TASK 2. Optimized Matrix Multiplication Approaches and Sparse Matrices

Samuel Déniz Santana

November 2024

GitHub Repository Link

1 Abstract

Efficient processing of matrix multiplications is a challenge in large-scale applications due to the execution time and memory-intensive nature of these computations. This study explores various optimisation techniques to improve the performance of dense and sparse matrix multiplication, comparing methods such as cache optimisation, loop unrolling, and CSC (column-wise) and CSR (row-wise) sparse matrix implementations. Experiments were performed varying the size of the matrices and sparsity levels, including a large-scale test with a 500,000 x 500,000 matrix and 99.99

The results show that the optimised methods for dense matrices (cache and loop unrolling) significantly reduce execution times, with loop unrolling being the most efficient. On sparse arrays, CSR proved to be highly effective at high sparsity levels, optimising both execution time and memory usage. The final test with CSR on the extremely large matrix confirmed its scalability and applicability in large data volume scenarios.

This analysis highlights the importance of choosing the right method to optimise computational resources, providing guidance for efficient implementation in high-demand applications.

2 Introduction

Matrix multiplication is a fundamental operation in many fields of computing, such as linear algebra, artificial intelligence and data processing. However, as the size of matrices increases, traditional methods for performing this operation can become slow and inefficient, both in terms of time and memory usage. Therefore, several techniques have been developed to optimise these operations in order to make them faster and less resource-intensive.

In particular, there are different optimisation strategies depending on the type of matrix. In dense arrays, where almost all elements are non-zero, optimisations usually focus on improving cache usage or applying techniques such as loop unrolling. Meanwhile, in sparse arrays, which contain many zeros, it is necessary to use more efficient storage methods, such as CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column) formats, which save memory without losing performance.

Several previous studies have worked on these issues, comparing different techniques and evaluating their performance depending on the size of the matrix and its sparsity level. In general, cache optimisations and loop unrolling have been found to improve performance on dense matrices, while formats such as CSR and CSC are key to work efficiently with sparse matrices. However, a more direct comparison of these approaches and their impact in terms of time and memory, especially on resource-constrained systems, is still lacking.

This work contributes to this area by performing a detailed evaluation of various optimisation techniques for dense and sparse matrices. Through experiments, both execution time and memory usage will be analysed, with the aim of providing practical recommendations. The main novelty of this study is the direct comparison of optimisation methods for both types of matrices, which provides a more complete and applied view to improve the performance of these operations.

3 Problem statement

The main objective of this work is to address the optimisation of matrix multiplication, a fundamental process in several areas of computing and data science. In particular, we aim to improve the performance of multiplication operations for both dense and sparse matrices. As the size of matrices increases, traditional multiplication methods become inefficient, leading to the need to explore techniques to reduce both execution time and memory usage.

In dense matrices, the main challenge lies in the high amount of non-zero data, which can generate a high computational cost. Optimisation operations, such as loop unrolling and cache optimisations, are presented as a potential solution to speed up such computations. However, these methods do not always improve memory usage, especially when handling large matrices, which can lead to efficiency problems in resource-constrained systems.

On the other hand, sparse arrays, which contain a large proportion of zeros, offer opportunities to optimise memory usage through specialised storage formats, such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC). The challenge here is to maximise the storage of this non-null data without sacrificing performance in terms of execution time. However, the efficiency of these formats varies depending on the sparsity level of the arrays, which introduces the need to explore different configurations and optimisation methods.

This paper focuses on addressing these problems by providing a comparative evaluation of different optimisation techniques for matrix multiplication. The aim is to determine which strategy is the most efficient in terms of execution time and memory usage, considering both dense and sparse matrices with different sparsity levels.

4 Methodology

This section describes the experimental approach used to evaluate and compare the performance of different optimization techniques for dense and sparse matrix multiplication. The experiments will be carried out using the IntelliJ IDEA development environment, aiming to measure both execution time and memory usage for each method. The following outlines the configuration of the environment, experimental methods, and evaluation metrics used to ensure reproducibility of the results.

4.1 Experimental Setup

The experiments will be executed on a system with the following specifications:

- **Hardware**:
  - **RAM**: 16 GB
  - **Processor**: 12th Gen Intel® Core™ i7-12700H @ 2.30 GHz
  - **Operating System**: Windows 11
- **Software**:
  - **Development Environment**: IntelliJ IDEA
  - **Programming Language**: Java
  - **Benchmarking Framework**: JMH (Java Microbenchmarking Harness)

JMH will be used to run the benchmarks and measure both execution time and memory usage for each matrix optimization technique.

4.2 Benchmark Configuration

To measure the performance of the optimizations, the benchmarks will be configured with the following parameters:

- **Benchmark Mode**:
  - `@BenchmarkMode(Mode.AverageTime)`: This will measure the average execution time of each operation.
  - **Output Time Unit**: `@OutputTimeUnit(TimeUnit.MILLISECONDS)` will output the time in milliseconds.
- **Warmup**:
  - `@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)` will perform 5 warmup iterations of 1 second each to allow the JVM to optimize the code before measurements are taken.

- **Measurement**:
  - `@Measurement(iterations = 10, time = 1, timeUnit = TimeUnit.SECONDS)` will perform 10 measurement iterations, each lasting 1 second.
- **Forking**:
  - `@Fork(1)` will ensure that each experiment runs in a separate JVM process to isolate the results and prevent interference from previous runs.

4.3 Matrix Sizes and Sparsity Levels

The matrix sizes and sparsity levels will be parameterized using the `@Param` annotation as follows:

- **Dense Matrix Sizes**:
  - `@Param{\10", \50", \100", \200", \500", \1000", \2000"}` will be used for matrices of different sizes in dense matrix multiplication experiments.
- **Sparse Matrix Sizes**:
  - `@Param{\10", \50", \100", \200", \500", \1000", \2000", \5000"}` will specify the matrix sizes for sparse matrix multiplication experiments.
- **Sparsity Levels**:
  - `@Param{\0.0", \0.2", \0.5", \0.9"}` will define the sparsity levels, representing the fraction of non-zero elements in the matrix.

4.4 Performance Evaluation Metrics

The main performance metrics for evaluating the optimizations are:

- **Execution Time**: The total execution time of each experiment will be measured in milliseconds using JMH. Times will be averaged over 10 iterations to obtain a representative value.
- **Memory Usage**: Memory usage will be measured during the experiments using system monitoring tools or JVM profiling features. This will allow the comparison of memory efficiency between the different methods, especially for sparse matrices, where optimizations like CSR and CSC are expected to improve memory usage depending on the sparsity level.

4.5 Experiment Repetition

Each experiment will be repeated several times to ensure the reliability of the results:

- **10 iterations** will be performed for each experiment configuration to obtain accurate measurements and reduce the impact of system fluctuations.

4.6 Data Analysis

Once the results are collected, they will be analyzed to draw conclusions about:

- The effectiveness of optimizations such as cache optimizations and loop unrolling in dense matrix multiplication.
- The performance and memory usage of CSR and CSC formats in sparse matrix multiplication, depending on the sparsity levels.
- The trade-offs between execution time and memory usage for each optimization method.

This methodology ensures that the results are reproducible and will help identify the most efficient optimization techniques for dense and sparse matrix multiplication based on matrix size and sparsity level.

5 Experiments

This study aims to evaluate the performance of optimization methods applied to matrix multiplication for both dense and sparse matrices. For dense matrices, we will analyze the effect of cache optimization and loop unrolling on improving execution time and memory usage compared to the basic multiplication approach. In parallel, experiments will be conducted with sparse matrices to assess the performance of these optimization techniques while varying the level of sparsity. By adjusting sparsity levels, we aim to observe how these optimizations perform as the proportion of zero elements in the matrices increases. The experiments will involve testing matrices of various sizes and sparsity levels, allowing us to examine the scalability and effectiveness of each method in different scenarios. This approach will provide a comprehensive view of the benefits and limitations of these optimizations, offering insights into the most efficient techniques for both dense and sparse matrix configurations.

5.1 Dense Algorithm Experiments

First, we will focus on optimizing the basic matrix multiplication algorithm to improve both execution time and memory usage. Two optimization methods were applied: cache optimization and loop unrolling. Cache optimization aims to make better use of the CPU cache, reducing the time spent accessing memory, while loop unrolling minimizes the overhead of loop control instructions, enhancing both speed and memory efficiency. To evaluate the impact of these optimizations, experiments will be conductedwith matrices of various sizes used to assess performance under different computational loads.

Execution Time

To assess the efficiency of the optimizations in terms of execution speed, the execution time for each algorithm will be analyzed. By measuring the time taken to perform matrix multiplication across different matrix sizes, we aim to evaluate how the optimizations improve processing speed compared to the basic multiplication approach. This analysis will provide insights into the effectiveness of cache optimization and loop unrolling, especially as matrix dimensions increase.

Table 1: Execution Time in different Optimisation Methods by Matrix Size (ms)

Matrix Size	Basic Multiplication	Cache Optimized Multiplication	Loop Unrolling Multiplication
10x10	4.258	3.907	4.327
50x50	4.785	3.932	4.016
100x100	5.465	4.732	4.422
200x200	13.881	8.446	6.078
500x500	134.287	63.869	34.534
1000x1000	1335.103	544.065	247.490
2000x2000	56415.435	4781.545	3819.039

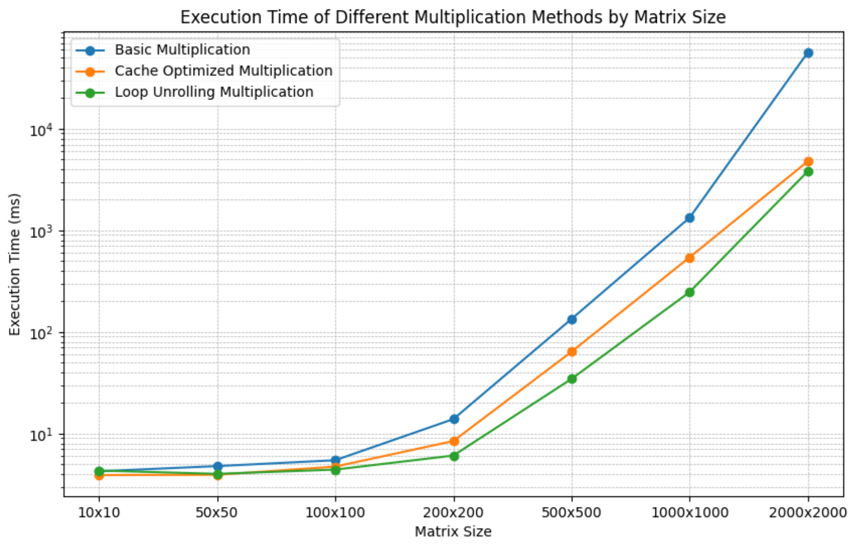

Figure 1: Graph of Execution Time in different Optimisation Methods

As the size of the matrices increases, the execution time for basic matrix multiplication grows rapidly from a few milliseconds with small matrices (10×10) to more than 56,000 milliseconds with large matrices (2000×2000). This is because the basic method has a high computational complexity, which makes it inefficient for handling large volumes of data. Without optimisation, the growth in processing time is approximately exponential, which is a problem in applications requiring fast computations with large matrices.

Optimised methods, such as multiplication with cache optimisation and loop unrolling, significantly reduce execution times, especially as arrays become larger. Cache optimisation improves speed by rearranging the order in which array elements are accessed to make better use of the processor's cache memory, which reduces overall data access time. For example, this technique shows a great advantage on 500×500 and larger arrays, with a time of 4781.545 ms on 2000×2000 arrays, compared to 56415.435 ms for the basic method. Loop unrolling is even faster in most cases, reaching only 3819.039 ms at the largest matrix size, as it minimises redundant operations by removing part of the loop structure and performing additional computations at each iteration.

Memory Usage

In addition to execution time, memory usage will also be evaluated to determine the impact of the optimizations on memory efficiency. By monitoring memory consumption during matrix multiplication, we can assess how well each method utilizes memory resources. This analysis will highlight how optimizations, particularly cache utilization and loop unrolling, can help reduce memory overhead and improve the overall memory efficiency of matrix multiplication.

Table 2: Comparison of Memory Usage in different Optimisation Methods by Matrix Size (MB)

Matrix Size	**Basic Multiplication**	**Cache Optimized Multiplication**	**Loop Unrolling Multiplication**
10x10	0.32	0.32	0.12
50x50	0.32	0.16	0.24
100x100	0.32	0.32	0.48
200x200	0.34	0.48	0.82
500x500	2.02	1.99	3.90
1000x1000	7.72	7.71	15.25
2000x2000	30.79	31.24	62.03

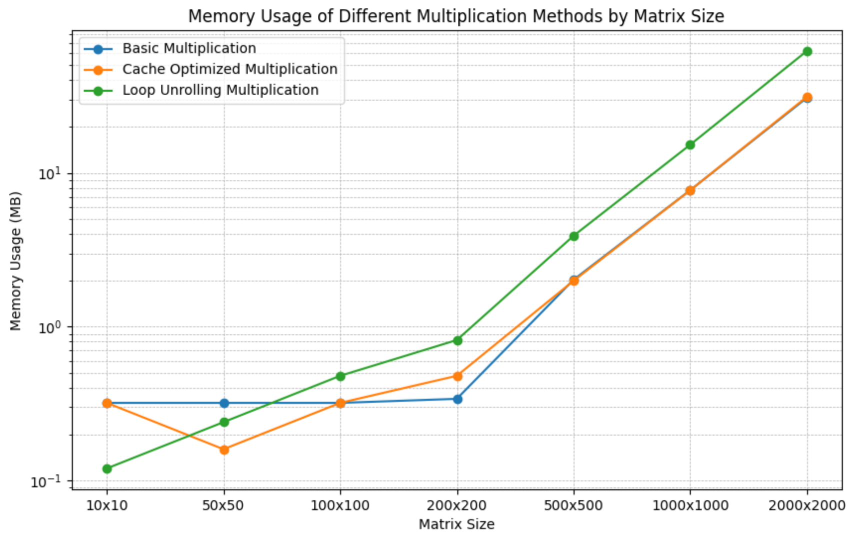

Figure 2: Graph of Memory Usage in different Optimisation Methods

In terms of memory usage, the basic multiplication method consumes less memory in general, especially for small and medium-sized arrays. For example, in arrays of 10×10 up to 500×500, the memory used hardly exceeds 2 MB. Even in large matrices, such as 2000×2000, the basic method consumes 30.79 MB, a relatively moderate amount compared to the other optimisation methods.

Cache optimisation uses slightly more memory as the matrix size increases, especially in large matrices such as 2000×2000, where the memory usage is 31.24

MB. This slight increase is because cache optimisation reorganises data access to make better use of the CPU cache, which may require additional buffering to make repeated access to certain data blocks more efficient.

Loop unrolling is the most memory-intensive method, using 62.03 MB in large 2000×2000 arrays.This is because loop unrolling introduces additional copies of the data into memory to avoid repetitive computations in inner loops and reduce execution time.However, this duplication of data and storage of intermediate results implies higher memory usage. In short, optimisation methods tend to consume more memory because they use additional structures or data duplication to improve computational speed, sacrificing memory in exchange for faster performance in terms of execution time.

5.2 Sparse Algorithm Experiments

The performance of the sparse matrix multiplication algorithms will be evaluated by conducting experiments that vary both the size of the matrices and their level of sparsity. The selected matrix sizes range from small dimensions, such as 10x10, to large sizes up to 5000x5000, allowing us to study how these algorithms behave under different computational loads. Additionally, various sparsity levels have been chosen to assess the impact of sparsity on algorithm performance. These sparsity values are as follows:

- **0.0 (0%)**: Completely dense matrix. This configuration serves as a baseline for comparison, representing the case where no elements are zero.
- **0.2 (20%)**: A low level of sparsity, where the benefits of optimizations for sparse matrices start to appear, although most elements are still non-zero.
- **0.5 (50%)**: Moderate sparsity, where half of the elements are zero. This level allows us to observe a balance in performance between dense and sparse matrix characteristics.
- **0.9 (90%)**: High level of sparsity, where most elements are zero. This case highlights the advantages of specialized algorithms for sparse matrices, which can significantly reduce execution time and memory usage by efficiently handling matrices with a high proportion of zeros.

Through these experiments, we aim to analyze the influence of sparsity and matrix size on the performance of these algorithms, identifying the most efficient methods for each configuration.

Execution Time

Now, the execution time of the sparse matrix multiplication algorithms will be analyzed. By measuring the time it takes for each algorithm to perform matrix multiplication at different sparsity levels and matrix sizes, we aim to assess their efficiency and scalability.

Table 3: Execution Time of Sparse Matrix Multiplication (CSC) (s)

Matrix Size (n)	0.0 Sparsity	0.2 Sparsity	0.5 Sparsity	0.9 Sparsity
10	19.425	24.840	16.926	16.717
50	15.795	20.810	18.555	22.212
100	17.713	16.472	17.364	18.150
200	37.407	28.491	32.283	28.649
500	197.664	179.119	143.115	71.731
1000	1816.239	1352.785	764.575	251.318
2000	13515.828	7659.191	6325.551	1477.246
5000	105841.438	39652.564	22360.970	3950.323

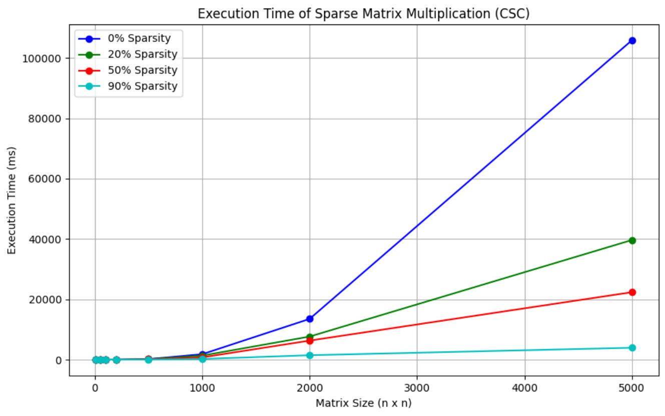

Figure 3: Graph of Execution Time in Multiplication of Sparse CSC Matrices

Table 4: Execution Time of Sparse Matrix Multiplication (CSR) (ms)

Matrix Size (n)	0.0 Sparsity	0.2 Sparsity	0.5 Sparsity	0.9 Sparsity
10	4.019	3.983	4.158	3.806
50	4.067	3.940	4.001	4.035
100	4.793	4.926	4.824	4.302
200	7.986	7.258	6.560	5.270
500	60.095	50.933	33.893	20.758
1000	397.208	346.787	212.812	72.477
2000	2754.173	2214.303	1334.728	331.846
5000	48489.400	35411.786	19409.126	2865.914

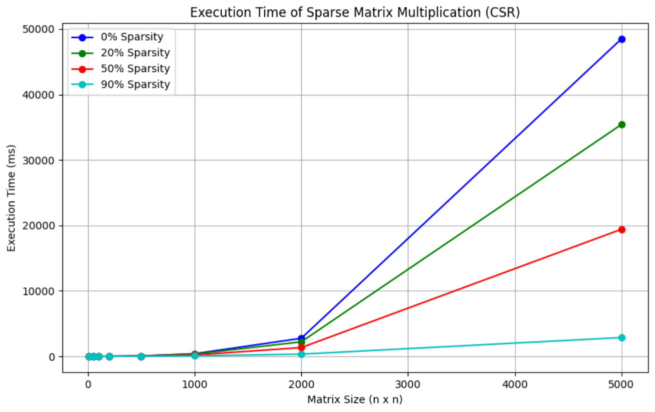

Figure 4: Graph of Execution Time in Multiplication of Sparse CSR Matrices

Comparing the execution times of the sparse matrix multiplication algorithms based on Compression by Column (CSC) and Compression by Row (CSR), it is observed that CSR is consistently faster than CSC at all matrix sizes and sparsity levels (percentage of null elements). For example, in matrices of size 5000×5000 and 0.9 sparsity, the CSR time is 2865.914 ms versus 3950.323 seconds in CSC, a considerable difference that highlights the higher efficiency of CSR in large and sparse matrices.

The impact of sparsity level is significant in both methods, but especially in CSC. In CSC, as sparsity increases (more zeros), the execution time decreases dramatically. This is because, in very sparse matrices, the CSC format must process fewer non-zero columns, which reduces the computation time. For example, in a 1000×1000 matrix with sparsity of 0.9, CSC takes only 251.318 seconds, compared to 1816.239 seconds in the case of a dense matrix (0.0 sparsity). In CSR, the time reduction with increasing sparsity is also noticeable, although less pronounced than in CSC, as the row format is better suited to processing individual rows regardless of their density.

In summary, the CSR method proves to be faster than CSC in almost all cases, especially in large and sparse matrices. This is because CSR handles row accesses better and avoids the overhead associated with compressing and decompressing entire columns in sparse matrices. CSC, however, has advantages in sparse matrices when the number of non-zero columns is low, due to the reduction in the processing of non-significant elements.

Memory Usage

In addition to execution time, the memory usage of sparse matrix multiplication algorithms will also be analysed. By monitoring the memory consumption during the matrix multiplication process, we aim to evaluate the efficiency of each algorithm in terms of memory management. This analysis will help us to understand how different levels of sparsity affect memory usage and to identify which algorithm optimises memory usage.

Table 5: Memory usage in sparse CSC matrix multiplication as a function of sparsity level (MB)

Matrix Size	0% Sparsity	20% Sparsity	50% Sparsity	90% Sparsity
10x10	0.16	0.16	0.16	0.16
50x50	0.80	0.56	0.48	0.44
100x100	2.00	1.94	1.59	1.46
200x200	6.85	6.58	5.86	5.60
500x500	34.87	28.59	25.44	21.01
1000x1000	130.03	116.17	113.41	95.11
2000x2000	565.73	563.51	420.09	322.49
5000x5000	2063.93	2063.95	2095.95	606.26

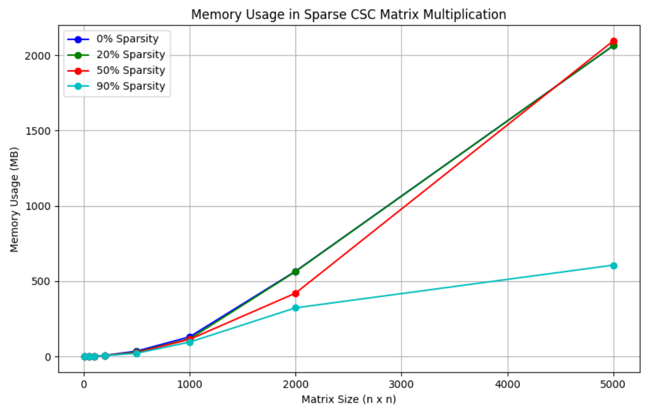

Figure 5: Graph of Memory Usage in Multiplication of Sparse CSC Matrices

Table 6: Memory usage in sparse CSR matrix multiplication as a function of sparsity level (MB)

Matrix Size	0% Sparsity	20% Sparsity	50% Sparsity	90% Sparsity
10x10	0.32	0.16	0.16	0.16
50x50	0.64	0.65	0.66	0.48
100x100	2.01	1.84	1.75	0.82
200x200	6.41	6.31	5.31	3.26
500x500	33.13	24.78	24.31	19.65
1000x1000	119.44	91.53	81.28	87.98
2000x2000	542.22	517.12	438.13	290.05
5000x5000	2166.85	2141.87	2047.90	1762.00

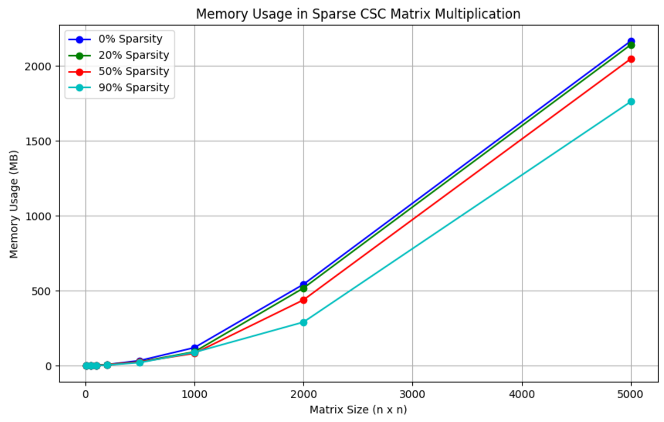

Figure 6: Graph of Memory Usage in Multiplication of Sparse CSR Matrices

Comparing the memory usage between the Compression by Column (CSC) and Compression by Row (CSR) methods for sparse matrix multiplication, we can observe that both methods reduce the memory usage as the sparsity level increases. In small size matrices, such as 10×10 and 50×50, the memory usage is similar in both methods, as both reach a minimum of 0.16 MB in high sparsity matrices (90%). However, as the matrices increase in size, significant differences in memory usage are noticed depending on the method and sparsity level.

In intermediate sized matrices, such as 500×500 and 1000×1000, the CSR method shows slightly lower memory usage than CSC at high sparsity levels (90%), with 19.65 MB and 87.98 MB respectively, compared to 21.01 MB and 95.11 MB in CSC. This suggests that CSR is more efficient in memory management on medium-sized sparse arrays. However, at low sparsity levels (0% and 20%), both methods have similar memory usage, indicating that the differences in storage are less significant when the matrix contains more non-zero data.

For large arrays such as 2000×2000 and 5000×5000, the difference in memory usage becomes more noticeable. In the case of a 5000×5000 matrix with 90% sparsity, CSR uses 1762 MB versus the 606.26 MB used by CSC, indicating that CSC is more efficient on highly sparse matrices. In general, CSC tends to be more efficient in memory usage in high sparsity and large matrices, as this method stores only the columns with non-null data, while CSR, by focusing on rows, requires more storage in large and sparse matrices.

500,000×500,000 Matrix Size Test

Given the high performance observed in the CSR optimisation method, an additional test was performed with an extremely large matrix size, 500,000×500,000, and a sparsity level of 99.99%. This matrix was obtained from the University of Florida's sparse matrix database, specifically from this page. The test was designed to evaluate CSR's ability to handle large matrices with a majority of null elements, since the method is optimised to work efficiently with sparse data. In this case, the multiplication process took approximately 4 minutes.

Considering the massive size of the matrix (250 billion potential positions) and the very high sparsity level, the time of 4 minutes is relatively optimal. Although it seems long, it is a reasonable time for a computation of this magnitude, where only 0.01% of the elements are non-zero, which still represents a large volume of data. The CSR method allows a significant reduction in time by avoiding calculations on the null elements, although the management of a matrix of this size is still demanding due to the complexity in handling the non-null rows.

In conclusion, the execution time of 4 minutes suggests that CSR is a viable and efficient option for extremely large sparse arrays. However, at this level of size and sparsity, there is still a considerable demand on resources and time, which could be improved with more advanced methods or more robust hardware infrastructure. For applications requiring real-time responses, this time could be considered high, but is optimal in contexts where processing large volumes of sparse data is the priority.

6 Conclusion

Matrix multiplication is a fundamental operation in many scientific and engineering applications, but presents a significant challenge when handling large matrices. The main problem is that the basic multiplication method is slow and consumes a considerable amount of resources, making it inefficient for large matrices. Our analysis explores how various optimisation methods - including cache-optimised multiplication, loop unrolling, and sparse matrix multiplication in CSC and CSR format - can improve both execution time and memory usage, with special attention to sparsity levels. This study addresses efficiency and scalability issues in dense and sparse matrix multiplication, proposing optimisations that take advantage of the structure and characteristics of the matrices to reduce computation time and memory usage.

To address these challenges, we have evaluated several techniques: cache optimisation and loop unrolling for dense arrays, and the use of CSC and CSR formats for sparse arrays. Cache optimisation reduces execution time by reorganising memory access, exploiting data locality and minimising memory access times. Loop unrolling further improves performance by reducing the number of iterations in the loop, allowing the processor to execute more operations in parallel. For sparse arrays, we tested the CSC (column-wise) and CSR (row-wise) formats, which only store non-null elements, optimising execution time as a function of sparsity and significantly reducing memory usage. In addition, we performed a final experiment with a matrix of size 500,000×500,000 and sparsity of 99.99% to evaluate CSR bounds, achieving a 4-minute execution in a test environment.

The results show that cache optimisations and loop unrolling significantly outperform the basic multiplication method in terms of execution time on dense matrices, especially as the matrix size increases. Loop unrolling is the fastest method in this context, standing out as the preferred choice for fast computations on large matrices, although it does not improve memory usage, as it increases the number of intermediate operations and requires more time space. For sparse matrices, CSR proved to be highly efficient in terms of execution time for high sparsity levels (above 90%), and also improves memory usage by taking advantage of the sparse structure, while CSC is more efficient on low sparsity matrices. The final test with CSR on the extremely large matrix confirmed its ability to handle large volumes of sparse data in a reasonable time, highlighting its applicability to large-scale problems.

In summary, this experimentation shows that each optimisation method has particular advantages depending on the matrix structure. Cache optimisation and loop unrolling are more suitable for dense matrices, while CSR and CSC are essential for sparse matrices, adapting well to different sparsity levels and matrix sizes, which maximises computational efficiency in terms of processing time and memory usage.

7 Future Work

As future work, an interesting option would be to explore additional parallelisation strategies using different tools and frameworks in addition to OpenMP. For example, testing with MPI (Message Passing Interface) would allow implementing a distributed parallelisation approach, which is particularly useful in multi-node systems, allowing to split the work of matrix multiplication over several networked machines. This could significantly optimise performance by leveraging shared resources in distributed computing configurations, which is crucial for applications that need to handle large volumes of data.

Another line of research could focus on vectorised optimisation using SIMD (Single Instruction, Multiple Data) instructions, further exploiting data parallelism in modern architectures. Integrating the use of GPUs through CUDA or high-performance libraries such as cuBLAS would also be relevant, as GPUs are specially designed to perform massively parallel operations, and in combination with SIMD could dramatically increase efficiency. These tools would make it possible to assess the improvement in execution speed by combining parallelisation and vectorisation in a hardware architecture suitable for these intensive computations.

It would also be useful to further analyse the scalability analysis of parallel efficiency. To this end, tests could be performed to evaluate the performance of multiplying arrays with a variable number of threads, on arrays of different sizes and sparsity levels. This would help identify the optimal performance point for each configuration in terms of resource consumption and execution speed, allowing the system to dynamically adjust the number of threads according to the workload. Furthermore, performing this analysis on different architectures, such as ARM or RISC-V, would provide valuable information on the adaptability of the approach to different hardware platforms.

Finally, it would be beneficial to include additional metrics it would be beneficial to include additional metrics in the analysis, such as data preparation time and cache usage, to assess how these optimisations impact actual performance in practical scenarios. Integrating these factors into the evaluation would allow for a more complete picture of the overall efficiency of each approach, identifying limitations and potential bottlenecks. These areas of experimentation can contribute significantly to the development of more robust and adaptive parallelisation and vectorisation techniques, essential for scientific computing and high-performance applications.