Language Benchmark of matrix multiplication

DIEGO JESÚS TORREJÓN CABRERA

October 2024

GitHub link: Language-Benchmark-of-matrix-multiplication

1 Abstract

Matrix multiplication is a computationally intensive task commonly used in various fields like machine learning and scientific computing. The challenge lies in identifying the most efficient programming language for handling large matrix operations. In this study, we compare the performance of a basic matrix multiplication algorithm across Python, Java, and C++, focusing on execution time and memory usage. Experiments were conducted by multiplying square matrices of sizes 100x100, 200x200, 300x300, 500x500 and 1000x1000 on a MacBook Pro with an Apple Silicon M3 Pro chip.

The results show a clear hierarchy: C++ significantly outperforms Java and Python, both in speed and memory efficiency. Java performs better than Python, benefiting from Just-In-Time (JIT) compilation but is slower than C++ due to garbage collection overhead. Python, being an interpreted language, showed the poorest performance due to its higher computational overhead.

In conclusion, C++ is the best option for performance-critical tasks like matrix multiplication, Java offers a balance between speed and ease of use, and Python is suitable for smaller tasks or when using optimized libraries.

2 Introduction

In recent years, benchmarking programming languages for computational tasks has become a crucial area of study, especially when evaluating performance-intensive operations like matrix multiplication. Matrix multiplication is a fundamental operation in various fields, from scientific computing to machine learning. By comparing how different programming languages handle this task, we can gain insights into the strengths and weaknesses of each language in terms of execution time, memory usage, and computational overhead.

Matrix multiplication involves the product of two matrices, a task that, depending on the algorithm and the matrix size, can quickly grow in computational complexity. The basic algorithm, typically taught in introductory courses, has a time complexity of O(n3), where n is the dimension of the matrices. Different languages approach this problem with varying levels of efficiency, due to factors like how they manage memory, optimize low-level operations, and handle computational overhead. Thus, benchmarking these languages allows us to make informed decisions about which language is better suited for specific use cases.

Several authors have approached this topic by comparing the performance of popular languages. For example, many studies have explored the performance gap between compiled languages like C++ and Java, which offer more control over memory management and optimization, and interpreted languages like Python, which are more user-friendly but often less efficient in computationally intensive tasks.

In this assignment, we aim to compare the performance of a basic matrix multiplication algorithm implemented in three popular programming languages: Python, Java, and C++. We will focus on analyzing execution time and memory usage to determine how each language handles this fundamental operation. The contribution of this paper lies in providing a clear and up-to-date comparison of these languages for matrix multiplication.

3 Methodology

To evaluate the performance of matrix multiplication across Python, Java, and C++, a series of experiments will be conducted. The experiments will involve multiplying two square matrices of varying sizes: 100x100, 200x200, 300x300, 500x500 and 1000x1000. These matrix sizes have been chosen to reflect small, medium, and large-scale problems, providing a comprehensive view of how each language performs under different computational loads.

For each matrix size, the experiments will measure two key metrics: execution time and memory usage. The execution time will indicate how efficiently each language handles the computation, while memory usage will give insight into how each language manages system resources during matrix multiplication. Benchmarking tools specific to each language, such as pytest-benchmark, will be used to capture these measurements accurately.

The experiments will be conducted on a MacBook Pro equipped with an Apple Silicon M3 Pro chip, which includes 11 CPU cores and 18 GB of unified memory. However, to execute this algorithm in C++, a virtual machine with the Fedora operating system has been used to execute it. It has been assigned 1 core and 4 GB of RAM. The emulator that has been used to create said virtual machine has been UTM.

4 Experiments

To carry out the following experiments, the following modifications have been made to the codes offered on campus. You can see the code for the following languages at the following link to GitHub: Language-Benchmark-of-matrix-multiplication. The modifications are as follows:

- In Python the code has been divided into two files. The first, **matrixMultiplicationAlgorithm.py**, which contains the basic matrix multiplication algorithm. The second, **test_algorithm.py**, which contains the function to perform the benchmark with pytest-benchmark. In addition, before doing the benchmark, the average memory used for 5 executions is calculated in order to obtain the memory usage by Python in matrix multiplication.

- In Java, 2 classes have also been created. The class **Matrix.java** that contains the function that is responsible for matrix multiplication, and the **MyBenchmark.java** class that contains the code necessary to use the JMH benchmark. Additionally, as in Python, before doing the benchmark, the average of the amount of memory that Java needs to do the multiplication for 5 executions is calculated.

- In C++ we have three files. First we have **matrixMultiplication.cpp** which is responsible for generating the matrices and multiplying them. Then we have **matrixMultiplication.h** which allows us to use the function of the file **matrixMultiplication.cpp** in **matrixBenchmark.cpp**.

Finally, we have the file **matrixBenchmark.cpp** which calculates the execution time for different matrix sizes, as well as memory usage. For C++ the benchmark tool does not only allow you to calculate how long it takes to perform the multiplication, as it also includes the time it takes to calculate the memory usage and the total time to execute the function for each size. This is why **gettimeofday()** is used for the execution time, before and after performing the multiplication.

The programs are executed in a way depending on the language:

- In Python, you first have to make sure you have the necessary libraries installed. Then, being inside the project, to be able to see the results of the benchmark you have to enter the command **pytest –benchmark-only -s**.

- In Java you must first do a **mvn clean install**, then install the jmh plugin if you do not have it installed, and finally go to the **MyBenchmark.java** code and click on the clock symbol that appears to the left of the **testMethod()** function.

- To compile the code in C++, you can use the command **g++ -O2 -o matrixMultiplication matrixMultiplication.cpp matrixBenchmark.cpp matrixMultiplication.h**. Then, to run the executable, use the command **perf stat ./matrixMultiplication**. However, since this command shows the overall execution time rather than the resulting multiplication time for each size, the measurements explained above and below have been used.

4.1 Execution Time

The first set of experiments measured the time (in seconds) taken by each language to multiply matrices of different sizes. The execution time represented is the one obtained with the different benchmarks used in each language. In Python, pytest-benchmark has been used with 5 warm-ups, 5 iterations and 5 rounds. In Java, jmh has been used with 5 warm-ups and 10 measurement iterations. In C++ you can compile and run the code explained above. Furthermore, in the case of C++, to obtain the average execution time, the program has been executed 5 times to obtain the average execution time for 5 different executions.

Size (n x n)	C++ (s)	Java (s)	Python (s)
100 x 100	0.000277	0.001	0.0323086
200 x 200	0.000965	0.005	0.271465
300 x 300	0.0026	0.018	1.1030705
500 x 500	0.00675	0.088	5.5695977
1000 x 1000	0.0196	0.979	46.832

Table 1: Execution time for different languages and sizes

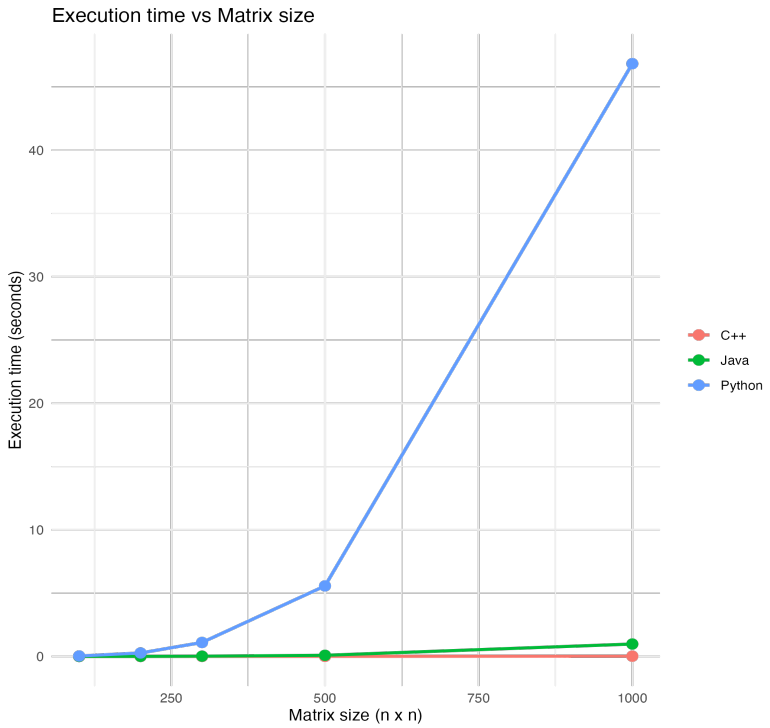

Figure 1: Graph that shows how the execution time evolves depending on the size of the matrices

As we can see, there are notable differences in the execution time in the different languages. We see how C++, being a compiled language, is the fastest. The next would be Java, which is a compiled language and uses the JVM, a virtual machine that introduces some overhead and makes it a little slower than C++. Finally we have Python, this being the slowest of the three because it is an interpreted language.

4.2 Memory Usage

This second experiment tries to show how much memory each language uses when executing the matrix multiplication algorithm. To measure memory in Python and Java, 5 iterations of the algorithm have been carried out and the average memory usage has been calculated. In C++, the average memory used is not necessary because once the array is defined, a fixed memory space is created that is always the same regardless of the execution if the type of data contained in the array or its size does not change. In this case, since both are constants, it does not vary.

Size (n x n)	**C++ (MB)**	**Java (MB)**	**Python (MB)**
100 x 100	0.1147	0.25	0.42
200 x 200	0.7208	1	1.07
300 x 300	1.2	2.18	1.74
500 x 500	3.838	5.98	2.95
1000 x 1000	18	14	4

Table 2: Memory usage for different languages and sizes

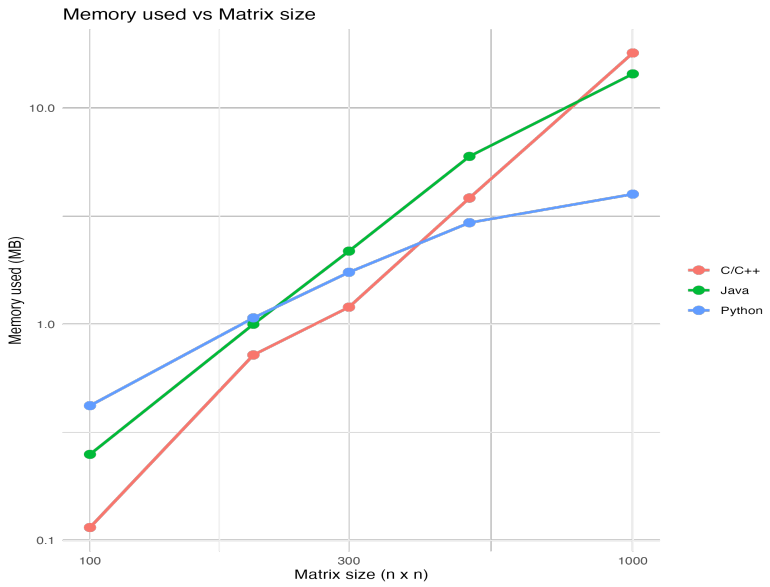

Figure 2: Graph that shows how the memory usage evolves depending on the size of the matrices

Here we see how C++ is the most efficient in most cases thanks to its direct control over memory allocation and lack of additional overhead. Java tends to

use more memory due to JVM overhead and object management, although at large sizes it can be more efficient thanks to JVM optimization. Finally, Python is less efficient at small and medium sizes due to its dynamic typing and internal structures, but at large sizes it improves its memory usage.

5 Conclusions

The experiments revealed a clear performance hierarchy for matrix multiplication: C++ is the fastest, followed by Java, with Python being the slowest. C++'s superior performance stems from its compiled nature, allowing low-level memory management and hardware optimizations, making it ideal for tasks that require efficiency, especially with large matrices.

Java, although slower than C++, performed better than Python due to Just-In-Time (JIT) compilation and multithreading capabilities, though its garbage collection introduces some overhead. Python, being an interpreted language, showed the poorest performance due to its computational overhead, dynamic typing, and reliance on an interpreter. However, Python's performance can be greatly improved by using optimized libraries like NumPy, although this was not the focus here.

In conclusion, C++ is the best option for raw computational tasks, while Java provides a good balance between performance and ease of use. Python is more suited for smaller-scale tasks or when paired with optimized libraries, offering flexibility and simplicity at the cost of speed.

6 Future Works

Future experiments could explore further optimizations to improve performance in matrix multiplication. One area is code optimization, such as utilizing advanced algorithms like Strassen's or blocking techniques, which can reduce computational complexity. Additionally, parallelization can be employed to take advantage of multicore processors, distributing the matrix multiplication workload across multiple CPU cores to speed up execution, particularly for large matrices.

Another promising direction is the use of distributed computing, where matrix operations are split across multiple machines or cloud-based infrastructure. This approach would be highly beneficial for extremely large matrices, where the computational and memory demands exceed the capacity of a single machine. These future optimizations could provide significant performance improvements across all the languages tested.