Performance Benchmark of matrix multiplication

Luna Yue Hernández Guerra
GitHub: github.com/lunahernandez/BigDataIndividualAssignments

November 2024

Abstract

This project explores optimized matrix multiplication methods, focusing on improving computational efficiency for large-scale matrix operations, which are essential in areas such as computer science, engineering, and data analysis. The study investigates several algorithms, including the naive, parallel, and vectorized approaches, to determine their effectiveness in reducing execution time and memory usage. Using Java implementations and performance benchmarking tools, we evaluate the speedup and efficiency of parallelization and vectorization at various matrix sizes. The results indicate that the vectorized algorithm consistently outperforms the others, particularly for larger matrices, in terms of both speed and memory usage. Among parallel algorithms, the 4-thread implementation provides the best performance, followed by the 2-thread approach. The 8-thread parallel algorithm, while still faster than the naive implementation, demonstrates diminishing returns in performance due to the overhead of managing additional threads. The naive algorithm, though simple, proves to be the least efficient, especially as matrix size increases. This work highlights the effectiveness of the vectorized algorithm for balancing memory efficiency and runtime, while parallel algorithms offer performance improvements for larger matrices, with the 4-threaded version being the most efficient for speed.

Contents

List of Figures

1 Introduction

Matrix multiplication is a fundamental operation in many scientific and engineering applications, often involving large data sets and requiring significant computational resources. As matrices grow in size, the computational cost increases, making matrix multiplication an ideal candidate for optimization. In this task, we explore several approaches to improve the performance of matrix multiplication, focusing on parallel computing techniques, including multi-threading and vectorized execution.

The main objective is to implement and evaluate a parallel version of matrix multiplication. By taking advantage of parallel processing capabilities, we aim to accelerate the computation of matrix products, in particular for large matrices. In addition, we investigate the potential benefits of matrix vector multiplication, which leverages SIMD (Single Instruction, Multiple Data) instructions for further optimization. While the parallel approach divides the workload among multiple processors or threads, the vectorized approach enables efficient data processing within a single instruction cycle, making it a powerful tool for high-performance computing.

This study compares the performance of parallel and vectorized implementations with a basic (naive) matrix multiplication algorithm. We evaluate the effectiveness of these approaches by measuring key performance metrics, such as speedup (the reduction in execution time compared to the naive algorithm), efficiency, and execution speed.

2 Problem Statement

Given the increasing computational demands of large-scale matrix operations, the selection of an optimal algorithm is crucial to minimize execution time and memory usage. Using Java, we will implement parallel and vectorized matrix multiplication algorithms, and compare their performance with that of a basic (naive) matrix multiplication approach. We will use benchmarking tools to evaluate the execution time, memory usage and efficiency of each algorithm. In addition, we will analyze how these algorithms scale with increasing matrix size and evaluate their efficiency in different parallelization scenarios. The goal is to identify the most efficient algorithm in terms of speed and resource consumption.

3 Methodology

In order to tackle the problem, we will create different classes that will represent the different types of algorithms as well as files to test the algorithms and use the measurement tools on them.

To compare the results, we will set the units of measurement in milliseconds for time and KB for memory.

Therefore, given the classes with the implementations of the algorithms used, in *MatrixMultiplicationBenchmarking.java* we will find the code to measure the execution time and in *MatrixMultiplicationMemory.java* we will find the corresponding one that measures the memory usage. All this is done thanks to **The Intellij JMH (Java Microbenchmark Harness)** plugin and Runtime class which belongs to Java Standard Library.

This data will then be saved in json files (the runtime) using the program argument `-rf json -rff benchmark_results.json` and csv (the memory used) through implemented code, which will then be processed with the *BenchmarkResultProcessor.java* class and plotted with *TimeDataPlotter.java* and *MemoryDataPlotter.java*.

4 Experiments

The following matrix multiplication algorithms are implemented:

- **NaiveMatrixMultiplication**: This class performs a standard matrix multiplication using three nested loops. It initializes an empty matrix c to store the result and asserts that the dimensions of matrices a and b are compatible.

- **ParallelMatrixMultiplication**: This class performs matrix multiplication using a given number of threads. Each thread processes a subset of the rows of the output matrix. Proper synchronisation is ensured to avoid race conditions during updates of shared data as each thread handles a different section.
- **VectorizedMatrixMultiplication**: This implementation uses the Apache Commons Math (1) library for matrix arithmetic. It uses the library's `RealMatrix` class to represent matrices and performs multiplication using the `multiply()` method, which is optimized for numerical efficiency. By using the robust and highly optimized linear algebra routines provided by the library, this approach eliminates the need for explicit loops and achieves significant performance improvements, especially for large matrices.

The benchmarking process is implemented in the `MatrixMultiplicationBenchmarking.java` file. This file contains the previous functions to evaluate different approaches to matrix multiplication.

For the benchmarking process, the average time mode was established, with the unit of measurement set to milliseconds and utilizing 1 fork. As well as one warmup iteration and 5 measurement iterations.

Also, the matrix sizes 64, 128, 512, 1024 and 2048 and the number of threads 2, 4 and 8 are compared.

First of all, looking at Figure 1, we can see the execution time as a function of the size of the matrix for the different algorithms. We can see that the vectorized algorithm is the best with larger matrices. We can also notice that the parallel algorithm with different number of threads improves the execution time compared to the naive algorithm.

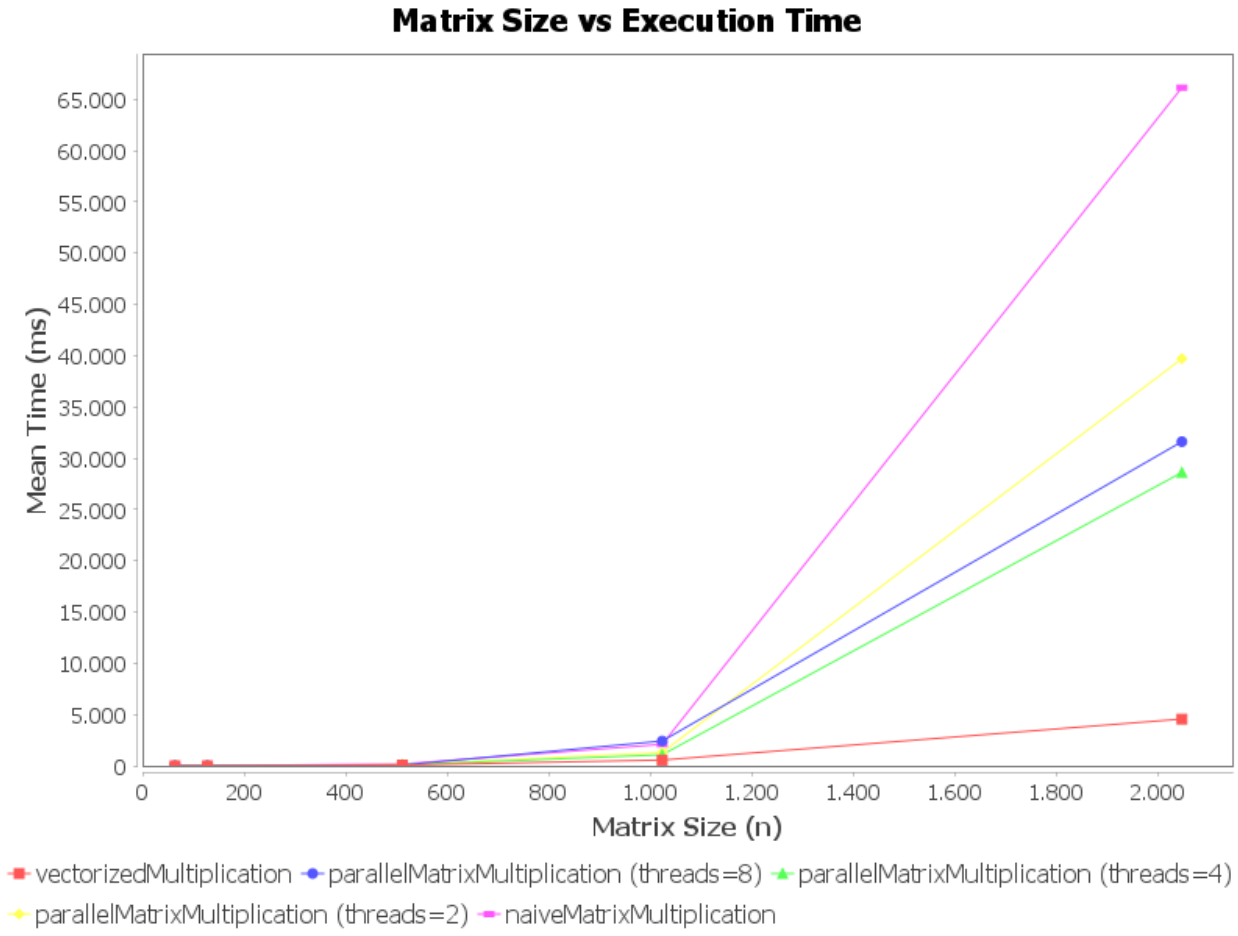

Figure 1: Execution Time Comparison

Then, among the different numbers of threads used, we observe that the most optimal is 4 threads, followed by 2, for large matrices.

Table 1 gives a better overview of the behavior. For the smallest matrix tested, 64, the basic algorithm is faster than the others. But already at size 128, improvements are found. Then, we can see that although with arrays of sizes 128 and 512 the parallel algorithm with 8 threads does not perform badly, with 1024 it is worse than the naive one. Meanwhile, the values for vectorized are much better than the others.

Table 1: Matrix Multiplication Benchmark Results

Algorithm	**Matrix Size**	**Number of Threads**	**Mean Execution Time (ms)**
Naive	64	-	0.1986
Parallel	64	2	0.3230
Parallel	64	4	0.4352
Parallel	64	8	0.7081
Vectorized	64	-	0.2425
Naive	128	-	1.9077
Parallel	128	2	1.7911
Parallel	128	4	1.3554
Parallel	128	8	1.3125
Vectorized	128	-	1.1604
Naive	512	-	177.4699
Parallel	512	2	119.9870
Parallel	512	4	90.1013
Parallel	512	8	69.6633
Vectorized	512	-	67.7176
Naive	1024	-	2088.7963
Parallel	1024	2	1304.1758
Parallel	1024	4	1069.2263
Parallel	1024	8	2415.6236
Vectorized	1024	-	558.5875
Naive	2048	-	66165.1263
Parallel	2048	2	39717.8006
Parallel	2048	4	28616.4044
Parallel	2048	8	31619.6562
Vectorized	2048	-	4558.2364

We will analyze the following performance metrics for parallel and vectorized algorithms:

$$\text{Speedup} = \frac{T_1}{T_n},$$

$$\text{Speedup} = \frac{T_1}{T_{vec}},$$

where T_1 is the execution time of the naive algorithm, T_n is the execution time of the parallel algorithm using n threads and , T_{vec} is the execution time of the vectorized algorithm. A higher speedup value indicates better performance, as it shows how much faster the parallel or vectorized algorithm is compared to the sequential one.

$$\text{Efficiency} = \frac{\text{Speedup}}{n} = \frac{T_1}{T_n \cdot n},$$

where n is the number of threads. Efficiency quantifies how effectively the parallel resources are utilized, with a value closer to 1 indicating high efficiency. A higher efficiency is desirable, as it reflects better utilization of computational resources.

In Table 2, we observe that the vectorised algorithm shows remarkable performance, significantly outperforming the other methods as the matrix size increases. For a matrix of size 2048, it achieves a speedup of 14.5155, demonstrating its great efficiency in handling large-scale computations. Additionally, we observe that, in terms of efficiency, the best performance is achieved by the algorithm utilizing 2 threads. This highlights its ability to effectively balance workload distribution and minimize overhead, resulting in optimal resource utilization.

Table 2: Speedup and Efficiency for Matrix Multiplication

Algorithm	**Matrix Size**	**Number of Threads**	**Speedup**	**Efficiency**
Parallel	64	2	0.6149	0.3074
Parallel	64	4	0.4563	0.1141
Parallel	64	8	0.2805	0.0351
Vectorized	64	-	0.8190	-
Parallel	128	2	1.0651	0.5325
Parallel	128	4	1.4075	0.3519
Parallel	128	8	1.4535	0.1817
Vectorized	128	-	1.1604	-
Parallel	512	2	1.4791	0.7395
Parallel	512	4	1.9697	0.4924
Parallel	512	8	2.5475	0.3184
Vectorized	512	-	2.6207	-
Parallel	1024	2	1.6016	0.8008
Parallel	1024	4	1.9536	0.4884
Parallel	1024	8	0.8647	0.1081
Vectorized	1024	-	3.7294	-
Parallel	2048	2	1.6659	0.8329
Parallel	2048	4	2.3121	0.5780
Parallel	2048	8	2.0925	0.2616
Vectorized	2048	-	14.5155	-

In terms of memory, a graph has been generated showing memory usage in relation to different algorithms and the number of threads. Computational cost is a crucial factor in the performance of applications, especially in resource-constrained environments. For this reason, it is important to find a balance between execution time and memory used.

As shown in Figure 2, naive and vectorized algorithms show the best memory efficiency, using the least amount of memory. This is probably because they do not require complex data structures that consume additional memory. Parallel algorithms with 2 and 4 threads have similar performance in terms of memory usage, as they probably scale well without excessive overhead. However, parallel algorithms with 8 threads tend to use more memory. This could be due to the larger number of threads, which may introduce additional memory overhead for synchronization, thread management or buffering during computation, resulting in higher memory demand.

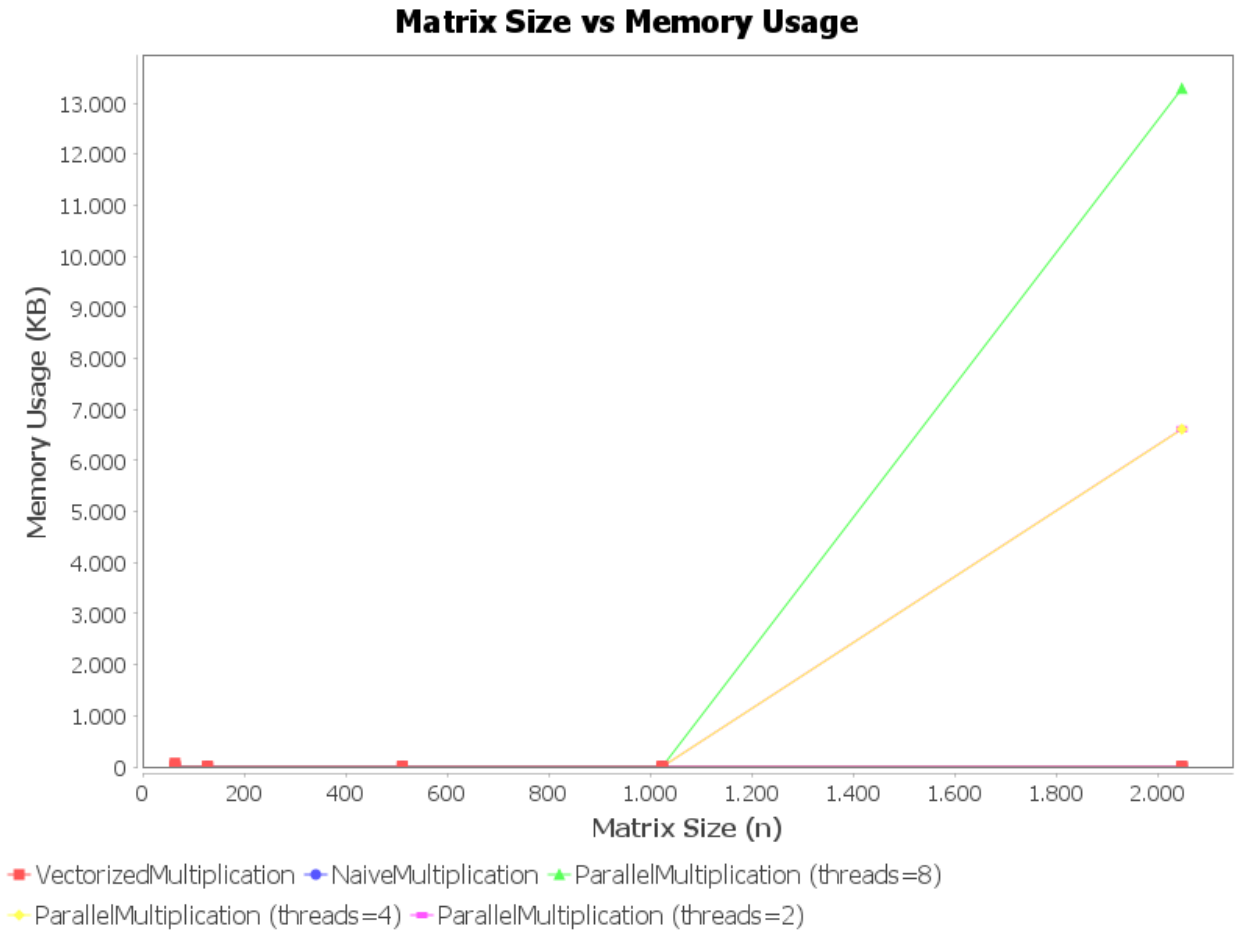

Figure 2: Memory Usage Comaprison

In conclusion, the vectorized algorithm is the most efficient in terms of memory, as it uses the least amount of memory across all matrix sizes. It also obtains better results in terms of execution time compared to the parallel and naive algorithms. It offers the best balance between memory efficiency and execution time.

5 Conclusion

In this study, several matrix multiplication algorithms were compared in terms of execution time, memory usage, speedup and efficiency. Specifically, naive, vectorized and parallel algorithms with different numbers of threads were analyzed to determine their performance at different matrix sizes.

In terms of memory efficiency, the vectorized algorithm outperforms the others, using the least amount of memory across all matrix sizes tested. This makes it the most efficient choice for memory-constrained environments.

When runtime is considered, the vectorized algorithm also performs better than the others. Also, the parallel algorithms demonstrate a clear advantage for larger matrix sizes over the basic algorithm, but fall short of the vectorized one. Among the parallel implementations, the 4-thread parallel algorithm shows the best performance, followed by the 2-thread algorithm. The 8-thread parallel algorithm, while still faster than the naive implementation, shows a decrease in performance compared to the 4-thread version. This could be attributed to the overhead introduced by handling more threads, which may negate some of the benefits for smaller arrays or less intensive computations.

Finally, the naive algorithm, although simple, is the slowest, especially as the size of the matrix increases.

Its performance is significantly outperformed by the vectorized and parallel algorithms, making it less suitable for larger computations.

Thus, the vectorized algorithm is the optimal choice for balancing memory efficiency and runtime, while the parallel algorithms perform better in runtime, with the 4-threaded version being the most efficient in terms of speed.

6 Future Work

Future work will focus on exploring distributed computing approaches to matrix multiplication, with an emphasis on scalability. As matrices grow and do not fit in the memory of a single machine no matter how much vectorized algorithm is used, distributed systems can spread the matrix across multiple machines to perform the computation in parallel.

This approach will help improve the efficiency of matrix multiplication for very large data sets. Major areas of future research include optimizing data partitioning, reducing inter-machine communication overhead, and balancing the computational load across the system.

This exploration will provide insights into high-performance computing and applications in machine learning, where large matrix operations are critical.

References

[1] A. C. Math, "Realmatrix (version 3.6.1) [api documentation]," 2017, accessed: 2024-11-24. [Online]. Available: https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/linear/RealMatrix.html