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 show that the vectorized algorithm delivers the best performance, especially for larger matrices, achieving significant speedup, with a remarkable increase of 16.9174 for a 2048 matrix. Among the parallel algorithms, the 2-threaded approach proves to be the most efficient in terms of performance, achieving an optimal balance between workload distribution and minimum overhead. On the contrary, if more threads are added (4 or 8, for example), performance decreases, and the 8-threaded implementation shows a decrease in efficiency compared to the 4-threaded version, probably due to increased synchronization and management overhead. The naive algorithm, while simple, remains competitive for smaller matrices but is outperformed as the matrix size grows.

This work demonstrates that the vectorized algorithm is the most effective in balancing memory efficiency and runtime performance. However, parallel algorithms offer clear advantages for larger matrices, especially when using 2 or 4 threads. Unless resources are limited, such as limited memory, in which case the use of parallel algorithms should be carefully considered.

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 vectorized matrix 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, execution speed and memory usage.

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 MB 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 the memory usage. All this is done thanks to **The Intellij JMH (Java Microbenchmark Harness)** plugin. In order to perform the memory calculation with this plugin, it is necessary to add the following instruction as an argument of the program:

```
-prof gc.
```

This data will then be saved in json files using the following program argument:

```
-rf json -rff benchmark_results.json
```

Then, we extract the data we need with the `BenchmarkResultProcessor.java` class and save it in a CSV file. Next, we plot both time and memory results with the `DataPlotter.java` class.

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 implements parallel matrix multiplication using two different approaches:

  - Explicit threads: Uses a fixed number of pre-specified threads. Each thread processes a subset of the rows of the output matrix.
  - ExecutorService: Employs a thread pool automatically managed by the Java framework to distribute the work among a specified number of threads.

  In both cases, synchronization is not necessary because each thread works independently on different sections of the output matrix, eliminating the possibility of race conditions.

- **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.

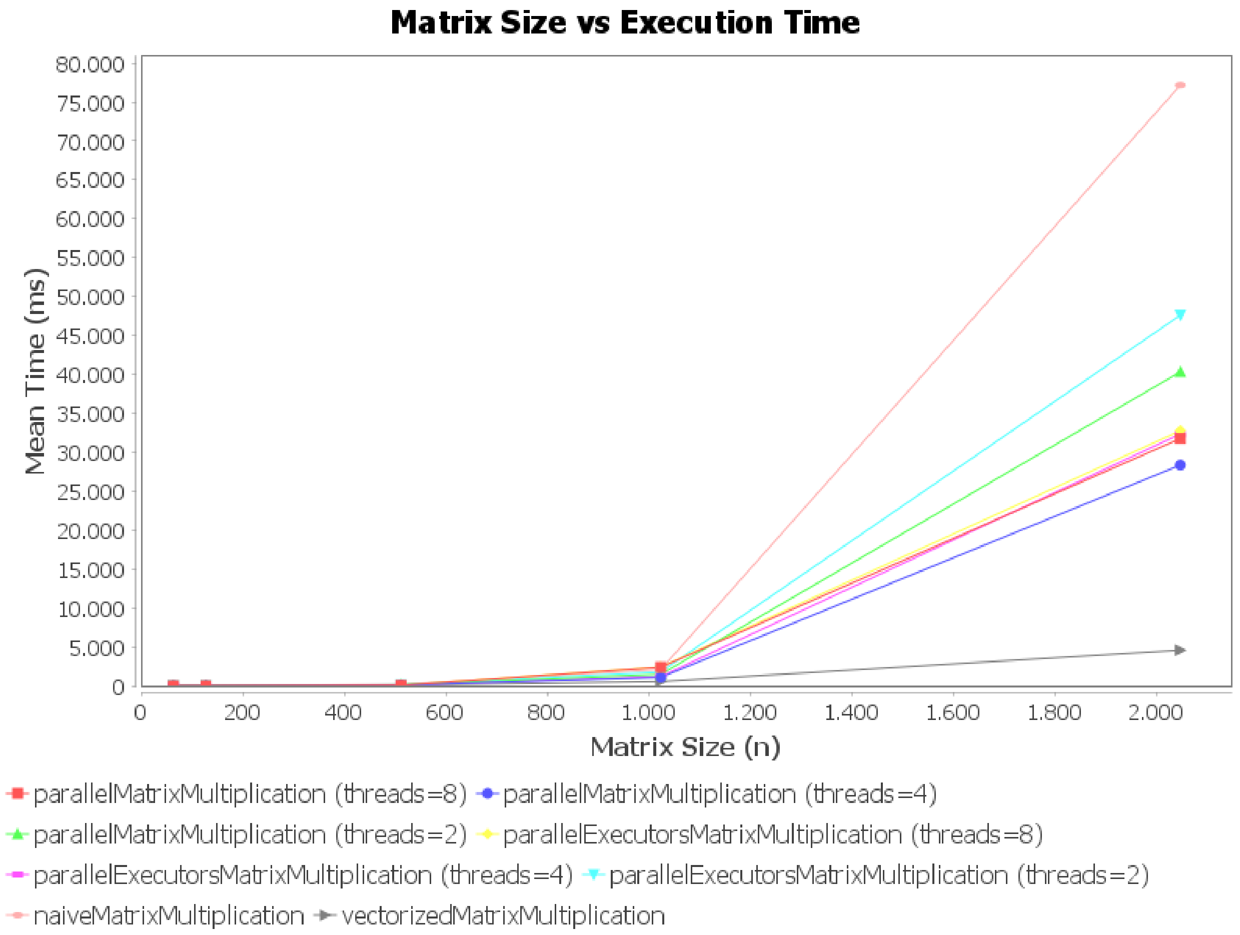

Figure 1: Execution Time Comparison

Table 1 gives a better overview of the behavior.

- **64x64:**
  - The *basic* algorithm (naive) performs better than the others due to the low computational cost, where the overhead of vectorized or parallel approaches is not justified.
  - The *vectorized* algorithm follows the naive in performance, while parallel algorithms, particularly the 2-threaded one, have an advantage over configurations with more threads or those using executors.
- **128x128:**
  - The *vectorized* outperforms the naive, indicating that optimizations in memory access and SIMD instructions are starting to show results.
  - In parallel algorithms, the *executors* gain advantage, and 8 threads turns out to be the most efficient configuration, possibly because they distribute the work better.
- **512x512:**
  - Parallel and vectorized algorithms are dominant. Both the *parallel with 8 threads* and vectorized approaches show a notable advantage, indicating that parallel and SIMD optimizations efficiently utilize computational resources for medium-sized matrices.
- **1024x1024 and 2048x2048:**

- The vectorized one is the most efficient, outperforming the others, possibly thanks to its ability to maintain high performance by means of SIMD operations and memory optimizations.
- The naive, on the other hand, becomes the worst in this case, since the absence of optimizations does not allow it to scale in the presence of increasing computational demands.

Table 1: Matrix Multiplication Benchmark Results

Algorithm	**Matrix Size**	**Number of Threads**	**Mean Execution Time (ms)**
Naive	64	-	0.2093
Parallel	64	2	0.3218
Parallel	64	4	0.3920
Parallel	64	8	0.6556
Executors	64	2	0.4058
Executors	64	4	0.4577
Executors	64	8	0.7019
Vectorized	64	-	0.2384
Naive	128	-	1.9160
Parallel	128	2	1.7551
Parallel	128	4	1.6083
Parallel	128	8	1.1319
Executors	128	2	1.4810
Executors	128	4	1.3944
Executors	128	8	1.2294
Vectorized	128	-	1.1025
Naive	512	-	169.8767
Parallel	512	2	114.7738
Parallel	512	4	75.4964
Parallel	512	8	60.0718
Executors	512	2	131.8042
Executors	512	4	83.5178
Executors	512	8	62.7806
Vectorized	512	-	65.3049
Naive	1024	-	2080.7164
Parallel	1024	2	1456.0443
Parallel	1024	4	1073.5301
Parallel	1024	8	2370.6405
Executors	1024	2	1764.4124
Executors	1024	4	1173.7998
Executors	1024	8	2403.5499
Vectorized	1024	-	532.0686
Naive	2048	-	77161.0016
Parallel	2048	2	40324.6224
Parallel	2048	4	28349.5935
Parallel	2048	8	31758.3743
Executors	2048	2	47609.0596
Executors	2048	4	32343.0477
Executors	2048	8	32767.6192
Vectorized	2048	-	4561.0511

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 16.9174, 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. When we add more threads, they do not take advantage of their full capacity, so the efficiency decreases.

Table 2: Speedup and Efficiency for Matrix Multiplication

Algorithm	**Matrix Size**	**Number of Threads**	**Speedup**	**Efficiency**
Parallel	64	2	0.6504	0.3254
Parallel	64	4	0.5339	0.1335
Parallel	64	8	0.3192	0.0399
Executors	64	2	0.5158	0.2579
Executors	64	4	0.4573	0.1143
Executors	64	8	0.2982	0.0373
Vectorized	64	-	0.8779	-
Parallel	128	2	1.0917	0.5458
Parallel	128	4	1.1913	0.2978
Parallel	128	8	1.6927	0.2116
Executors	128	2	1.2937	0.6469
Executors	128	4	1.3741	0.3435
Executors	128	8	1.5585	0.1948
Vectorized	128	-	1.7379	-
Parallel	512	2	1.4801	0.7400
Parallel	512	4	2.2501	0.5625
Parallel	512	8	2.8279	0.3535
Executors	512	2	1.2889	0.6444
Executors	512	4	2.0340	0.5085
Executors	512	8	2.7059	0.3382
Vectorized	512	-	2.6013	-
Parallel	1024	2	1.4290	0.7145
Parallel	1024	4	1.9382	0.4846
Parallel	1024	8	0.8777	0.1097
Executors	1024	2	1.1793	0.5896
Executors	1024	4	1.7726	0.4432

Continued on next page...

Algorithm	**Matrix Size**	**Number of Threads**	**Speedup**	**Efficiency**
Executors	1024	8	0.8657	0.1082
Vectorized	1024	-	3.9106	-
Parallel	2048	2	1.9135	0.9567
Parallel	2048	4	2.7218	0.6804
Parallel	2048	8	2.4296	0.3037
Executors	2048	2	1.6207	0.8104
Executors	2048	4	2.3857	0.5964
Executors	2048	8	2.3548	0.2943
Vectorized	2048	-	16.9174	-

The results reflect that runtime-based performance depends significantly on the size of the matrix. For small sizes, naive remains competitive due to its simplicity, but as the size increases, more advanced techniques such as vectorization and parallelization show clear advantages. This highlights the importance of selecting the right approach depending on the size of the data and the available hardware. However, as our interest is to find the most optimal algorithm to apply on large volumes of data, those that improve the basic one will benefit us.

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, the *naive* algorithm demonstrates the best memory efficiency, using the least amount of memory. This is likely because it does not rely on complex data structures or additional memory overhead. The other algorithms, including the parallel ones with 2, 4, and 8 threads, show similar memory usage, as they scale efficiently without introducing significant overhead.

However, the *vectorized algorithm* uses more memory compared to all other approaches, particularly as the matrix size increases. This could be due to the additional memory required for precomputations, data alignment, and temporary storage during vectorized operations, which becomes more significant with larger matrices.

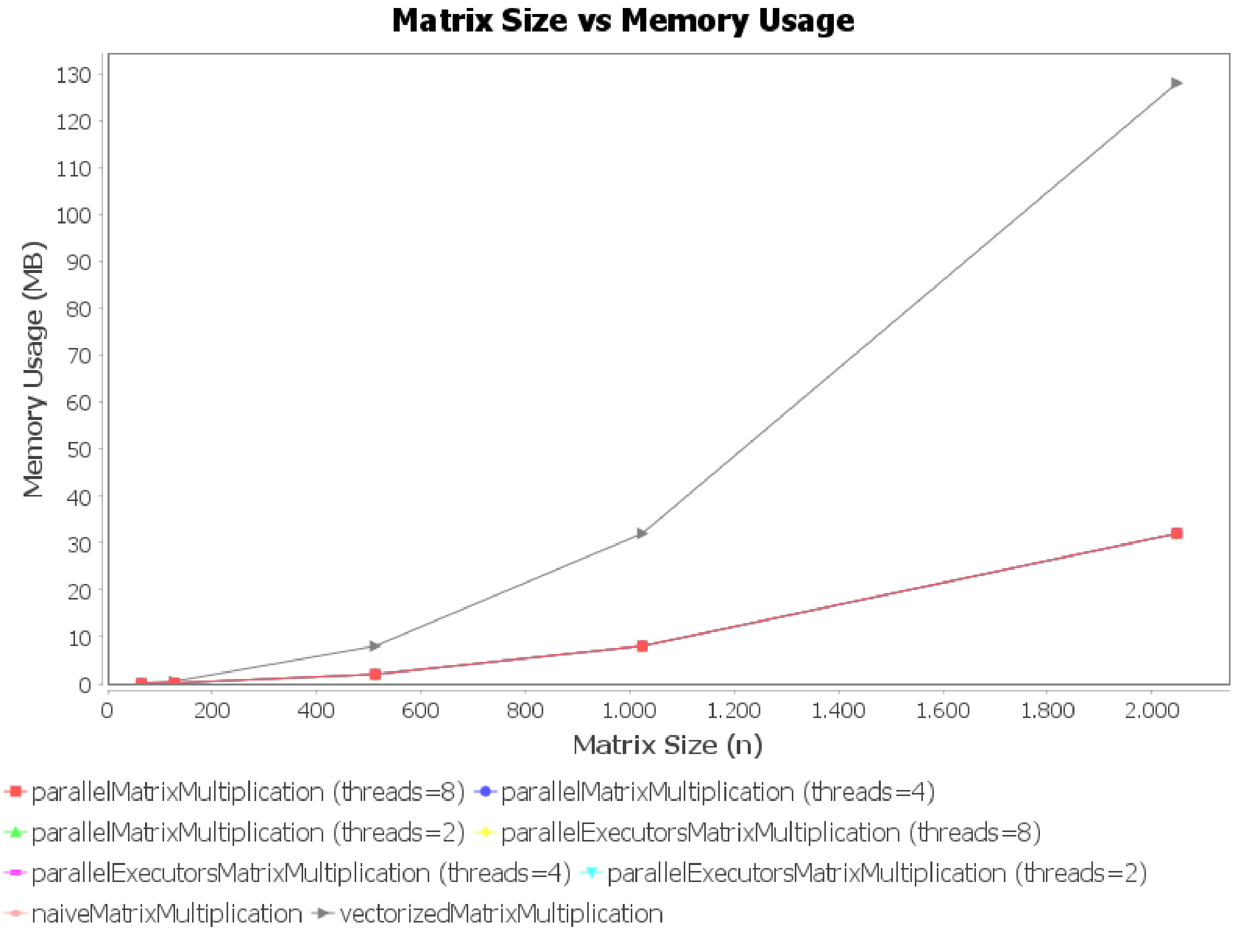

Figure 2: Memory Usage Comaprison

In conclusion, the vectorized algorithm is the least efficient in terms of memory, as it uses the most memory at all matrix sizes, especially as the matrix size increases. However, it achieves better results in terms of execution time compared to the parallel and naive algorithms, making it a good choice for performance optimization when memory usage is not a big issue. It offers a good balance between runtime and speedup, however at the cost of higher memory consumption.

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 is the least efficient, using the most memory across all matrix sizes tested. This makes it less suitable for memory-constrained environments.

When runtime is considered, the vectorized algorithm demonstrates superior performance, significantly outperforming the other methods as the matrix size increases.

The parallel algorithms also exhibit a clear advantage over the naive implementation as matrix size grows, but they fall short of the vectorized method. Among the parallel implementations, the algorithm utilizing 2 threads achieves the best efficiency, effectively balancing workload distribution and minimizing overhead, resulting in optimal resource utilization. However, as more threads are introduced, their full capacity is not utilized effectively, leading to a decrease in efficiency. For instance, the 8-thread algorithm, while still faster than the naive implementation, performs worse than the 4-thread version, likely due to increased overhead and synchronization costs.

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 execution time.

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