Compilation time of matrix multiplication

Gil Bernal, Víctor

GITHUB link: https://github.com/BeepBoopVictor/MatrixMultiplicationBD

***Abstract*—This document details the study of performance and its optimization in algorithms related to matrix multiplication. The fields analyzed are different algorithm approaches, number of iterations, execution time and the languages used.**

I. Introduction

FOR the last decades, the increase of data disponibility and the increasing speed of its collection has lead to significant challenges in the fields of computer and data science. As datasets continue to increase in size and complexity, the demand for efficient processing techniques has become critical.

A fundamental operation that is subject to this problem is matrix multiplication. This technique is used in most of deep-learning algorithms. However, the computational cost of multiplying matrices of large dimensions increases along with its size, becoming a bottleneck in some situations.

For matrices of size n×n, the basic multiplication algorithm has a time complexity of $O(n^3)$, making it slow for modern datasets with large dimensions. This issue has inspired multiple number of researches focused on optimization techniques aimed at reducing the computational cost. Techniques such as parallel computing, alternative algorithms, and cache optimization are just a few of the many strategies developed to improve performance. However, with the continuous expansion of data and the increasing demand for real-time processing, finding efficient solutions remains a crucial problem in the field.

This study explores various optimization techniques in matrix multiplication, evaluating their effectiveness in addressing the challenge of large dimensions, size and computational limitations.

II. Basic Algorithm in Different Languages

THIS section details the study of performance of the basic matrix multiplication algorithm in different programming languages –Python, Java, C++ and Rust–, the result will give a superficial view of the efficiency when handling the task for different matrix sizes.

The performance metrics used in this study include:

- Execution Time (milliseconds): The time taken by the algorithm to perform the matrix multiplication task, which is a key indicator of how quickly each language can process matrices of different sizes.
- Memory Usage (MB): The amount of memory consumed during the execution, which highlights the language's efficiency in handling resources.
- CPU Usage (%): The percentage of CPU resources used during the execution, which shows how much computational power each language utilizes for the task.

Firstly we will benchmark based on the number of iterations×rounds, with the objective of studying the execution time for every language and matrix size.

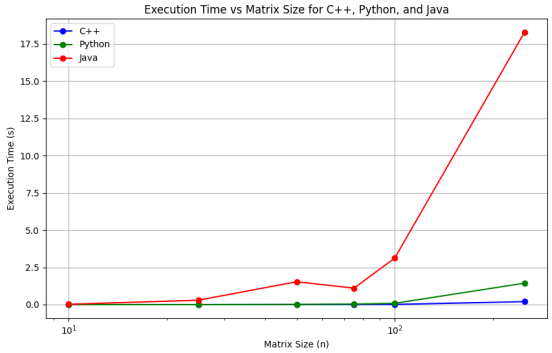

Figure 1. Execution time based on language and size.

As seen in figure 1 we have the different results on execution time for the different languages, measured in milliseconds per execution.

Java takes significantly longer execution times for larger matrices, with an exponential increase in runtime as matrix size grows, which could indicate inefficiencies in handling large datasets. Python and C++ show more consistent execution times, with Python slightly higher than C++ at larger matrix sizes.

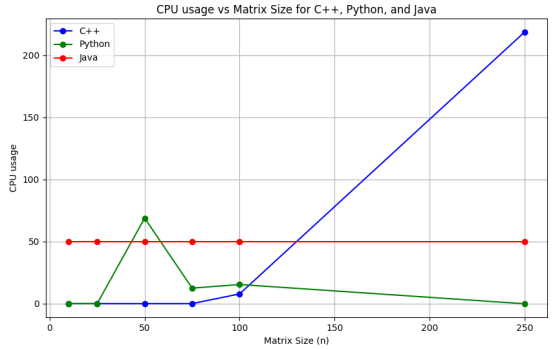

Figure 2. CPU usage based on language and size.

C++ shows a rise in CPU usage for larger matrix sizes, especially beyond size 250. Python demonstrates fluctuating CPU usage, reaching peaks around size 50, followed by a decrease as the matrix size increases. Java maintains a constant CPU usage regardless of matrix size, indicating a more stable and predictable behavior.

Regarding C++, memory usage starts low and increases significantly at larger matrix sizes, particularly at size 250 where it jumps to 0.996 MB. This shows that C++ manages memory efficiently for smaller matrices but requires more memory as matrix size increases, which is expected for a statically compiled language.

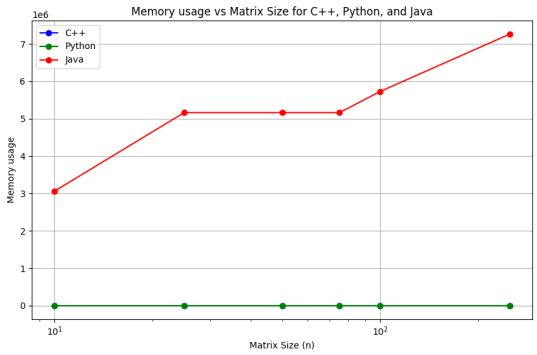

Figure 3. Memory usage based on language and size.

Python's memory usage fluctuates less and stays generally low, except for matrix size 25 where it spikes. This may reflect Python's interpreter overhead and dynamic memory allocation, which can occasionally lead to higher memory use for certain matrix sizes, but overall it's efficient with larger matrices.

In Java the data does not indicate significant memory usage for Java, which likely means it manages memory conservatively or the garbage collection system is efficiently handling the allocation.

III. Optimized Matrix Multiplication Approaches and Sparse Matrices

FOR the past few decades, mathematicians and computer scientists have intensively studied matrix multiplication algorithms in order to develop faster and more efficient methods, a challenge with very important implications in fields like machine learning, physics, and computer graphics. The classical approach, which involves a naive multiplication of each element, has a time complexity of $O(n^3)$, making it computationally expensive for large matrices. In 1969, the Strassen algorithm revolutionized the field with an algorithm that reduced the complexity to $O(n^{2.81})$. Since then, algorithms like the Coppersmith-Winograd algorithm and recent advancements using deep learning approaches have pushed the complexity even lower with each new discovery.

In this section we are going to analize the performance of different algorithms regarding average matrix and sparse matrix in Java and explain a little bit about them.

A. Normal matrix multiplication

We begin by examining normal (dense) matrix multiplication using matrices filled with randomly generated double values. Three algorithms are implemented and analyzed:

Naive Algorithm: The baseline method for matrix multiplication operates with a cubic complexity of $O(n^3)$ by iterating over each element. While inefficient for large matrices, it remains a common choice for smaller matrices due to its simplicity and low memory overhead.

Block Matrix Multiplication: The block matrix algorithm divides each matrix into smaller submatrices (blocks) and performs multiplications within these blocks. This approach not only reduces cache misses but also enables better utilization of CPU cache memory, resulting in speedup over the naive approach. Block matrix multiplication has shown particular advantages in handling larger matrices efficiently.

Strassen's Algorithm: Strassen divides matrices into quadrants and applies a divide-and-conquer strategy to reduce the number of multiplications required. However, it introduces additional memory requirements and may suffer from overhead for smaller matrix sizes. In practice, Strassen's algorithm often underperforms on smaller matrices and becomes advantageous only for larger matrices.

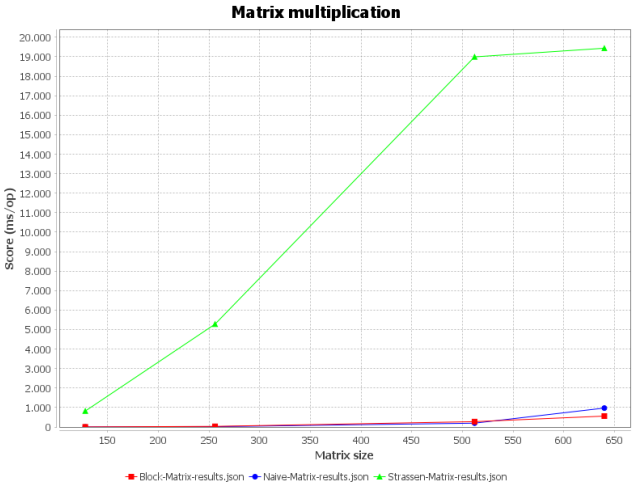

Figure 4. Execution time for different algorithms.

From the results, the Strassen algorithm is the slowest due to its recursive overhead, particularly noticeable with smaller matrices. The block algorithm slightly outperforms the naive approach, indicating its suitability for optimizing cache usage.

B. Sparse matrix multiplication

Sparse matrices are characterized by a high proportion of zero elements, enabling optimizations that reduce both memory usage and computation. We tested each algorithm on sparse matrices with various sparsity levels, which measure the percentage of zero elements in the matrix.

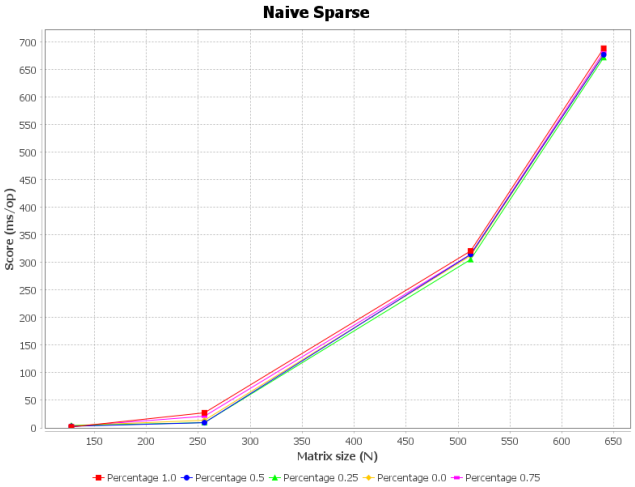

Figure 5. Execution time for naive algorithm.

For the naive algorithm on sparse matrices, we observe that the trend remains consistent across different sparsity levels. The sparse matrix structure allows for skipping unnecessary multiplications, yielding performance gains relative to dense matrices.

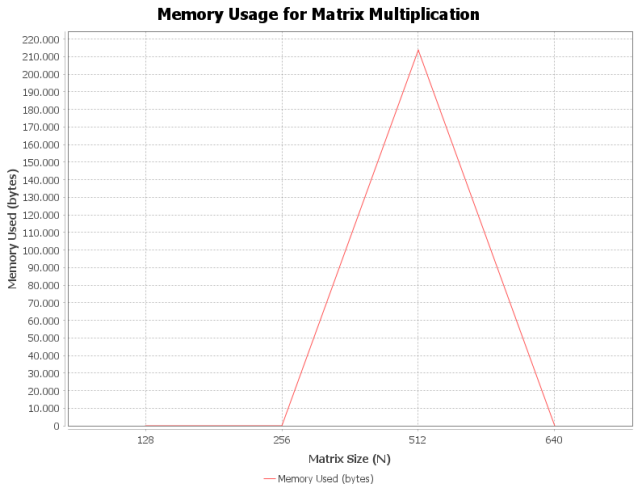

Figure 6. Memory for naive algorithm.

The memory footprint for the naive algorithm remains relatively stable, though some outliers occur at larger matrix sizes, particularly for size 512, where memory usage shows a slight increase.

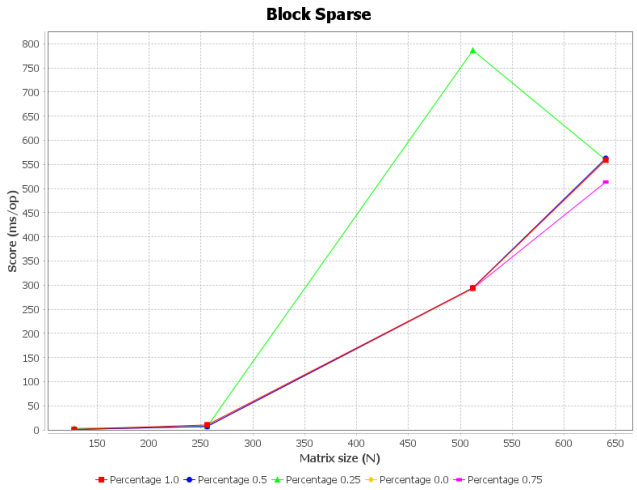

Figure 7. Execution time for block algorithm.

In contrast, the block algorithm exhibits a notable outlier in the 25% zero matrix, indicating that certain sparsity levels may introduce irregularities in memory access patterns. Overall, the block method shows stable performance across different sparsity configurations, affirming its robustness for larger, sparse matrices.

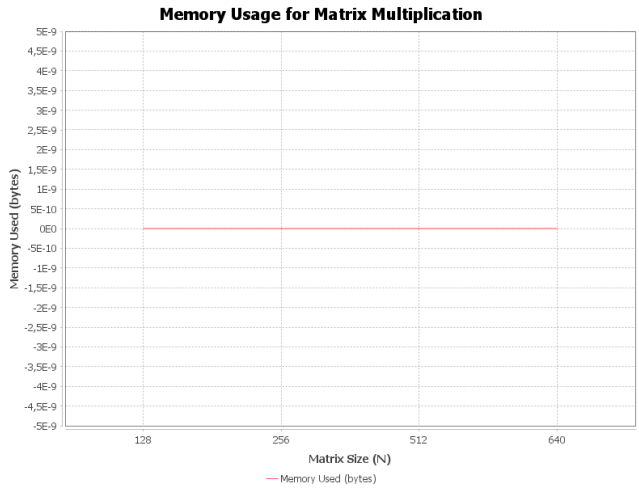

Figure 8. Memory for block algorithm.

The memory usage for the block algorithm remains close to zero, thanks to efficient memory management within the block divisions, further enhancing its efficiency in handling sparse matrices.

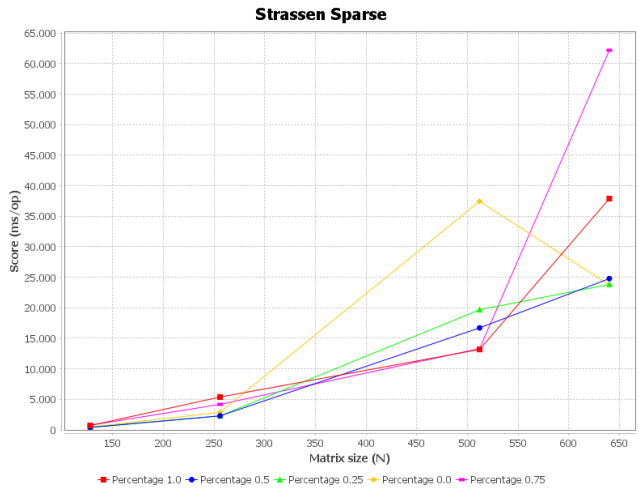

Figure 9. Execution time for Strassen algorithm.

Finally, the Strassen algorithm performs inconsistently with sparse matrices, exhibiting varied execution times across different sparsity levels. The added complexity of recursive calls, combined with sparse data, appears to hinder its efficiency, particularly for matrices with higher zero percentages.

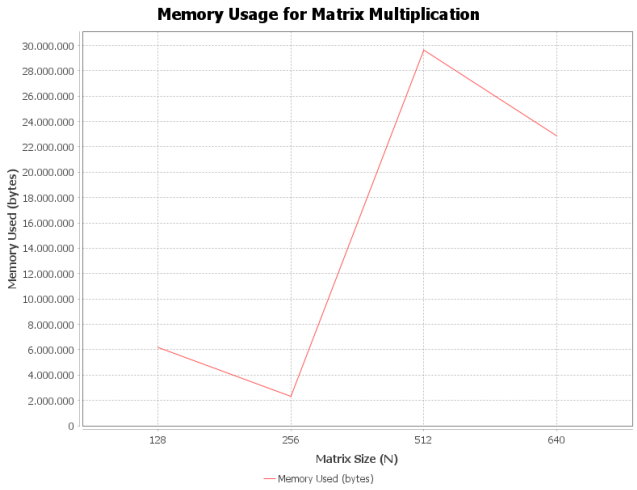

Figure 10. Memory for Strassen algorithm.

The memory usage for Strassen's algorithm tends to increase with matrix size, revealing a higher memory overhead compared to the other methods. This result suggests that Strassen's algorithm may not be optimal for sparse matrix multiplication, where memory efficiency is crucial.