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.

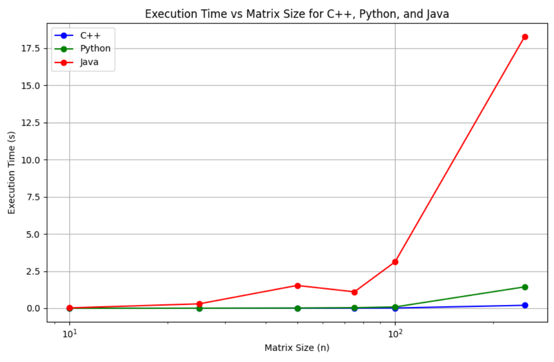

Figure 1. Execution time based on language and size.

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.

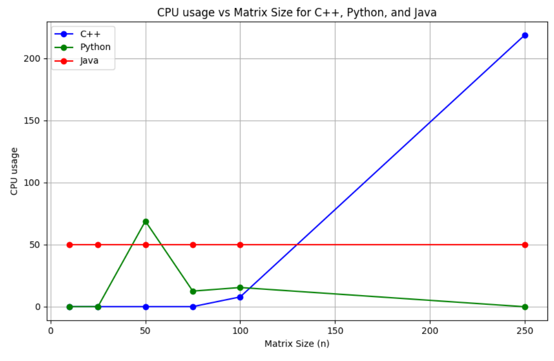

Figure 2. CPU usage based on language and size.

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.

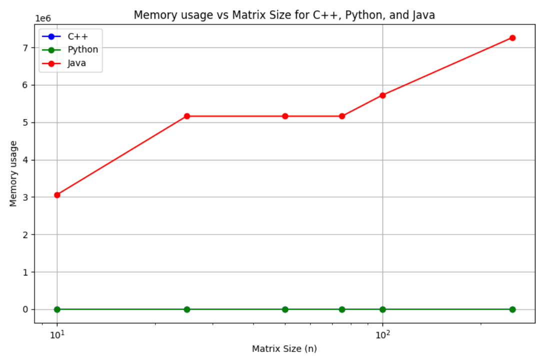

Figure 3. Memory usage based on language and size.

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.