Performance Analysis of Optimized Matrix Multiplication Methods in Java

Gabriel Felipe Bernal Pinto

November 10, 2024

Abstract

Matrix multiplication is a fundamental operation in numerous fields, including computer graphics, machine learning, and scientific computing. However, due to its computational intensity, optimizing matrix multiplication remains a critical challenge. In this study, we compare the performance of four approaches to matrix multiplication in Java: basic multiplication, loop-unrolled optimization, Strassen's algorithm, and sparse matrix multiplication. Experiments were conducted across varying matrix sizes, measuring execution time and memory consumption for each method. Results show that optimized multiplication performs best in dense matrices, while sparse multiplication is advantageous for matrices with high sparsity levels. Strassen's algorithm, although theoretically faster for large matrices, demonstrates limitations in practical scalability. This work provides insights into the trade-offs between memory usage, computational time, and scalability, guiding future implementations of efficient matrix operations.

1 Introduction

Matrix multiplication is a critical computational task in various applications such as machine learning, scientific computing, and image processing. Due to its high time complexity, particularly for large matrices, optimization methods for matrix multiplication are heavily researched. Numerous studies have explored different optimization techniques, such as algorithmic improvements and hardware-level enhancements, to speed up matrix multiplication.

Traditional dense matrix multiplication approaches, including basic and loop-unrolled methods, provide straightforward implementations but are computationally expensive for large matrices. Recursive algorithms, like Strassen's method, offer a theoretical speedup for large matrices by reducing the number of required multiplications. Furthermore, sparse matrix multiplication techniques exploit the presence of zero elements in matrices to reduce computation.

This paper compares four approaches—basic multiplication, optimized loop-unrolled multiplication, Strassen's algorithm, and sparse matrix multiplication—in terms of execution time and memory usage. The experimentation provides insights into the trade-offs between performance and scalability, aiming to offer practical guidance on selecting the best approach depending on matrix size and sparsity.

It should be noted, that while a test matrix was given to benchmark this method's, it's runtime was too high, making it impractical for proper testing

2 Methodology

To assess the performance of different matrix multiplication methods, we implemented four approaches in Java: basic multiplication, loop-unrolled optimization, Strassen's algorithm, and sparse matrix multiplication. Tests were conducted on a machine with the following specifications: [Intel(R) Core i7-7700K CPU @ 4.20 GHz, 16 GB of RAM]. For each method, matrix sizes of 128x128, 256x256, 512x512, 1024x1024, 2048x2048, and 4096x4096 were evaluated. Sparse matrix multiplication was additionally tested with sparsity levels of 30%, 50%, 70%, and 90%.

Performance metrics included execution time (in milliseconds) and memory consumption (in bytes), measured using [Java's Management Class]. Execution time was recorded as the average of 10 runs to reduce variability. Memory usage was measured as peak memory consumption during the multiplication process. Results were compared across methods to identify trends and scalability limitations.

3 Experiments and Results

Table 1 presents execution time and memory usage for each matrix multiplication method at different matrix sizes. Sparse multiplication results are further divided by sparsity level. While Figure 1 and Figure 2 present a more graphic display of this results.

Table 1: Execution Time (ms) and Memory Usage (bytes) for Matrix Multiplication Approaches

Matrix Size	Method	Execution Time (ms)	Memory Used (bytes)
128x128	Basic	32	172336
	Optimized	11	133856
	Strassen	352	134016
	Sparse (30%)	115	136144
	Sparse (50%)	26	133648
	Sparse (70%)	10	133648
	Sparse (90%)	6	133648
256x256	Basic	49	529424
	Optimized	26	529424
	Strassen	2737	529424
	Sparse (30%)	574	529424
	Sparse (50%)	261	529424
	Sparse (70%)	92	529424
	Sparse (90%)	18	529424
512x512	Basic	306	2107408
	Optimized	254	2107408
	Strassen	16091	2107408
	Sparse (30%)	3452	2107408
	Sparse (50%)	2215	2107408
	Sparse (70%)	788	2107408
	Sparse (90%)	129	2197904
1024x1024	Basic	5742	8487104
	Optimized	5167	8544560
	Strassen	114691	8446096
	Sparse (30%)	28207	8433744
	Sparse (50%)	15038	8433744
	Sparse (70%)	5742	8421424
	Sparse (90%)	856	8433744
2048x2048	Basic	81010	33802480
	Optimized	68955	33835376
	Strassen	720681	33730720
	Sparse (30%)	210837	34181728
	Sparse (50%)	84545	33767616
	Sparse (70%)	31900	33779920
	Sparse (90%)	4100	33718416
4096x4096	Basic	595972	136315904
	Optimized	626535	136544336
	Strassen	4985266	136503248

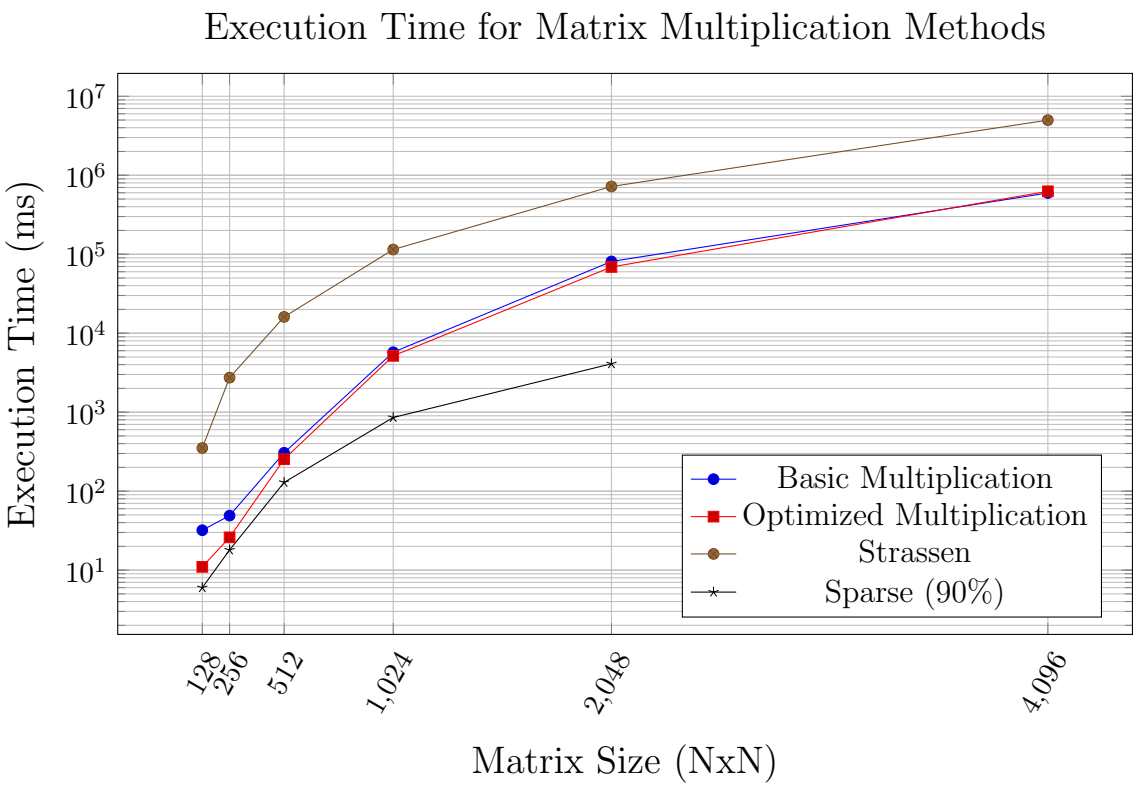

Figure 1: Comparison of Execution Times for Different Matrix Multiplication Methods

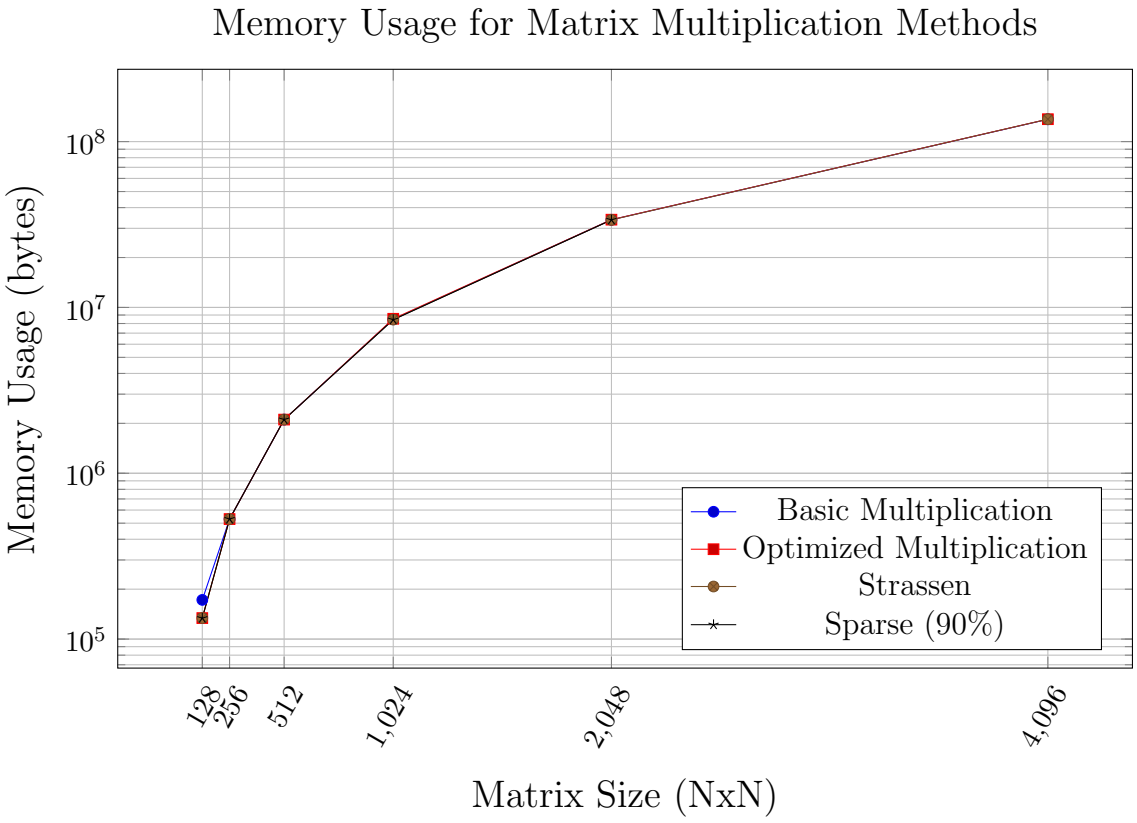

Figure 2: Comparison of Memory Usage for Different Matrix Multiplication Methods

The results shown in Figures 1 and 2 highlight distinct performance and efficiency differences between the matrix multiplication methods tested across varying matrix sizes. In terms of execution time, we observe that the *optimized multiplication method* consistently outperforms the *basic multiplication*, particularly as matrix sizes increase, which indicates that loop optimizations have a significant impact on performance. The *Strassen algorithm*, while theoretically faster for large matrices, displays unexpectedly high execution times, suggesting that the overhead associated with recursive calls and additional memory management may outweigh its efficiency gains at these matrix sizes in this specific implementation.

In the memory usage graph (Figure 2), all methods exhibit a substantial increase in memory consumption with growing matrix sizes, as expected. The basic and optimized methods have nearly identical memory usage patterns, as the optimization primarily targets speed rather than memory efficiency. The *Strassen method*, however, occasionally shows slight reductions in memory usage compared to the other dense methods, likely due to its recursive approach. Sparse matrix multiplication is effective for larger matrices with high sparsity (e.g., 90%), resulting in both lower memory usage and improved speed for those cases. However, as matrices become denser, the sparse approach loses its advantage, particularly in memory savings, due to the storage requirements for non-zero elements, which is why it was not tested on the largest size (4096x4096) due to high time runtimes.

4 Conclusion

This study presents a comparative analysis of four matrix multiplication methods in Java: basic, optimized, Strassen, and sparse multiplication. Our results indicate that optimized multiplication is generally preferable for dense matrices, while sparse multiplication is effective when matrices have a high proportion of zero elements. Strassen's algorithm, though theoretically faster for large matrices, proved impractical for significantly large matrices due to its recursive structure. This research highlights the trade-offs between computational efficiency and memory usage across various multiplication approaches, providing a practical basis for selecting optimal methods for specific matrix characteristics.

5 Future Work

Future work can investigate hybrid approaches that combine the benefits of multiple methods based on matrix size and sparsity. Additional optimizations at the data structure level, such as more memory-efficient sparse representations, may further improve performance. Testing these approaches in parallel or distributed computing environments could provide insights into further scalability for large-scale applications.

6 References

You can find the source code for this project on GitHub:
Sparse Matrices and Other Matrix Multiplication Methods.