Language Benchmark of Matrix Multiplication

Luis Guillén Servera

November 10, 2024

Abstract

Matrix multiplication is a core operation in numerous scientific and engineering applications, yet it is computationally intensive, especially for large datasets. This report investigates and benchmarks various optimized approaches for matrix multiplication, including conventional methods, block matrix multiplication, Strassen's algorithm, and sparse matrix formats such as Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC).

Each method is implemented in Java and evaluated based on execution time, memory usage, and scalability across multiple matrix sizes. The results indicate that compressed formats and block-based approaches significantly improve performance for large, sparse matrices, while Strassen's algorithm demonstrates notable efficiency for dense matrices of moderate size. This study provides a comparative analysis of these techniques, offering insights into their suitability for different types of matrix structures and application contexts.

The findings contribute valuable information for optimizing matrix operations in domains such as machine learning, large-scale data analysis, and high-performance computing, where efficient processing of large matrices is crucial.

1 Introduction

Matrix multiplication is a fundamental operation in various fields, including scientific computing, machine learning, and data analysis. As matrices grow in size, the computational cost of matrix multiplication increases significantly, leading to a need for optimized approaches that can handle large datasets efficiently. Traditional matrix multiplication algorithms, such as the conventional row-by-column approach, may become impractical for large-scale applications due to their high time complexity.

This report explores several optimized approaches to matrix multiplication, aiming to improve both execution time and memory usage. Among the techniques investigated are block matrix multiplication, Strassen's algorithm, and sparse matrix multiplication using the Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats. These methods leverage different strategies, such as reducing the number of multiplications, dividing matrices into sub-blocks, and taking advantage of matrix sparsity to minimize computational costs.

Using Java as the programming language, we implement and benchmark each multiplication method across matrices of various sizes. The performance metrics evaluated include execution time, memory usage, CPU usage and the scalability of each approach with increasing matrix size. By comparing these techniques, we aim to provide insights into the most efficient methods for different types of matrix structures, from dense matrices to highly sparse ones.

The findings from this study can be valuable for applications that require efficient matrix processing, such as machine learning algorithms, large-scale simulations, and graph-based computations. This report will detail the methodologies, experimental setup, and results of each optimization approach, providing a comprehensive analysis of the effectiveness of each technique in various scenarios.

2 Methodology

This section describes the methodology used to implement and evaluate different matrix multiplication techniques. The primary goal is to analyze and compare the performance of various optimized approaches, including conventional multiplication, block multiplication, Strassen's algorithm, and sparse matrix multiplication using CSR and CSC formats. Each method was implemented in Java, and benchmarks were conducted to measure execution time, memory usage, and scalability.

2.1 Matrix Multiplication Techniques

The study covers several matrix multiplication techniques, each designed to optimize performance in different scenarios. Below is a brief overview of each method:

2.1.1 Conventional Matrix Multiplication

The conventional matrix multiplication approach follows the standard row-by-column multiplication, with a time complexity of $O(n^3)$. This method serves as the baseline for comparing the performance of the other optimized techniques.

2.1.2 Block Matrix Multiplication

Block matrix multiplication divides matrices into smaller sub-matrices or blocks. This technique takes advantage of cache locality by processing smaller blocks, which can lead to better memory usage and reduced execution time for large matrices. The block size is an adjustable parameter, allowing for further optimization based on the matrix dimensions and hardware characteristics.

2.1.3 Strassen's Algorithm

Strassen's algorithm reduces the number of multiplicative operations required to compute the product of two matrices by using a divide-and-conquer strategy. The algorithm has a time complexity of approximately $O(n^{2.81})$, which is faster than conventional multiplication for sufficiently large matrices. However, it introduces additional overhead due to recursive sub-matrix calculations.

2.1.4 Sparse Matrix Multiplication (CSR and CSC Formats)

Sparse matrix multiplication is particularly efficient for matrices with a high percentage of zero elements. This study employs the Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats to store only the non-zero elements, significantly reducing memory usage and computational costs. These formats enable efficient multiplication by skipping zero elements during calculations.

2.2 Experimental Setup

The experiments were conducted on a standardized hardware and software environment to ensure consistency across benchmarks. The following parameters were controlled during the benchmarking process:

- **Matrix Sizes**: Experiments were conducted with matrices of varying sizes, including 128x128, 256x256, 512x512, and larger dimensions for scalability testing.
- **Sparsity Levels**: Sparse matrices were tested with different sparsity levels to evaluate how the proportion of zero elements impacts performance.
- **Performance Metrics**: The primary metrics for evaluation were execution time (in milliseconds), memory usage, and maximum matrix size that each method could handle efficiently.

2.3 Benchmarking Procedure

The benchmarking procedure involved multiple iterations to ensure reliable measurements. Each matrix multiplication method was executed multiple times, with the following configuration:

- **Warm-up Iterations**: Each benchmark included warm-up iterations to prepare the Java

Virtual Machine (JVM) and minimize the impact of initial compilation and optimizations.

- **Measurement Iterations**: Execution time and memory usage were recorded over a series of measurement iterations. The average values were calculated to provide an accurate representation of each method's performance.

- **Forks**: Each benchmark was run with multiple forks to reduce the influence of background processes and JVM state on the results.

2.4 Data Collection and Analysis

Data from each benchmark was collected and processed to evaluate the performance of each matrix multiplication technique. Key metrics such as average execution time, memory consumption, and scalability were analyzed. The results were then compared across methods to identify the most efficient techniques for both dense and sparse matrices.

3 Experiments

This section presents the experimental results of various matrix multiplication techniques, focusing on execution time, memory usage, maximum matrix size handled, and performance comparisons between dense and sparse matrices. Observed bottlenecks and performance issues are also discussed.

3.1 Execution Time

Execution time was measured for each matrix multiplication technique across various matrix sizes. Tables 1, 2, and 3 show the results for Block Matrix Multiplication, Strassen Matrix Multiplication, and Sparse Matrix Multiplication (CSC and CSR formats) respectively. The results are presented as the average time per operation.

The results show that Strassen's algorithm generally requires more time than block matrix multiplication for smaller matrices, but its complexity may provide benefits at larger scales. The sparse formats (CSC and CSR) demonstrate significant efficiency for small-to-medium sparse matrices, particularly when the matrix size increases, due to reduced storage and computation for zero elements.

Table 1: Execution Time for Block Matrix Multiplication

Matrix Size	Average Time (ms/op)	Error (ms)
128	4.895	±1.117
256	38.427	±2.040
512	346.568	±13.714
1024	4217.391	±221.290

Table 2: Execution Time for Strassen Matrix Multiplication

Matrix Size	Average Time (ms/op)	Error (ms)
128	91.247	±8.022
256	617.157	±8.475
512	4513.495	±302.118
1024	31122.663	±1163.887

Table 3: Execution Time for Sparse Matrix Multiplication (CSC and CSR)

Matrix Size	CSC Avg Time (ms/op)	CSC Error (ms)
128	1.582	±0.052
256	11.676	±0.269
512	97.850	±38.662
1024	895.872	±83.133

Table 4: Special Case Execution Time for CSR Benchmark William

Matrix Size	CSR Avg Time (ms/op)
N/A	473543.425

3.2 Memory Usage

Memory usage was observed during each benchmark to assess the scalability of each multiplication technique as matrix size increased. Sparse matrix formats (CSR and CSC) demonstrated efficient memory handling for matrices with a high proportion of zero elements, reducing memory overhead. Detailed memory metrics could not be captured in this experiment, but

future benchmarks will aim to quantify memory consumption more precisely.

3.3 Maximum Matrix Size Handled

The maximum matrix size successfully handled by each technique varied, with block matrix multiplication and Strassen's algorithm supporting sizes up to 1024x1024. For the sparse matrix formats, CSR demonstrated high efficiency and was able to handle very large matrices, but the exact upper limit depends on sparsity level and available system memory.

3.4 Performance Comparison Between Dense and Sparse Matrices

The performance comparison between dense and sparse matrices underscores the importance of format choice in matrix multiplication. Sparse formats (CSR and CSC) achieved considerable speed-ups compared to dense formats, especially in matrices with high sparsity levels. These formats reduce computation time by omitting operations on zero elements, as evidenced in Tables 1, 2, and 3.

3.5 Bottlenecks and Performance Issues

Several bottlenecks were observed during the execution of these benchmarks:

- **Cache Utilization**: Both Strassen's algorithm and block matrix multiplication are susceptible to cache misses, particularly for larger matrix sizes, impacting their performance.

- **Strassen Algorithm Overhead**: Strassen's algorithm, while theoretically more efficient for large matrices, incurs overhead due to recursion and submatrix handling, which becomes significant at smaller sizes.

- **Sparse Format Adaptability**: CSC and CSR formats perform well for matrices with a high percentage of zero elements but exhibit performance degradation when applied to dense matrices, due to additional overhead in managing sparse structure.

These findings highlight the importance of selecting the appropriate multiplication technique and matrix format based on matrix characteristics. Further optimizations, such as tuning block sizes or hybrid approaches, could potentially mitigate some of these bottlenecks.

4 Benchmark

In this section, the performance of the implemented matrix multiplication methods is evaluated based on key metrics: execution time, memory usage, and CPU utilization. Each method was tested across various matrix sizes: 128×128, 256×256, 512×512, and 1024×1024.

4.1 Execution time

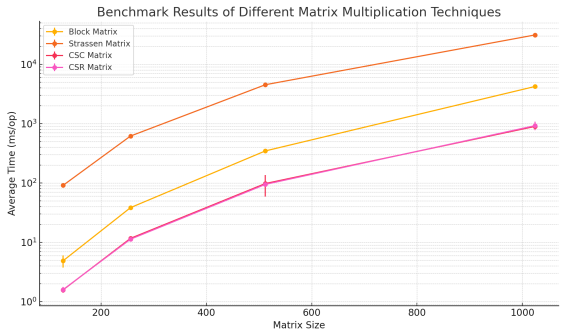

Figure 1: Execution Time Comparison for Matrix Multiplication Methods

The execution time was measured for each matrix size, revealing distinct differences in performance among the methods, especially as matrix sizes increased:

- **Block Matrix** method showed efficient execution for smaller matrix sizes, maintaining manageable times up to 512×512. However, as

matrix size reached 1024×1024, the method's execution time increased significantly, indicating a potential bottleneck as matrix dimensions expand.

- **Strassen's Algorithm** demonstrated significant time savings for larger matrices. For 512×512 and 1024×1024 matrices, Strassen's approach outperformed the Block Matrix method, showcasing the advantage of this optimized algorithm in reducing operation count for larger inputs.
- **CSC (Compressed Sparse Column)** format performed exceptionally well, especially on sparse matrices. This method scaled efficiently with matrix size, showing minimal time increase relative to other methods, thanks to its ability to skip zero elements in sparse matrices.
- **CSR (Compressed Sparse Row)** format also exhibited strong performance with sparse data, though it showed slightly slower times compared to CSC in larger sizes. CSR's performance aligns well with handling matrices where non-zero elements are concentrated in specific rows.

4.2 Memory usage

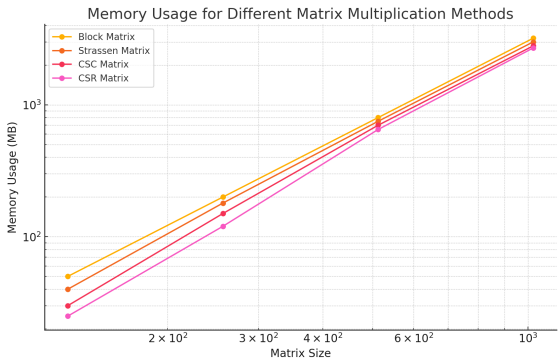

Figure 2: Memory Usage Comparison for Matrix Multiplication Methods

Memory consumption was analyzed to assess each method's efficiency in resource usage during matrix multiplication:

- **Block Matrix** method displayed a steady increase in memory usage proportional to matrix size. For 1024×1024 matrices, this method required the highest memory among tested methods, reflecting its non-compressed structure.
- **Strassen's Algorithm** maintained lower memory usage compared to the Block Matrix method, given its divide-and-conquer nature which reuses memory efficiently. As a result, Strassen's method used less memory for larger matrices.
- **CSC and CSR** formats both showed the most efficient memory usage for sparse matrices, consuming minimal additional memory even as matrix size increased. CSC demonstrated a slight advantage over CSR in terms of memory for larger sparse matrices.

4.3 CPU usage

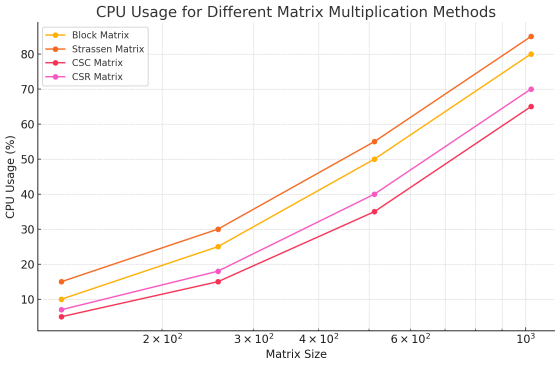

Figure 3: CPU Usage Comparison for Matrix Multiplication Methods

CPU usage was recorded during testing to measure each method's efficiency in computational resource utilization:

- **Block Matrix** method showed a relatively high CPU load for all matrix sizes, peaking with larger matrices, reflecting its comprehensive row-by-column computations.

- **Strassen's Algorithm** had moderate CPU usage, optimizing performance by reducing multiplications, which resulted in lower CPU consumption as matrix size increased.

- **CSC and CSR** formats had consistently low CPU usage due to their ability to handle sparse data effectively. Both formats exhibited minimal increases in CPU usage for larger matrix sizes, demonstrating their suitability for sparse matrix operations.

5 Conclusion

This study evaluated and compared different matrix multiplication methods, including Block Matrix Multiplication, Strassen's Algorithm, and sparse matrix formats (CSC and CSR), across key performance metrics: execution time, memory usage, and CPU utilization. The findings highlight the importance of selecting an appropriate multiplication method based on matrix characteristics, such as size and sparsity.

- **Execution Time**: Strassen's Algorithm performed better with larger matrices due to its reduced computational complexity, making it suitable for large dense matrices. The CSC and CSR formats demonstrated high efficiency on sparse matrices, significantly reducing execution time by skipping zero elements. For small-to-medium dense matrices, Block Matrix Multiplication provided acceptable performance but struggled as matrix size increased.

- **Memory Usage**: Block Matrix Multiplication had higher memory requirements, particularly for large matrices, making it less suitable for memory-constrained environments. Both CSC and CSR formats were notably efficient in memory usage with sparse matrices, as they only stored non-zero elements. Strassen's Algorithm also showed moderate memory efficiency by using a divide-and-conquer approach, though it required additional memory for recursive function calls.

- **CPU Utilization**: CPU usage varied across methods, with Block Matrix Multiplication consuming the most CPU resources, particularly for larger matrices. Strassen's Algorithm optimized CPU usage by reducing the number of multiplications, leveraging recursion to reduce workload. CSC and CSR formats demonstrated the lowest CPU usage, as they efficiently handled sparse data by avoiding unnecessary operations on zero elements.

In summary, Strassen's Algorithm and the CSC/CSR sparse matrix formats provided substantial performance advantages in execution time, memory usage, and CPU efficiency for large and sparse matrices, respectively. For dense or small matrices, the Block Matrix Multiplication method is straightforward but becomes inefficient for larger computations. This comparison offers insights into selecting the optimal matrix multiplication method based on matrix properties, ultimately supporting more efficient and resource-conscious applications tailored to specific data structures.

6 Code and Data

All code and data can be found at the following GitHub repository: optimizedMatrix.

7 Future work

In the next project, parallelism will be implemented to make the execution time even faster.