Optimized Matrix Multiplication Approaches and Sparse Matrices

DIEGO JESÚS TORREJÓN CABRERA

November 2024

GitHub link: Optimized-Matrix-Multiplication-Approaches-and-Sparse-Matrices

1 Abstract

This study addresses the challenge of optimizing matrix multiplication, a computationally intensive operation with broad applications. Traditional dense matrix multiplication can be inefficient, particularly with large matrices or those containing many zero elements. To improve performance, we investigated alternative multiplication techniques, including sparse matrix representations (CSC and CSR formats) and optimizations like block and loop unrolling in dense matrices. Experiments were conducted on matrices of increasing sizes (from 256x256 to 1764x1764) and varying sparsity levels, allowing comparison across execution times and memory usage.

Results show that sparse matrices, while sometimes using more memory than dense ones at lower sparsity levels, achieve faster execution times by skipping operations on zero elements. Notably, using the CSR format, a very large matrix (525,825 x 525,825) was multiplied in just 4 minutes, confirming the efficiency of sparse formats for massive datasets. For dense matrices, the block method proved optimal for larger sizes up to 7056 x 7056. Overall, sparse matrix multiplication in high sparsity and large-scale scenarios offers significant time savings, making it advantageous for time-critical applications. This study underscores the trade-off between execution time and memory use, with sparse matrices emerging as highly effective for scalable operations.

2 Introduction

Matrix multiplication is a fundamental operation in various fields of computer science, used in applications such as graphics processing, machine learning and scientific simulations. As matrix sizes and the amount of data to be processed increase, the efficiency of matrix multiplication algorithms becomes critical to reduce computational costs and improve performance. The optimization of this operation has been widely studied and has given rise to different approaches, ranging from improvements in the basic algorithms to advanced optimization techniques in the use of caching and the efficient handling of sparse matrices.

Among the approaches proposed by the literature, Strassen's algorithm is notable for its ability to reduce runtime complexity compared to conventional matrix multiplication, albeit at the cost of higher implementation complexity. Other authors have explored loop unrolling techniques, which allow better exploitation of instruction-level parallelism, as well as cache optimizations, which minimize memory access and improve efficiency on modern processors. In addition, the use of sparse arrays represents an efficient strategy when most of the elements in the arrays are zeros, thus reducing the number of operations required and memory consumption.

This paper presents an implementation of several optimized approaches to matrix multiplication, including a block algorithm that optimizes cache usage, a version that employs loop unrolling to improve instruction-level parallelism, and approaches for sparse matrices using the Column and Row Compression (CSC and CSR) format. The contribution of this project lies in a comprehensive analysis of how these optimization techniques and the level of matrix sparsity affect computational performance, providing a practical and quantitative view of efficiency scaling in large-scale matrix multiplication.

3 Methodology

To address the problem of optimized matrix multiplication, different algorithms have been implemented and evaluated using matrices of increasing sizes: 256x256, 576x576, 784x784, 1024x1024 and 1764x1764. The experiments were designed to measure and compare both the execution time and memory usage of each approach, with the goal of understanding their efficiency and scalability as the size of the arrays grows.

First, a benchmark has been implemented to compare the performance of three methods of dense matrix multiplication: a classical block algorithm, which optimizes cache usage by dividing the matrices into sub-matrices; a loop unrolling algorithm that improves instruction-level parallelism; and a basic approach that serves as a reference. Each of these implementations is executed and evaluated on arrays of the specified sizes, recording the execution time and memory used for each case.

In addition, a second benchmark has been designed to evaluate the multiplication of sparse matrices, stored in Column Compression (CSC) and Row Compression (CSR) format. In this case, experiments are performed with different sparsity levels, that is, different percentages of null elements in the matrix, allowing to analyze how the sparsity level affects the performance. The sparsity levels have been adjusted in a controlled manner to reflect different usage scenarios, from less sparse matrices to matrices with a high proportion of zeros. The execution time and memory used in each test are also recorded to compare the impact of sparsity on the efficiency of the algorithm.

Additionally, to further assess the scalability and efficiency of sparse matrix multiplication, a third benchmark has been conducted using two extremely large sparse matrices of size 525,825 x 525,825, containing 2,100,225 non-zero elements. For this test, both CSC and CSR formats were used to determine which provides the most efficient execution time with matrices of such magnitude. In contrast, for dense matrix multiplication optimizations (such as block and loop unrolling), we used matrices of size 7056 x 7056, as this is the maximum size supported by JMH for dense matrices in our setup. By comparing the performance on these substantial matrix sizes, we aim to evaluate the feasibility of optimized sparse and dense multiplication approaches in handling large-scale data efficiently, particularly in terms of execution time. This assessment provides insights into the potential of sparse matrix formats for real-world applications that involve massive datasets where computational efficiency is critical.

The experiments were performed on a MacBook Pro equipped with an Apple Silicon M3 Pro chip, which includes 11 CPU cores and 18 GB of unified memory. Additionally, for each benchmark, JMH has been used, using 5 warm-up iterations and 10 measurement iterations. The methodology described here provides a reproducible framework for comparing different optimization techniques in matrix multiplication, evaluating performance in terms of time and memory for different size and sparsity configurations.

4 Experiments

This section presents the experiments performed to evaluate the performance of the implemented matrix multiplication algorithms, both for dense and sparse matrices. Each subsection details the experiments aimed at measuring execution time and memory usage, with the objective of analyzing the efficiency of the optimizations applied in each case.

4.1 Dense matrices

For dense matrix analysis, experiments have been designed to compare the performance of three versions of matrix multiplication:

- **Block matrix multiplication:** This method divides arrays into smaller subarrays, allowing more efficient cache access and reducing the cost of memory accesses.
- **Loop unrolling matrix multiplication:** This approach optimizes instruction-level parallelism, reducing the number of loop control operations and enabling faster execution.
- **Original matrix multiplication:** This method serves as a reference to evaluate the improvements achieved through optimization techniques.

Each of these experiments will be run on the arrays mentioned in the previous section, providing a comprehensive comparison of efficiency as a function of array size and algorithm used.

4.1.1 Execution time

Size (n x n)	**Matrix multiplication(s)**	**Block matrix multiplication(s)**	**Loop unrolling matrix multiplication (s)**
256 x 256	0.011	0.008	0.006
576 x 576	0.129	0.090	0.085
784 x 784	0.352	0.225	0.366
1024 x 1024	1.6	0.518	1.392
1764 x 1764	8.979	2.861	7.945

Table 1: Execution time for different matrix multiplication algorithm

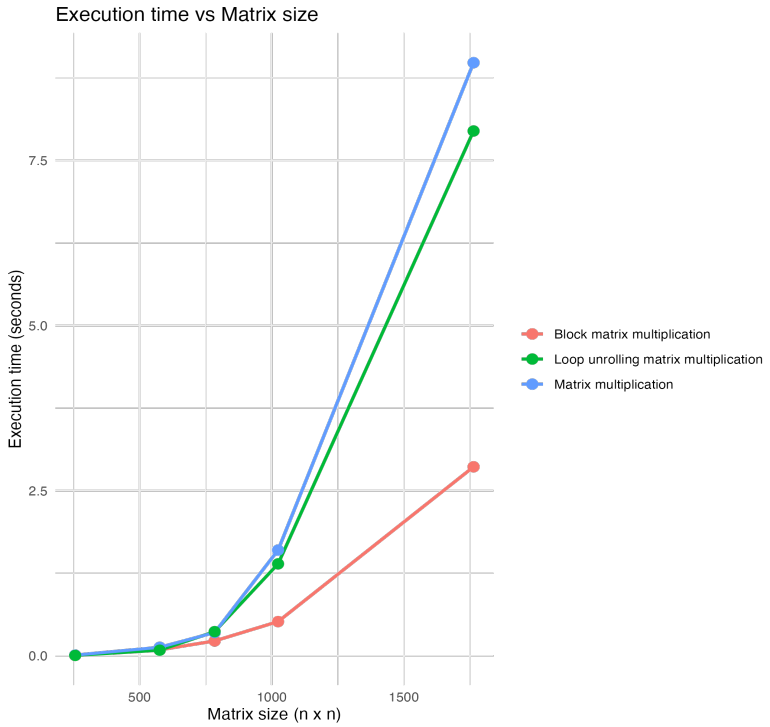

Figure 1: Graph that shows the execution time for different matrix multiplication algorithm

In table 1 and graph 1, we see how the difference in execution time for small matrices is minimal, with loop unrolling being the fastest, but as the size of the matrix increases, block multiplication begins to show significant advantages. For 1764x1764 matrices, block multiplication requires 2.861 seconds, compared to 8.979 seconds for the basic method and 7.945 seconds for loop unrolling.

From this we can deduce that, taking the basic matrix multiplication method as a reference, the block multiplication method is more efficient for large matrices due to its optimised cache management and that the loop unrolling method performs well for small matrices but does not scale well with larger sizes.

4.1.2 Memory used

Size (n x n)	Matrix multiplication(MB)	Block matrix multiplication(MB)	Loop unrolling matrix multiplication (MB)
256 x 256	6.350	6.345	6.343
576 x 576	12.870	12.872	12.883
784 x 784	19.800	19.726	19.812
1024 x 1024	29.700	29.635	29.684
1764 x 1764	78.700	78.546	78.432

Table 2: Memory usage for different matrix multiplication algorithm

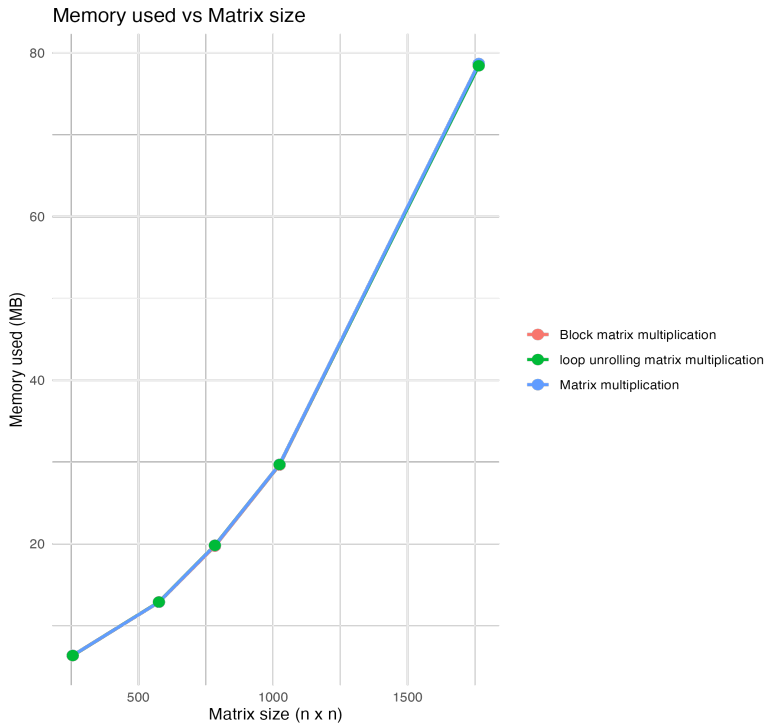

Figure 2: Graph that shows the memory used for different matrix multiplication algorithm

In table 2 and graph 2, we see how the block multiplication method shows a very slight advantage in terms of memory efficiency, but there is no noticeable difference between the three methods, so we conclude that these methods focus more on optimising execution time than memory usage.

4.2 Sparse matrices

For sparse matrices, the experiments focus on evaluating the multiplication efficiency of matrices stored in Column Compression (CSC) and Row Compression (CSR) format with different levels of sparsity, which implies varying the percentage of null elements in the matrix.

4.2.1 Execution time for CSC format

Size (n x n)	**Sparsity level = 30%(s)**	**Sparsity level = 50%(s)**	**Sparsity level = 70%(s)**	**Sparsity level = 90%(s)**
256 x 256	0.005	0.003	0.001	0.0001
576 x 576	0.045	0.032	0.018	0.009
784 x 784	0.106	0.065	0.040	0.025
1024 x 1024	0.226	0.138	0.082	0.032
1764 x 1764	1.082	0.625	0.271	0.121

Table 3: Execution time for different sparsity levels for sparse matrices with CSC format

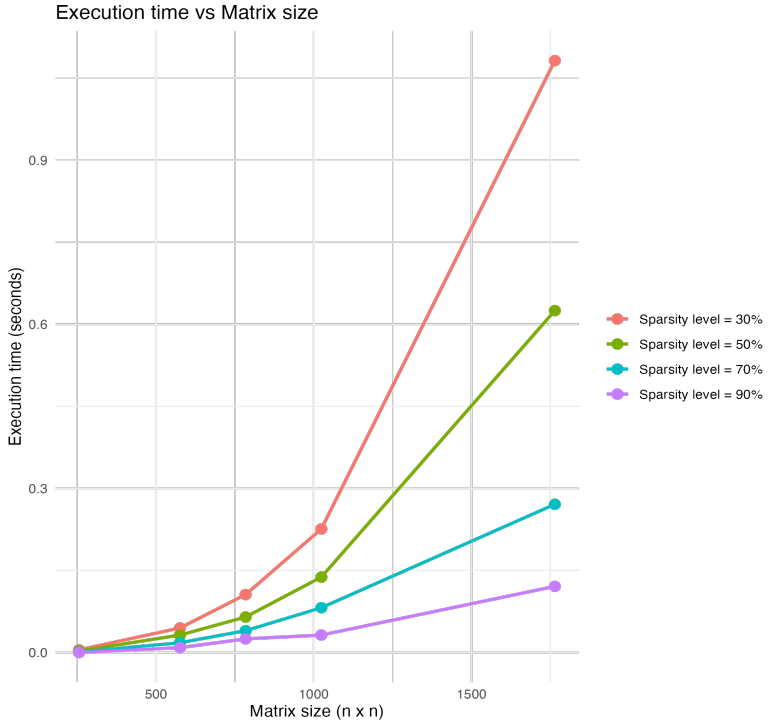

Figure 3: Graph that shows the execution time for different sparsity levels for sparse matrices with CSC format

The table 3 and graph 3 shows the execution time in seconds for the multiplication of sparse matrices with different sizes and levels of sparsity.

We can see how as the size of the matrix increases, the execution time increases significantly at all levels of dispersion, reflecting the computational cost of processing larger arrays, even in the sparse format. Additionally, a decrease in execution time is observed as the level of sparsity increases, suggesting that the higher the dispersion, the faster the processing, which is consistent with the smaller number of non-zero elements to be multiplied.

The difference in execution time is more noticeable for larger arrays. For smaller matrices, such as 256x256, the reduction is smaller between different sparsity levels. In contrast, for 1764x1764 matrices, each increase in the sparsity level generates a considerable reduction in execution time, indicating that the impact of sparsity is more pronounced in larger matrices.

4.2.2 Memory used for CSC format

Size (n x n)	**Sparsity level = 30%(MB)**	**Sparsity level = 50%(MB)**	**Sparsity level = 70%(MB)**	**Sparsity level = 90%(MB)**
256 x 256	10.424	10.127	9.838	9.304
576 x 576	30.797	29.652	26.303	28.706
784 x 784	70.824	69.813	66.409	61.387
1024 x 1024	120.226	117.822	108.809	95.927
1764 x 1764	296.086	289.375	280.490	274.274

Table 4: Memory used for different sparsity levels for sparse matrices with CSC format

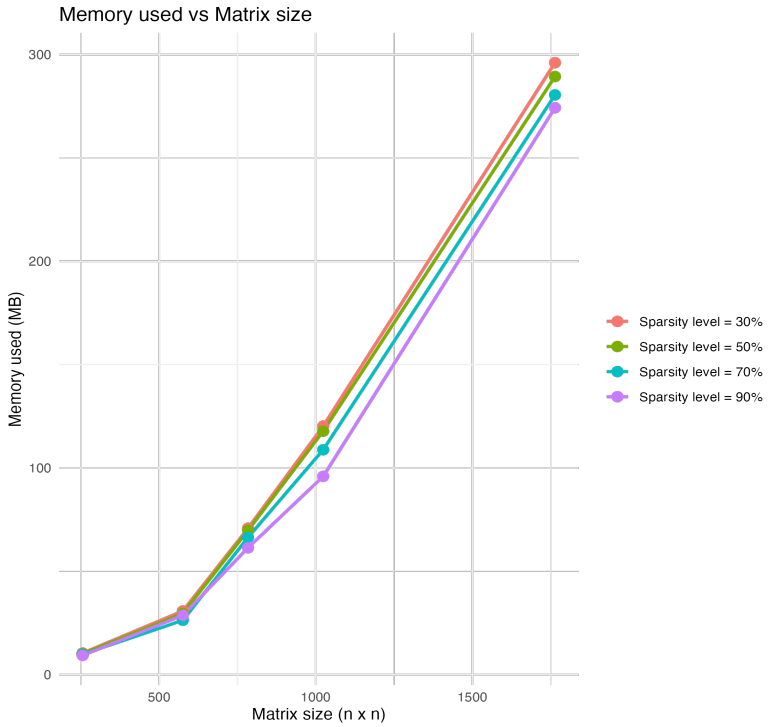

Figure 4: Graph that shows the memory used for different sparsity levels for sparse matrices with CSC format

In table 4 and graph 4 we see that as the size of the matrix increases, the memory usage increases significantly at all levels of dispersion. However, we can also see how the higher the sparsity, the memory usage decreases at each matrix size, because the CSC format allows to store only the non-zero elements. Finally, we can also see how the difference in memory usage between different levels of sparsity becomes more pronounced as the matrix size increases.

4.2.3 Execution time for CSR format

Size (n x n)	**Sparsity level = 30%(s)**	**Sparsity level = 50%(s)**	**Sparsity level = 70%(s)**	**Sparsity level = 90%(s)**
256 x 256	0.005	0.003	0.002	0.001
576 x 576	0.039	0.029	0.018	0.009
784 x 784	0.107	0.073	0.037	0.026
1024 x 1024	0.219	0.139	0.073	0.043
1764 x 1764	1.072	0.606	0.280	0.103

Table 5: Execution time for different sparsity levels for sparse matrices with CSR format

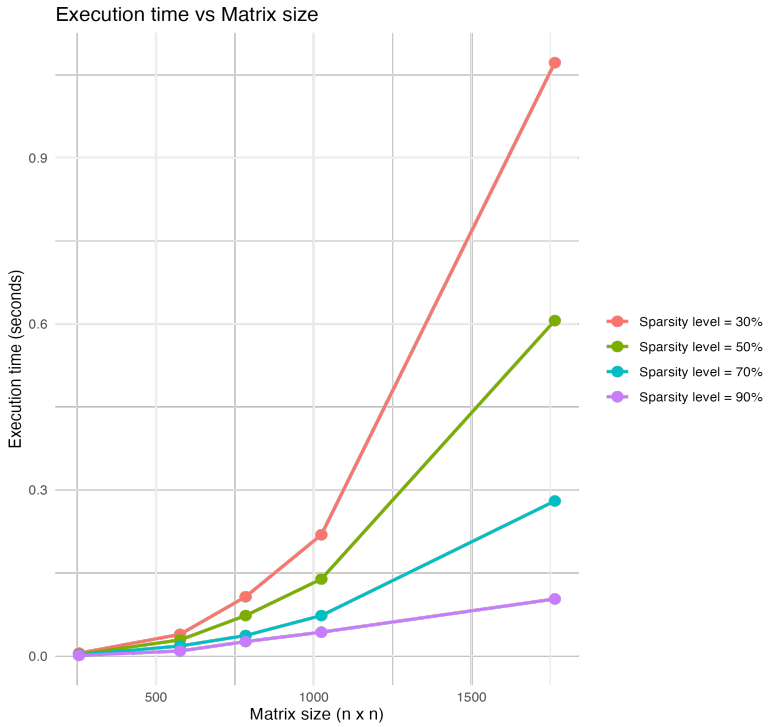

Figure 5: Graph that shows the execution time for different sparsity levels for sparse matrices with CSR format

As with the CSC format, we can see in the 5 table and 5 graph how for smaller matrices the influence of the sparsity level is not noticeable, but as the size of the matrices increases, the higher the sparsity level, the shorter the execution time.

4.2.4 Memory used for CSR format

Size (n x n)	**Sparsity level = 30%(MB)**	**Sparsity level = 50%(MB)**	**Sparsity level = 70%(MB)**	**Sparsity level = 90%(MB)**
256 x 256	11.512	10.892	10.299	9.445
576 x 576	41.473	31.097	28.230	29.552
784 x 784	81.752	79.914	70.720	62.768
1024 x 1024	149.921	123.498	122.094	98.256
1764 x 1764	376.433	336.293	316.762	279.468

Table 6: Memory used for different sparsity levels for sparse matrices with CSR format

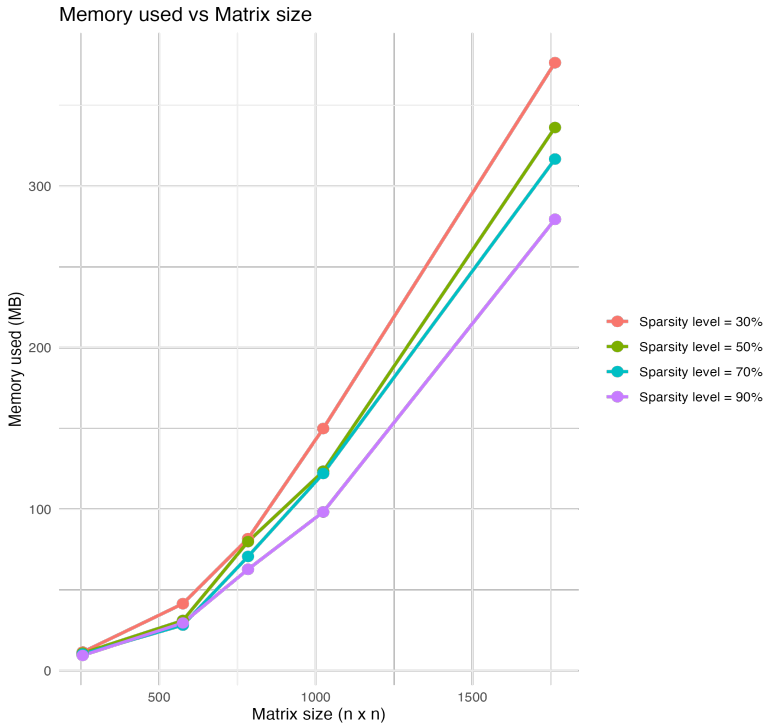

Figure 6: Graph that shows the memory used for different sparsity levels for sparse matrices with CSR format

In terms of memory used, the same situation as for the CSC-formatted sparse matrices is repeated. We see in table 6 and figure 6 how the amount of memory used increases the larger the size of the matrix, as well as how this size decreases the higher the sparsity level due to the fact that it does not store null elements.

In conclusion, the comparison between the CSC and CSR formats reveals that both are effective for storing sparse arrays, but show slight differences in terms of memory usage and execution time. In terms of memory, the CSC format tends to be slightly more efficient than CSR on smaller, less sparse arrays, but this advantage diminishes on larger, sparse arrays, where both formats are close in their storage performance.

Regarding execution time, the results indicate that CSR tends to be faster than CSC at various levels of sparsity, especially on larger arrays. This improvement in execution time is due to the row-oriented structure of CSR, which facilitates operations such as the multiplication of sparse matrices by row vectors. However, the choice of the optimal format may depend on the characteristics of the application, as CSC may be preferable in cases where column-level operations are frequent.

4.3 Maximum matrix size handled

To thoroughly evaluate the scalability and efficiency of our matrix multiplication methods, experiments were conducted with the largest matrix sizes manageable within our setup for both dense and sparse formats.

4.3.1 Dense Large Matrix Multiplication

For dense matrices, three matrix multiplication methods were tested on matrices of size 7056 x 7056. Each method demonstrated varying levels of efficiency:

- The block matrix multiplication method achieved the best performance among dense methods, with a total runtime of approximately 4 minutes, benefiting from cache optimization through sub-matrix division.
- The loop unrolling method, which enhances instruction-level parallelism, completed the multiplication in around 20 minutes.
- The standard dense multiplication method, used as a baseline, required approximately 28 minutes to complete the operation.

These results highlight the performance benefits of the block method, confirming its suitability for large-scale dense matrix operations within time-sensitive applications.

4.3.2 Sparse Large Matrix Multiplication

For sparse matrices, two storage formats—Compression by Row (CSR) and Compression by Column (CSC)—were tested using matrices of size 525,825 x 525,825, containing 2,100,225 non-zero elements. The results were as follows:

- CSR format demonstrated the fastest execution time, completing the multiplication in 4 minutes.
- CSC format closely followed, with a runtime of around 4 minutes and 10 seconds.

These findings underscore the efficiency of the CSR format in handling large-scale sparse matrices, as it offers a slight advantage in execution time over CSC. Both formats, however, allow for reasonable runtimes despite the matrix's extreme size, showcasing the feasibility of using sparse matrices in large-scale applications where rapid processing and efficient resource management are essential.

5 Conclusions

In conclusion, the experiments carried out show that, although sparse matrices may take up more memory space compared to dense matrices, they offer a significant advantage in terms of execution time. This difference is particularly relevant when working with large matrices and with high levels of sparsity, as CSC and CSR storage formats reduce the number of operations required by omitting zero values, thus speeding up calculations. In the cases analysed, the multiplication of sparse matrices in both CSC and CSR formats resulted in significantly shorter execution times compared to dense matrices, demonstrating that the sparse approach is superior in time efficiency, especially in time-critical applications.

Furthermore, while CSC and CSR sparse arrays may consume more memory on smaller arrays or arrays with low sparsity levels, this disadvantage is largely offset by the reduction in computation time. On large arrays, the difference in execution time is even more evident, making sparse arrays a preferable choice for operations in large data-volume contexts. Overall, the use of sparse matrices significantly optimises the performance of matrix multiplication when sparsity is high, making it a highly efficient strategy for improving execution time in practical applications.

As for the memory used, the results show that the storage format has a considerable impact. In dense matrices, memory usage is constant, as it stores all values, regardless of whether they are zeros. In contrast, sparse matrices can vary in their memory consumption depending on the sparsity level and the storage format. At low levels of sparsity, both CSC and CSR can use a similar or even larger amount of memory than that required by dense matrices. However, as sparsity increases, memory usage in sparse arrays is reduced, which becomes advantageous in large, high sparsity arrays. Ultimately, the choice between dense and sparse formats involves a trade-off between execution time and memory usage, with sparse arrays being preferable for environments that prioritise computational speed and can tolerate variable memory usage.

Finally, in the experiment with the extremely large matrix (525,825 x 525,825), both CSR and CSC formats demonstrated the feasibility and efficiency of sparse matrix multiplication at scale. The CSR format achieved the best performance with a multiplication time of 4 minutes, while the CSC format closely followed at 4 minutes and 10 seconds. This successful handling of a matrix with over 2 million non-zero elements confirms the scalability of sparse matrix formats, particularly the CSR format, in high-dimension scenarios where computational resources are limited. Furthermore, the block method for dense matrices, tested on matrices up to 7056 x 7056, also demonstrated efficiency with a runtime of 4 minutes, though it was less optimal than CSR in extremely large sparse cases. These findings reinforce the suitability of sparse matrices for applications requiring rapid processing and efficiency with massive data sizes, underscoring their role in high-performance computing contexts.

6 Future Works

A possible extension of this work is the implementation of parallel computing techniques for matrix multiplication. Parallelization can significantly improve performance by distributing multiplication operations over multiple cores, especially on modern hardware architectures such as multi-core processors or GPUs. In this sense, techniques such as multi-threading or the use of parallelization libraries, such as OpenMP, offer opportunities to explore how the workload can be efficiently divided among different threads.

An interesting approach would be to analyze the efficiency of vectorization in combination with parallelization, taking advantage of the ability of vector processing units to perform operations on multiple data elements simultaneously. Implementing optimized algorithms for modern processor architecture would allow investigating how the combination of thread-level parallelism and vectorized operations affects performance, compared to the sequential optimization techniques implemented in this project.

This future work could include additional benchmarks to compare performance between optimized sequential implementations and their parallel and vectorized versions, measuring execution time, memory utilization, and core utilization. In addition, comparison of different parallelization libraries and their adaptation to dense and sparse arrays could provide a comprehensive view of the potential of parallel computing for large-scale applications.