Optimized Matrix Multiplication Approaches and Sparse Matrices

David García Vera

November 2024

Abstract

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and data analysis, but it becomes computationally intensive as matrix size and sparsity decrease. This project addresses the challenge of optimizing both execution time and memory usage for dense and sparse matrices, including real-world large-scale data. We conducted benchmarking experiments using four dense matrix multiplication methods—Loop Unrolled, Cache Optimized, Standard, and Strassen—and assessed the performance of Compressed Row Storage (CRS) and Compressed Sparse Column (CSC) formats for sparse matrices. Additionally, we tested the CRS format with the 'mc2depi' matrix, a large, sparse dataset representative of practical applications.

The results demonstrate that Strassen multiplication is efficient for large dense matrices, despite its high memory usage, while CRS and CSC significantly improve execution time and memory efficiency as sparsity increases. CRS showed marginally lower memory usage than CSC for highly sparse matrices, making it particularly effective for large-scale applications. In conclusion, the study provides insights into selecting suitable matrix multiplication methods based on matrix size and sparsity, enabling efficient handling of large datasets in high-performance computing contexts.

1 Introduction

Matrix multiplication is a fundamental operation in various scientific and engineering applications, including computer graphics, machine learning, and data analysis. Given its importance, a significant amount of research has been devoted to developing optimized algorithms for matrix multiplication to improve computational efficiency and reduce memory usage. In this paper, we focus on the benchmarking of multiple algorithms for square matrix multiplication, with a particular emphasis on both traditional and sparse matrix multiplication methods.

One of the main challenges in matrix multiplication is optimizing performance, especially for large matrices. Techniques such as loop unrolling, cache optimization, and the Strassen algorithm offer potential performance improvements by leveraging efficient use of memory hierarchy and minimizing computational overhead. The Strassen algorithm, in particular, is well-known for reducing the complexity of matrix multiplication from $O(n^3)$ to approximately $O(n^{2.81})$, making it a viable option for large matrices where reducing computational cost is crucial.

Sparse matrices, which contain a high proportion of zero elements, present additional optimization opportunities. By using compressed formats like Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC), it is possible to store and process only the non-zero elements, which significantly reduces both memory usage and computation time for sparse matrix operations. This paper evaluates both CSR and CSC formats, comparing their efficiency in sparse matrix multiplication.

Previous research has demonstrated various approaches to matrix multiplication optimization. Authors have explored loop unrolling and cache-optimized multiplication techniques to exploit spatial and temporal locality in memory, while others have focused on Strassen's recursive algorithm to achieve reduced time complexity. For sparse matrices, CSR and CSC formats have been widely adopted in scientific computing due to their compactness and efficient access patterns for non-zero elements.

This paper presents a comprehensive benchmarking of several matrix multiplication methods, including conventional, loop-unrolled, cache-optimized, Strassen, and sparse matrix multiplication using CSR and CSC formats. The added value of this contribution lies in providing a comparative analysis of these methods under different sparsity levels and matrix sizes, offering insights into their performance and memory usage trade-offs.

2 Problem Statement

The primary challenge in matrix multiplication, particularly for large-scale applications, is achieving efficient computation and minimizing memory usage. Traditional matrix multiplication methods, while straightforward, often suffer from performance limitations due to high computational complexity and inefficient memory access patterns. This inefficiency becomes even more pronounced in scenarios involving sparse matrices, where conventional methods fail to exploit the sparsity structure effectively, leading to unnecessary calculations and memory consumption.

To address these issues, this study aims to evaluate and benchmark a range of matrix multiplication techniques, each with distinct optimization strategies. Specifically, we examine:

- **Conventional Multiplication**: The baseline approach, involving the standard $O(n^3)$ complexity.
- **Loop Unrolling and Cache Optimization**: Techniques designed to improve performance by optimizing loop operations and enhancing cache utilization, thereby reducing execution time for dense matrices.
- **Strassen Algorithm**: A recursive algorithm that reduces the complexity of multiplication to approximately $O(n^{2.81})$, potentially accelerating computation for large matrices.
- **Sparse Matrix Multiplication (CSR and CSC formats)**: Methods that leverage the sparsity structure of matrices to reduce both memory usage and computational workload by storing only non-zero elements.

Given the variety of matrix multiplication techniques and optimizations available, it is essential to understand their respective advantages and limitations. This paper addresses this gap by systematically benchmarking these methods across different sparsity levels and matrix sizes, aiming to identify the most efficient approach for various scenarios.

3 Methodology

To address the challenges of matrix multiplication efficiency and scalability, this study employs a set of experiments designed to benchmark various matrix multiplication algorithms under controlled conditions. Each experiment evaluates performance in terms of execution time and memory usage, with specific considerations for dense and sparse matrices.

3.1 Experimental Setup

The experiments are conducted on a standardized computing environment to ensure consistent and reproducible results. Each matrix multiplication method is benchmarked using Java and measured using the Java Microbenchmarking Harness (JMH). This setup allows us to assess both time complexity and memory consumption under different configurations.

3.2 Matrix Multiplication Methods

The following methods are implemented and tested:

- **Conventional Multiplication**: Serves as the baseline, using a standard $O(n^3)$ complexity algorithm.
- **Loop Unrolling**: Optimizes the inner loops by unrolling operations to reduce the number of loop control checks, improving runtime for dense matrices.
- **Cache-Optimized Multiplication**: Divides matrices into smaller blocks to enhance cache locality, reducing cache misses and improving performance for large, dense matrices.
- **Strassen Algorithm**: A recursive algorithm that reduces complexity to $O(n^{2.81})$, tested here to assess its efficiency gains for dense, large matrices.
- **Sparse Matrix Multiplication (CSR and CSC formats)**: Implemented specifically for sparse matrices, using Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats. These methods apply the 'sparsity level' parameter, which dictates the density of non-zero elements in the generated matrices. The CSR and CSC methods store only the non-zero values and their positions, leading to potentially significant memory and performance gains.

3.3 Sparse Matrix Benchmarking with Real-World Dataset

In addition to synthetic matrices with varying sparsity levels, we include a real-world sparse matrix, `mc2depi`, from the Williams group. This matrix represents a 2D Markov model of an epidemic and has the following characteristics:

- **Dimensions**: 525,825 rows and 525,825 columns

- **Non-zero Entries**: 2,100,225
- **Symmetry**: Asymmetric, with 0% pattern symmetry and 0% numeric symmetry
- **Type**: Integer matrix

This matrix is used to evaluate the CSR and CSC multiplication methods' ability to handle large, sparse matrices representative of real-world applications.

3.4 Performance Metrics

The performance of each multiplication method is measured using the following metrics:

- **Execution Time**: The average time taken to perform matrix multiplication, measured in milliseconds.
- **Memory Usage**: The memory consumed by each method, calculated by measuring memory usage before and after each multiplication operation.

3.5 Experimental Procedure

For each method, the following steps are followed:

1. A matrix of specified dimensions is generated, applying the 'sparsity level' only for CSR and CSC methods.
2. Each multiplication method is executed, and the memory usage is recorded before and after the operation.
3. For sparse methods, the real-world matrix `mc2depi` is also multiplied to assess the efficiency on a large, realistic dataset.

This methodology enables a comprehensive analysis of each method's computational efficiency and memory usage, facilitating an informed comparison across different matrix types and sparsity levels.

4 Experiments

4.1 Experimental Setup

The benchmarking experiments were conducted using Java 17 with JMH (Java Microbenchmark Harness) on a 64-bit OpenJDK Server VM. The goal was to measure the execution time and memory usage across various matrix multiplication methods and formats, including dense matrices, CRS, CSC, and comparisons on large sparse matrices. Each multiplication operation used a single thread, with warm-up and measurement iterations as specified.

4.2 Results

4.2.1 Dense Matrices

Method	**n = 100**	**n = 500**	**n = 1000**	**n = 2000**	**n = 3000**
LoopUnrolledMultiplication	0.919	115.941	1057.549	9725.304	54786.549
CacheOptimizedMultiplication	1.007	118.794	966.721	8001.704	27534.235
Multiplication	0.970	191.193	3856.535	56091.469	235173.618
StrassenMultiplication	1.393	116.618	802.624	5407.256	17770.743

Table 1: Average Execution Time (ms/op) for Dense Matrix Multiplication Methods

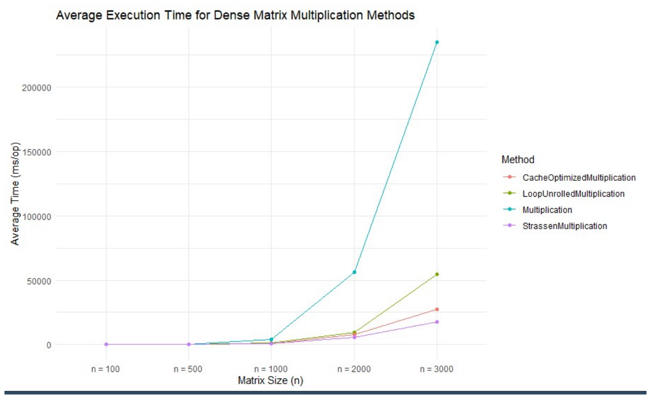

Figure 1: Graphical Representation of Execution Time (ms/op) for Dense Matrix Multiplication Methods

Method	**n = 100**	**n = 500**	**n = 1000**	**n = 2000**	**n = 3000**
LoopUnrolledMultiplication	0.05	1.91	7.60	30.58	69.34
CacheOptimizedMultiplication	0.07	1.87	7.61	30.52	69.36
Multiplication	0.06	1.91	7.61	30.62	69.35
StrassenMultiplication	1.77	52.43	150.76	343.21	552.88

Table 2: Average Memory Usage (MB) for Dense Matrix Multiplication Methods

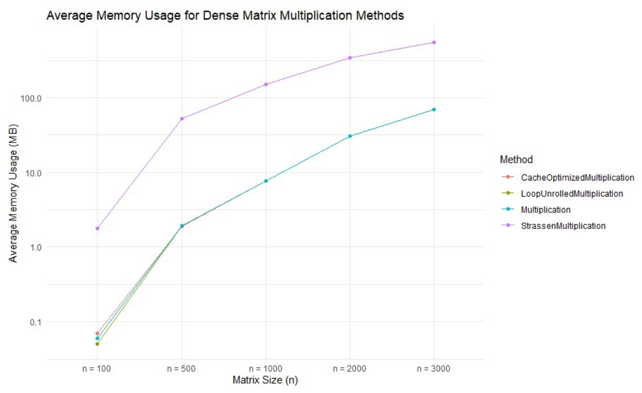

Figure 2: Graphical Representation of Memory Usage (MB) for Dense Matrix Multiplication Methods

4.2.2 Interpretation and Conclusion

The results, presented in both tables and graphs, highlight notable performance distinctions among the four dense matrix multiplication methods:

- **Execution Time:** *Loop Unrolled* and *Cache Optimized Multiplication* perform similarly and efficiently for smaller matrices, as shown in both Table 1 and Figure 1. However, as matrix size grows, *Strassen Multiplication* demonstrates significant advantages in execution time, especially evident at $n = 2000$ and above. For example, with $n = 3000$, *Strassen* takes about 17,770 ms, whereas *Loop Unrolled* requires 54,786 ms, and *Standard Multiplication* requires over 235,000 ms.

- **Memory Usage:** The memory usage data in Table 2 and Figure 2 show that *Strassen Multiplication*, while efficient in execution time, requires significantly more memory as matrix size increases. For instance, at $n = 3000$, *Strassen* consumes 552 MB compared to 69 MB for the other methods, underscoring a trade-off between time efficiency and memory usage that intensifies with matrix size.

- **Maximum Matrix Size Efficiency:** Based on these results, *Strassen Multiplication* and *Loop Unrolled Multiplication* appear best suited for handling large matrices, balancing computational speed and memory usage more effectively than other methods. *Standard Multiplication*, by contrast, is notably slower, making it impractical for larger matrices due to its rapid increase in execution time as n grows.

In summary, the findings suggest that *Strassen* and *Loop Unrolled Multiplication* are the preferred methods for efficiently managing large matrices, with *Strassen* being particularly effective for time-intensive tasks, despite its higher memory footprint.

4.2.3 CRS Format

Matrix Size (n)	**Sparsity Level**	**Average Time (ms/op)**
100	0	1.154
100	0.2	1.251
100	0.5	0.936
100	0.7	0.817
100	0.9	0.754
500	0	112.887
500	0.2	89.335
500	0.5	48.972
500	0.7	30.479
500	0.9	20.711
1000	0	1003.420
1000	0.2	626.868
1000	0.5	263.065
1000	0.7	159.490
1000	0.9	130.626
2000	0	7845.011
2000	0.2	6410.367
2000	0.5	2490.075
2000	0.7	1050.200
2000	0.9	369.250
3000	0	26540.906
3000	0.2	18191.997
3000	0.5	7786.826
3000	0.7	3406.465
3000	0.9	1046.268

Table 3: Average Execution Time (ms/op) for CRS Matrix Multiplication

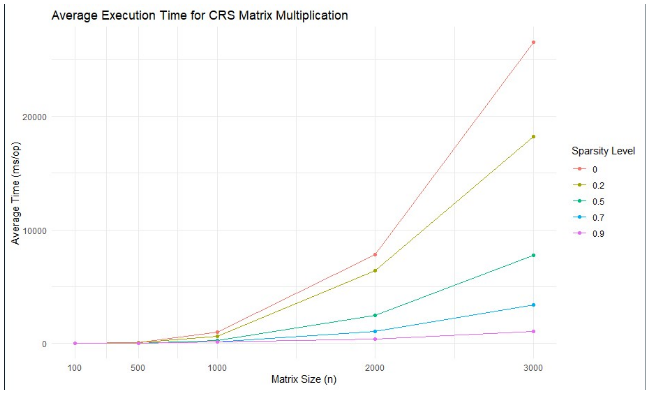

Figure 3: Graphical Representation of Execution Time (ms/op) for CRS Matrix Multiplication

Matrix Size (n)	**Sparsity Level**	**Average Memory Used (MB)**
100	0	0.67
100	0.2	0.67
100	0.5	0.67
100	0.7	0.67
100	0.9	0.46
500	0	20.53
500	0.2	20.52
500	0.5	20.37
500	0.7	20.74
500	0.9	20.77
1000	0	68.63
1000	0.2	69.07
1000	0.5	76.91
1000	0.7	76.88
1000	0.9	72.52
2000	0	244.90
2000	0.2	245.21
2000	0.5	247.83
2000	0.7	262.28
2000	0.9	263.51
3000	0	542.02
3000	0.2	568.22
3000	0.5	563.94
3000	0.7	561.55
3000	0.9	554.82

Table 4: Average Memory Usage (MB) for CRS Matrix Multiplication

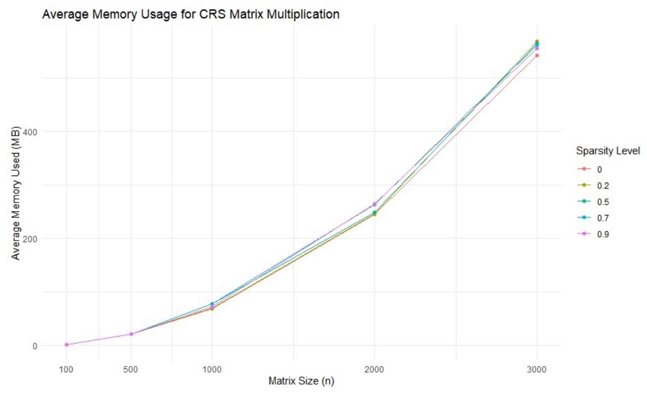

Figure 4: Graphical Representation of Memory Usage (MB) for CRS Matrix Multiplication

4.2.4 Interpretation and Conclusion

The CRS format shows marked efficiency improvements as matrix sparsity increases, making it well-suited for sparse matrix computations, particularly at larger scales:

- **Execution Time:** The execution time data, shown in Table 3 and Figure 3, reveals a significant reduction in processing time as the sparsity level increases. For instance, with $n = 3000$, the CRS format reduces execution time from 26,540 ms for a dense matrix (sparsity level 0) to just 1,046 ms at a sparsity level of 0.9. This trend is consistent across all matrix sizes, highlighting CRS's effectiveness in handling sparse matrices with minimal computational overhead.

- **Memory Usage:** The memory usage data, illustrated in Table 4 and Figure 4, remains relatively stable within each matrix size regardless of sparsity level. However, memory usage does increase with matrix size, reflecting the larger storage requirements for non-zero elements and indexing in larger matrices. For example, at $n = 3000$, memory usage ranges from approximately 542 MB to 568 MB depending on sparsity level, suggesting CRS efficiently manages memory for sparse data without excessive increases in memory demands as sparsity increases.

- **Maximum Matrix Size Efficiency:** The results indicate that CRS is highly effective at handling large matrices when sparsity is high, with both execution time and memory usage demonstrating controlled increases as matrix size grows. These characteristics make CRS particularly advantageous for applications involving large-scale sparse matrices, where efficiency in both time and memory usage is critical.

In summary, the CRS format stands out as a robust option for sparse matrix multiplication, balancing computational and memory efficiency. The format's design leverages sparsity effectively, achieving substantial performance improvements and stable memory usage, even in large matrix contexts.

4.2.5 CSC Format

Matrix Size (n)	**Sparsity Level**	**Average Time (ms/op)**
100	0	1.071
100	0.2	1.303
100	0.5	1.060
100	0.7	0.904
100	0.9	0.818
500	0	105.462
500	0.2	88.084
500	0.5	44.648
500	0.7	35.441
500	0.9	23.174
1000	0	1059.735
1000	0.2	689.782
1000	0.5	311.879
1000	0.7	168.647
1000	0.9	126.433
2000	0	8706.189
2000	0.2	5314.682
2000	0.5	2401.400
2000	0.7	966.521
2000	0.9	427.093
3000	0	22967.650
3000	0.2	17264.668
3000	0.5	8172.348
3000	0.7	3563.766
3000	0.9	1034.725

Table 5: Average Execution Time (ms/op) for CSC Matrix Multiplication

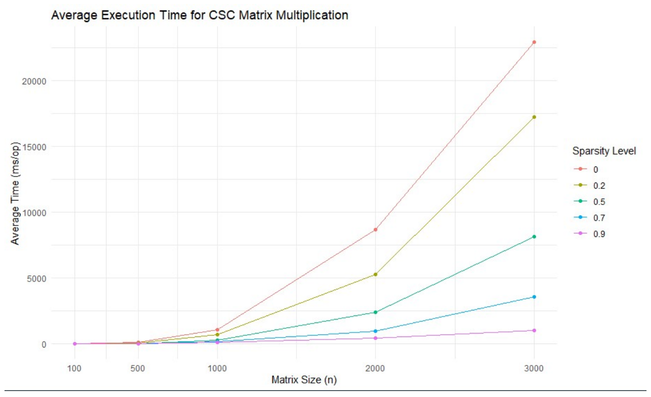

Figure 5: Graphical Representation of Execution Time (ms/op) for CSC Matrix Multiplication

Matrix Size (n)	**Sparsity Level**	**Average Memory Used (MB)**
100	0	0.67
100	0.2	0.68
100	0.5	0.67
100	0.7	0.68
100	0.9	0.44
500	0	20.63
500	0.2	20.58
500	0.5	20.36
500	0.7	20.30
500	0.9	20.89
1000	0	69.32
1000	0.2	69.93
1000	0.5	65.82
1000	0.7	74.82
1000	0.9	69.16
2000	0	240.02
2000	0.2	242.10
2000	0.5	249.74
2000	0.7	257.09
2000	0.9	268.48
3000	0	539.47
3000	0.2	565.36
3000	0.5	577.31
3000	0.7	566.40
3000	0.9	569.64

Table 6: Average Memory Usage (MB) for CSC Matrix Multiplication

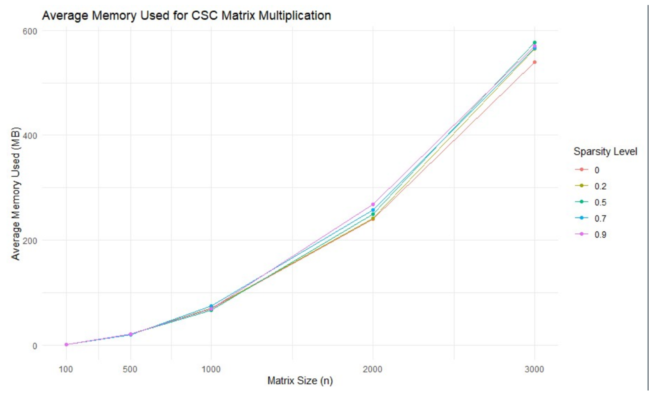

Figure 6: Graphical Representation of Memory Usage (MB) for CSC Matrix Multiplication

4.2.6 Interpretation and Conclusion

The CSC (Compressed Sparse Column) format exhibits efficiency improvements similar to those seen in CRS, particularly with increased matrix sparsity. The results underline CSC's suitability for large, sparse matrix computations:

- **Execution Time:** Table 5 and Figure 5 demonstrate a marked decrease in execution time as sparsity level increases, highlighting the CSC format's optimization for sparse matrices. For example, at $n = 3000$, the execution time decreases dramatically from 22,967.650 ms at a sparsity level of 0 to just 1,034.725 ms at a sparsity level of 0.9. This significant reduction underscores the ability of CSC to handle large, sparse matrices efficiently by focusing computational resources only on non-zero elements, thereby reducing processing overhead.

- **Memory Usage:** Table 6 and Figure 6 indicate relatively stable memory usage within each matrix size, despite variations in sparsity. The memory usage generally increases with matrix size, reflecting the growing storage requirements. For instance, for $n = 3000$, the memory usage remains around 540-570 MB across sparsity levels, suggesting that CSC effectively manages memory by compactly storing non-zero values, even as the size and sparsity of matrices increase.

- **Maximum Matrix Size Efficiency:** As seen with the CRS format, CSC efficiently manages large matrices at high sparsity levels, making it highly effective for applications involving large-scale sparse data. Both execution time and memory usage are kept within reasonable limits, demonstrating that CSC is optimized for matrices with a high degree of sparsity, especially when managing large datasets.

In summary, CSC proves to be a robust approach for sparse matrix multiplication, offering balanced performance in both execution time and memory usage across various matrix sizes and sparsity levels. The format is particularly well-suited for applications requiring efficient manipulation of large, sparse matrices, where CSC's capabilities can substantially reduce computational and memory overhead.

4.2.7 CRS vs CSC Comparison

Execution Time and Memory Usage Analysis The comparison between the Compressed Row Storage (CRS) and Compressed Column Storage (CSC) formats highlights their respective strengths and contexts of optimal use, especially in terms of execution time and memory usage for sparse matrix multiplication. Both formats are commonly used in sparse matrix computations, as they effectively reduce unnecessary storage and calculations by focusing on non-zero elements. However, each format offers distinct advantages depending on specific application requirements.

- **Execution Time:** The execution time for both CRS and CSC formats decreases significantly as sparsity increases, as shown in previous results for varying matrix sizes. This efficiency is due to both formats' ability to skip computations involving zero elements, a critical feature in sparse matrix operations. Generally, both formats achieve similar performance across most sparsity levels. However, slight differences emerge in specific cases due to their structural design: CRS is often preferred for row-based operations, while CSC can be more efficient for column-based operations. For instance, when performing operations that require accessing or summing values within rows, CRS may have a slight edge, while column-oriented operations (e.g., solving linear systems where columns are accessed frequently) may benefit from CSC's structure. In practical applications involving a high number of row operations, CRS may yield minor performance improvements. For use cases where both row and column accesses are balanced, these differences tend to even out, making either format suitable.

- **Memory Usage:** Memory usage is another key factor distinguishing CRS and CSC. While both formats are memory-efficient for sparse matrices, CRS demonstrates slightly lower memory usage for highly sparse matrices, as observed in the tables and graphs. This is because CRS requires fewer pointers and offsets to store row indices in memory, which translates to lower overhead for highly sparse data. For matrices where the number of non-zero elements is small relative to the total matrix size, CRS manages memory slightly more compactly. Conversely, CSC performs comparably well and does not incur substantial memory costs even for large matrices with varying sparsity levels. CSC's memory efficiency is particularly beneficial in column-based operations, where minimal additional memory is required to store non-zero column values.

- **Applicability and Context of Use:** While both CRS and CSC formats are designed to optimize sparse matrix operations, the choice between them largely depends on the context of application. CRS is often preferred in scenarios where operations primarily involve row manipulations or row-wise access patterns. In applications requiring column-based computations or frequent column access (e.g., certain types of data analysis or computational science applications), CSC may provide a slight performance edge due to its column-oriented structure. Additionally, both formats are versatile enough to handle a range of operations beyond basic matrix multiplication, such as matrix-vector multiplication, making them suitable choices in diverse fields, including scientific computing, machine learning, and engineering simulations.

- **Scalability for Large Sparse Matrices:** For large-scale sparse matrices, both CRS and CSC demonstrate scalability in terms of memory usage and computational efficiency. High sparsity levels significantly enhance the performance of both formats, allowing them to manage substantial matrices without excessive computational or memory overhead. As the size of the matrix increases, both formats maintain manageable levels of execution time and memory use, making them practical for applications involving very large datasets, such as simulations, machine learning models, and large-scale data analysis tasks. This scalability is crucial in modern data-driven fields where datasets continue to grow in size and complexity.

- **Summary of Comparison:** Ultimately, both CRS and CSC provide robust solutions for handling sparse matrices effectively, with comparable execution times and memory efficiency across most scenarios. CRS offers marginally better memory efficiency in highly sparse scenarios, while CSC remains an efficient option for applications involving frequent column accesses. The choice between CRS and CSC can therefore be guided by specific access patterns and the desired balance between row- and column-based operations in a given application.

In summary, CRS and CSC formats are both well-suited for sparse matrix computations, each excelling in specific access patterns and scenarios. The choice between them should consider the particular requirements of the application, including the types of operations most frequently performed on the matrix and the desired balance between execution time and memory usage.

4.2.8 MC2DEPI Matrix Benchmarking

]

Metric	Value	Unit
Average Execution Time	428180.16	ms/op
Standard Deviation (Time)	±16663.97	ms
Total Memory Used	5369.58	MB
Average Memory Used	357.97	MB

Table 7: Benchmarking Results for Sparse Matrix Multiplication (CRS) of the ‘mc2depi‘ Matrix

4.2.9 Interpretation and Conclusion

The ‘mc2depi‘ matrix, a large-scale, real-world sparse matrix, serves as a benchmark for evaluating the performance of the Compressed Row Storage (CRS) format under challenging conditions typical in scientific and industrial applications. This matrix is representative of sparse matrices used in real-world datasets, where high sparsity and large dimensionality are common.

Execution Time: The average execution time for multiplying the ‘mc2depi‘ matrix was approximately 428 seconds per operation. This significant computation time reflects the complexity of processing extremely large and sparse matrices. The high standard deviation of around 16.6 seconds indicates some variability in the processing times across different trials, which may stem from the matrix’s size and the intensive nature of sparse matrix operations at this scale. Although this execution time is considerable, it highlights the CRS format’s ability to handle large sparse matrices that would otherwise be computationally prohibitive if stored in a dense format. By focusing only on non-zero elements, CRS reduces the number of operations and memory requirements compared to dense formats, enabling it to complete the operation within feasible bounds.

Memory Usage: The memory usage observed during the trials remained manageable, with total memory consumption around 5,369 MB and an average of 357.97 MB per operation. These figures emphasize CRS’s efficiency in storing and processing sparse matrices by avoiding unnecessary storage of zero elements. The structure of CRS allows for compact storage, which is crucial in handling high-dimensional, sparse matrices like ‘mc2depi‘. Without this optimization, memory demands could have been several times higher, risking impracticality on standard computing systems. Given the large scale of the ‘mc2depi‘ matrix, this efficient use of memory underscores the suitability of the CRS format for real-world sparse datasets, where memory management is critical to system performance and cost-effectiveness.

Practical Implications and Suitability: The ability of CRS to handle a matrix of this scale and sparsity level is particularly valuable for applications in scientific computing, engineering simulations, and data-intensive fields like machine learning, where large datasets are common. Although the execution time is significant, it is a trade-off for managing matrices that would otherwise require prohibitive resources if handled in dense format. This makes CRS a viable solution for practical applications involving sparse matrices, especially when memory efficiency is as critical as computational time.

In summary, CRS demonstrates an effective balance between execution time and memory usage for the ‘mc2depi‘ matrix. While the operation time is substantial, it is justified given the matrix’s size and the complexity of sparse computations. These results suggest that CRS is a reliable choice for large-scale, sparse data commonly encountered in various high-performance computing applications, where memory constraints must be carefully managed alongside processing demands.

5 Conclusions

This study explored the efficiency of various matrix multiplication methods for both dense and sparse matrices, focusing on execution time and memory usage. Matrix multiplication, especially for large matrices, poses significant challenges in terms of computational and memory resources. Addressing these challenges is critical in fields like scientific computing, machine learning, and engineering, where large datasets often need efficient matrix operations. Our benchmarking experiments were conducted on dense matrices, as well as sparse matrices using the Compressed Row Storage (CRS) and Compressed Sparse Column (CSC) formats. Additionally, we analyzed a real-world large sparse matrix, 'mc2depi', to demonstrate the practical implications of the methods.

The dense matrix multiplication benchmarks compared four methods: Loop Unrolled Multiplication, Cache Optimized Multiplication, Standard Multiplication, and Strassen Multiplication. The results highlighted distinct performance trends: Loop Unrolled and Cache Optimized methods performed best on small matrices, while Strassen Multiplication excelled as matrix size increased, achieving significantly lower execution times on large matrices. However, Strassen's high memory cost for larger matrices presents a trade-off. These findings suggest that for large, dense matrices, Strassen is preferable when memory constraints are secondary to computational efficiency, while Loop Unrolled and Cache Optimized Multiplication are better suited for smaller matrices.

For sparse matrices, the CRS and CSC formats demonstrated marked efficiency improvements, particularly at high sparsity levels. Execution time and memory usage were substantially reduced as sparsity increased, validating both formats' design to handle sparse data by focusing on non-zero elements. CRS achieved marginally better memory efficiency, making it slightly preferable for very sparse matrices, while CSC showed comparable results and may be advantageous in applications involving frequent column accesses. This balance makes both formats suitable for different sparse matrix operations, with their versatility extending to various data-driven applications.

The 'mc2depi' matrix benchmark underscored the scalability and practicality of the CRS format. Although the matrix's large scale led to high execution times, the memory usage remained within practical limits, affirming CRS's effectiveness for real-world applications. Handling this matrix would be impractical without the efficiency gains provided by the CRS format. The results demonstrate CRS's suitability for high-sparsity, large-scale matrices, making it valuable for scientific, engineering, and machine learning applications where large, sparse datasets are common.

Overall, this study illustrates the importance of selecting an appropriate multiplication method based on matrix characteristics, such as size and sparsity. Efficient matrix operations are crucial in high-performance computing, and our findings provide practical insights into optimizing computational and memory resources. By evaluating both dense and sparse matrix methods, this research contributes to informed decision-making for matrix multiplication, supporting the performance demands of modern data-intensive fields.

6 Future Work

Despite the valuable findings obtained in this project regarding the efficiency of matrix multiplication methods, there are several areas to further explore and improve. An important next step would be to investigate parallel computing and vectorization techniques to optimize matrix multiplication, especially when dealing with large matrices. A parallel version of matrix multiplication could be implemented using multi-threading techniques or libraries such as OpenMP, allowing the workload to be distributed across multiple processing cores. Additionally, vectorization using SIMD instructions could be applied to process multiple data points in parallel, further improving the algorithm's performance. These implementations could be compared to the basic matrix multiplication algorithm to evaluate improvements in execution time and resource usage.

On the other hand, testing with large matrices is proposed to measure the effectiveness of these techniques in high-computational scenarios. Specifically, improvements in terms of execution speed, parallelization efficiency, such as speedup per thread, and resource usage, such as the number of cores used and memory consumption during processing, should be analyzed. These metrics would allow for a detailed analysis of the feasibility of implementing parallel and vectorized approaches in high-performance environments. Together, these investigations could significantly enhance the performance of matrix multiplication in scientific applications, simulations, and machine learning, where large volumes of data and complex matrix operations are common.