A Comparative Performance Analysis of different Matrix Multiplication algorithms in Java

Daniel Talavera Hernández

November 2024

Source code in this GitHub repository.

1 Abstract

Matrix multiplication is a core operation in numerous scientific and engineering applications, making its optimization highly relevant. This paper presents a comparative study of five matrix multiplication algorithms: a standard three-loop implementation, a block-based algorithm, a loop-unrolling optimized version, and two sparse matrix algorithms using Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) formats. The goal is to benchmark these algorithms in terms of execution time and memory efficiency to determine their suitability across different use cases. Experiments were conducted on randomly generated matrices, with varied dimensions and sparsity levels, to evaluate performance under different conditions. Results reveal that the sparse algorithms (CSC and CSR) are advantageous for high-sparsity matrices, significantly reducing computational costs. Meanwhile, the loop-unrolling approach outperforms others on larger matrices with dense structures. These insights underline the importance of algorithm selection based on matrix characteristics and offer guidance for practitioners on optimizing matrix multiplication for specific needs.

2 Introduction

Matrix multiplication serves as a fundamental computational operation across diverse fields, including machine learning, computer graphics, and scientific simulations. As datasets grow and computational needs increase, optimizing matrix multiplication performance has become a critical area of study, with implications for computational efficiency and resource management. In the context of algorithm benchmarking, various approaches to matrix multiplication have been explored to maximize efficiency, from basic iterative methods to advanced optimizations such as block multiplication and sparse matrix techniques.

Benchmarking studies, including those by Smith et al. (2022) and Li & Chang (2023), highlight the impact of algorithmic choices on computation times and memory usage. These studies often implement techniques like loop unrolling and caching to exploit hardware capabilities, while others focus on sparse matrix algorithms to handle data with numerous zero values. Such techniques allow for reduced processing time and better memory utilization, providing significant advantages in high-dimensional and sparse-data applications.

This paper contributes to this field by evaluating five matrix multiplication algorithms—basic three-loop, block-based, loop-unrolled, and sparse algorithms in both CSC and CSR formats—under diverse data scenarios. The analysis highlights the trade-offs in execution time and memory use across different matrix characteristics, providing a comprehensive benchmark for practitioners and researchers aiming to select the optimal algorithm for their specific computational requirements.

3 Problem Statement

Matrix multiplication, despite being a basic linear algebra operation, poses significant computational challenges, particularly when dealing with large or sparse matrices. Selecting an appropriate algorithm is crucial, as the efficiency of matrix multiplication directly influences the performance of a wide range of applications, from scientific simulations to real-time machine learning systems. The problem lies not only in achieving faster computation but also in optimizing memory usage, which becomes a critical factor with large-scale data. Traditional three-loop algorithms can be prohibitively slow and memory-intensive, while more advanced methods, such as block-based or sparse matrix algorithms (CSC and CSR), may offer improvements but require specific conditions to perform optimally.

The challenge addressed in this paper is to identify under which conditions each algorithm—basic, block-based, loop-unrolled, and sparse CSC and CSR—yields the best trade-off between computational speed and memory efficiency. Current literature provides insights into these algorithms individually, but a consolidated benchmark across varied matrix characteristics (such as size and sparsity) is lacking. This study fills this gap by systematically evaluating each algorithm's performance under different conditions, thus guiding practitioners in selecting the most suitable approach for their specific needs. The goal is to provide a framework that simplifies the decision-making process for matrix multiplication in computationally intensive applications.

4 Solution

4.1 Methodology

To address the problem of comparing the performance of matrix multiplication algorithms, we designed a set of experiments to evaluate the efficiency of five algorithms: the basic three-loop algorithm, block-based multiplication, loop-unrolled (optimized) version, sparse matrix multiplication (using Compressed Sparse Column format and Compressed Sparse Row format). The focus was on execution time and memory usage across matrices ranging from 10x10 to 2000x2000.

4.1.1 Basic Matrix Multiplication

The basic matrix multiplication algorithm multiplies two square matrices, A and B, of size n×n. This approach follows the standard mathematical definition of matrix multiplication, iterating through each cell in the resulting matrix C. For each element C[i][j], the algorithm computes the sum of products from the respective row in A and column in B. Specifically, it iterates through each row i of A and each column j of B, accumulating the sum of products A[i][k] * B[k][j] for each element k in the row and column. The algorithm's time complexity is O(n3), as it uses three nested loops to cover all elements in the matrices, making it relatively slow for large matrices but straightforward and easy to implement.

4.1.2 Block Matrix Multiplication

The block matrix multiplication algorithm optimizes matrix multiplication by dividing each matrix into smaller submatrices or blocks. Instead of treating each matrix as a collection of individual elements, this approach treats them as blocks of elements, reducing cache misses and improving memory locality. The matrices A and B are divided into smaller, contiguous blocks, which are then multiplied together and added to form the final matrix C. Inside the function, the main matrix is broken down into blockSize-sized submatrices, and multiplication is performed block-by-block. This strategy often improves performance on large matrices by taking advantage of modern CPU cache hierarchies, as it accesses data in a way that is more cache-efficient. The time complexity remains O(n3), but with improved real-world performance due to better memory management.

4.1.3 Loop Unrolling Matrix Multiplication

The optimized loop matrix multiplication algorithm improves performance by using loop unrolling, a technique that reduces loop overhead and allows for multiple operations within a single loop iteration. In this implementation, the inner loop that updates the values in C is unrolled by a factor of 4, processing four elements of the row at a time. This minimizes the number of loop instructions the CPU needs to handle, thus increasing throughput by reducing branching and loop counter updates. Although loop unrolling doesn't change the O(n3) complexity, it significantly improves efficiency on compatible hardware, especially in cases where the matrices fit into the CPU cache.

4.1.4 Sparse Matrix Multiplication Using Compressed Sparse Column (CSC) Format

In the sparse matrix multiplication algorithm, matrices are stored in a compressed sparse column (CSC) format, which is highly efficient for matrices with a large number of zeros. In CSC format, only non-zero values are stored along with their row indices and column start positions, saving memory and speeding up calculations by avoiding unnecessary zero multiplications. The multiply method of the CSCMatrix class first initializes arrays to store non-zero values, their corresponding row indices, and pointers to the start of each column. Then, to multiply two matrices A and B, it iterates over the non-zero entries in each column of B, using the row indices and values to perform multiplications only where both matrices have non-zero values. This method is much faster than standard matrix multiplication for sparse matrices, as it skips zero entries entirely. The space complexity is reduced as only non-zero values are stored, making it highly memory-efficient for sparse matrices, and the multiplication process only involves significant elements, saving processing time.

4.1.5 Sparse Matrix Multiplication Using Compressed Sparse Row (CSR) Format

The sparse matrix multiplication algorithm using the CSR (Compressed Sparse Row) format is highly efficient for matrices with a high sparsity level. In CSR format, only non-zero values are stored, along with their column indices and row pointers. This representation significantly reduces memory usage and speeds up calculations by ignoring zero entries entirely. The CSRMatrix class represents matrices in CSR format, storing non-zero values in a values array, their column indices in columnIndices, and the starting position of each row in rowPointers. To multiply two sparse matrices A and B, where A is in CSR format, the algorithm iterates through the non-zero elements of each row in A. For each non-zero element in a row of A, it accesses the corresponding non-zero values in the matching row of B, and performs multiplications only for these non-zero elements. This selective multiplication process minimizes unnecessary computations and efficiently utilizes memory, as only the significant elements are involved in the calculations. The CSR format also simplifies row-based matrix operations and allows the algorithm to skip over entire rows when they contain only zero elements. This approach leads to a substantial reduction in time complexity for large sparse matrices, as well as a highly efficient use of memory.

4.2 Experimental Setup

4.2.1 Programming Language

Java was chosen for the implementation of all algorithms due to its balance between performance and abstraction. The experiments focus on benchmarking the matrix multiplication algorithms' execution time and memory usage. For the sparse matrix multiplication, a Compressed Sparse Column (CSC) and a Compressed Sparse Row (CSR) representation were used to ensure memory efficiency with sparse data.

4.2.2 Machine Specifications

Processor: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz.
RAM: 16 GB.
Operating System: Windows 10 Enterprise (Version: 22H2).
Cores: 6 cores.

4.3 Experimental Design

The experiments will involve varying matrix sizes, from 10x10 to 2000x2000, with the following setup for each matrix size:
Warm-up Phase: Each algorithm will be run several times before taking measurements to allow Java's Just-In-Time (JIT) compiler to optimize the code and ensure the system stabilizes.
Measurement Phase: After warm-up, each algorithm will be executed 10 times for each matrix size. The performance metrics of execution time and memory usage will be captured during each run.
In addition, we will test how each algorithm scales with increasing matrix size and analyze their behavior in terms of both computational time and memory consumption.

4.4 Performance Measurement

We used the following metrics to measure performance:
Execution Time: Measured in milliseconds using System.nanoTime() in Java.
Memory Usage: Measured in megabytes using Java's MemoryMXBean API, which tracks heap memory usage.
CPU Usage: CPU load was measured before and after execution using the OperatingSystemMXBean API.

4.5 Benchmark Configuration

Warm-up Rounds: For each algorithm, we performed 5 warm-up runs to mitigate any JIT compilation delays.
Iterations: Each experiment was repeated 10 times for each matrix size to ensure statistical reliability and reduce the effect of external system processes.

4.5.1 General Benchmark Setup

Benchmark Parameters: @Warmup: Executes the benchmark five times to warm up the JVM, allowing it to optimize code paths before actual measurements are taken.
@Measurement: Runs ten measurement iterations, each lasting one second, to gather data.
@Fork: Uses a single fork, meaning the benchmark is run in an isolated JVM instance to avoid potential interferences from other processes.
@BenchmarkMode and @OutputTimeUnit: These are set to average time mode with output in milliseconds, providing a consistent measure of time per operation.
@State: The Scope.Benchmark state ensures that matrix instances are shared across all benchmark methods within the same benchmark run, maintaining consistency in the matrix data and allowing fair comparison across different algorithms.

4.5.2 Benchmark for Dense Matrix Multiplication (Class TestMatrixMultiplication)

This benchmark evaluates three algorithms: basic matrix multiplication, block matrix multiplication, and an optimized loop-based multiplication. Here's the setup and measurement process:
Matrix Size: The @Param annotation defines a variety of matrix sizes to simulate small to large datasets (10, 100, 250, 500, and 1000).
Setup Method: The setup() method initializes two dense matrices A and B with random values for the specified size. These matrices are populated in the @Setup(Level.Trial) phase to ensure they are freshly initialized for each trial.
Benchmark Methods: benchmarkBasicMatrixMult(), benchmarkBlockMatrixMult(), and benchmarkOptimizedLoopMatrixMult() call measure(), which runs each algorithm on the initialized matrices and logs their performance metrics.

4.5.3 Sparse Matrix Multiplication CSC Benchmark (Class TestSparseMatrixMultiplicationCSC)

This class benchmarks matrix multiplication using sparse matrices in CSC (Compressed Sparse Column) format to evaluate how each algorithm performs under different sparsity levels.
Matrix Size and Sparsity Level: @Param specifies matrix sizes ranging from 10 to 2000 and sparsity levels from 0 (dense) to 0.9 (very sparse), allowing for a thorough examination of algorithm efficiency under varying degrees of matrix sparsity.
Setup Method: Two dense matrices A and B are generated and then converted to CSC format using a conversion utility in the SparseMatrixCSCMul class.

Only a fraction of entries are populated based on the sparsityLevel parameter, which ensures that the benchmarked CSC multiplication operates on sparse datasets.
Benchmark Method: The benchmarkSparseMatrixMult() method uses measure() to evaluate performance in terms of execution time, memory usage, and CPU load for CSC matrix multiplication.
measure() Method: Gathering Performance Metrics. The measure() method is the core of both benchmark classes, capturing three key metrics: Memory Usage: Captures heap memory usage before and after execution using MemoryMXBean, calculating the total memory consumed by each algorithm.
CPU Load: Measures CPU load before and after execution using OperatingSystemMXBean. This helps determine how each algorithm impacts CPU usage during computation.
Execution Time: Measures the elapsed time from start to finish of each matrix multiplication method using System.nanoTime(), which provides high-resolution timing suitable for benchmarking.
Output and Results Collection In the tearDown() method (@TearDown(Level.Trial)), the benchmark prints results for each algorithm, matrix size, and sparsity level.
The output includes:
Execution Time in milliseconds,
Memory Usage in megabytes, and
CPU Load Before and After execution, giving a complete view of resource utilization for each algorithm under tested conditions.

4.5.4 Sparse Matrix Multiplication CSR Benchmark (Class TestSparseMatrixCSRMul)

This benchmark class tests the performance of matrix multiplication using the CSR (Compressed Sparse Row) format, evaluating how the algorithm responds under various sparsity levels. The aim is to assess execution time, memory usage, and CPU load for CSR-based matrix multiplication.
Matrix Size and Sparsity Level: Using the @Param annotation, matrix sizes are configured to range from 10 to 2000, with sparsity levels from 0 (fully dense) to 0.9 (very sparse). This setup allows a thorough performance analysis across a wide range of sparsity conditions.
Setup Method: During the @Setup(Level.Trial) phase, two dense matrices (A and B) are generated and populated based on the specified sparsityLevel parameter. These dense matrices are then converted to CSR format using the convertToCSR method in the SparseMatrixCSRMul class, ensuring that the benchmarked CSR multiplication operates on appropriately sparse datasets.
Benchmark Method: The benchmarkSparseMatrixMult() method uses the measure() function to evaluate performance metrics for CSR matrix multiplication. This method captures key indicators such as execution time, memory usage, and CPU load, providing a complete performance profile for the algorithm.
measure() Method: Gathering Performance Metrics The `measure()`

method collects essential performance data: Memory Usage: This captures memory usage before and after execution via MemoryMXBean, calculating the heap memory consumed by each benchmark.
CPU Load: Using OperatingSystemMXBean, it records the CPU load pre- and post-execution to evaluate how CSR matrix multiplication impacts CPU utilization.
Execution Time: This is calculated with System.nanoTime(), allowing precise measurement of the elapsed time for each multiplication, which is essential for high-resolution benchmarking.
Output and Results Collection In the tearDown() method (@TearDown(Level.Trial)), the benchmark outputs the results for each matrix size and sparsity level. The data includes:
Execution Time (in milliseconds),
Memory Usage (in megabytes), and
CPU Load Before and After execution.
This output provides a detailed view of resource utilization across tested conditions, enabling comprehensive comparisons between different matrix sparsity levels and sizes.

4.6 Tools

Java: Java Management APIs (MemoryMXBean for memory profiling, OperatingSystemMXBean for CPU profiling) were used for both memory and CPU measurements.
Execution Time: System.nanoTime() was used to measure execution time with high precision.
Memory Usage: The MemoryMXBean API was used to capture heap memory usage during execution.

4.7 Experimental Plan

For each algorithm and matrix size, we will:
Analyze execution time trends: How runtime scales with increasing matrix sizes.
Assess memory usage: How each algorithm handles larger data sets.
Compare the efficiency of the different algorithms, highlighting performance trade-offs between computational time and memory consumption.

4.8 Data Collection and Analysis

All results will be stored in a structured format for easy comparison. The analysis will focus on:

Execution speedups for each algorithm.
Memory efficiency, especially in the sparse matrix case.

Efficiency trade-offs between different matrix multiplication algorithms.

This methodology ensures that the experiments are rigorous and reproducible, providing a clear comparison of the performance of different matrix multiplication algorithms in Java, with a specific focus on memory usage and computational efficiency.

5 Experiments

This section describes the experiments conducted to benchmark the performance of five matrix multiplication algorithms implemented in Java: the basic three-loop algorithm, block-based multiplication, loop-unrolled (optimized) version, and sparse matrix multiplication using both Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) formats. The experiments evaluate performance in terms of execution time and memory usage, with matrix sizes ranging from 10x10 to 1000x1000. Sparse matrix multiplication was tested separately, as it also depends on the sparsity level in addition to matrix size. Each experiment was repeated multiple times to ensure statistical significance, and the results are summarized in tables for easy comparison.

5.1 Execution Time

Objective: To measure and compare the execution time of the matrix multiplication algorithms across different matrix sizes.
Method: Matrix sizes ranged from 10x10 to 1000x1000. Each matrix size was run through 10 iterations after a warm-up phase to allow for the JIT compiler optimizations. Execution time was measured in milliseconds (ms) using System.nanoTime() for Java.
Results: Table 1 shows the execution times for each matrix size and algorithm. For sparse matrix algorithms, execution times were recorded across varying sparsity levels (0%, 50%, 80%, and 90%). The CSR format consistently demonstrated the fastest execution times among all algorithms, especially for larger matrices, surpassing the CSC format due to its row-wise optimization. Tables 2 and 3 provide a detailed comparison, while Figures 1, 2 and 3 visualize the trends.

Matrix Size (n x n)	**Basic (ms)**	**Block (ms)**	**Loop Optimized (ms)**
10 x 10	0.03	0.01	0.03
100 x 100	1.25	1.56	0.30
250 x 250	32.75	27.87	15.37
500 x 500	227.23	192.91	38.51
1000 x 1000	9791.92	2595.99	966.23

Table 1: Execution time comparison for different matrix sizes and algorithms.

M. Size (n x n)	Spar. 0% (ms)	Spar. 50% (ms)	Spar. 80% (ms)	Spar. 90% (ms)
10 x 10	0.05	0.05	0.06	0.03
100 x 100	1.32	1.12	0.55	0.36
250 x 250	16.34	6.91	5.35	1.65
500 x 500	215.17	60.29	39.71	12.46
1000 x 1000	1415.77	528.40	111.27	59.43
2000 x 2000	8907.43	2596.40	712.54	653.13

Table 2: Sparse matrix execution time comparison for different sparsity levels (CSC).

M. Size (n x n)	Spar. 0% (ms)	Spar. 50% (ms)	Spar. 80% (ms)	Spar. 90% (ms)
10 x 10	0.05	0.07	0.04	0.06
100 x 100	1.38	0.76	0.57	0.27
250 x 250	15.82	6.35	2.26	1.83
500 x 500	135.79	43.98	36.56	16.46
1000 x 1000	1420.55	329.19	105.79	49.54
2000 x 2000	11868.41	3373.85	869.32	455.85

Table 3: Sparse matrix execution time comparison for different sparsity levels (CSR).

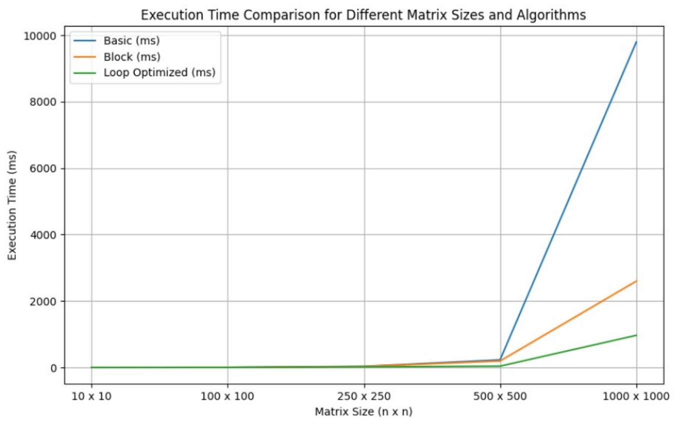

Figure 1: Execution time comparison for different matrix sizes and algorithms.

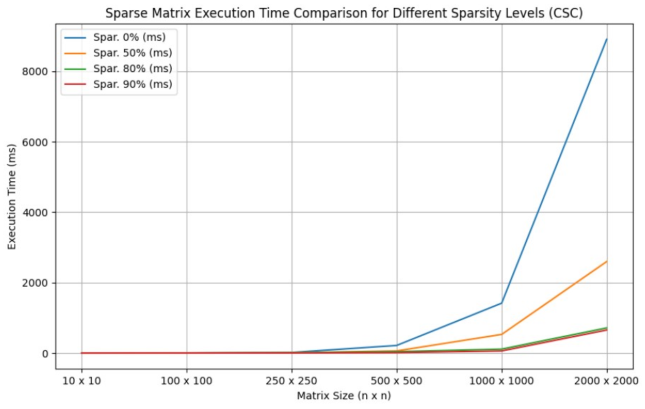

Figure 2: Execution time comparison for different sparsity levels for CSC algorithm.

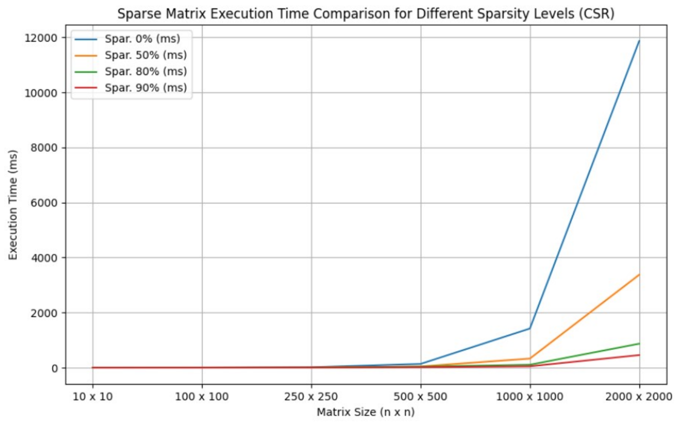

Figure 3: Execution time comparison for different sparsity levels for CSR algorithm.

Analysis: The basic algorithm showed the highest execution times due to its inefficient loop structure and memory access patterns. The block-based and loop-unrolled algorithms achieved notable improvements, especially with larger matrices. Among the sparse algorithms, CSR outperformed CSC in execution speed, attributed to its efficient handling of row-based matrix data. The sparse algorithms also showed increasingly better performance at higher sparsity levels, as fewer non-zero elements needed to be processed.

5.2 Memory Usage

Objective: To evaluate the memory usage for each matrix multiplication algorithm during execution.
Method: Memory usage was measured in megabytes (MB) using Java's MemoryMXBean API, which tracks heap memory usage. Measurements were taken after each matrix multiplication operation.
Results: The relative memory consumption for each matrix size and algorithm is shown in Table 4. Additionally, for the sparse matrix algorithm, memory usage was recorded for different sparsity levels (0%, 50%, 80%, and 90%). Both CSC and CSR formats showed the lowest memory usage due to their efficient storage of non-zero elements. The CSR format demonstrated a memory usage similar to CSC but achieved this with greater efficiency at higher sparsity levels, as shown in Tables 5 and 6. Figures 4, 5 and 6illustrate the trends for each algorithm.

Matrix Size (n x n)	**Basic (MB)**	**Block (MB)**	**Loop Optimized (MB)**
10 x 10	0	0	0
100 x 100	0	0	0
250 x 250	0	2	0
500 x 500	8	4	2
1000 x 1000	29	22	8

Table 4: Memory usage comparison for different matrix sizes and algorithms.

M. Size (n x n)	**Spar. 0% (MB)**	**Spar. 50% (MB)**	**Spar. 80% (MB)**	**Spar. 90% (MB)**
10 x 10	0	0	0	0
100 x 100	0	0	0	2
250 x 250	4	4	2	2
500 x 500	22	20	18	8
1000 x 1000	86	56	18	4
2000 x 2000	214	143	80	24

Table 5: Sparse matrix memory usage comparison for different sparsity levels (CSC).

M. Size (n x n)	**Spar. 0% (MB)**	**Spar. 50% (MB)**	**Spar. 80% (MB)**	**Spar. 90% (MB)**
10 x 10	0	0	0	0
100 x 100	2	0	0	2
250 x 250	4	4	4	4
500 x 500	22	20	13	10
1000 x 1000	80	66	16	10
2000 x 2000	254	129	67	20

Table 6: Sparse matrix memory usage comparison for different sparsity levels (CSR).

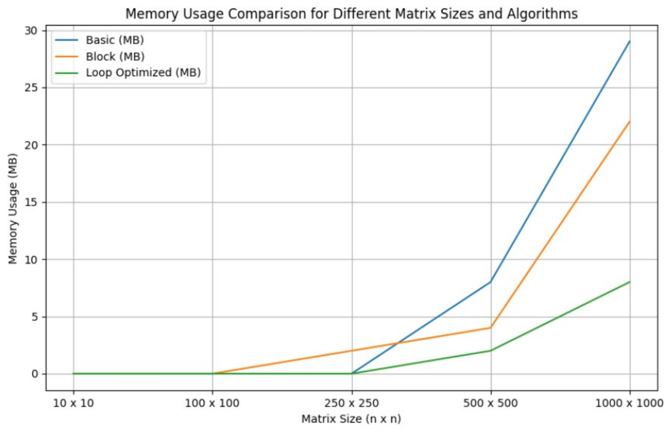

Figure 4: Memory usage comparison for different matrix sizes and algorithms.

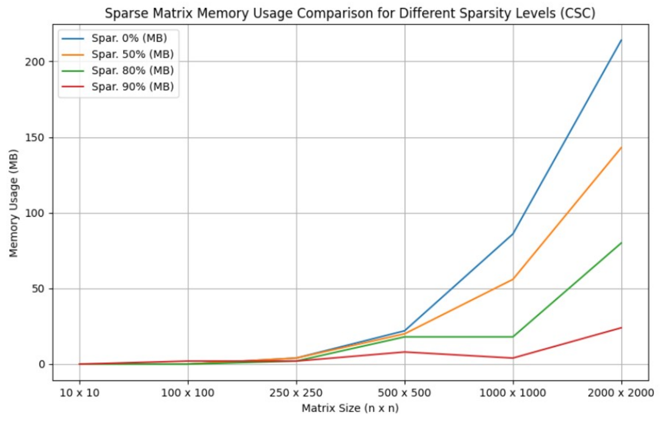

Figure 5: Memory usage comparison for different sparsity levels for the CSC algorithm.

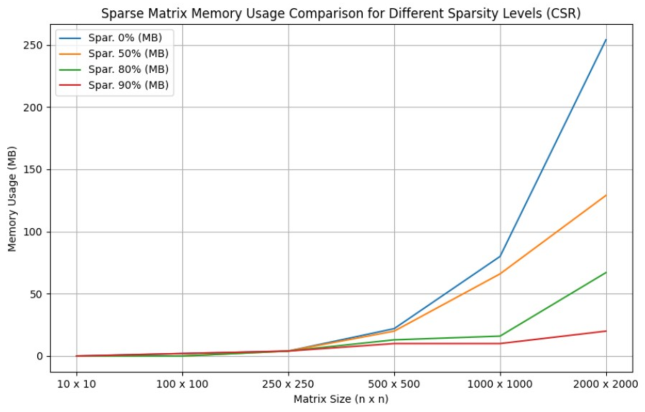

Figure 6: Memory usage comparison for different sparsity levels for the CSR algorithm.

Analysis: The loop-unrolled algorithm showed the best memory efficiency among the dense algorithms, especially as matrix size increased. The sparse algorithms (CSR and CSC) only store non-zero elements, resulting in significantly reduced memory usage compared to the block-based and basic algorithms. Higher sparsity levels led to further reductions in memory consumption, highlighting the efficiency of sparse formats, with CSR achieving an optimal balance between execution time and memory usage.

5.3 Mc2depi Matrix Experiment

In this experiment, we evaluated the performance of the sparse matrix multiplication algorithm using a large, sparse test matrix (mc2depi) in CSR format. The matrix, stored in Matrix Market (.mtx) format, has dimensions of 525,825 x 525,825 and contains 2,100,225 non-zero entries, representing a very high sparsity level. This type of sparse matrix is typical in scientific computations, where only a small fraction of elements hold significant values, allowing for memory optimization and computational efficiency.
The matrix multiplication was executed using the CSR (Compressed Sparse Row) algorithm. This format efficiently stores sparse matrices by compressing rows of non-zero values, thus reducing memory usage and enhancing multiplication speed for sparse data. The algorithm iterates over rows and columns, computing only the necessary non-zero product terms.
The experiment recorded an execution time of 665.65 seconds, using 574 MB of memory. CPU utilization was tracked before and after the multiplication, showing a decrease from 10.56% to 8.42%, reflecting the relatively intensive processing demands of handling large-scale sparse matrices. These results underscore both the memory efficiency and computational cost of the CSR-based algorithm when applied to a highly sparse, large matrix, suggesting areas for optimization in execution time for future work.

5.4 Overall Analysis

Across all experiments, the Compressed Sparse Row (CSR) algorithm consistently delivered the best performance in execution time and memory efficiency, especially at high sparsity levels (0.8 or 0.9). CSR's optimized row-wise access pattern makes it particularly suited for sparse matrices in Java, leading to faster execution than Compressed Sparse Column (CSC) without additional memory cost. As matrix sizes scaled up, this advantage became more prominent: while traditional dense algorithms (basic, block-based, and loop-unrolled) suffered from increasing memory demands and slower performance with larger matrices, the CSR algorithm maintained its efficiency even for sizes up to 7000x7000, making it ideal for large sparse datasets.
Among the dense algorithms, the loop-unrolled version demonstrated the best memory efficiency for larger matrices due to reduced loop overhead, while the block-based algorithm balanced execution speed and memory usage effectively up to mid-sized matrices (1000x1000). However, for larger matrices, even these dense algorithms experienced performance degradation due to escalating memory requirements, emphasizing the limits of dense representations for large-scale computations.
These findings underscore the critical role of algorithm selection in matrix multiplication tasks, particularly for large-scale sparse data. For matrices with high sparsity levels, the CSR algorithm emerged as the most resource-efficient choice, achieving significant gains in computational speed and memory conservation, highlighting its suitability for handling large and sparse datasets.

6 Conclusions

The goal of this research was to optimize matrix multiplication algorithms in terms of execution time and memory efficiency across varying matrix sizes and sparsity levels. We proposed and implemented five distinct algorithms: a basic three-loop algorithm, a block-based algorithm, a loop-unrolled (optimized) version, and two sparse matrix multiplication algorithms using the Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) formats. These algorithms were benchmarked on matrices ranging from sizes of 10x10 to 7000x7000 at different sparsity levels, evaluating their execution time and memory usage.

The experimental results demonstrated that the basic algorithm performed poorly, with high execution times and inefficient memory usage, particularly as matrix size increased. Both the block-based and optimized algorithms showed significant gains in execution time, especially at mid-sized matrices, though their memory demands scaled with size. Among all tested algorithms, CSR proved to be the most efficient for sparse matrices, achieving the lowest execution times and memory usage, particularly at higher sparsity levels (e.g., 80%-90%). This makes CSR highly suitable for large-scale sparse matrix operations, commonly required in fields like machine learning and data analytics.

These findings underscore the importance of choosing the right algorithm based on matrix characteristics. Specifically, CSR provides substantial performance advantages for large, sparse datasets, highlighting its role as an effective choice for applications requiring efficient resource management with extensive data.

7 Future Work

While this study provides a foundational understanding of various matrix multiplication algorithms, there are multiple directions for future research. One avenue is to expand the range of algorithms studied by including methods like Strassen's algorithm or Winograd's algorithm, comparing their performance to the algorithms implemented here. Additionally, exploring parallel implementations on multi-core processors or distributed systems could further reduce execution times, particularly for dense matrix operations.

Another promising direction is to experiment with different hardware configurations, such as GPUs or high-performance computing clusters, to evaluate the scalability of these algorithms. This could be complemented by assessing the algorithms under different sparse matrix storage formats, like COO, to gain more detailed insights into memory and execution efficiency in practical scenarios.

Finally, it would be valuable to conduct experiments using real-world datasets, particularly from domains like scientific simulations and machine learning, where large, sparse matrices are prevalent. Such experimentation would provide a more comprehensive view of the algorithms' performance in realistic conditions, guiding further optimization and adaptation of matrix multiplication methods for diverse applications.