across a distributed environment. Additionally, the use of distributed systems in cloud computing environments, such as Apache Spark or MPI (Message Passing Interface), presents an interesting area for further exploration.

In conclusion, future work will focus on scaling matrix multiplication techniques to distributed systems, investigating how to effectively manage and compute very large matrices that do not fit into the memory of a single machine. The insights gained from these studies will be valuable for applications in fields like big data analytics, scientific simulations, and machine learning, where the size of the datasets continues to grow exponentially.