Optimized Matrix Multiplication Approaches and Sparse Matrices in Java

Adonai Ojeda Martín

November 10, 2024

Abstract

Matrix multiplication is a core operation in various computational domains, but its computational cost increases with matrix size. Optimizations such as Strassen's algorithm, loop unrolling, and cache optimizations are frequently employed to improve performance. This paper investigates the effects of these optimizations on both dense and sparse matrices in Java. We evaluate performance in terms of execution time, memory usage, and matrix size limitations, with a specific focus on the role of sparsity in enhancing performance efficiency. Sparse matrices, which contain a high percentage of zero elements, offer additional opportunities for performance improvements. We analyze the impact of varying sparsity levels on computational efficiency across different matrix multiplication techniques.

1 Introduction

Matrix multiplication is widely used in fields such as scientific computing, machine learning, and graphics processing. However, as matrix sizes increase, the computation becomes intensive. This paper explores optimized approaches in Java, focusing on dense and sparse matrix multiplications to reduce computational costs.

2 Problem Statement

The aim is to optimize matrix multiplication by implementing and comparing different approaches, including a basic matrix multiplication, loop unrolling, and Strassen's algorithm. Additionally, we analyze sparse matrices, which contain a high percentage of zero elements, to study how varying sparsity levels impact computational efficiency.

3 Methodology

The goal of this study is to compare three matrix multiplication techniques implemented in Java. Each technique represents a distinct approach to optimize the computational workload involved in matrix multiplication, particularly when working with large matrices and sparse data structures. Java's object-oriented nature and built-in memory management (e.g., garbage collection) have implications for how these algorithms perform, particularly in memory-intensive operations such as Strassen's algorithm. The three methods implemented for comparison are described below:

- **Basic Matrix Multiplication:** This is the standard approach to matrix multiplication. It uses nested loops to iterate over each row of the first matrix and each column of the second matrix, computing the dot product for each element in the resulting matrix. The complexity of this approach is $O(n^3)$, making it computationally expensive for large matrices. While simple and easy to implement, it becomes inefficient as the matrix size increases.

- **Loop Unrolling:** This optimization technique aims to reduce the overhead associated with looping constructs. By manually expanding the loops, the number of loop control instructions is reduced, allowing for more efficient execution. Loop unrolling helps decrease the number of iterations and thus reduces the time spent on loop management, improving performance especially for large matrices. However, this technique can lead to an increase in code size and might not always result in significant speedups for all use cases.

- **Strassen's Algorithm:** This is a more advanced approach that improves the computational complexity of matrix multiplication. Strassen's algorithm recursively divides matrices into smaller submatrices, reducing the number of multiplicative operations required. Instead of performing $O(n^3)$ operations like the basic approach, Strassen's algorithm reduces the complexity to approximately $O(n^{\log_2 7})$, which is a significant improvement for large matrices. The trade-off is the increased memory usage and the need for additional operations to combine the submatrices.

For the purpose of this study, the matrices were considered sparse, meaning they contain a significant number of zero elements. Sparse matrices are common in many real-world applications, such as scientific computing, where

many elements of a matrix may have no effect on the outcome of a computation. To simulate this, zero elements were assigned randomly across the matrix according to specified sparsity levels: 0.25, 0.5, 0.75, and 0.9. These levels correspond to matrices where 25

The methods were tested on matrices of increasing size, from 128x128 to 512x512, to explore the performance limits and the impact of matrix sparsity on execution time and memory usage. By examining these variables, we can gain insights into how the different algorithms scale with matrix size and how they handle the challenges posed by sparse data.

4 Experiments

This section details the experimental setup, metrics used, and observations drawn from the data.

4.1 Experimental Setup

- **Matrix Sizes**: We evaluated matrix sizes of 128x128, 256x256, 512x512, and 1024x1024 to examine how performance scales with increased matrix dimensions.
- **Sparsity Levels**: Four levels of sparsity (0.25, 0.5, 0.75, and 0.9) were used to simulate matrices with various percentages of zero elements, mimicking real-world sparse matrices.
- **Optimizations**: We implemented three variations of matrix multiplication: Basic, Loop Unrolling, and Strassen. These methods vary in complexity, computational overhead, and efficiency.

4.2 Metrics Evaluated

- **Execution Time**: Measured in milliseconds (ms), indicating the time taken to complete matrix multiplication.
- **Memory Usage**: Measured in megabytes (MB), showing the memory footprint of each multiplication algorithm.
- **Scalability**: The maximum matrix size that each method could handle efficiently.

5 Results

5.1 Results for 128x128 Matrix

Table 1: Performance on 128x128 Matrix with varying sparsity levels.

Algorithm	**Execution Time (ms)**	**Memory Usage (MB)**	**Sparsity Level**
Basic	420	62.25	0.25
Loop Unrolling	370	7.74	0.25
Strassen	642	59.22	0.25
Basic	341	75.62	0.5
Loop Unrolling	342	6.75	0.5
Strassen	288	61.65	0.5
Basic	277	19.14	0.75
Loop Unrolling	257	19.21	0.75
Strassen	269	54.76	0.75
Basic	218	18.23	0.9
Loop Unrolling	386	36.58	0.9
Strassen	280	53.67	0.9

5.2 Results for 256x256 Matrix

Table 2: Performance on 256x256 Matrix with varying sparsity levels.

Algorithm	Execution Time (ms)	Memory Usage (MB)	Sparsity Level
Basic	2541	15.11	0.25
Loop Unrolling	2509	100.49	0.25
Strassen	1963	238.68	0.25
Basic	2653	58.92	0.5
Loop Unrolling	2543	163.24	0.5
Strassen	1776	132.12	0.5
Basic	2292	79.00	0.75
Loop Unrolling	2235	36.82	0.75
Strassen	1639	30.00	0.75
Basic	1844	51.01	0.9
Loop Unrolling	1759	71.69	0.9
Strassen	1781	40.60	0.9

5.3 Results for 512x512 Matrix

Table 3: Performance on 512x512 Matrix with varying sparsity levels.

Algorithm	Execution Time (ms)	Memory Usage (MB)	Sparsity Level
Basic	28378	120.90	0.25
Loop Unrolling	28535	122.03	0.25
Strassen	12356	420.45	0.25
Basic	25425	179.22	0.5
Loop Unrolling	24892	71.72	0.5
Strassen	11674	208.70	0.5
Basic	24451	265.43	0.75
Loop Unrolling	25788	220.76	0.75
Strassen	11162	62.04	0.75
Basic	18234	166.94	0.9
Loop Unrolling	19240	267.47	0.9
Strassen	10894	290.38	0.9

5.4 Results for 1024x1024 Matrix

Table 4: Performance on 1024x1024 Matrix with varying sparsity levels.

Algorithm	**Execution Time (ms)**	**Memory Usage (MB)**	**Sparsity Level**
Basic	242616	162.60	0.25
Loop Unrolling	215907	232.69	0.25
Strassen	77908	813.20	0.25
Basic	228630	148.66	0.5
Loop Unrolling	254137	326.89	0.5
Strassen	73643	1028.53	0.5
Basic	216715	161.91	0.75
Loop Unrolling	218067	170.82	0.75
Strassen	71951	730.08	0.75
Basic	164447	165.87	0.9
Loop Unrolling	166598	199.21	0.9
Strassen	74286	162.13	0.9

6 Discussion and Analysis

This section interprets the results, highlighting insights, patterns, and potential optimizations.

6.1 Performance Patterns

As the matrix size increases, the differences between the basic algorithm and optimized methods become more apparent, particularly for lower sparsity levels. This suggests that Strassen's algorithm benefits significantly from larger problem sizes, where the reduction in multiplicative operations outweighs the increased memory usage. For higher sparsity levels, performance improvement is less noticeable across the methods. This is expected as fewer non-zero elements reduce the overall computational workload, making the basic method less computationally expensive even for larger matrices.

6.2 Memory Trade-offs

Strassen's algorithm, while the fastest in terms of execution time for large matrices with lower sparsity, comes at the cost of increased memory usage. This is due to the additional memory required for the submatrices and intermediate results. For matrices with very high sparsity, the increased memory overhead can negate the benefits in execution time, especially in environments with constrained resources. - Loop unrolling, while offering reduced loop overhead, did not consistently outperform the basic method in terms of execution time, suggesting that its benefits may be limited in the context of sparse matrices where computation is dominated by the presence of non-zero elements.

6.3 Optimization Opportunities

Further memory optimizations could be achieved by improving how submatrices are stored in Strassen's algorithm. For example, using in-place matrix operations or optimizing the way memory is allocated and freed could reduce memory overhead. Another possible improvement is hybridizing Strassen's algorithm with block matrix multiplication, which could combine the efficiency of Strassen with the memory benefits of traditional methods. Additionally, adjusting the threshold at which Strassen is used (e.g., switching between traditional and Strassen methods based on matrix size) could provide a better trade-off between memory usage and computation time.

7 Conclusion

The experiments conducted demonstrate that optimized matrix multiplication methods significantly impact performance for large, dense matrices. Strassen's algorithm provided the most efficient solution in terms of time for large matrices, though at a higher memory cost. Sparse matrix techniques effectively reduced both time and memory usage as the sparsity level increased. This study emphasizes the importance of choosing an appropriate algorithm based on matrix size and sparsity, showing that algorithmic optimizations like Strassen and Loop Unrolling can yield considerable performance gains when applied thoughtfully.

8 Future Work

Future work could explore the following:

- **Hybrid Algorithms**: Developing hybrid approaches that adapt dynamically based on matrix size and sparsity, switching between dense and sparse algorithms.
- **Parallel Implementations**: Implementing parallel processing (e.g., multithreading) for Strassen's algorithm to reduce execution time further, especially for large matrices.
- **Machine Learning Integration**: Exploring machine learning models to predict the best algorithm based on matrix properties, optimizing the choice of multiplication method dynamically.
- **Alternative Sparse Storage Formats**: Examining various sparse matrix storage formats (e.g., CSR, COO) to determine their impact on performance in different sparsity scenarios.
- **Energy Efficiency Analysis**: Studying the energy consumption of each algorithm, particularly in large-scale computations, to identify the most sustainable approach.

9 References

- Java Documentation: https://docs.oracle.com/javase/8/docs/
- Strassen's Algorythm: https://www.geeksforgeeks.org/strassens-matrix-multiplication
- Loop Unrolling Algorythm: https://blogs.oracle.com/javamagazine/post/loop-unrolling
- ChatGPT: https://chatgpt.com/
- Stack Overflow: https://stackoverflow.com/
- GitHub Repository: Optimized Matrix Multiplication Approaches and Sparse Matrices