Performance Benchmark of matrix multiplication

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

November 2024

Abstract

This project evaluates optimised matrix multiplication methods to handle the high computational costs associated with large matrices, which are crucial in fields such as computer science and engineering. We analyse several algorithms, including naive, blocking, Strassen and sparse matrix formats such as CSR, CSC and COO, evaluating their execution time and memory usage at different sparsity levels. Through Java implementations and comparative tests with JMH, CSR, CSC and locking appeared as the most efficient methods, achieving a reliable balance between speed and memory usage. These three approaches not only performed well with dense and sparse matrixes, but also demonstrated their scalability, handling large matrices efficiently. Although Strassen's algorithm was more time-consuming than expected, the CSR, CSC and blocking methods maintained efficiency even with large matrices.

Contents

List of Figures

1 Introduction

Matrix multiplication is a fundamental operation in a number of scientific and engineering disciplines, providing the basis for a wide range of applications across computer science, physics, engineering, and data analysis. As a fundamental mathematical process, matrix multiplication is essential for complex computations in fields such as machine learning, image processing, and scientific simulations. The challenge is that as matrices grow in size, the computational cost of multiplication increases substantially. For large matrices, the standard approach has a time complexity of $O(n^3)$, which is not viable in terms of both time and memory. Therefore, developing efficient algorithms to handle large-scale matrix multiplication is crucial for the performance and feasibility of many applications.

Therefore, the aim is to find other matrix multiplication algorithms capable of performing this task in less time and at lower computational cost. Some algorithms have already been evaluated in the previous stage, such as the basic or naive algorithm and the blocking algorithm; and others have been added, such as the Strassen algorithm and the loop unrolling algorithm. Likewise, the behaviour of algorithms with different sparsity values will be evaluated, as well as algorithms designed to work with sparse matrices such as CSC, CSR and COO.

2 Problem Statement

Given the importance of choosing a good algorithm to reduce costs, in this study we will use Java to implement different algorithms and use benchmarking tools to measure execution time as well as memory usage. On the other hand, we will look for computational limits in the algorithms, if they exist. In addition, we will compare the behaviour of these algorithms with both dense and sparse matrices.

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 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.

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.
- **BlockMatrixMultiplication**: This class implements block matrix multiplication, which divides the matrices into smaller blocks to optimize cache usage and reduce memory latency. The `multiply()` method iterates over the matrix in blocks of size *block_size* and calls `multiplyBlock()` to handle the multiplication of each block efficiently.
- **StrassenMatrixMultiplication**: This class uses the Strassen's matrix multiplication algorithm that recursively divides the matrix into sub-matrices and combines the results, as explained on javatpont's website (1).

- **LoopUnrollingMatrixMultiplication**: This class optimises computation by *loop unrolling*. It expands the inner loop to reduce the number of iterations and improve efficiency. The `multiply()` function combines multiple operations into a single iteration of the loop, which reduces loop overhead and improves performance in systems with optimised instruction pipelines.

Sparse matrix multiplication algorithms are also implemented:

- **SparseMatrixCOOMul**: This class multiplies a matrix represented by rows of non-zero values and a matrix represented by columns and the result is expressed in COO format.
- **SparseMatrixCSCMul**: This class utilises the Compressed Sparse Column (CSC) format for multiplication. The CSC format stores non-zero values by columns, making it an efficient solution for column-based operations, particularly for sparse matrices with clustered non-zero elements.
- **SparseMatrixCSRMul**: This class performs multiplication using the Compressed Sparse Row (CSR) format, which is designed to store non-zero values row by row. The CSR format is particularly well-suited for row-based operations and is commonly used in applications where matrices are large and sparse.

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 sparsity levels 0.0, 0.5 and 0.9 are compared.

First, looking at graphs 1, 2 and 3, we can see the execution time according to the size of the matrix with sparsity 0.0, 0.5 and 0.9, respectively, for the different algorithms. We can see that Strassen's algorithm (the blue line) has by far the longest execution time.

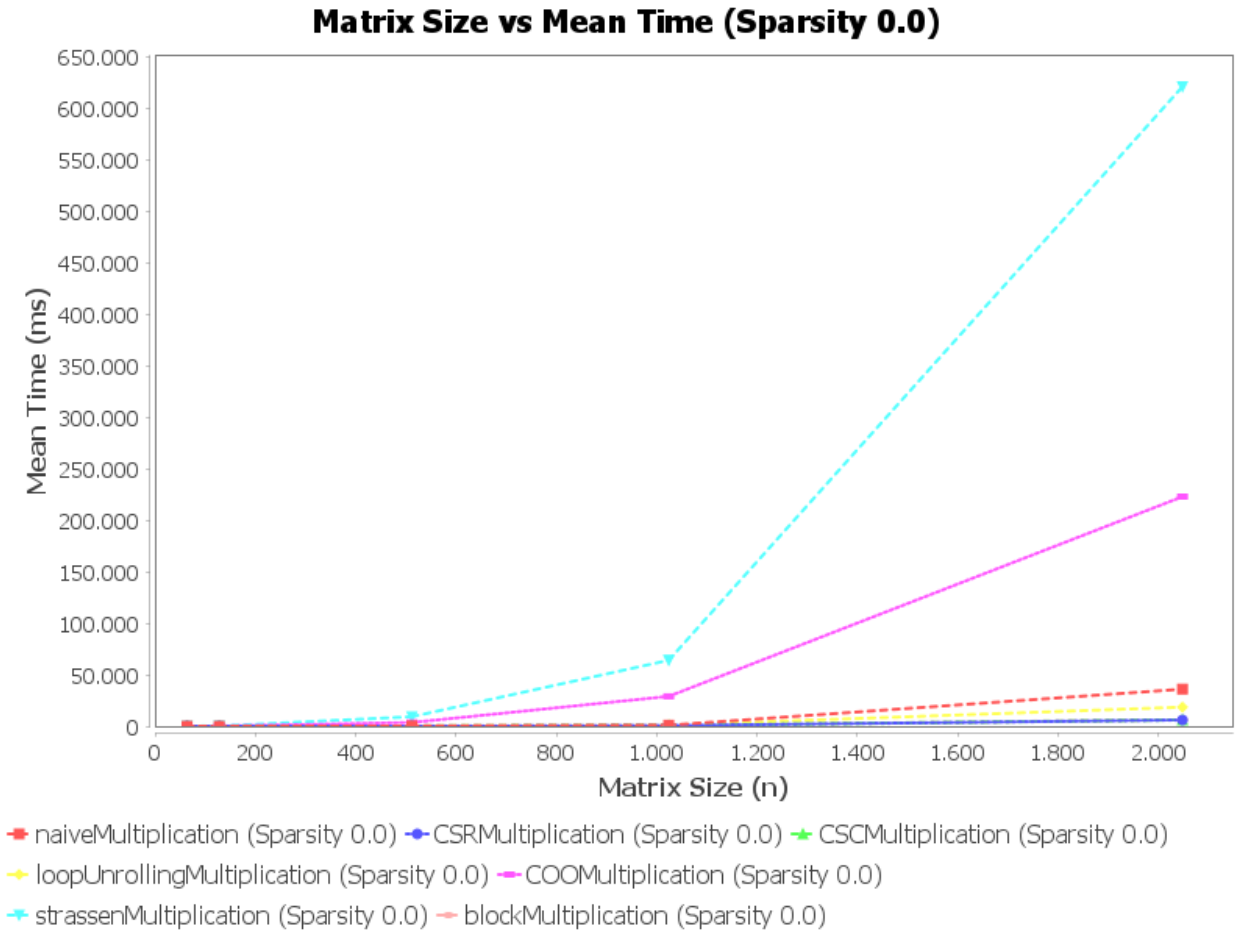

Figure 1: Time Execution with sparsity 0.0

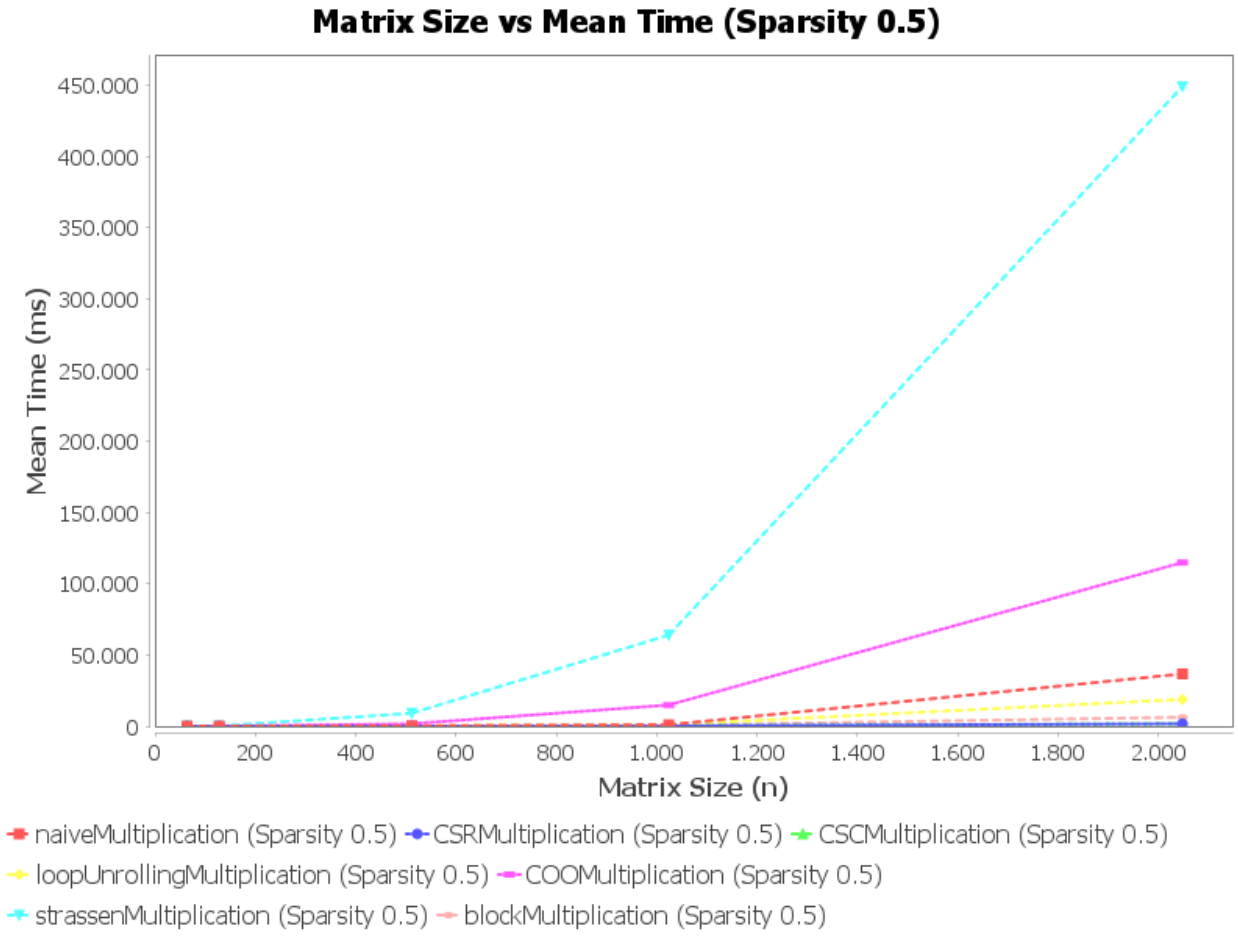

Figure 2: Time Execution with sparsity 0.5

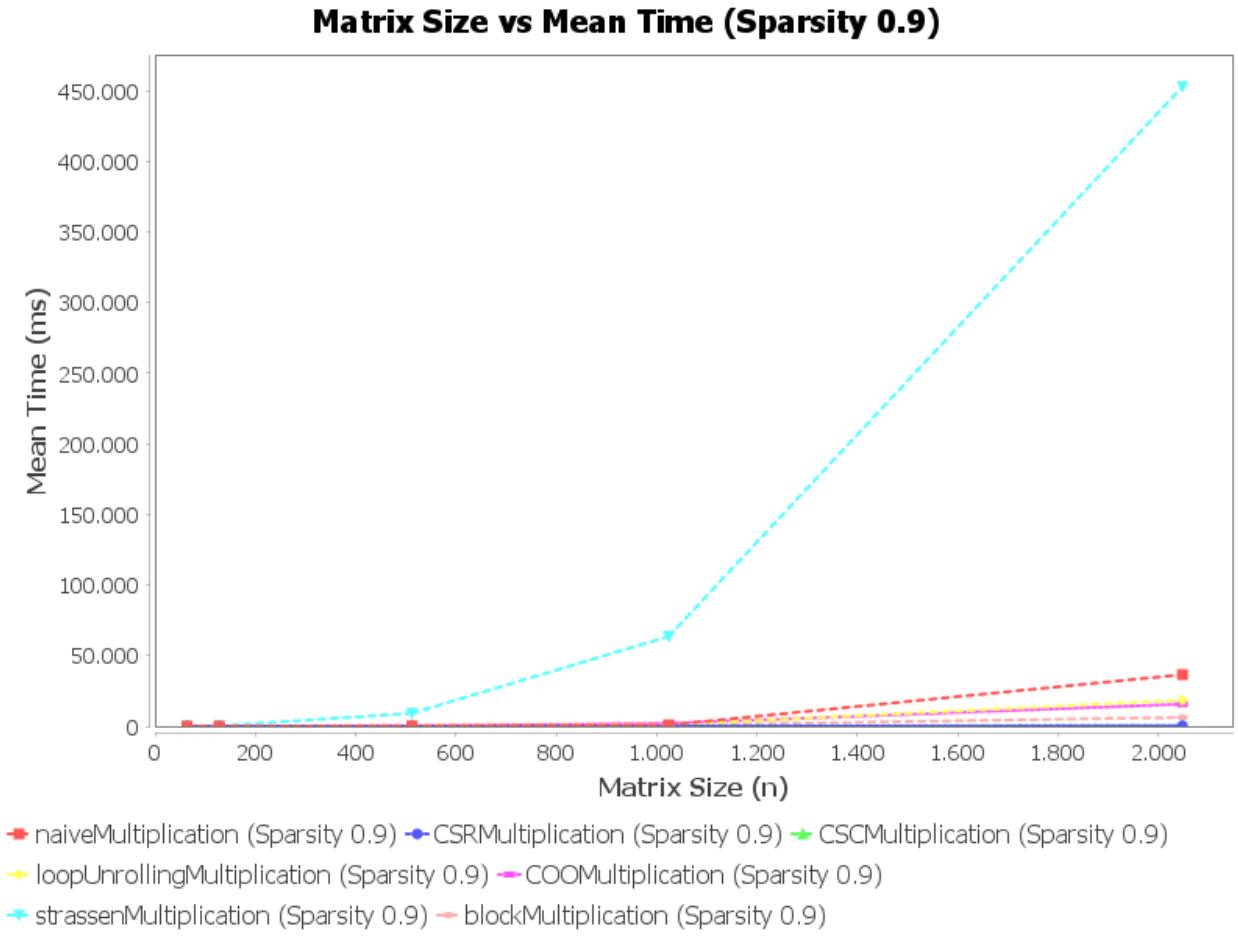

Figure 3: Time Execution with sparsity 0.9

Therefore, we will discard this algorithm since for larger matrices the execution time will increase and we have other algorithms that improve this result. As we can now see in figure 4, the COO algorithm is the worst performer with sparsity 0.0, which corresponds to a dense matrix. However, in figure 6 we observe a great improvement of this algorithm due to the fact that the sparsity level is 0.9, that is, the matrix contains quite a lot of zeros and this algorithm ignores these values. We also observe that the CSR and CSC algorithms are the best in all measurements, and with quite small times. Then the blocking algorithm obtains very good results as well.

One observation we can make is that the change between 0.0 and 0.5 is not very significant for the algorithms, as they keep more or less the same, as can be seen in figures 4 and 5.

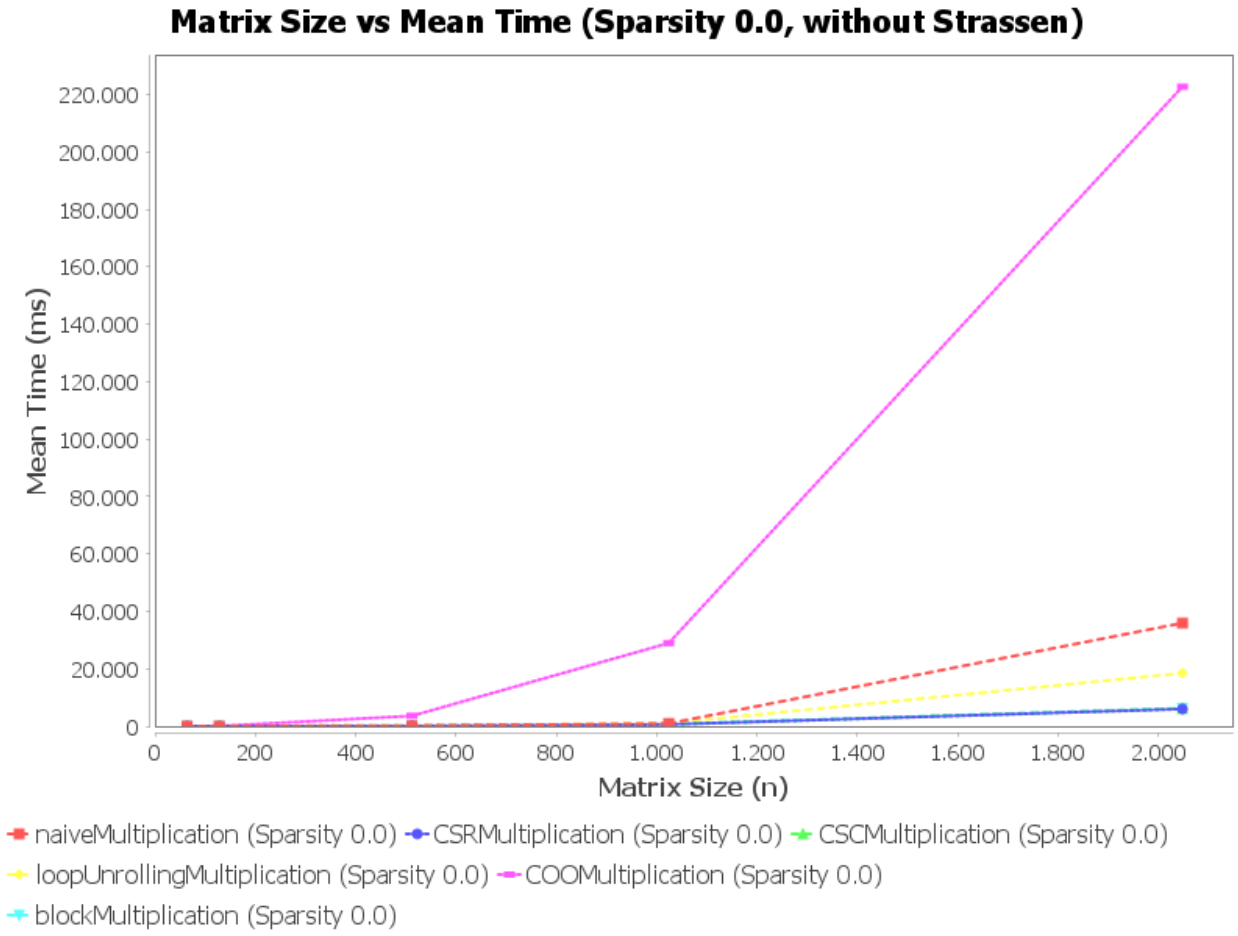

Figure 4: Time Execution with sparsity 0.0 and without Strassen's algorithm

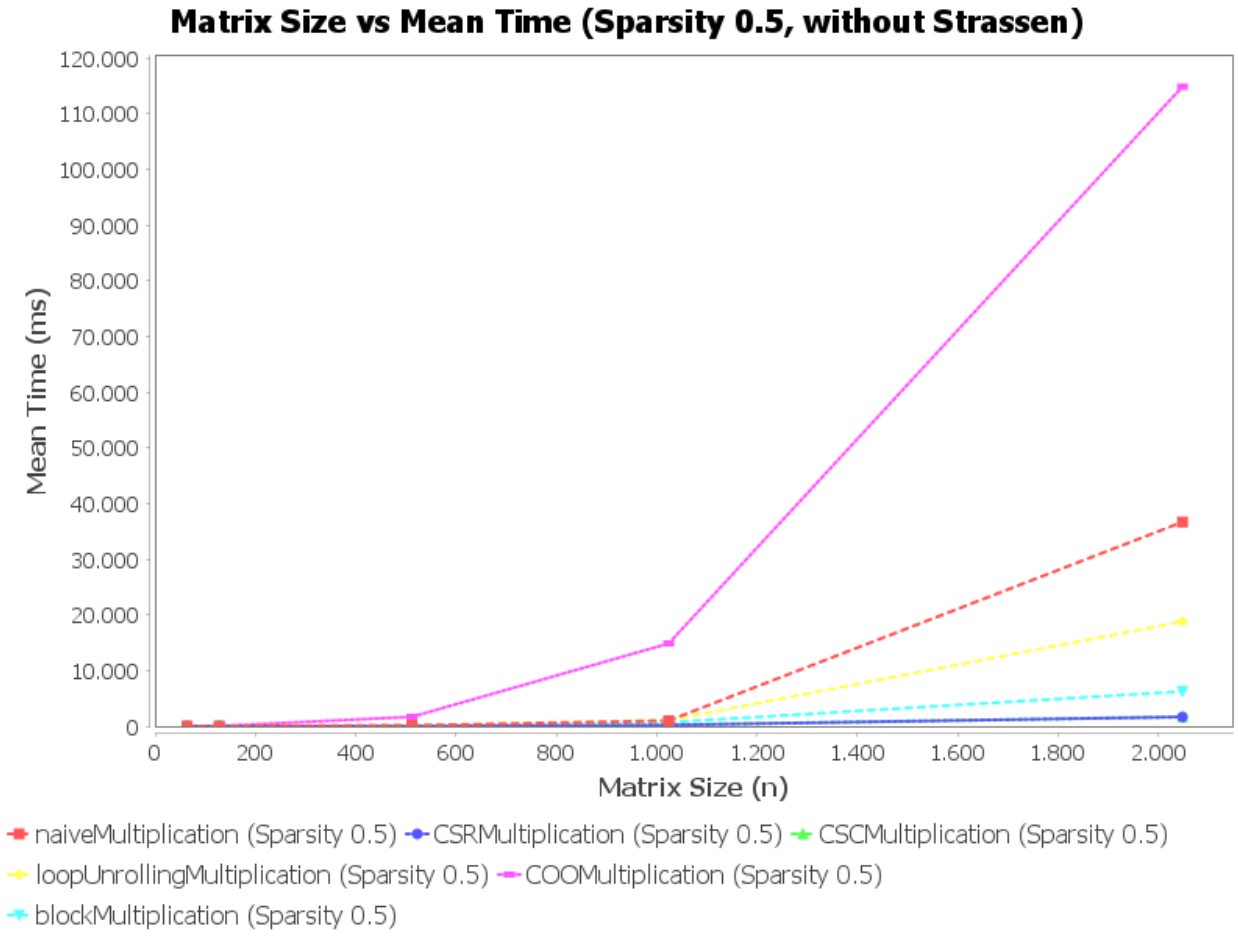

Figure 5: Time Execution with sparsity 0.5 and without Strassen's algorithm

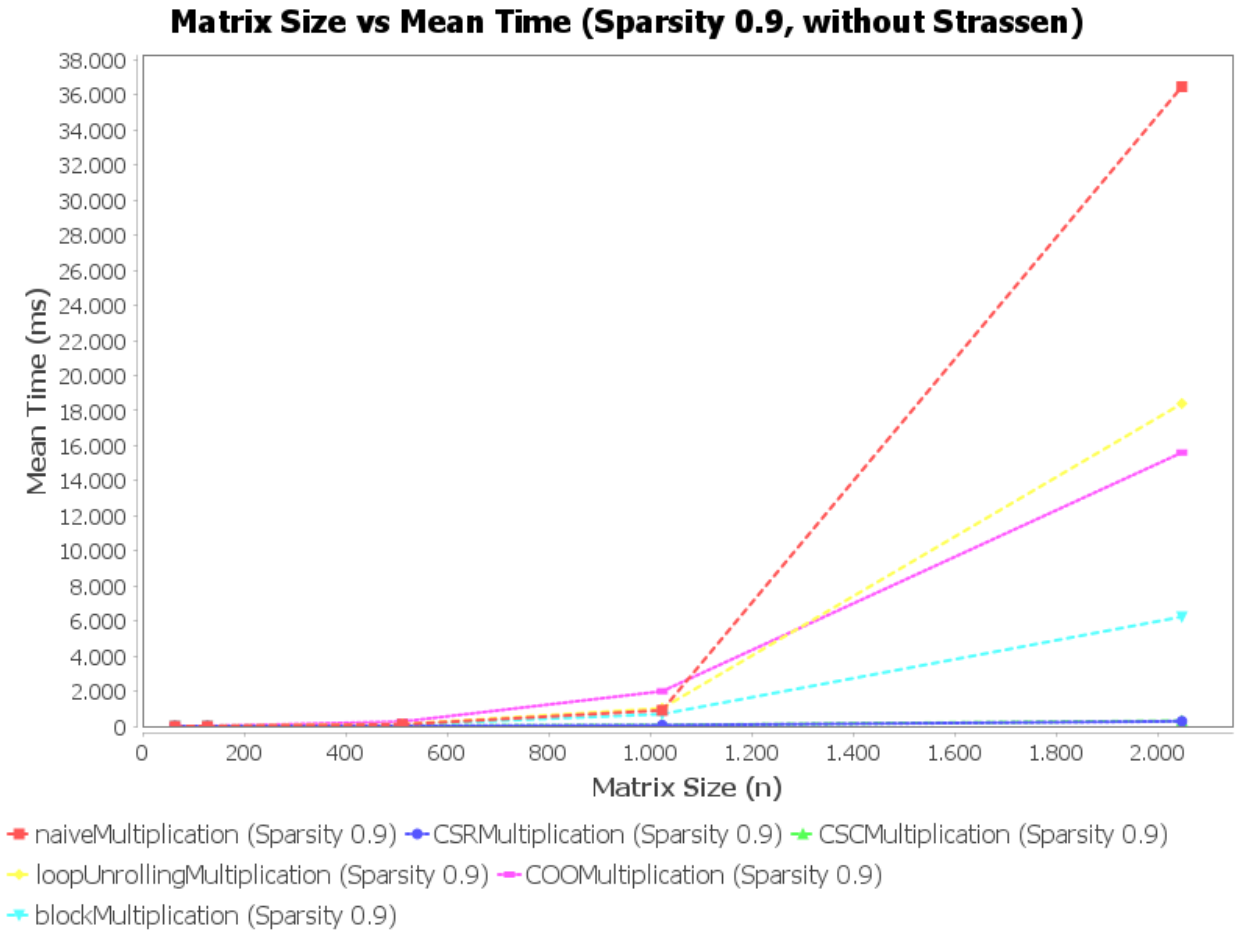

Figure 6: Time Execution with sparsity 0.9 and without Strassen's algorithm

In terms of memory, several comparative graphs have been generated showing memory usage in relation to different algorithms and sparsity configurations. 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 an optimisation in one of these aspects may imply an increase in the other.

According to Figure 7, it is evident that, in terms of memory efficiency, naive and blocking algorithms have the best performance, utilizing the least amount of memory. Next, CSR and CSC algorithms perform similarly. Finally, COO algorithm needs more memory than the others.

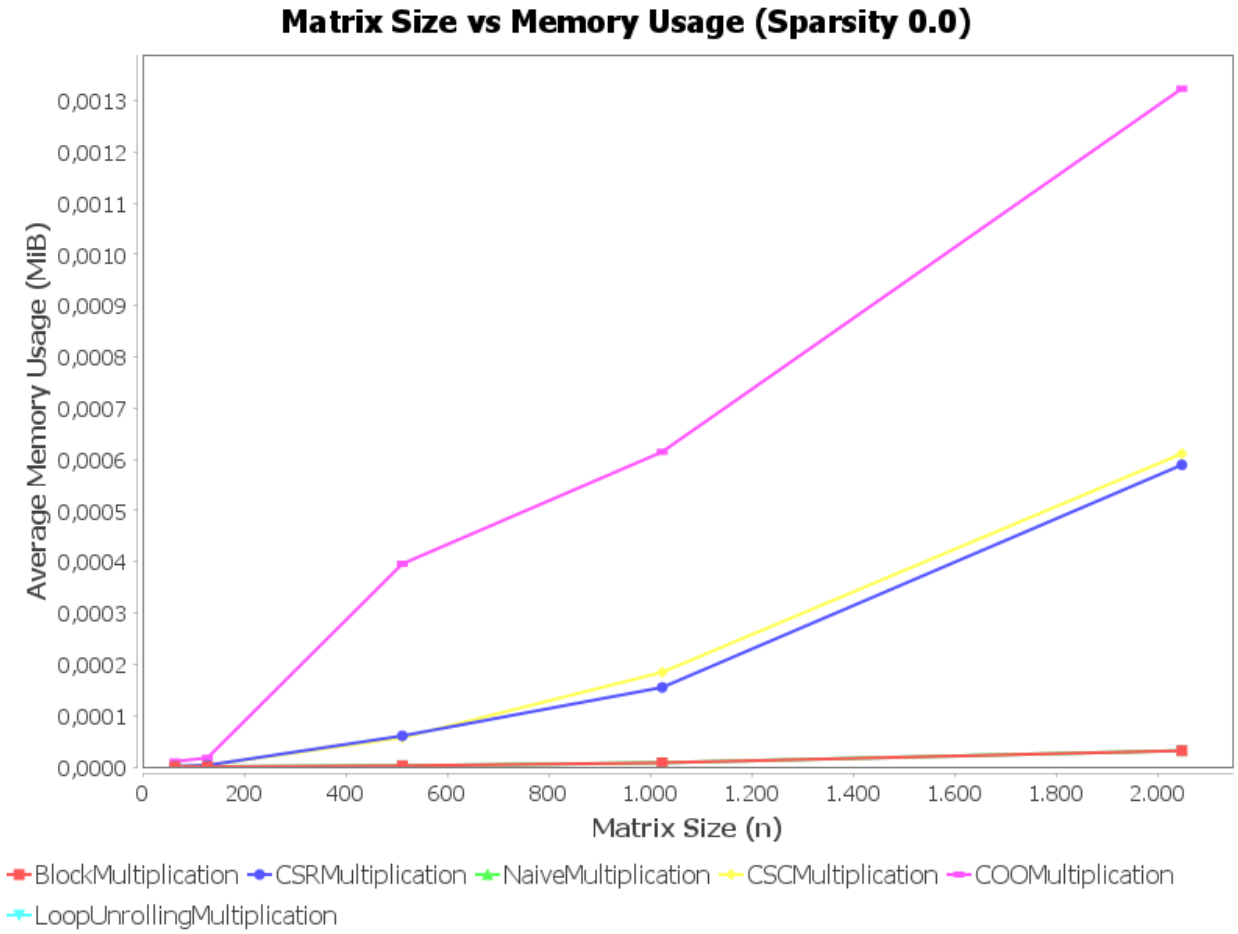

Figure 7: Memory Usage with sparsity 0.0

The following figure 8, with a sparsity of 0.5, shows a similar behavior to the one with sparsity 0.0, just as we saw with the execution time.

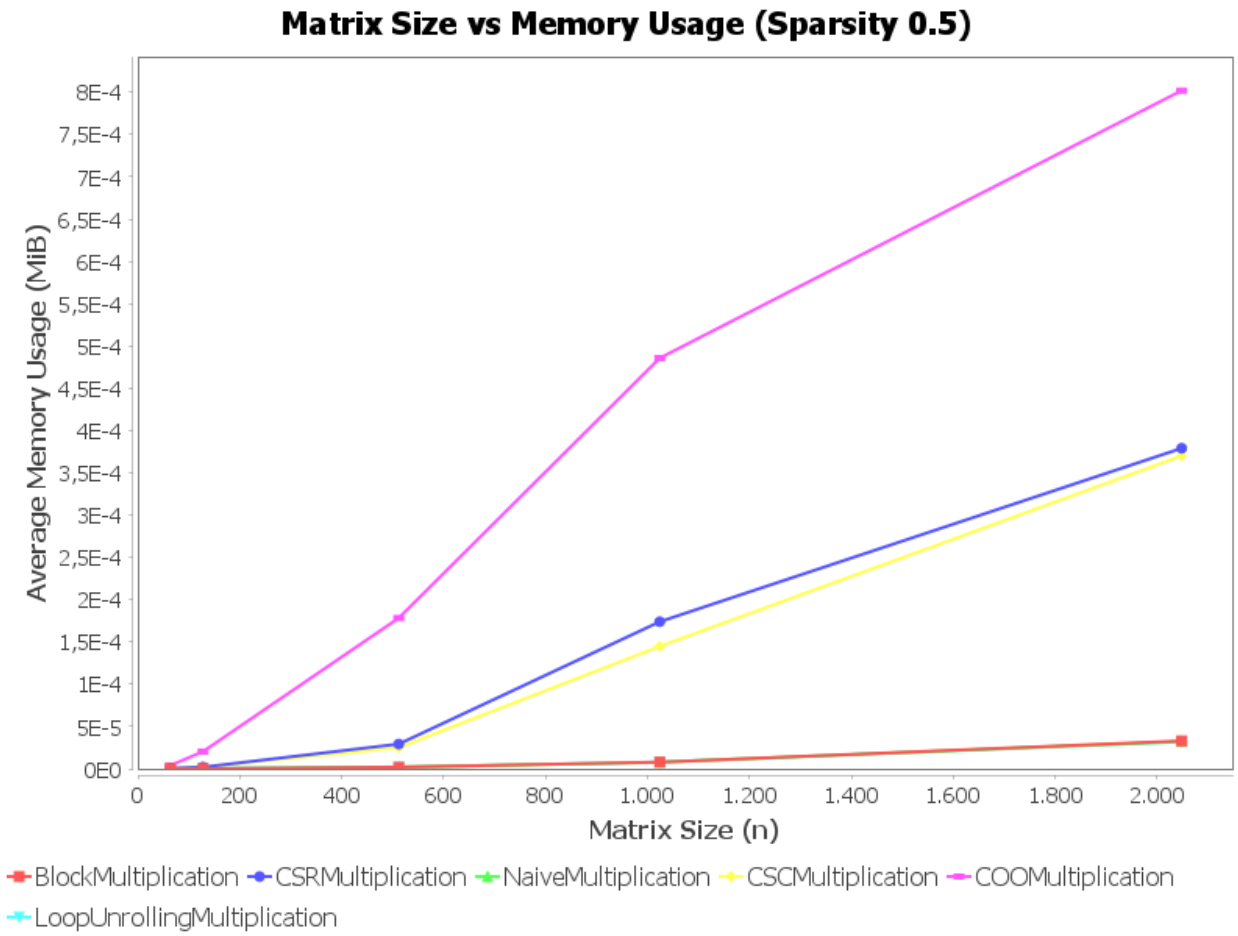

Figure 8: Memory Usage with sparsity 0.5

Finally, in the figure 9 we observe that, in general, memory usage is reduced, especially in the CSC and CSR algorithms. These methods show a noticeable decrease in memory consumption as the sparsity level increases. This trend suggests that both the CSC and CSR algorithms are more memory efficient when the matrices are sparse, making them suitable for applications that require manipulating large amounts of data with a high proportion of zeros.

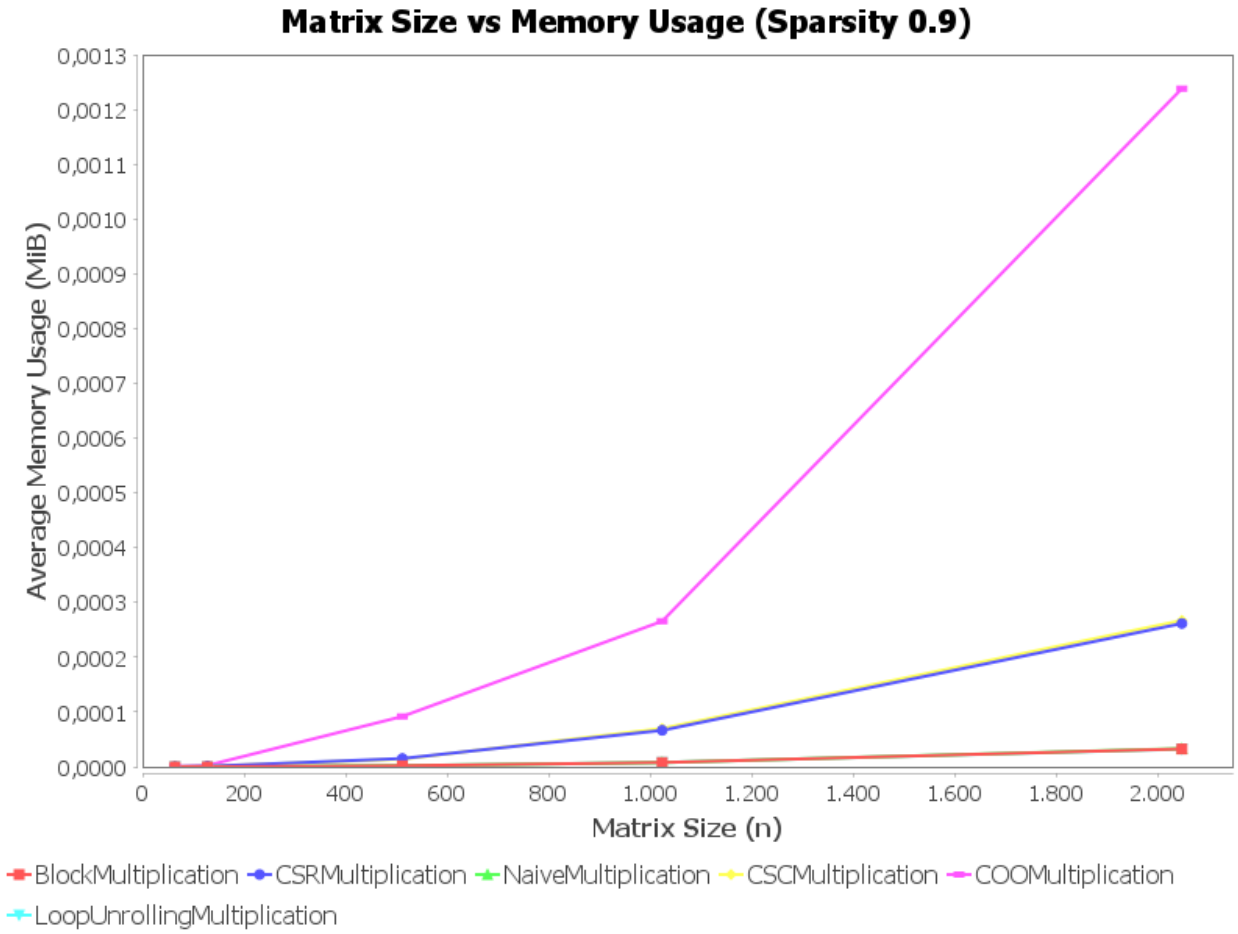

Figure 9: Memory Usage with sparsity 0.9

In conclusion, the most stable and efficient methods for balancing execution time and memory usage in Java are the CSR (Compressed Sparse Row), the CSC (Compressed Sparse Column) and the blocking matrix multiplication method. These methods not only provide consistent performance on arrays of different sizes, but also optimise memory usage, which makes them well suited for large-scale matrix operations. CSR and CSC, in particular, are well suited to sparse matrices, while the locking method improves execution time by taking advantage of cache locality. Together, these methods offer a reliable trade-off between speed and memory efficiency in matrix multiplication tasks.

5 Conclusion

In this work, we tested several matrix multiplication methods, looking at both execution time and memory usage for dense and sparse matrices. The methods that worked best were CSR (Compressed Sparse Row), CSC (Compressed Sparse Column), and the blocking method. These approaches showed strong performance and efficient memory usage, even for large matrices. CSR and CSC worked especially well with sparse matrices, where many elements are zero, and the blocking method was faster by using cache memory effectively.

On the other hand, Strassen's algorithm was less useful because its execution time was very high for large matrices (sizes 1024 and 2048), which would make it impractical for even larger datasets. The COO method showed promise, but it would need more data to match the results of Strassen's algorithm. Meanwhile, CSR, CSC, and blocking performed so well with 2048-sized matrices that they are likely to stay efficient as matrix size increases, making them the most reliable options for larger-scale matrix operations.

6 Future Work

For future research, more advanced parallelisation algorithms that take advantage of multi-core architectures and GPU processing can be explored. GPU implementation would significantly reduce execution time on large matrices, as this architecture is designed to perform intensive mathematical operations in a massively parallel fashion. In addition, combining parallelisation techniques with specific optimisations, such as block processing and automatic adaptation to matrix density, could provide even greater efficiency in data-intensive environments with a variety of sparse characteristics.

7 Bibliography and References

References

[1] JavaTpoint, "Strassen's matrix multiplication," n.d., accessed: 2024-11-06. [Online]. Available: https://www.javatpoint.com/strassens-matrix-multiplication