UNIVERSITY OF LAS PALMAS DE GRAN CANARIA

Optimized Matrix Multiplication Approaches and Sparse Matrices

Casimiro Torres, Kimberly

Data Science and Engineering

November 10, 2024

Abstract

This project explores and evaluates various optimization techniques in matrix multiplication, both for dense and sparse matrices. Starting with basic matrix multiplication as a reference, optimizations such as the Strassen algorithm, the Winograd algorithm, loop unrolling, parallelization, vectorization, cache blocking, and specific formats for sparse matrices like Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) are applied to improve performance in terms of execution time and memory usage. Additionally, the impact of sparsity levels on both sparse and dense matrices is studied, examining how these levels affect computational efficiency in advanced processing architectures. The results show significant performance improvements compared to the basic approach and highlight limitations and bottlenecks when handling large matrices, providing a comprehensive view of how optimization techniques can influence the efficiency of matrix multiplication.

Keywords: Matrix multiplication, optimization, dense matrices, sparse matrices, Strassen algorithm, Winograd algorithm, loop unrolling, parallelization, vectorization, cache blocking, CSR, CSC, Benchmark, sparsity levels, execution time, memory usage, bottlenecks, Big Data

1 Introduction

The growing demand for high-performance applications has driven the industry toward designing systems with processors specialized in handling large amounts of data efficiently. Dense and sparse matrices are essential components in a variety of scientific, industrial, and artificial intelligence applications. Matrix multiplication, a fundamental operation in scientific computing, becomes an optimization challenge when dealing with large matrices, as these calculations require careful management of computational resources to maximize performance and minimize energy consumption.

Dense matrices, characterized by a high proportion of non-zero values, present specific challenges in terms of memory usage and data access. Despite their potential to leverage the inherent parallelism of modern architectures, operations on dense matrices can quickly saturate memory and processing resources, especially when scaled to very large sizes. In contrast, sparse matrices, dominated by zero values, allow for a more efficient approach in terms of storage and computation by focusing only on non-zero elements. These matrices are highly useful in applications where data is mostly sparse, such as in graphs and machine learning models.

In this project, various optimization techniques for matrix multiplication are explored and implemented for both dense and sparse matrices. The optimizations include ba-

sic matrix multiplication as a foundation; Strassen's algorithm, which reduces computational complexity by dividing the matrix into smaller submatrices; Winograd's algorithm, which decreases the number of multiplicative operations for certain matrix configurations; loop unrolling, which improves performance by reducing cycle overhead; parallelization, which distributes multiplication operations across multiple execution threads; vectorization, which utilizes SIMD (Single Instruction, Multiple Data) instructions to process multiple data elements in parallel; and cache blocking, which organizes memory access into cache-optimized blocks, reducing cache misses in large matrices. Additionally, specific formats for sparse matrices, such as Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC), are used, enabling efficient handling of matrices where most elements are zeros. These techniques, along with the analysis of the impact of sparsity levels on both sparse and dense matrices, provide a comprehensive view of how to improve computational efficiency in advanced processing architectures.

These optimization techniques were implemented and evaluated in different contexts, considering both dense and sparse matrices with varying sparsity levels. By examining how each technique performs based on the type and size of the matrix, significant improvements were achieved in terms of execution time and memory usage compared to the basic matrix multiplication approach. Additionally, the limitations and bottlenecks associated with each method were analyzed, including memory bandwidth and cache access issues, which are critical aspects when dealing with advanced architectures aimed at maximizing computational performance.

This study provides a comprehensive view of how optimization techniques can be applied to improve performance in large-scale applications, highlighting the practical implications of each approach and offering recommendations on how to maximize resource efficiency for applications involving intensive matrix computations.

2 Problem Statement

Matrix multiplication, while fundamental in scientific computing, faces inherent limitations that hinder its performance in large-scale applications. Operations with dense and sparse matrices require different approaches and optimization techniques due to their distinctive characteristics in terms of data density and memory requirements. Without effective optimization strategies, these operations can result in excessive resource consumption and prolonged execution times, which is unsustainable for applications demanding efficiency and speed.

One of the primary problems to address is how to optimize resource usage in both dense and sparse matrix multiplication. For dense matrices, efficient memory and processing management is crucial, as the high proportion of non-zero data can quickly deplete resources on conventional systems. On the other hand, sparse matrices present the challenge of leveraging their sparse structure to reduce computation time without sacrificing result accuracy, which requires specialized storage and processing techniques.

Another significant challenge is balancing the complexity of optimization techniques with their effectiveness for different matrix sizes. Advanced techniques, such as the Strassen algorithm and loop unrolling, can improve performance, but their implementation and adjustment for various architectures require thorough analysis to maximize benefits. Additionally, the sparsity level in sparse matrices introduces an additional variable, where very low or very high sparsity levels can significantly impact performance and efficiency.

Finally, the project also focuses on analyzing and evaluating the performance of each implemented optimization technique across different scenarios, considering factors like memory usage and processing efficiency. Through testing on matrices of various sizes and sparsity levels, the limits of each technique are explored, identifying optimal con-

figurations to maximize performance in each case. This detailed analysis allows for recommendations on more efficient and adaptable matrix multiplication, capable of meeting the demands of data-intensive and computational resource-intensive applications.

Furthermore, the project examines how each technique handles memory access and processing requirements, aiming to minimize bottlenecks and ensure more efficient resource utilization. This enables the development of scalable and optimized solutions for matrix multiplication, adaptable to a wide range of scientific and industrial applications, as well as diverse hardware configurations.

3 Methodology

3.1 Development Environment and System Specifications

The project was developed in the integrated development environment IntelliJ IDEA, a robust and widely used platform for Java application development. This environment enables effective project management, facilitating code organization, integration of external libraries, and testing. Additionally, it provides advanced tools for performance analysis, debugging, and benchmarking, essential aspects for a comprehensive evaluation of the implemented optimization techniques.

The system used for development and testing has the following specifications:

- **Processor:** 13th Gen Intel(R) Core(TM) i7-13700H, 2.40 GHz. This high-performance processor allows efficient execution of multiple threads, making it ideal for parallelization tests and other optimization techniques implemented in the project.
- **RAM:** 16 GB. The memory capacity is suitable for handling memory-intensive operations, especially when working with large matrices and varying levels of sparsity.
- **Operating System:** Windows 11 Home, 64-bit edition. It provides a stable and compatible environment with the development tools and libraries used, facilitating the implementation and evaluation of optimization techniques in the project.

This configuration is suitable for conducting performance evaluations in matrix multiplication operations of different sizes and levels of complexity, allowing for the measurement of the impact of each optimization technique in terms of efficiency and resource usage.

3.2 Implementation of Optimization Techniques

The project has been developed to implement and evaluate multiple optimization techniques in matrix multiplication, both for dense and sparse matrices. The implementation and code organization are structured into several modules, each designed to fulfill a specific function in the testing and optimization system.

The project structure is organized into the packages Benchmark, Dense, Sparse, Utils, and Resources. Each package serves a specific function in the development and execution of optimization and benchmarking techniques.

3.2.1 Benchmark

The project is organized in a structure that allows the evaluation of the performance of various optimized matrix multiplication techniques, clearly distinguishing between dense and sparse matrices, and providing tools to compare their efficiency in terms of execution time and memory usage. The main results folder, benchmarkResults, is divided into three subfolders: dense, sparse, and combined. Each subfolder contains two CSV files, one for storing the execution time benchmark results and another for recording memory usage. These subfolders reflect the different

categories of matrices and comparisons conducted in the project.

Additionally, in the benchmark folder, there are three source code subfolders: dense, sparse, and combined. Each of these folders contains two Java classes: MatrixMultiplicationBenchmark, dedicated to running the execution time benchmark for the various matrix multiplication techniques, and MemoryUsageBenchmark, designed to measure and record memory usage during the execution of each multiplication technique. Each of these classes performs an in-depth analysis of the performance of the optimized techniques on different types of matrices. The classes in dense focus on fully dense matrices, those in sparse work with sparse matrices and their specific formats, and those in combined enable a comparison between dense and sparse matrices by adjusting the sparsity level in dense matrices.

The combined subfolder has a special purpose: it allows the comparison of multiplication techniques when dense matrices are subjected to levels of sparsity. This is achieved by adjusting a sparsityLevel parameter that introduces zeros into the dense matrices, simulating the behavior of sparse matrices. This approach enables direct comparison between dense matrices and their sparse versions, and helps observe how different levels of sparsity affect the performance of various multiplication techniques.

The MatrixMultiplicationBenchmark class in each subfolder performs execution time benchmarks using the JMH (Java Microbenchmark Harness) library, which provides high-precision microbenchmarking tools in Java. This class configures and runs multiple iterations and warm-up phases before taking measurements, ensuring that the results are stable and consistent. The benchmarks are set to measure average execution time in milliseconds, recording detailed data for each technique.

The MemoryUsageBenchmark class in each subfolder is designed to measure memory usage during the execution of multiplication techniques. This class uses specific methods to capture the memory used before and after executing each technique, calculating the total memory consumption in megabytes. This memory analysis is crucial for identifying techniques that, while fast, may have high memory usage, which is relevant when working with large matrices.

In summary, the CSV files within benchmarkResults are essential for analyzing and comparing the results. These reports allow a detailed analysis of how each technique performs in terms of efficiency and resource consumption, helping identify the most suitable technique for different types of matrices and configurations. Additionally, these reports serve as a basis for comparing the performance of optimization techniques in both dense and sparse matrices, providing a comprehensive view of each algorithm's behavior in various usage scenarios.

3.2.2 Dense

The dense package consists of algorithms and specific optimizations for performing dense matrix multiplication, covering both basic and advanced techniques aimed at analyzing and improving computational efficiency. Within this package, there are two subpackages: algorithms and optimizations, each focused on a category of optimization techniques.

In algorithms, the BasicMultiplication file implements the $O(n^3)$ complexity matrix multiplication algorithm, which serves as a reference to compare the impact of applied optimizations in terms of execution time and memory usage. This implementation establishes a fundamental benchmark, as its $O(n^3)$ complexity makes it ideal for assessing efficiency improvements when applying optimized methods.

The StrassenAlgorithm file implements Strassen's algorithm, a technique that optimizes multiplication by dividing matrices into smaller submatrices, thus reducing theoretical complexity compared to the basic

method. This algorithm enables observation of performance improvements in large matrices by reducing the number of required operations. WinogradAlgorithm also implements an optimized method, focusing on reducing multiplications and performing operations more efficiently, providing another point of comparison regarding resource usage and time in dense matrices.

The optimizations subpackage includes techniques that enhance performance at the level of memory access and parallel processing. CacheBlockedMultiplication implements cache blocking, which divides matrices into small blocks to optimize memory access and reduce cache misses, particularly useful in large matrices where sequential access is critical. LoopUnrollingMultiplication, on the other hand, applies loop unrolling to minimize iteration overhead by processing multiple steps in a single iteration, improving cache access and reducing control cycles.

Another key optimization is ParallelMultiplication, which distributes operations across multiple threads, leveraging available processing cores and significantly improving execution time for larger matrices. This technique is especially effective in multi-core architectures. Finally, VectorizedMultiplication uses SIMD (Single Instruction, Multiple Data) instructions to perform parallel calculations at the data level, enabling the simultaneous processing of multiple elements, thereby maximizing performance in dense matrices where data load is uniform.

These implementations of algorithms and optimizations allow for comparison and evaluation of how each technique impacts the performance of dense matrix multiplication, providing valuable insights into the efficiency and scalability of the methods under high computational demand conditions.

3.2.3 Sparse

The sparse package focuses on implementing specific optimization techniques and data structures for efficiently handling sparse matrices, optimizing both storage and computation time by processing only the non-zero elements of matrices. This package is divided into two main subpackages: optimization and structures, each containing essential components for managing sparse matrices in various contexts.

In optimization, the SparseMatrix file defines the base data structure for sparse matrices, allowing only the relevant (non-zero) elements of a matrix to be stored. This significantly reduces memory usage and speeds up calculations by avoiding the processing of zero elements, which is crucial for applications that handle large volumes of sparse data. SparseMultiplication, in turn, implements the multiplication algorithm for sparse matrices, operating exclusively on non-zero elements. This technique is fundamental in applications where matrices contain a high proportion of zeros, as it enables efficient data manipulation by minimizing both memory consumption and execution time.

The structures subpackage includes several classes designed for the efficient storage and management of sparse matrices. MatrixMultiplicationCSRCSC provides an implementation of sparse matrix multiplication using the CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) formats. These formats are ideal for the compact and efficient storage of sparse matrices, enhancing multiplication performance and reducing memory usage. The SparseMatrixCSR class specializes in the CSR format, which is particularly effective for row-based access operations, while SparseMatrixCSC uses the CSC format, optimized for column-based access. Both structures facilitate efficient handling of sparse matrices in different processing contexts and allow selecting the most suitable format based on the characteristics of the operation.

The Main file within the structures subpackage serves as an entry point for executing sparse matrix multiplication using the CSR and CSC structures. This file is responsible for loading matrices in MTX format from an input file and converting them into the

appropriate representations (CSR and CSC) for manipulation. It first loads the matrix in CSR format and then converts a copy of it to CSC format for multiplication. During execution, Main measures processing time and memory usage, allowing for an evaluation of the performance and efficiency of sparse multiplication in various configurations. This process not only provides a practical implementation of the structures and optimizations but also enables a quantitative assessment of the impact of these techniques in terms of computational resources.

3.2.4 Utils

The utils package plays a fundamental role in the organization and functionality of the project, providing essential support tools for configuration, matrix generation and loading, as well as for the operational management of tests. This package includes several classes designed to facilitate the development and execution of the matrix multiplication optimization testing system.

ConfigLoader enables flexible and efficient project configuration management by loading essential parameters from the config.properties file located in the resources folder. This configuration file contains critical information such as the matrix sizes to be tested, the desired sparsity levels, and the number of threads available for parallelization. With ConfigLoader, the system's behavior can be adjusted without modifying the source code, making it easy to customize tests to suit various configurations and experimental conditions.

The MatrixGenerator class creates dense and sparse test matrices to evaluate multiplication techniques in the project. For dense matrices, it generates two-dimensional arrays filled with random values through generateDenseMatrix. For sparse matrices, it provides two methods: generateSparseMatrix, which creates instances of SparseMatrix optimized to store only non-zero values, and generateSparseMatrixAsDense, which represents sparse matrices as dense by adjusting the sparsity level. This approach simulates sparsity in dense matrices, allowing observation of its impact on performance and memory usage, thus facilitating a comprehensive evaluation of each optimization technique's efficiency.

MatrixLoader complements these functionalities by enabling matrix loading from .mtx files, commonly used in scientific and data analysis applications. This feature is particularly useful for tests that require large matrices or real data, allowing for more realistic and representative evaluations. MatrixLoader supports standardized formats such as Matrix Market, enabling the system to work with pre-existing data and evaluate technique performance under conditions that mimic practical usage scenarios.

3.2.5 Resources

The resources folder in the project plays a fundamental role by providing essential configuration files and sample matrices, allowing for customization and testing of matrix multiplication optimizations. This structure facilitates the adaptation of test parameters without the need to modify the source code, offering a flexible and adaptable system.

Within resources, the matrix_samples subfolder contains large-scale matrix files in .mtx format, such as mc2depi.mtx and rajat23.mtx. These files represent large sparse matrices and are specifically used to measure execution time and memory usage by applying the sparse matrix multiplication algorithm in CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) formats. The implementation of these formats allows processing of only the non-zero elements of the matrices, significantly optimizing resources. Tests conducted with mc2depi.mtx and rajat23.mtx provide a realistic and detailed evaluation of algorithm performance under high data demand conditions, which is essential for measuring efficiency and scalability in sparse matrix applications.

The config.properties file, also located in the resources folder, contains a series of critical configurations for the system. This file defines the matrix sizes to be used in the tests, with values ranging from 64 to 2048, and the sparsity levels (sparsity.levels) for sparse matrices, in a range from 0.1 to 0.9, allowing for evaluation of how sparsity affects the efficiency of each technique. Additionally, config.properties specifies the number of threads (num.threads) to be used in parallelization tests, a key aspect for leveraging parallel processing and adapting tests to different hardware architectures.

Together, the resources folder and its content provide a comprehensive and adaptable testing environment, allowing developers to quickly adjust system parameters and use representative matrices to validate the efficiency of optimization techniques in both dense and sparse matrix multiplication.

4 Experiments

This section presents and analyzes the graphs generated from the results obtained after implementing various matrix multiplication techniques. Execution time was measured using benchmarking techniques with the JMH (Java Microbenchmark Harness) library, which enabled high-precision and consistent measurements across various conditions. Subsequently, the memory usage of each technique was calculated by capturing the values before and after execution to determine the total consumption in megabytes.

Each graph displays the results in terms of memory usage and execution time, considering different matrix sizes and sparsity levels, providing a detailed view of the performance of each technique under different scenarios.

4.1 Dense Matrices

4.1.1 Execution Time vs. Matrix Size

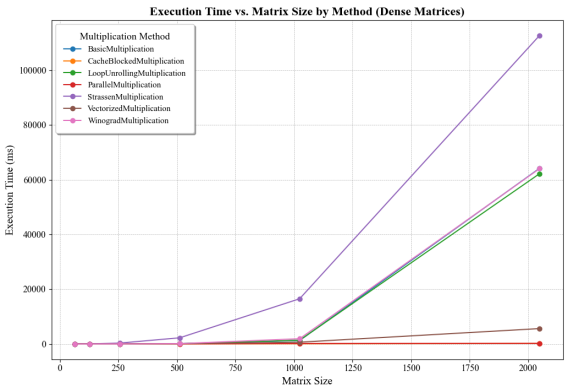

Figure 1: Execution Time vs. Matrix Size

The figure 1 shows the execution time, in milliseconds, for different dense matrix multiplication methods as a function of matrix size. The methods included in this evaluation are Basic Matrix Multiplication (blue), Cache Blocked Matrix Multiplication (orange), Loop Unrolling Matrix Multiplication (green), Parallel Matrix Multiplication (red), Strassen Matrix Multiplication (purple), Vectorized Matrix Multiplication (brown), and Winograd Matrix Multiplication (pink).

As matrix size increases, the execution time for all multiplication methods rises. This is expected due to the greater number of operations required to multiply larger matrices. However, significant differences are observed in the slope of increase among the different methods, which reflects the relative efficiency of each technique concerning matrix size.

In the graph, the Strassen method (represented in purple) shows the worst performance, with an exponential increase in execution time as matrix size grows, especially for matrices of size 1024x1024 and larger. This is due to the additional complexity of the Strassen algorithm, which, although it reduces the number of multiplications, introduces additional operations and high memory consumption, limiting its efficiency for large dense matrices.

Next, we observe the Loop Unrolling Multiplication method (green), which, although it optimizes loop structures to reduce processing time, still shows a considerable increase in execution time as matrix size increases. This method is efficient for medium-sized matrices but loses its advantage for larger sizes due to the accumulated computational load.

Cache Blocked Multiplication (orange) improves performance by dividing the matrix into blocks that optimize cache usage, reducing cache misses and increasing efficiency for large matrices. However, even though it is more efficient than previous methods, it still shows a noticeable increase in execution time for larger matrices.

Winograd Multiplication (pink) exhibits moderate behavior, with controlled execution time for intermediate matrix sizes, though it loses efficiency with larger sizes. This method is useful when reducing the number of multiplicative operations is needed, but its advantage diminishes as matrix size grows considerably.

The Parallel (red) and Vectorized (brown) methods stand out for their efficiency compared to the others. Parallel Multiplication distributes operations across multiple processing cores, taking advantage of multicore hardware and showing a significant reduction in execution time for large matrices. This method is especially beneficial for systems with multiple cores, maintaining low execution time even for matrix sizes of 2048x2048.

Finally, Basic Multiplication (blue) shows a less steep increase in execution time than Strassen or Loop Unrolling, which is notable for an unoptimized algorithm. While not the fastest, it serves as a reference for standard behavior compared to optimized methods.

In summary, the graph demonstrates that optimized methods, such as Parallel and Vectorized Multiplication, excel in terms of efficiency for large matrices. Techniques that optimize cache usage and hardware resources, like cache blocking and parallelization, offer significant advantages, while algorithms like Strassen, although theoretically efficient, show practical limitations for large dense matrices due to their computational complexity and memory consumption.

4.1.2 Memory Usage vs. Matrix Size

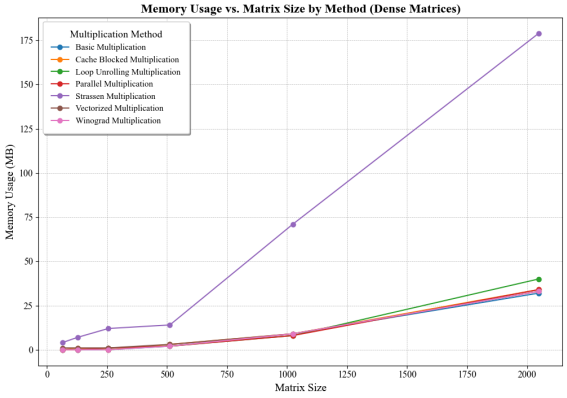

Figure 2: Memory Usage vs. Matrix Size

The Figure 2 illustrates memory usage, in megabytes, for different dense matrix multiplication methods according to matrix size.

In general terms, similar to execution time, memory usage increases as matrix size grows. However, differences in memory usage between methods become more pronounced with larger matrices. This is due to the different memory management and optimization strategies employed by each method.

In the graph, the Strassen method (represented in purple) shows the highest memory consumption, increasing exponentially as the matrix size grows. This is due to the structure of the Strassen algorithm, which decomposes the matrix into submatrices, resulting in intensive memory usage, especially in large matrices. Although this algorithm is known for reducing the number of multiplications, its memory cost becomes considerably high, limiting its applicability in large dense matrices. Comparing with the first execution time graph, Strassen also shows poor performance in terms of time for large matrix sizes, making it an unfavorable option in resource-limited environments.

Next, the Loop Unrolling (green) and Cache Blocking (orange) methods show moderate memory consumption compared to Strassen. These methods are designed to optimize memory access and reduce cycle overhead. In particular, cache blocking is effective in improving cache usage by dividing the matrix into blocks, while loop unrolling optimizes the structure of the loops. Although both methods demonstrate reasonable execution time performance (as seen in the first graph), their memory efficiency is moderate, increasing notably in large matrices.

Parallel Multiplication (red) and Vectorized Multiplication (brown) maintain controlled memory consumption that remains relatively low, especially for large matrices. Parallel Multiplication distributes the computation across multiple cores without significantly increasing memory usage, while Vectorized Multiplication leverages SIMD instructions to optimize time without substantially impacting memory consumption. In terms of execution time, both methods offer a notable improvement compared to non-optimized methods, making them attractive options for large matrices.

Finally, Basic Multiplication (blue) and Winograd Multiplication (pink) stand out for their efficiency in terms of memory usage. Basic Multiplication shows the lowest memory consumption profile, remaining almost constant across all matrix sizes. Although it is the reference method and does not include complex optimizations, its memory efficiency makes it the most stable option in resource-limited environments. Comparing with the first graph, Basic Multiplication also has moderate execution time, making it suitable when a balance between time and memory is required. Winograd, while also showing low memory consumption, is particularly efficient for intermediate-sized matrices, but its time performance is surpassed by other optimized methods in larger matrices.

In summary, the graph shows that Basic Multiplication is the most memory-efficient method, followed by Winograd and the parallel and vectorized methods. While Strassen performs poorly in both time and memory for large matrices, cache blocking and parallelization-optimized methods provide good time efficiency with moderate memory consumption. In comparison, the most time-efficient methods are not always the most memory-efficient, so the optimal method choice depends on resource priorities and matrix sizes within the system.

4.2 Sparse Matrices

4.2.1 Execution Time vs. Matrix Size

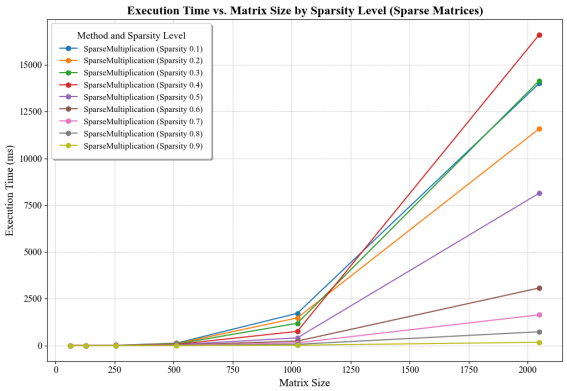

Figure 3: Execution Time vs. Matrix Size

The Figure 3 expands on the previous analysis, now focusing on sparse matrices with varying levels of sparsity from 0.1 to 0.9. In this graph, execution time is shown as a function of matrix size and sparsity level, allowing us to observe how the structure of sparse matrices affects multiplication performance compared to the dense matrices previously evaluated.

In general, we observe that execution time tends to decrease as the sparsity level increases. This is because a higher level of sparsity implies a lower number of non-zero elements in the matrix, reducing the number of operations required in the multiplication. This behavior is especially notable in larger matrices, where differences between sparsity levels become more pronounced.

The case with a sparsity level of 0.4, shown in red, presents the longest execu-

tion time, particularly for large matrices like 2048x2048. This is because, with a low sparsity level, the matrix contains a greater number of non-zero elements, making its computational load more similar to that of a dense matrix. As the sparsity level increases, as seen in the purple, brown, and pink curves (representing 0.5, 0.6, and 0.7, respectively), the execution time progressively decreases, reflecting the efficiency gained from having fewer elements to process.

For the highest sparsity levels (0.8 and 0.9, represented in gray and yellow, respectively), the execution time is significantly lower across all matrix sizes. This behavior illustrates the advantage of using sparse matrices in situations where most elements are zero, allowing for considerable savings in computation time.

This analysis complements the results observed in Figures 1 and 2, where execution time efficiency is fundamentally different when working with sparse matrices compared to dense matrices. For sparse matrices, the reduction in the number of arithmetic operations due to higher sparsity leads to superior performance, especially in large matrices. In contrast, with dense matrices, it was necessary to use optimized methods, such as vectorization and parallelization, to efficiently manage memory usage and improve execution time, which is less critical in sparse matrices due to the natural reduction in the number of calculations.

In conclusion, the choice of a multiplication method and its configuration should consider both the density and size of the matrix, as optimized methods for dense matrices may not be necessary or efficient when working with sparse matrices. This graph demonstrates that, with high levels of sparsity, significant benefits in execution time are achieved, making the use of sparse matrices highly advisable in contexts where most elements are zero.

4.2.2 Memory Usage vs. Matrix Size

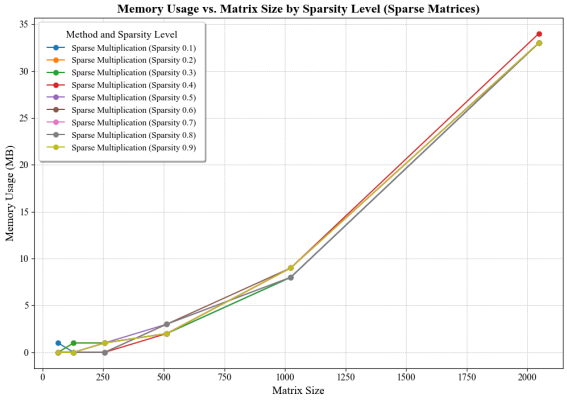

Figure 4: Memory Usage vs. Matrix Size

The Figure 4 illustrates memory usage in megabytes for sparse matrices at different sparsity levels as a function of matrix size. This chart complements the analysis in Figure 3 (execution time for sparse matrices) by providing insight into how the sparse structure of the matrix and its sparsity levels affect memory consumption during multiplication.

Unlike dense matrices, where memory usage quickly increases with matrix size, here we observe that in sparse matrices, memory consumption is significantly lower and remains fairly controlled, even in large matrices. This is due to the efficient representation of sparse matrices, which stores only non-zero elements, thereby reducing the space required for multiplication operations.

For higher sparsity levels, such as 0.9 (yellow line), memory usage is greater compared to lower sparsity levels like 0.5 (purple line). This behavior is consistent with what is observed in the execution time in Figure 3; with higher sparsity, the matrix contains fewer non-zero elements, which reduces execution time but increases memory consumption due to how the sparse elements are stored and managed. As the sparsity level increases (from 0.1 to 0.9), memory usage rises notably since more data is needed to represent the sparse structure and perform the necessary operations.

A comparative analysis with dense matrices (Figure 2) reveals that, while dense

matrices require significantly greater and increasing memory usage as matrix size grows, sparse matrices allow for more efficient memory management. This is particularly beneficial in applications where most of the matrix elements are zero, as it avoids unnecessary storage and processing of those elements.

In summary, sparse representation results in considerable optimization in terms of memory usage. This memory saving, combined with the reduction in execution time previously observed, underscores the advantage of sparse matrices in contexts where the matrix contains a high percentage of zeros. The relationship between these graphs confirms that efficient handling of sparse matrices positively impacts not only execution time but also memory consumption, making this technique an optimal choice for certain types of applications.

4.3 Dense Matrices vs Sparse Matrices

4.3.1 Execution Time vs. Matrix Size by Sparsity Level

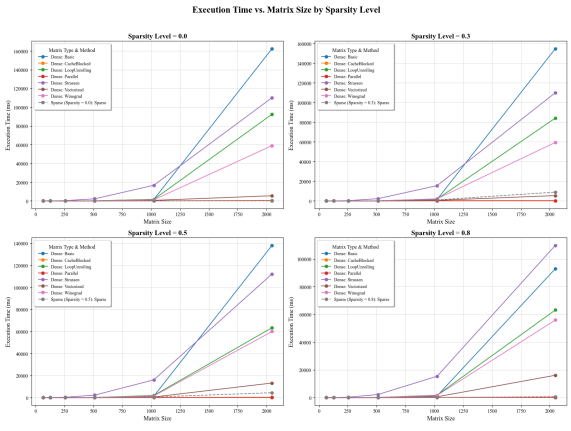

Figure 5: Execution Time vs. Matrix Size by Sparsity Level

In Figure 5, the execution time is illustrated as a function of matrix size, considering different sparsity levels (0.0, 0.3, 0.5, and 0.8) for both dense and sparse matrices. In this study, sparsity was applied to dense matrices to analyze how different levels affect the performance of multiplication methods. This breakdown provides a detailed view of the influence of sparsity on the efficiency of each technique, in relation to the density and size of the matrix.

At low sparsity levels, such as 0.0 (upper left corner), it is observed that the execution time for dense methods is significantly higher than for sparse methods. This is because, with few sparse elements, the matrix behaves almost like a dense matrix, limiting the advantages of methods specifically for sparse matrices. As sparsity increases, as seen at the 0.8 level (lower right corner), sparse multiplication methods show a notable improvement in performance compared to dense methods, especially in execution times. At this level, sparse methods leverage the high number of zeros, processing matrices more quickly and efficiently.

Additionally, it is observed that some dense methods, such as the Strassen method, perform worse compared to optimization methods like parallelized and vectorized approaches, especially in large matrices. This highlights the importance of optimization techniques that leverage hardware architecture and matrix sparsity, providing a substantial improvement in performance and significantly reducing execution times under high sparsity conditions.

Comparing with previous analyses where only dense matrix methods were evaluated without applying sparsity, it is evident here how sparsity considerably reduces execution time in sparse methods as sparsity increases. For dense methods, the increase in sparsity only provides slight improvement in some cases, remaining inferior to the performance of methods optimized for sparse matrices.

In conclusion, the analysis of this figure highlights the superiority of sparse methods as sparsity levels increase, while dense methods do not show significant improvements under these conditions. This underscores the importance of choosing the appropriate method based on the matrix structure and the optimization of available computational

resources.

4.3.2 Memory Usage vs. Matrix Size by Sparsity Level

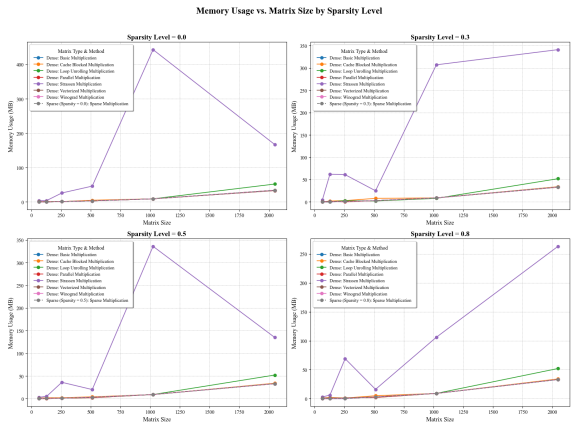

Figure 6: Memory Usage vs. Matrix Size by Sparsity Level

Figure 6 presents memory usage, measured in megabytes (MB), is shown as a function of matrix size, considering different sparsity levels: 0.0, 0.3, 0.5, and 0.8. Similar to the execution time analysis, different sparsity levels are applied to dense matrices to observe how memory usage varies with data sparsity in each multiplication method.

At the 0.0 sparsity level (top left corner), memory usage is moderate and follows an upward trend as matrix size increases. The Strassen method (purple line) stands out for its considerably higher memory consumption compared to other methods, due to its complexity in terms of temporary storage for submatrices. This behavior is consistent with the previous execution time analysis, where it also showed less efficient resource performance.

As sparsity increases to 0.3 (top right corner), the impact on memory usage becomes more pronounced for the Strassen method, reaching up to 350 MB for larger matrices, while other methods maintain relatively low and constant consumption. This suggests that the memory efficiency of basic and optimized methods (such as cache blocking and vectorization) significantly benefits from increased sparsity, minimizing resource usage by reducing the amount of non-zero data.

At higher sparsity levels, such as 0.5 and 0.8 (bottom graphs), memory usage in optimized methods remains low, particularly at the 0.8 sparsity level, where even the Strassen method shows a notable reduction in memory consumption. This is due to the large number of zero elements in the matrices, allowing optimized methods to avoid unnecessary data storage, leveraging the sparse structure more efficiently. Sparse multiplication methods, at these levels, show considerably reduced and more uniform memory usage, aligning with their execution time results and highlighting their efficiency in highly sparse matrices.

Compared with Figure 5, this memory usage analysis complements the understanding of how different methods perform in terms of both time and memory under various sparsity levels. In general, optimization methods in cache access and sparse matrix processing prove to be the most efficient in both aspects, especially in large matrices with high sparsity levels.

In conclusion, this figure reinforces the importance of selecting the appropriate method based on sparsity level when considering memory usage. Methods such as cache blocking and parallelization show significant advantages, while the Strassen method, despite its theoretical efficiency in operations, is unfavorable in large, dense matrices due to its high memory consumption.

4.4 Results of the mc2depi and rajat23 Matrices

To evaluate the efficiency of the sparse matrix multiplication algorithm in Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats, tests were conducted with two large matrices: mc2depi and rajat23. These matrices were selected due to their high number of rows and columns, which presents a significant challenge in terms of performance and memory usage for sparse matrix multiplication algorithms. The

results obtained, including execution time and memory usage, are detailed in Table 1. These results allow us to analyze the performance of the CSR and CSC formats in handling large-scale sparse data, providing valuable information on the algorithm's efficiency and feasibility in environments requiring efficient processing of large matrices.

	mc2depi	**rajat23**
Size (rows x columns)	525825 x 525825	110355 x 110355
Execution Time (ms)	467	695
Memory Usage (MB)	140	107

Table 1: Results of the multiplication for large sparse matrices

The execution of these matrices demonstrates the ability of the sparse matrix multiplication algorithm to efficiently handle large-scale data. The mc2depi matrix, with dimensions 525825 x 525825, required an execution time of 467 ms and a memory usage of 140 MB, while the rajat23 matrix, sized 110355 x 110355, took 695 ms and consumed 107 MB of memory.

The Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats are particularly suitable for this type of matrices, as they allow storing and processing only the non-zero elements, thus optimizing memory usage and reducing computational complexity. Instead of multiplying every element of the matrix, the algorithm processes only the non-zero values and their corresponding row and column indices, which is essential for efficient operations on such large matrices.

Additionally, this implementation enables matrix multiplication operations on a large scale, which would not be practical using traditional methods due to high memory consumption and computation time. In this context, the execution times of 467 ms and 695 ms indicate the algorithm's efficiency, allowing the handling of substantial matrix sizes in a reasonable time frame with optimized memory usage.

These results illustrate how the use of CSR and CSC formats, along with efficient data organization via the sparse matrix multiplication algorithm, make it feasible to work with extremely large matrices in Big Data and high-performance computing applications. This is particularly relevant in environments where memory is limited, and performance must be maximized.

5 Conclusion

In conclusion, the detailed analysis of the results obtained in these graphs provides a deeper understanding of the performance of various dense and sparse matrix multiplication techniques, taking into account matrix size and sparsity levels.

For dense matrices, it is observed that the Basic Multiplication and Winograd methods are efficient in terms of memory usage, maintaining stable consumption as matrix size increases. However, in terms of execution time, Basic Multiplication shows a gradual increase, making it less efficient compared to optimized methods like Vectorization and Parallelization. These two techniques stand out for their ability to significantly reduce execution time while maintaining relatively low memory usage.

However, a significant bottleneck is identified in the Strassen method, which, although it reduces the theoretical number of multiplications, shows exponential growth in memory usage and execution time as matrix size increases. This is due to the need for temporary storage for additional submatrices, which limits its applicability for large matrices, especially in memory-constrained environments. For matrix sizes greater than 1024 x 1024, the Strassen method becomes inefficient, making it evident that other methods should be considered for large dense matrices.

In the case of sparse matrices, high sparsity levels (0.8 and 0.9) demonstrate a significant improvement in execution time and memory usage. This is because the high number of zero elements reduces the required operations, maximizing the efficiency of sparse

multiplication methods. In particular, Sparse Multiplication methods show a notable decrease in both execution time and memory consumption for large and highly sparse matrices, outperforming dense methods under these conditions.

Additionally, a bottleneck in sparse matrices is observed at low sparsity levels (such as 0.1 or 0.3), where the matrices still contain a significant number of non-zero elements. This increases the computational load and limits the advantages of sparse methods, making them approach the behavior of dense methods. However, as sparsity increases, these methods achieve optimal utilization of the sparse structure, considerably reducing execution time and memory usage.

During the tests, the maximum matrix size efficiently handled by the methods was 2048 x 2048. Beyond this size, both execution time and memory consumption increase substantially, especially for methods like Strassen and Basic Multiplication in dense matrices. This limitation suggests that, in Big Data applications or systems with limited resources, it is necessary to carefully consider matrix size and the methods used to avoid overloading the system.

Cache Blocking and Parallelization methods proved effective for dense matrices by optimizing cache memory access and distributing operations across multiple cores, respectively. Vectorization also offers solid performance, especially in terms of execution time, leveraging SIMD instructions to process data in parallel. These methods not only optimize execution time but also allow for more efficient use of hardware resources, such as processing cores and cache memory.

This study demonstrates that the optimal choice of multiplication method largely depends on matrix size, sparsity level, and system characteristics. For large dense matrices, parallelized and vectorized methods are the most suitable, maximizing efficiency in execution time while moderating memory consumption. In contrast, for sparse matrices with high levels of zero elements, sparse multiplication methods are clearly superior in both execution time and memory usage.

Additionally, it is important to note that certain methods, such as Strassen, although theoretically efficient, present practical limitations due to memory usage bottlenecks, especially for large matrices. This limitation highlights the importance of selecting the appropriate method not only based on theoretical efficiency but also by considering memory constraints and the resources available in the system.

In conclusion, the results obtained emphasize that there is no universally optimal multiplication method. The selection of the method should be adapted to the specific characteristics of the matrix and system resources, balancing both execution time and memory usage based on application requirements and the available hardware architecture.

6 Future Work

Future work can expand this research by exploring additional optimization strategies and advanced computational paradigms to further improve matrix multiplication performance. A promising area is the integration of machine learning models to predict the optimal multiplication method based on matrix characteristics and system constraints. This would enable the development of adaptive algorithms that dynamically adjust their approach according to data structure and computational environment, thereby enhancing overall efficiency.

Additionally, matrix multiplication on distributed systems, briefly discussed in the context of large-scale matrices, warrants further investigation. Implementing matrix multiplication in distributed systems, such as clusters or cloud infrastructures, would allow processing extremely large matrices that exceed the memory capacity of a single machine. This approach would involve optimizing network communication and data distribution among

nodes to minimize latency and maximize performance.

Exploring GPU-based multiplication and hardware accelerators such as Tensor Processing Units (TPUs) is another avenue for future research. These hardware solutions are designed to handle large-scale parallel operations, and their application in matrix multiplication could provide significant performance improvements, particularly for large, dense matrices in real-time applications.

Finally, investigating hybrid approaches that combine multiple optimization techniques, such as parallelization with cache optimization or vectorization with sparse matrices, could offer new perspectives for achieving even greater efficiency. The incorporation of hybrid models would contribute to a more nuanced understanding of how different optimization techniques complement each other and impact performance in various computational scenarios. This ongoing exploration would enable the development of highly efficient and scalable matrix multiplication solutions tailored to the demands of Big Data and high-performance computing applications.

GitHub Link

All the code implemented for this project can be found on my GitHub repository at the following link: GitHub Repository.