Matrix Multiplication Benchmarking

Alonso León, María

Data Science and Engineering

Big Data - ULPGC

GitHub Repository

November 10, 2024

Abstract: Matrix multiplication is a core operation in scientific computing, machine learning, and Big Data, where optimizing performance is essential for managing large datasets efficiently. This study benchmarks multiple matrix multiplication algorithms in Java, assessing execution time and memory usage across dense and sparse matrices of varying sizes and sparsity levels. Experiments reveal that specialized sparse matrix formats, CSR and CSC, yield notable improvements in execution time but do not significantly reduce memory consumption, particularly as sparsity increases. Conversely, Strassen's algorithm demonstrates poor performance in both metrics, making it unsuitable for large-scale applications where both speed and memory efficiency are critical. In conclusion, CSR and CSC formats are ideal for time-sensitive sparse matrix applications, while block-based and row-column major algorithms scale effectively in dense cases. These insights support selecting the most suitable matrix multiplication approach based on data characteristics to optimize computational performance and resource management for large-scale applications.

Key words: Matrix multiplication, sparse matrices, algorithm efficiency, performance benchmarking, execution time, memory consumption, Java, Strassen algorithm, CSR, CSC, computational optimization

1 Introduction

Matrix multiplication is a fundamental operation in various scientific and industrial applications, particularly in the field of big data, machine learning, and numerical simulations. As the size of datasets and the complexity of algorithms grow, optimizing matrix multiplication has become a critical challenge. This paper investigates the performance of different algorithms for matrix multiplication, focusing on the impact of parameters such as sparsity, parallelism, and memory usage. The goal is to identify the optimal approach for maximizing performance, particularly within a carefully selected programming language.

In a previous study, we explored the performance of four widely used programming languages—Java, Rust, Python, and C++—for matrix multiplication. The results demonstrated that Java outperforms the others in terms of execution time. Building upon these findings, the present work focuses on further optimizing matrix multiplication within the Java programming environment. We examine various algorithms to determine which one delivers the best performance in terms of both execution time and memory efficiency, considering factors like sparsity and parallelism.

Several studies have explored similar problems. Notably, Williams et al. (2018) optimized sparse matrix-vector multiplication on emerging multicore platforms, incorporating techniques such as block-based parallelism. These methods are relevant to our research as they offer potential solutions for enhancing the efficiency of matrix operations on modern hardware.

The contribution of this paper lies in the comprehensive comparison of multiple matrix multiplication algorithms, with a particular emphasis on sparse and parallelized approaches. The insights gained will help guide future implementations, providing valuable information not only about the computational speed but also about memory optimization techniques in matrix operations.

2 Problem Statement

The problem addressed in this paper is to determine the optimal algorithm for matrix multiplication that minimizes execution time and memory consumption, especially when working with sparse matrices and utilizing parallelism.

Matrix multiplication between two matrices A of size $m \times n$ and B of size $n \times p$ results in a matrix C of size $m \times p$, where each element c_{ij} of C is computed as follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

This basic operation has a computational complexity of $O(m \cdot n \cdot p)$. For large matrices, this cubic complexity implies a rapid increase in execution time and memory usage, especially as matrix dimensions grow.

2.1 Dense vs. Sparse Matrices

In matrix multiplication, the distinction between *dense* and *sparse* matrices is critical for understanding performance trade-offs in both computational time and memory usage. A **dense matrix** is one in which most of the elements are non-zero. For instance:

$$A_{\text{dense}} = \begin{bmatrix} 5 & 1 & 3 \\ 2 & 7 & 6 \\ 4 & 8 & 9 \end{bmatrix}$$

In dense matrix multiplication, every element of A and B contributes to the result, meaning the memory requirements grow proportionally to $m \cdot n + n \cdot p + m \cdot p$. This is because all elements, including zeros, are stored and computed.

A **sparse matrix**, on the other hand, contains a high proportion of zero elements. For example:

$$A_{\text{sparse}} = \begin{bmatrix} 0 & 0 & 3 \\ 0 & 7 & 0 \\ 4 & 0 & 0 \end{bmatrix}$$

Sparse matrix representations, such as the Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats, store only non-zero elements along with their indices, significantly reducing the memory footprint and enabling more efficient computation by avoiding operations on zero elements.

When matrix sparsity is high, meaning most of the elements are zeros, memory usage and computation time can be reduced considerably. For instance, if matrix A of size 1000×1000 has a sparsity level of 0.9 (90% of its elements are zeros), only about 10% of the elements need to be stored and processed. This reduction enables optimizations that are not possible with dense matrices and can greatly improve performance in large-scale applications.

2.2 Challenges in Optimizing Matrix Multiplication

The primary challenge in optimizing matrix multiplication stems from the need to balance computational efficiency with memory usage. While sparsity in matrices can offer significant reductions in both memory requirements and computation time, handling large-scale matrices with sparse data structures presents challenges in terms of algorithm design and hardware utilization. Parallelism further complicates the problem, as it requires efficient distribution of computations across multiple processing units while maintaining data coherence and minimizing memory bottlenecks.

Current research on matrix multiplication often focuses either on performance optimization or memory efficiency, but few studies provide an exhaustive comparison of algorithms considering both factors simultaneously. This gap in the literature calls for a detailed analysis of existing algorithms in a way that accounts for execution time, memory usage, and sparsity levels. This paper aims to fill this gap by evaluating and comparing various algorithms for matrix multiplication, providing a comprehensive understanding of their performance under different conditions, particularly in the context of modern computational environments.

This research is particularly important for applications in big data and machine learning, where matrix operations are critical for tasks such as training models, data transformations, and solving large-scale systems of equations. By identifying the most efficient algorithm for matrix multiplication, this paper aims to contribute valuable insights into optimizing computational performance for real-world applications.

3 Methodology

In this study, an experimental approach has been adopted to evaluate and compare various matrix multiplication algorithms. The primary goal is to identify the most efficient algorithm in terms of both execution time and memory consumption under different conditions, such as varying matrix sizes and sparsity levels. The experiments were conducted using a custom-built Java project structured into several modules, as described below:

- **Benchmarking Module**: This module includes three benchmark tests. One test is dedicated to dense matrices, another to sparse matrices, and a comprehensive test to measure both execution time and memory usage. The results of all these tests are saved in JSON format for further analysis.
- **MatrixMultiplicationFile Module**: This module allows for matrix multiplication using matrices loaded from files, specifically with the COO algorithm. It also tracks the execution time and memory usage for each multiplication operation.

- **MatrixGenerator Module**: This module provides a class for generating dense and sparse matrices based on a specified size. It can be used to create matrices with different dimensions and sparsity levels for testing purposes.
- **MatrixMultiplication Module**: This module contains multiple matrix multiplication algorithms, all of which implement the MatrixMultiplier interface. These algorithms include the basic matrix multiplication, optimized versions, and more advanced methods such as Strassen's algorithm. Each algorithm provides a method for multiplying double-type matrices.

For more detailed information, please refer to the README in the GitHub repository, which contains further explanations and instructions for using each module.

3.1 Algorithm Variants

In this study, several matrix multiplication algorithms have been tested, each designed to address different performance needs, such as reducing computation time or improving memory usage. The algorithms tested include standard approaches, optimized versions, and methods specifically adapted for sparse matrices. Below is an in-depth overview of each algorithm variant:

- **Basic Matrix Multiplication**: The basic matrix multiplication algorithm, also known as the *triple-loop* algorithm, is a straightforward approach that multiplies two matrices A and B by iterating over each element. For matrices of size $n \times n$, it has a computational complexity of $O(n^3)$, as it requires three nested loops to compute the result matrix C. While simple, this method can be inefficient for large matrices due to its high computational cost and memory requirements.
- **Strassen's Matrix Multiplication**: Strassen's algorithm is a divide-and-conquer method that reduces the number of multiplications needed for matrix multiplication. Instead of performing $O(n^3)$ operations, Strassen's method reduces the complexity to $O(n^{\log_2 7}) \approx O(n^{2.81})$, by breaking down the matrices into smaller sub-matrices and recursively applying the algorithm. This reduction in complexity makes Strassen's method more efficient than the basic algorithm for large matrices, though it may require additional memory due to the overhead of storing intermediate matrices.
- **Optimized Cache Multiplication**: The optimized cache multiplication algorithm aims to improve performance by minimizing cache misses. In large-scale matrix operations, memory access times can be a bottleneck. This algorithm reorganizes matrix access patterns to leverage spatial and temporal locality, thereby improving cache utilization. This is achieved by dividing matrices into blocks, performing computations within these blocks, and ensuring that data loaded into the cache is reused before being evicted.
- **Winograd's Algorithm**: Winograd's algorithm is an optimization of the basic matrix multiplication method that reduces the number of required multiplications by using additional additions and subtractions. Winograd's method is particularly useful when the cost of multiplications is higher than that of additions, as it reduces the total multiplication count, leading to potential speedups. Although Winograd's algorithm offers computational advantages, it may increase the number of required addition operations, which could affect performance depending on the hardware used.
- **Row-Column Major Multiplication**: This algorithm adapts the standard matrix multiplication to improve memory access patterns by handling the matrices in either row-major or column-major order. By aligning memory access patterns to the matrix's storage layout, this method aims to reduce memory access delays and improve cache efficiency, particularly when matrices are stored in a specific memory layout. This optimization can be useful in performance-sensitive applications where consistent memory access is critical.
- **Block-Based Matrix Multiplication**: The block-based matrix multiplication algorithm divides the input matrices into smaller blocks or sub-matrices, and performs multiplication within these blocks rather than directly across the entire matrix. This approach takes advantage of cache locality by working on smaller chunks of data that fit in the processor's cache. The computational complexity of block-based multiplication remains $O(n^3)$, but by focusing on smaller sub-matrices, it reduces the number of cache misses, leading to better overall performance in practical implementations, especially for large matrices.
- **Parallel Matrix Multiplication**: The parallel matrix multiplication algorithm leverages multiple processors or cores to divide the matrix multiplication task. Each processor works on a portion of the matrix, performing the multiplications and summing the partial results. Parallelizing the matrix multiplication process reduces the overall execution time and allows for scaling the algorithm to larger matrix sizes and higher processor counts. The parallel algorithm typically uses divide-and-conquer strategies or row-wise and column-wise partitioning to distribute the computation. The performance of this method is highly dependent on the number of available processors, the communication overhead between processors, and the matrix size, with larger matrices showing significant performance gains when using multiple cores.

- **COO (Coordinate Format) Sparse Matrix Multiplication**: In applications where matrices are sparse, the Coordinate List (COO) format provides a more efficient storage structure by only storing non-zero elements and their respective row and column indices. COO sparse matrix multiplication leverages this structure to skip computations involving zero elements, reducing both memory usage and computation time. The COO algorithm is well-suited for sparse matrices with few non-zero elements distributed across the matrix, though it may become inefficient if the non-zero elements are highly clustered.
- **CSC (Compressed Sparse Column) Sparse Matrix Multiplication**: The Compressed Sparse Column (CSC) format organizes sparse matrices by storing non-zero elements in a single column-wise structure. This format is particularly advantageous when the matrix multiplication involves operations concentrated in columns, as it allows rapid access to non-zero elements within each column. CSC multiplication is often preferred in applications that perform column-based operations, such as certain scientific and engineering simulations.
- **CSR (Compressed Sparse Row) Sparse Matrix Multiplication**: Similar to CSC, the Compressed Sparse Row (CSR) format optimizes storage and access for sparse matrices by storing non-zero elements row-wise. This format is beneficial for row-based operations, enabling efficient access to non-zero elements within each row. CSR is commonly used in applications requiring fast row-based processing, such as machine learning and data analysis tasks. This algorithm can significantly reduce computational time and memory usage for sparse matrices with a high row-based sparsity structure.

4 Experiments

In this section, the experiments conducted to evaluate the performance of various matrix multiplication algorithms are described. The experiments focus on two main factors: execution time and memory usage. Different matrix sizes and sparsity levels were tested to understand how these factors impact the performance of each algorithm. A set of experiments was carried out using dense matrices with sizes ranging from 64 to 2048, and sparse matrices with sparsity levels of 0.7, 0.8, and 0.9, tested up to matrix sizes of 512.

For both dense and sparse matrices, benchmarks were conducted. However, for sparse matrices, due to computational costs, the benchmarking was limited to matrix sizes up to 512. Additionally, an exhaustive test was performed using `System.nanoTime()` to measure execution time and memory usage for various sparsity levels ranging from 0.1 to 0.9, with matrix sizes from 64 to 2048. Memory usage was monitored using `ManagementFactory.getMemoryMXBean()`.

All tests were performed on a system running Windows 11 with 32 GB of RAM. A fixed number of threads was used for all parallelized algorithms to maintain consistency across experiments. To ensure accuracy, each experiment was preceded by two warm-up iterations, followed by five measurement iterations for dense matrices and three for sparse matrices.

The performance data were recorded in line graphs, stored in JSON files for future reference, and can be accessed on GitHub for additional detail. The JSON files include indicators of whether each algorithm was efficient in terms of time or memory usage, based on variable thresholds applied according to matrix size.

4.1 Benchmarks

4.1.1 Dense Matrices

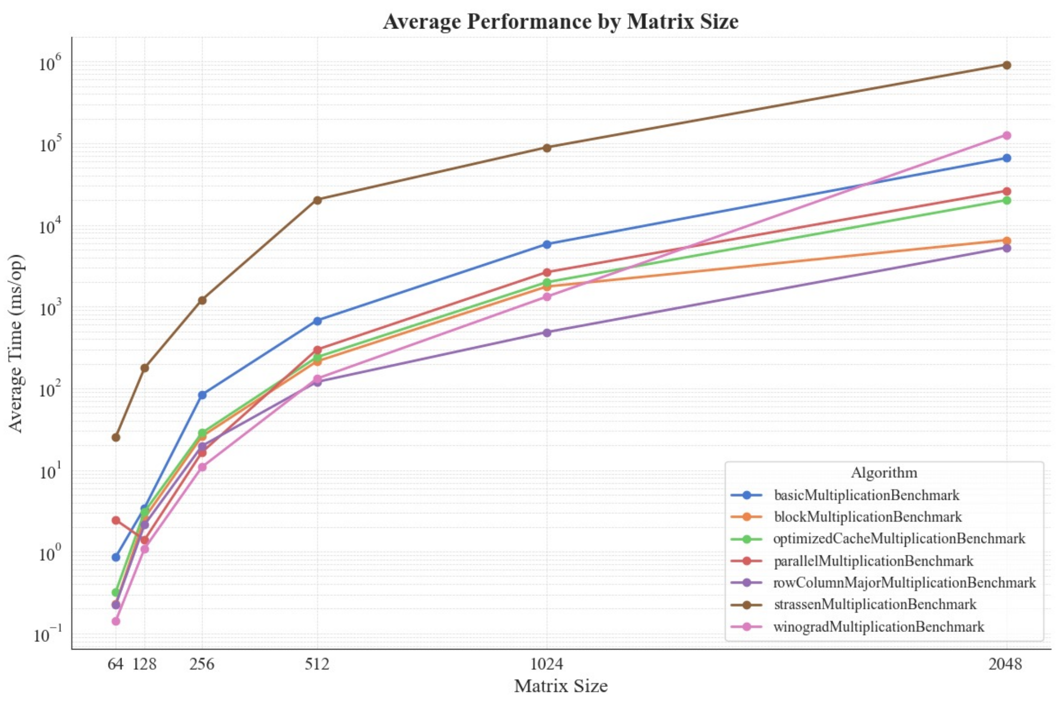

Figure 1: Average Performance by Matrix Size

In this experiment, the performance of seven matrix multiplication algorithms (Basic, Block, OptimizedCache, Parallel, RowColumnMajor, Strassen, and Winograd) was evaluated on dense matrices of various sizes. The matrix sizes varied from 64 to 2048, and the performance was measured in terms of average execution time (ms/operation) for each algorithm. The results of this experiment are shown in Figure 1.

The key observations from the experiment are as follows:

- **Strassen's Algorithm** shows a notably higher execution time compared to the other algorithms, especially for larger matrices (e.g., 2048). This indicates that Strassen's algorithm does not scale well with increasing matrix size and performs poorly in this case.
- **Basic Matrix Multiplication** also exhibits relatively poor performance, especially as the matrix size increases. This is expected since the basic algorithm has cubic time complexity.
- **Winograd's Algorithm** performs well for smaller matrices, but its execution time increases significantly as the matrix size grows, eventually surpassing other algorithms. This suggests that although Winograd's algorithm has an advantage for small-sized matrices, its performance is not optimal for larger matrices.
- The **Block Multiplication** and **RowColumnMajor Multiplication** algorithms perform significantly better, with RowColumnMajor Multiplication being the most efficient. These algorithms show a much lower execution time as the matrix size increases, demonstrating better scalability.
- **Optimized Cache Multiplication** also performs very well, though not quite as efficiently as the Block algorithm, but still surpasses the basic and Strassen algorithms in terms of performance.

One surprising result is the performance of Strassen's algorithm, which was expected to outperform the basic method but instead showed a higher execution time for larger matrices. This may be attributed to the algorithm's overhead for matrix division and the fact that it doesn't take full advantage of the hardware.

As the matrix size increases, the execution time for all algorithms increases, which is expected due to the cubic complexity of matrix multiplication. However, the Block and RowColumnMajor algorithms scale better than the others, showing significantly lower execution times even for the largest matrix sizes.

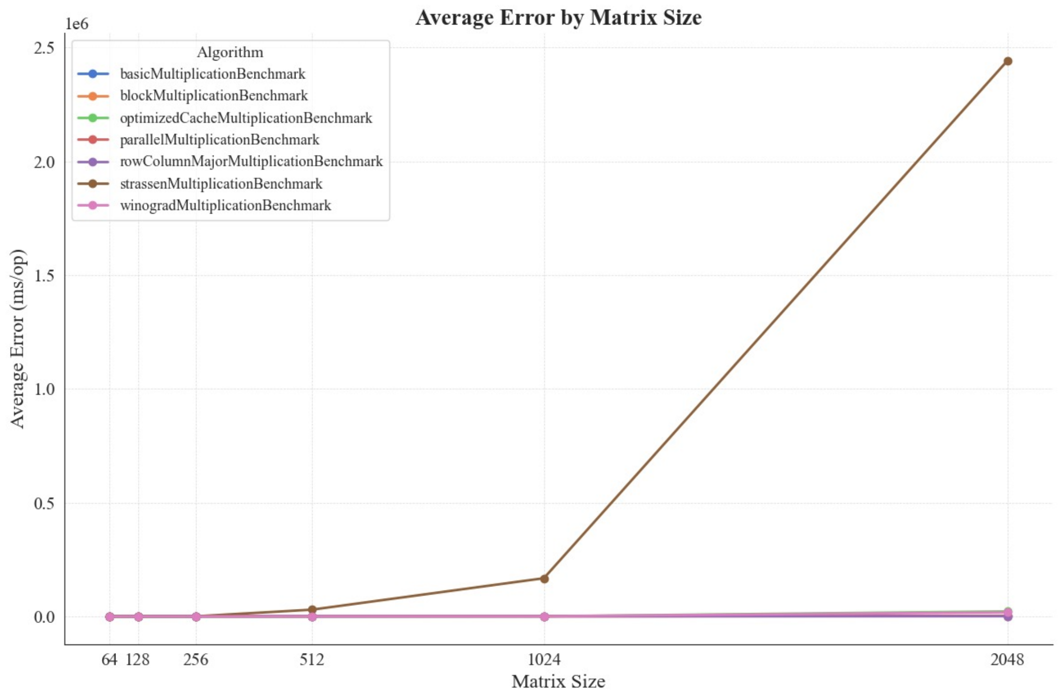

Figure 2: Average Error by Matriz Size

In this experiment, the average error (ms/operation) for the matrix multiplication algorithms (Basic, Block, Optimized-Cache, Parallel, RowColumnMajor, Strassen, and Winograd) was evaluated using dense matrices of varying sizes. The matrix sizes ranged from 64 to 2048, and the results are shown in Figure 2.

The key observations from this experiment are as follows:

- **Strassen's Algorithm** shows a significantly higher average error, especially as the matrix size increases. This suggests that Strassen's algorithm introduces greater computational error as the problem size grows, which is a surprising result considering its theoretical performance advantages.
- The **Basic Matrix Multiplication** and other algorithms (Block, OptimizedCache, Parallel, RowColumnMajor, Winograd) show relatively similar behavior, with their average error remaining relatively stable and low across different matrix sizes. This indicates that these algorithms maintain accuracy well as matrix size increases.
- **Block Multiplication** and **OptimizedCache Multiplication** perform similarly, showing minimal error even as matrix sizes increase. These algorithms are more stable in terms of error than Strassen.

Overall, the error analysis confirms that most algorithms exhibit stable error behavior, with Strassen being the clear outlier. As expected, the error increases slightly as the matrix size grows, but the increase is far more pronounced for Strassen, especially for larger matrices. This indicates that Strassen's algorithm might be less suitable for scenarios requiring high accuracy, particularly as the matrix size increases.

4.1.2 Sparse Matrices

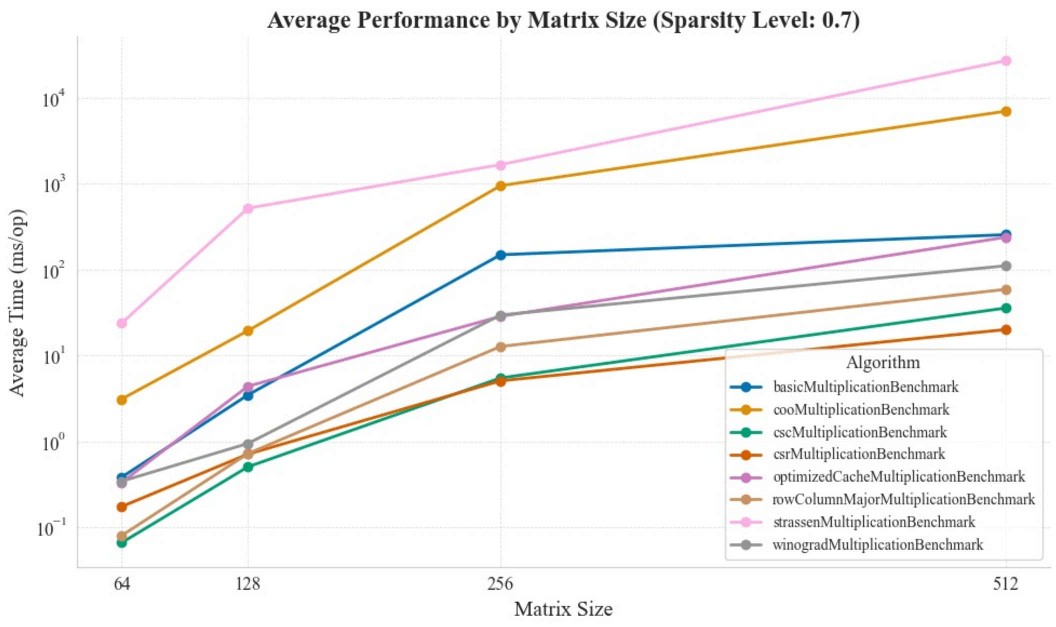

Figure 3: Average Performance by Matrix Size (Sparsity Level: 0.7)

In this experiment, the performance of the matrix multiplication algorithms—Basic, OptimizedCache, Parallel, RowColumnMajor, Strassen, Winograd, as well as the sparse matrix algorithms COO, CSR, and CSC—was evaluated on sparse matrices with a sparsity level of 0.7. The matrix sizes varied from 64 to 512, and the performance was measured in terms of average execution time (ms/operation) for each algorithm. The results of this experiment are shown in Figure 3.

The key observations from the experiment are as follows:

- **Strassen's Algorithm** continues to show the highest execution time among all algorithms, especially as the matrix size increases. This highlights that Strassen's algorithm is not well-suited for sparse matrices and suffers from performance issues when handling high sparsity.
- **COO (Coordinate List)** also exhibits a high execution time compared to other sparse matrix algorithms, though it performs better than Strassen. The COO format is not the most efficient for this experiment's matrix sizes and sparsity level.
- The **Basic Matrix Multiplication** also shows a relatively high execution time, following a similar trend to the COO format.
- The **CSR (Compressed Sparse Row)** and **CSC (Compressed Sparse Column)** show significantly lower execution times than the other algorithms. These sparse matrix formats are much more efficient in terms of memory and computation for matrices with a sparsity of 0.7.
- The **RowColumnMajor**, and **OptimizedCache** algorithms show moderate execution times. These algorithms perform better than the basic and COO methods, but not as efficiently as CSR and CSC for sparse matrices.

As with dense matrices, the execution time for all algorithms increases as the matrix size grows, but the sparse matrix algorithms (CSR and CSC) manage to keep their execution times lower, thanks to the more efficient handling of sparsity.

In summary, the CSR and CSC formats are the most efficient algorithms when working with sparse matrices, particularly for high sparsity levels like 0.7, while Strassen and COO perform poorly in comparison.

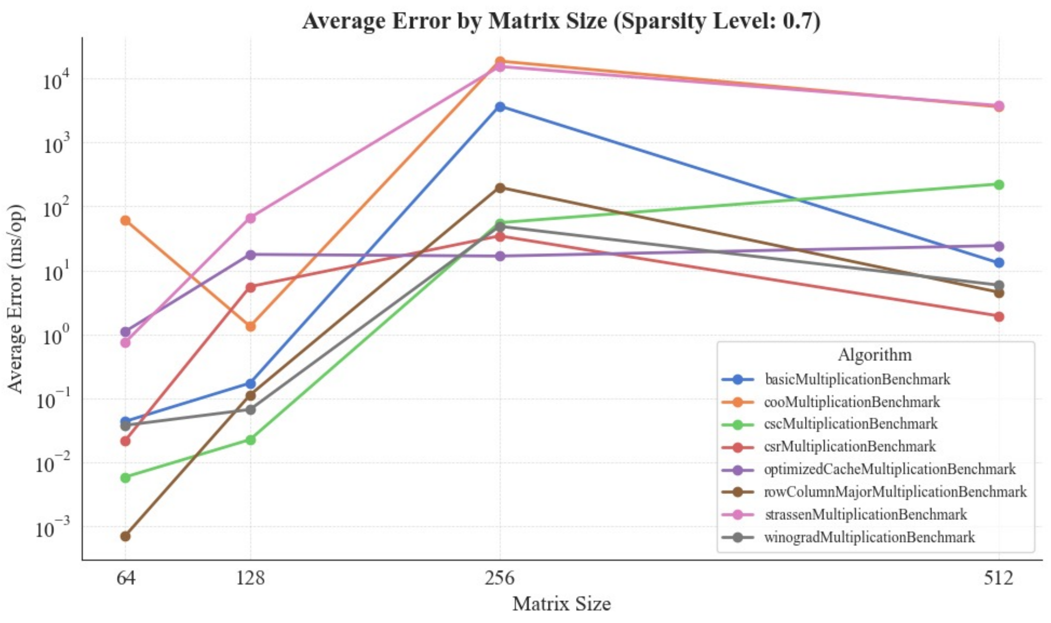

Figure 4: Average Error by Matrix Size (Sparsity Level: 0.7)

In this experiment, the error (measured in ms/operation) for the matrix multiplication algorithms—Basic, OptimizedCache, Parallel, RowColumnMajor, Strassen, Winograd, COO, CSR, and CSC—was evaluated on sparse matrices with a sparsity level of 0.7. The matrix sizes ranged from 64 to 512. The results of this experiment are shown in Figure 4.

The key observations from the experiment are as follows:

- **COO (Coordinate List)**: The error behavior for COO is quite unusual, as it initially decreases, then increases, and eventually reaches its peak, showing one of the highest error rates among all algorithms. This fluctuating error pattern indicates that COO may not perform consistently as the matrix size increases.
- **Strassen's Algorithm**: Strassen consistently presents a high error, especially for larger matrices. This suggests that although Strassen improves performance in terms of computational complexity, it incurs additional overhead in terms of error, which may outweigh its potential benefits in this case.
- **CSR (Compressed Sparse Row)**: For larger matrices, CSR shows the lowest error, indicating that this algorithm is particularly effective for handling sparse data structures with larger dimensions.
- **RowColumnMajor Multiplication**: For smaller matrices, the RowColumnMajor algorithm exhibits the lowest error, suggesting that it is highly efficient for small-scale sparse matrix operations. However, its performance increases as the matrix size grows, which leads to higher error compared to CSR.
- **Other Algorithms**: The remaining algorithms (Basic, OptimizedCache, Parallel, Winograd, and CSC) exhibit relatively stable error patterns throughout the matrix sizes, with no significant variations in error rates.

As the matrix size increases, the error typically becomes larger, as is expected with more complex algorithms that involve more calculations. However, some algorithms, such as CSR, maintain low error rates even as the matrix size grows, making them well-suited for large sparse matrices.

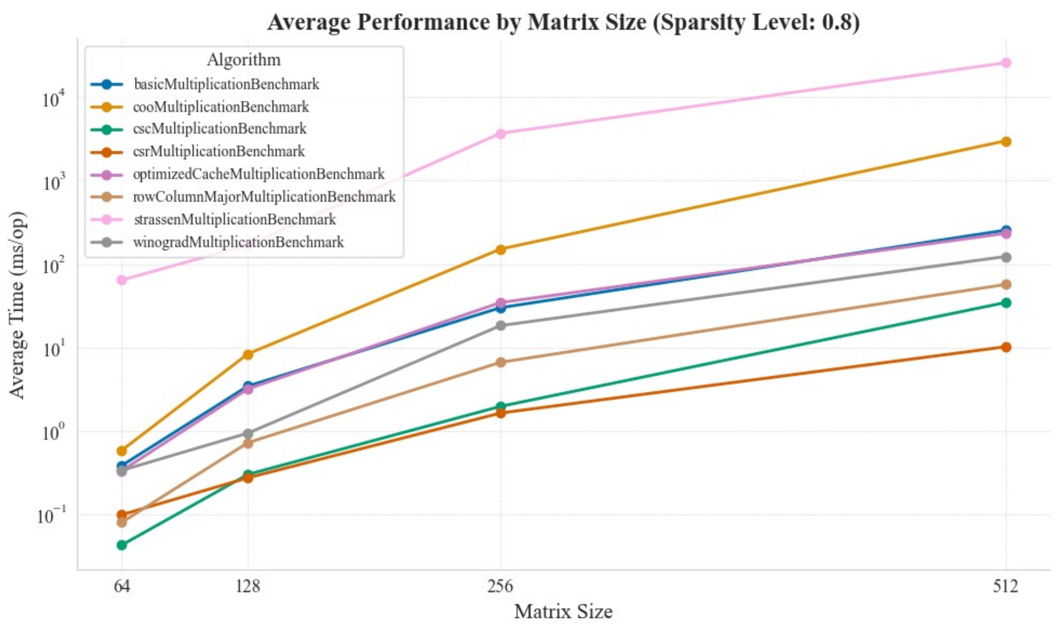

Figure 5: Average Performance by Matrix Size (Sparsity Level: 0.8

In this experiment, the performance of matrix multiplication algorithms on sparse matrices with a sparsity level of 0.8 has been evaluated. The matrix sizes ranged from 64 to 512, and the performance was measured in terms of average execution time (ms/operation) for each algorithm. The results of this experiment are shown in Figure 5.

The key observations from the experiment are as follows:

- **Strassen's Algorithm** continues to show the highest execution times, consistently underperforming compared to other algorithms, especially for larger matrices.
- **COO**, **CSR**, and **CSC** algorithms benefit significantly from the increased number of zero elements in the matrices. Their performance improves as sparsity increases, showing lower execution times compared to the previous experiment with a sparsity of 0.7.
- **Basic Matrix Multiplication** performs better in this case compared to the previous sparse matrix experiment. However, it still does not show the improvements that the specialized sparse matrix algorithms exhibit.
- The other algorithms, such as **Optimized Cache Multiplication**, and **RowColumnMajor**, do not show significant performance gains with the increased sparsity. These algorithms still benefit from improved cache performance, but they do not fully leverage the sparsity in the matrix.

This experiment demonstrates that sparse matrix algorithms like COO, CSR, and CSC become more efficient as the number of zero elements increases, confirming that they take better advantage of sparsity. However, Strassen's algorithm remains inefficient in this context, and the non-sparse-optimized algorithms do not exhibit significant improvements.

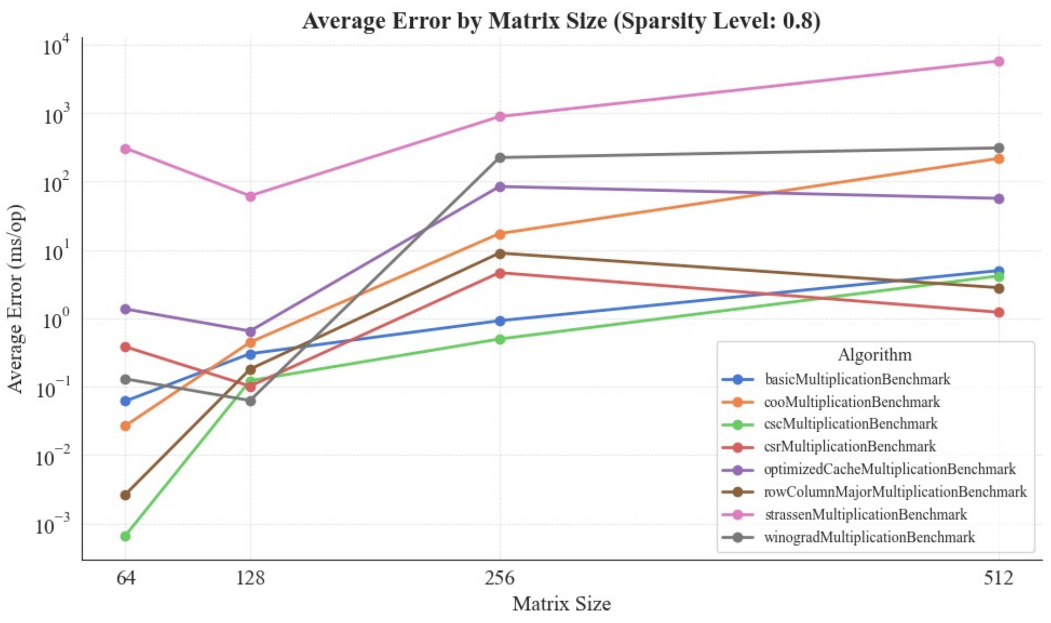

Figure 6: Average Error by Matrix Size (Sparsity Level: 0.8)

In this experiment, the error performance of the matrix multiplication algorithms—Basic, OptimizedCache, Parallel, RowColumnMajor, Strassen, Winograd, COO, CSR, and CSC—was analyzed on sparse matrices with a sparsity level of 0.8. The error metric, obtained from the JHM benchmark, reflects the deviation from the expected performance.

The key observations from the experiment are as follows:

- In comparison to the sparse matrix with a 0.7 sparsity level, the **COO, CSR, and CSC algorithms** show significantly lower error values than the other algorithms, benefiting from the higher number of zero elements in the matrix. These sparse-specific algorithms perform better in terms of error as the sparsity increases, as expected.
- **Winograd's Algorithm** continues to show high error values, especially for larger matrices, although the error is slightly lower for smaller matrices compared to the experiment with sparsity 0.7.
- **Strassen's Algorithm** remains the highest in terms of error, which suggests that the overhead introduced by the algorithm doesn't benefit from the sparsity structure.
- **OptimizedCache Algorithm** also presents higher error values compared to sparse-specific algorithms, although it shows a slight improvement compared to its performance with sparsity 0.7.
- The **CSR, CSC, Basic, and Row Major Column algorithms** exhibit similar behavior, with the basic algorithm even appearing to benefit from a higher number of zeros.

When compared to the experiment with sparsity 0.7, the error values for most algorithms are lower in this case, likely due to the increased number of zero elements in the matrix. However, **Winograd**, **Strassen**, and **OptimizedCache** still show significant error values, especially for larger matrices. These algorithms are not optimized for sparse matrices and do not benefit as much from the sparsity level of 0.8.

The results reinforce the idea that algorithms specifically designed for sparse matrices (such as COO, CSR, and CSC) are much more effective at minimizing error, particularly as the sparsity increases.

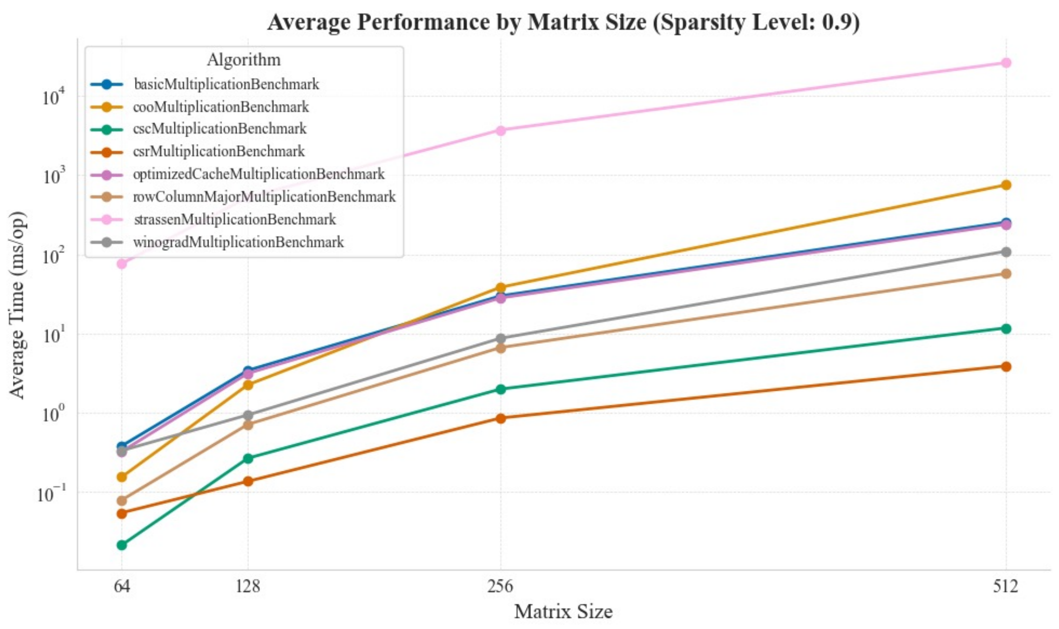

Figure 7: Average Performance by Matrix Size (Sparsity Level: 0.9

In this experiment, the performance of matrix multiplication algorithms was measured for sparse matrices with a sparsity of 0.9. The same metrics as in the previous experiments were used, focusing on the average execution time (ms/operation). The results were compared with the performance for matrices with sparsity levels of 0.7 and 0.8.

The key observations from this experiment are as follows:

- **General Decrease in Execution Time:** Compared to the results for sparsity 0.7 and 0.8, the overall execution time for matrices with sparsity 0.9 has decreased. This is especially noticeable in the **COO** algorithm, which performs significantly better as the number of zeros in the matrix increases.
- **Algorithms Optimized for Sparse Matrices:** The algorithms designed to handle sparse matrices, such as **COO**, **CSR**, and **CSC**, benefit most from the higher sparsity. These algorithms show better performance as the sparsity increases, with **COO** being the most improved in terms of execution time.
- **Other Algorithms:** For algorithms that are not specifically optimized for sparse matrices, such as **Strassen**, **Winograd**, and the basic algorithms, the performance remains relatively unaffected by the increase in sparsity. These algorithms do not benefit as much from the increase in the number of zeros in the matrices, and their performance is largely similar to the results observed with sparsity 0.7 and 0.8.
- **Comparison with Previous Sparsity Levels:** While there is a noticeable improvement for **COO**, **CSR**, and **CSC**, the rest of the algorithms behave similarly to the 0.7 and 0.8 sparsity levels. The performance improvement due to higher sparsity is more pronounced in the specialized sparse matrix algorithms.

In conclusion, increasing the sparsity from 0.7 to 0.9 leads to a decrease in execution time for sparse matrix algorithms, particularly for **COO**, while other algorithms that do not take advantage of sparsity still exhibit similar performance. The results highlight the importance of using optimized algorithms for sparse matrices to fully capitalize on the benefits of high sparsity levels.

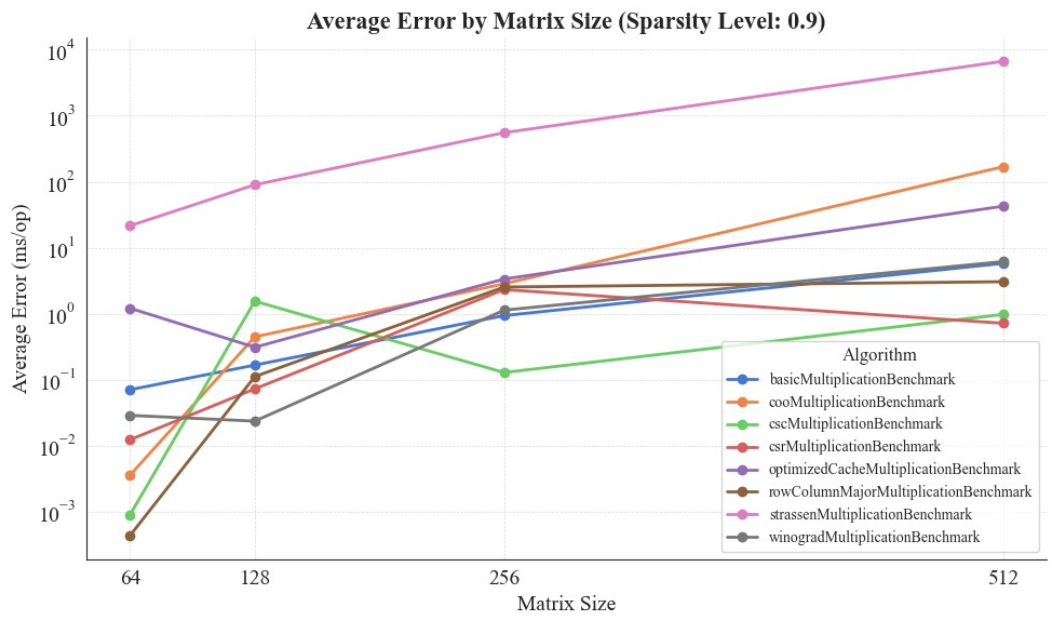

Figure 8: Average Error by Matrix Size (Sparsity Level: 0.9)

In this experiment, we measured the error of various matrix multiplication algorithms, including Basic, Block, Optimized-Cache, Parallel, RowColumnMajor, Strassen, Winograd, COO, CSR, and CSC, using sparse matrices with a sparsity level of 0.9. The error metric used is the same as in previous experiments, provided by the JHM benchmark. The results are shown in Figure 8.

The key observations from the experiment are as follows:

- In comparison to other sparsity levels, **Winograd and OptimizedCache** show surprising improvements with a higher number of zero elements. These algorithms, which previously performed poorly for sparse matrices with lower sparsity levels (e.g., 0.7), seem to benefit from the higher sparsity, as their error rates decrease.
- **COO** also shows improved performance, with a significant reduction in error compared to earlier experiments. This behavior is consistent with the trends observed in previous sparsity levels, where COO performs well as the number of zero elements increases.
- **Strassen** exhibits a marked improvement in error as well, which is surprising given its previous poor performance at lower sparsity levels. This suggests that Strassen may benefit from larger sparsity but still lags behind other algorithms in terms of overall error.
- **CSC** shows a more irregular behavior compared to other algorithms. Its performance does not improve as consistently as COO or CSR, with some unexpected spikes in error for certain matrix sizes.
- The **CSR** maintains its position as one of the most efficient algorithms, with the smallest error across most matrix sizes. This is consistent with its performance in earlier experiments with sparsity levels 0.7 and 0.8.
- The other algorithms, including **Basic** and **RowColumnMajor**, do not show significant improvements as the sparsity increases, and their error rates remain relatively stable across the various sparsity levels.

Comparing the error results for sparsity levels 0.7, 0.8, and 0.9, it is clear that many algorithms benefit from higher sparsity, particularly those optimized for sparse matrices (COO, CSR, and CSC). In contrast, algorithms that are not designed to take advantage of sparsity, such as Strassen and Winograd, show noticeable improvements at 0.9, although they still do not outperform the sparse-optimized methods.

Overall, the results indicate that as the sparsity level increases, the algorithms optimized for sparse matrices (COO, CSR, CSC) continue to provide the lowest error rates, especially for larger matrix sizes. However, certain algorithms, like Winograd and OptimizedCache, show potential for improvement when working with highly sparse matrices, making them more competitive in these scenarios.

4.2 Memory and Time

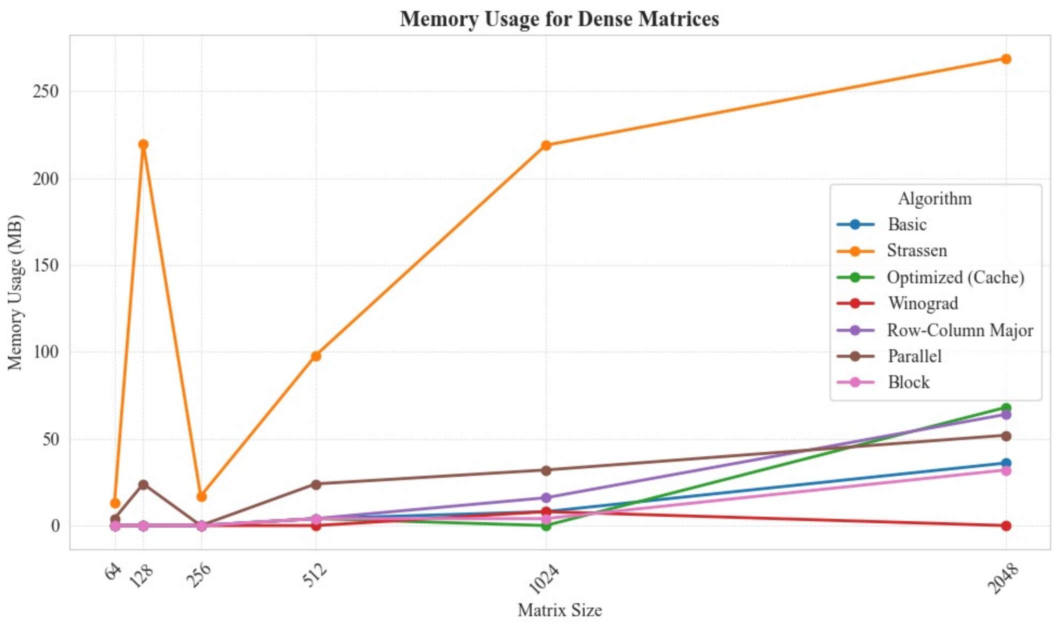

Figure 9: Memory Usage for Dense Matrices

In this experiment, we measured the memory usage (in MB) of the seven matrix multiplication algorithms—Basic, Block, OptimizedCache, Parallel, RowColumnMajor, Strassen, and Winograd—on dense matrices of various sizes. The matrix sizes ranged from 64 to 2048, and the memory usage was plotted against these sizes. The results are shown in Figure 9.

The key observations from this experiment are as follows:

- **Strassen's Algorithm** exhibits highly irregular behavior, with significant fluctuations in memory usage, making it the most memory-intensive algorithm overall. As the matrix size increases, these fluctuations become more pronounced, and the algorithm consumes significantly more memory than the others, especially for larger matrices (e.g., 2048).
- The **Winograd's Algorithm** consumes the least memory, maintaining a more stable memory usage across all matrix sizes. This makes Winograd the most memory-efficient algorithm among those tested.
- The **Optimized Cache Multiplication** performs well in terms of memory consumption for smaller matrices, but for larger matrix sizes (e.g., 2048), its memory usage increases substantially, making it less efficient in memory compared to other algorithms.
- The **Basic Matrix Multiplication** demonstrates competitive memory usage, with the memory consumption growing steadily with matrix size. Surprisingly, it does not show significant inefficiency when compared to the other algorithms.
- The **Block Multiplication** and **RowColumnMajor Multiplication** show similar behavior, with steady increases in memory usage as matrix size increases, but not as irregular as Strassen.

As expected, the memory usage of all algorithms increases as the matrix size grows, but with some important differences. The Strassen algorithm is a notable outlier, displaying significant spikes in memory usage, especially for larger matrices, while most of the other algorithms exhibit more consistent growth. The Winograd algorithm consistently uses the least memory across all matrix sizes, making it the most efficient in terms of memory consumption.

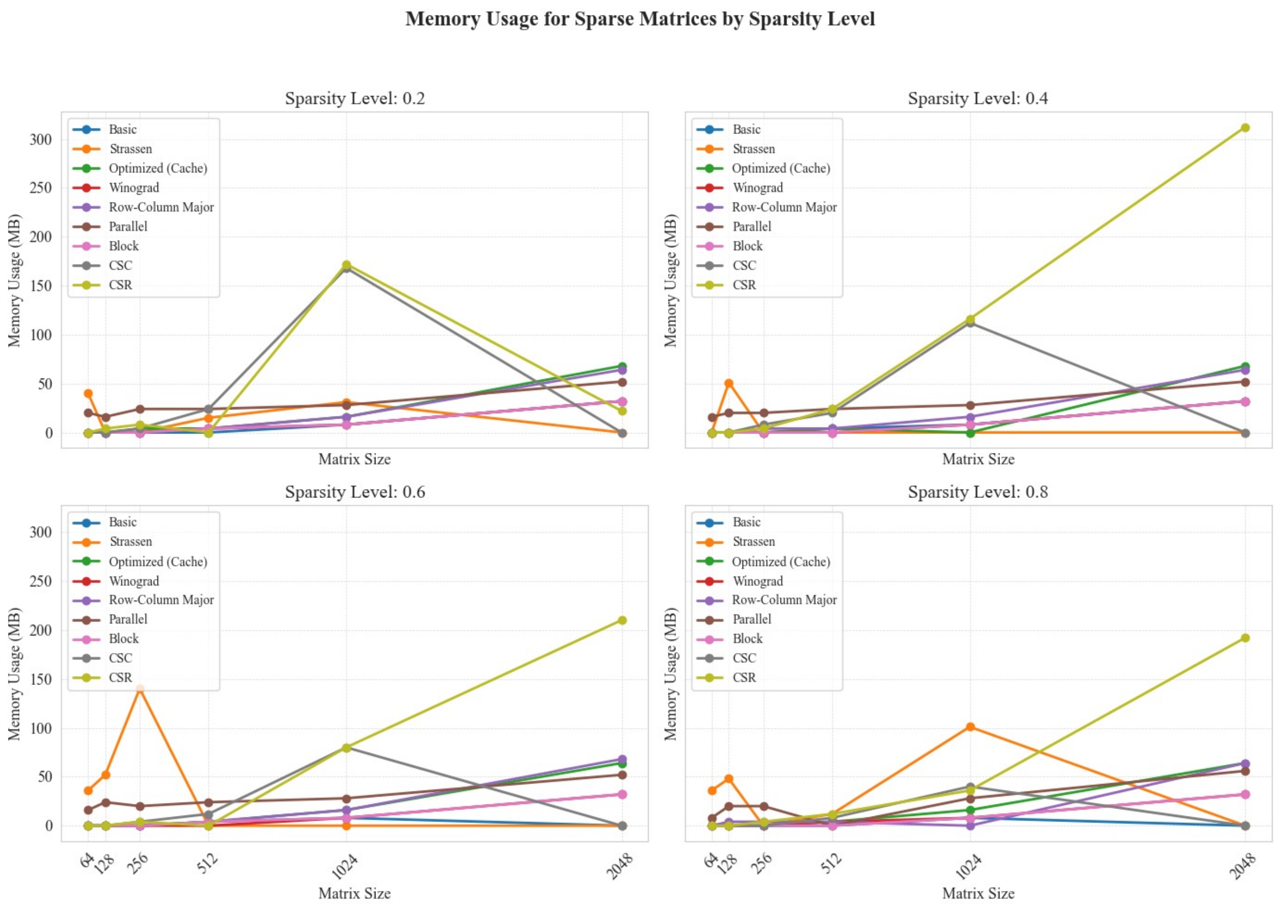

Figure 10: Memory Usage for Sparse Matrices by Sparsity Level

In this experiment, we analyzed the memory usage of various matrix multiplication algorithms—Basic, Block, Optimized-Cache, Parallel, RowColumnMajor, Strassen, Winograd, CSR, and CSC—across sparse matrices with different sparsity levels (0.2, 0.4, 0.6, and 0.8). The memory usage was measured in megabytes (MB), and the matrix sizes ranged from 64 to 2048. The results for each sparsity level are presented in four subplots, with each subplot representing one sparsity level. The memory usage for each algorithm is shown as a line, and the behavior of each algorithm is analyzed in relation to the increasing matrix size.

The key observations from this experiment are as follows:

- **CSR** generally consumes the most memory across all sparsity levels, with the exception of sparsity 0.2. For sparsity 0.2, CSR shows a comparatively lower memory consumption, but it increases dramatically with higher sparsity levels, making it the most memory-intensive algorithm overall.
- **CSC** shows a fluctuating pattern in its memory usage for sparsity levels 0.2, 0.4, and 0.6, but maintains a more stable behavior at sparsity 0.8. This suggests that the CSC algorithm is more sensitive to the sparsity level, with its performance stabilizing at higher levels of sparsity.
- **Strassen's Algorithm** exhibits irregular memory usage patterns, especially for sparsity 0.2, where it performs better compared to higher sparsity levels. As the sparsity increases, Strassen's memory consumption becomes less predictable, with noticeable peaks and drops.
- **Block, OptimizedCache, Parallel, RowColumnMajor, and Winograd** algorithms demonstrate relatively stable memory usage across the sparsity levels. The differences in memory consumption between these algorithms are minimal, with Winograd being the most memory-efficient at lower sparsity levels.
- In general, as the matrix size increases, memory consumption also increases across all algorithms. However, the more irregular patterns are observed in Strassen's and CSR, where sudden spikes in memory usage are visible.

The most significant observation is that the algorithms optimized for sparse matrices, such as CSR and CSC, show a notable improvement in memory consumption as the sparsity increases, particularly for the highest sparsity level (0.8). In contrast, algorithms that are not specifically optimized for sparse matrices, such as Strassen and the basic methods, show less improvement and even exhibit irregular memory usage patterns at higher sparsity levels.

These results highlight the importance of using algorithms designed for sparse matrices, especially as the sparsity increases, to achieve more efficient memory utilization.

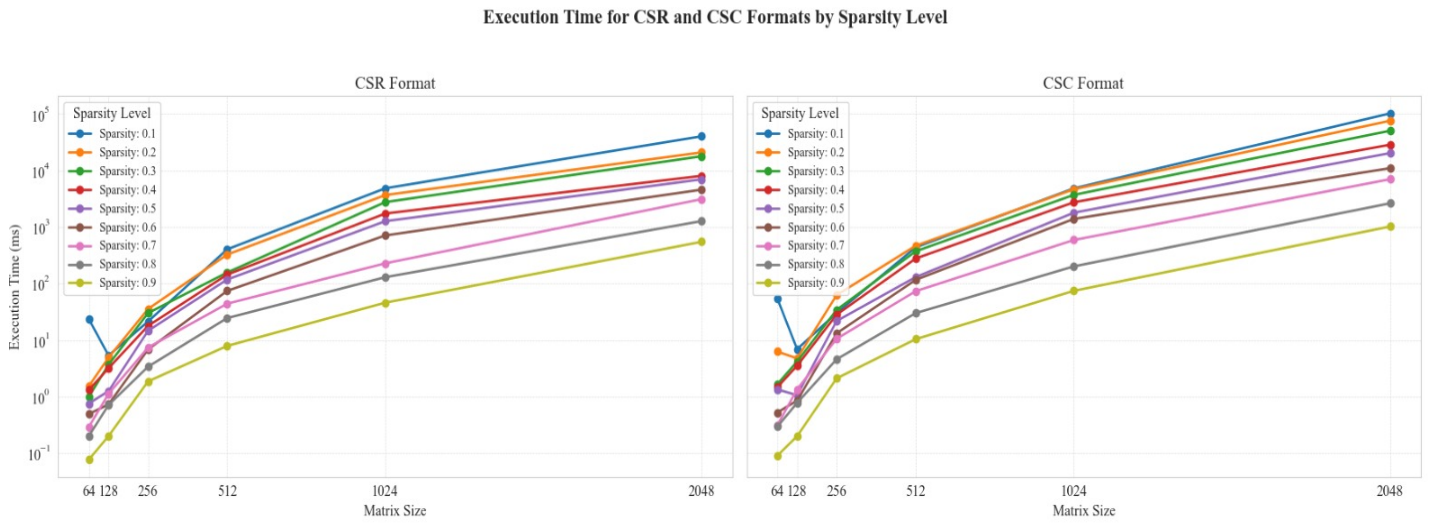

Figure 11: Execution Time for CSR and CSC Formats by Sparsity Level

In this experiment, we measured the execution time (in milliseconds) of the CSR and CSC algorithms for sparse matrices with varying sparsity levels. The sparsity levels tested were 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9. The matrix sizes ranged from 64 to 2048. The results are shown in Figure 11.

The key observations from the experiment are as follows:

- Both the **CSR** and **CSC** algorithms show similar behavior in terms of execution time across different sparsity levels.
- **CSR** consistently outperforms **CSC** in terms of execution time, particularly at higher sparsity levels. This suggests that CSR is more efficient for sparse matrix multiplication compared to CSC.
- As the sparsity increases, the execution time for both algorithms generally decreases. This is expected, as more zero elements in the matrix allow the algorithms to skip unnecessary computations.
- The matrix size from 64 to 2048 also affects the execution time. Larger matrices lead to higher execution times, though this trend is more pronounced in lower sparsity levels.
- There is a strong consistency in the behavior of both algorithms across all sparsity levels. The execution time decreases steadily as the sparsity increases, confirming that both CSR and CSC effectively take advantage of matrix sparsity.

This experiment shows the effectiveness of the CSR and CSC algorithms in handling sparse matrices. The results indicate that both algorithms benefit from increasing sparsity, with CSR showing a more stable and consistently better performance than CSC.

4.3 Experiment with the Williams/mc2depi Matrix Using the COO Algorithm

In this experiment, the COO (Coordinate List) algorithm was utilized to perform matrix multiplication with the Williams/mc2depi matrix. This matrix comes from the field of epidemiology and represents a 2D Markov model of an epidemic, as described by Williams et al. The matrix is part of a set of benchmark problems used to evaluate the performance of sparse matrix algorithms.

Matrix Characteristics:

- **Matrix Name:** mc2depi
- **Group:** Williams

- **Matrix ID:** 2377
- **Number of Rows and Columns:** 525,825
- **Nonzeros:** 2,100,225
- **Pattern Entries:** 2,100,225
- **Kind:** 2D/3D Problem
- **Symmetric:** No
- **Structural Rank:** 525,825
- **Numeric Symmetry:** 0%
- **Positive Definite:** No

This matrix represents a large sparse structure, where the nonzero entries (2,100,225) are relatively sparse compared to the total number of entries in the matrix (which is approximately 276 billion, as the matrix is square with 525,825 rows and columns). The matrix is asymmetric, which rules out certain optimization techniques like those used for symmetric matrices, and is not positive definite, meaning it does not exhibit properties that would make it suitable for certain types of factorization methods like Cholesky decomposition.

The use of the COO format in this case is particularly suitable for representing sparse matrices. The COO format stores only the nonzero entries, along with their row and column indices. This reduces both the memory footprint and computational cost associated with sparse matrix operations. The nonzero entries represent the critical interactions between states in the epidemiological model, and efficient multiplication of such a sparse matrix is key for simulating the system dynamics.

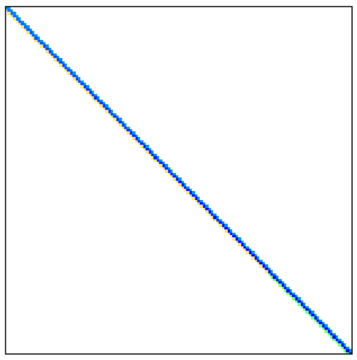

Figure 12: The Williams/mc2depi matrix structure.

Performance Results: The results for this experiment are as follows:

- **Execution Time:** 342 ms
- **Memory Usage:** 320 MB

These results show that the COO algorithm can handle relatively large sparse matrices efficiently. Given that the matrix is quite large, with over 500,000 rows and columns, the fact that the execution time is relatively low (342 ms) suggests that the COO format is well-suited for this type of sparse matrix multiplication. This is further supported by the fact that the memory usage is manageable at 320 MB, indicating that only the nonzero elements are being stored and processed, which is a significant advantage over dense matrix representations.

Implications of Results: The results highlight several important considerations:

- **Scalability of COO for Large Sparse Matrices:** Despite the large size of the matrix (525,825 rows and columns), the COO format demonstrated good performance, with both reasonable execution time and memory usage. This suggests that the algorithm can scale effectively with matrices that have a large number of rows and columns, as long as the matrix remains sparse.

- **Memory Efficiency:** The memory usage of 320 MB is quite efficient given the large size of the matrix. This is due to the sparse nature of the matrix, where only the nonzero entries (around 2.1 million) are stored in memory. This is a clear advantage of using sparse matrix formats like COO, as opposed to dense formats that would require substantially more memory for matrices of this size.
- **Execution Time Performance:** The execution time of 342 ms is relatively fast, which is promising for real-time simulations or computations that require rapid matrix multiplication, such as those used in epidemiological modeling. It demonstrates that even large, sparse matrices can be processed quickly using the appropriate algorithm and representation (in this case, COO).

In summary, the use of the COO algorithm for multiplying the Williams/mc2depi matrix proves to be an efficient approach for handling large sparse matrices. The algorithm performs well in terms of both execution time and memory usage, making it an appropriate choice for applications dealing with large-scale sparse matrices, particularly in fields such as epidemiology where such matrices often arise.

4.4 Matrix Size Limitations and Benchmark Justification

The maximum matrix sizes tested can be found in the test_size_results file. Overall, it is evident that the Strassen algorithm is particularly inefficient for this case.

Additionally, there are specific cases where the basic algorithm also encountered bottlenecks. For this reason, and to ensure that computational limits of the available equipment would not present issues, the benchmarks were designed with these constraints in mind.

5 Conclusions

This research provides an in-depth benchmarking of matrix multiplication algorithms, focusing on execution time, memory consumption, and the effects of sparsity across various algorithmic approaches. Matrix multiplication is essential in numerous domains, including machine learning, scientific simulations, and big data analytics, where efficient computation directly impacts overall system performance. This study's findings underscore the critical role of algorithm choice in optimizing matrix multiplication for different data characteristics.

In sparse matrix scenarios, the CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) formats demonstrated notable improvements in execution time, benefiting applications with high sparsity by reducing computation through selective processing of non-zero elements. However, these algorithms did not achieve substantial reductions in memory usage, as their design focuses on processing efficiency rather than memory optimization. This distinction highlights the need for careful algorithm selection based on specific requirements—favoring CSR and CSC for scenarios where execution speed is paramount but memory efficiency is secondary.

In contrast, Strassen's algorithm performed poorly across both metrics, exhibiting inefficiencies in execution time and memory consumption that limit its applicability in large-scale matrix operations. This underscores the algorithm's limitations when handling large, complex matrices, where its theoretical performance gains are overshadowed by practical drawbacks in both computation and resource usage.

For dense matrices, row-column major and block multiplication algorithms showed superior scalability and consistent performance, maintaining lower execution times and efficient memory usage as matrix sizes increased. These algorithms are well-suited for dense matrix operations where scalability and balanced resource utilization are critical.

The implications of this study extend to practical applications in big data and machine learning, where selecting a matrix multiplication algorithm tailored to the data's density and scale can significantly enhance computational performance. By offering a detailed comparative analysis across diverse matrix structures and algorithm types, this research provides valuable insights into algorithm selection that meet the demands of data-intensive environments. These findings offer a foundation for future work aimed at further optimizing matrix multiplication, supporting the development of high-performance solutions for large-scale computational tasks.

6 Future Work

Future work will focus on the exploration of parallel computing techniques to improve matrix multiplication performance. This includes implementing a parallelized version of matrix multiplication using multi-threading and libraries such as OpenMP. The primary goal is to leverage multiple processing cores to reduce computation time significantly, particularly for large matrices. Additionally, an optional vectorized approach will be investigated, potentially using SIMD (Single Instruction, Multiple Data) instructions to further enhance performance by processing multiple data points in parallel.

The effectiveness of these optimizations will be measured by key performance metrics, including:

- **Speedup:** Comparison of execution time between the parallelized and basic matrix multiplication algorithms.
- **Efficiency of Parallel Execution:** Assessment of speedup per thread to determine scalability and the effectiveness of resource utilization.
- **Resource Usage:** Analysis of core usage and memory consumption during parallel execution.

Testing with large matrices will allow for a comprehensive evaluation of the performance gains achieved through both parallelization and vectorization. This work aims to establish a more efficient matrix multiplication approach suitable for large-scale computations, highlighting the potential of parallel and vectorized methods for enhancing computational performance.