Comparative Performance Analysis of Matrix Multiplication in C, Python, and Java

Pablo Cabeza Lantigua
University of Las Palmas de Gran Canaria
pabcablan2005@email.com
https://github.com/pabcablan/Language_Benchmark_of_matrix_multiplication

October 23, 2025

Abstract

This study examines how the choice of programming language affects the efficiency of matrix multiplication, a fundamental operation in scientific computing. We compare C, Python, and Java by developing independent programs in each language, using a clear and modular structure to ensure fairness and reproducibility. Each implementation was tested with different matrix sizes, measuring both how fast the calculations were completed and how much computer memory was used. To get reliable results, each test was repeated several times and the averages were analyzed. The results show that C is the fastest and uses the least memory, especially for large matrices, thanks to its low-level handling of resources. Python, although easy to use and code, required more time and memory as the problem size increased. Java's performance was in between, offering a balance between programming simplicity and efficiency. These findings highlight the importance of selecting the right language.

Keywords: matrix multiplication, benchmarking, performance analysis, memory usage, C, Python, Java

1 Introduction

Matrix multiplication is an operation in mathematics, computer science, and engineering. The way matrix multiplication is programmed can affect how quickly and efficiently is.

Although there is a lot of research on making matrix multiplication algorithms faster, not much work has focused on comparing how different programming languages perform this operation in practice. Understanding these differences is important to choose the best language.

The aim of this paper is to evaluate and compare how well matrix multiplication works in three popular programming languages: C, Python, and Java. For each language, I created separate programs that follow good coding practices, making sure to clearly organize the code and allow tests with different matrix sizes. The main contributions are:

- Present a way to benchmark matrix multiplication in C, Python, and Java.
- Compare how fast each language completes the calculations and how much computer memory they use for different matrix sizes.
- Discuss the advantages and disadvantages of each language in terms of implementation difficulty, execution speed, and memory usage.
- Share all code, data, and analysis scripts publicly at: https://github.com/pabcablan/Language_Benchmark_of_matrix_multiplication

This work shows why the choice of programming language matters for high-performance and data-intensive computing, and aims to help others make informed decisions for future projects.

2 Problem Statement

The main problem addressed in this paper is to evaluate and compare how efficiently matrix multiplication can be performed in three different programming languages, C, Python, and Java. Specifically, I want to find out how much time and computer memory each language needs to multiply two square matrices of different sizes.

I hypothesize that lower-level languages like C will be faster and use less memory than higher-level languages such as Python and Java, especially when working with larger matrices. This expectation is based on how these languages handle memory and the amount of extra work done by their runtimes.

To test this, I measure the execution time and memory usage for matrix multiplication in each language, using the same method and experimental setup. The goal is to understand the practical differences in performance and resource usage, and to help others choose the best language for similar tasks.

3 Methodology

To compare how matrix multiplication performs in C, Python, and Java, I designed a simple but thorough benchmarking setup that anyone can reproduce. All programs were written to keep production code separate from the testing and benchmarking code. This clear structure makes it easier to understand, maintain, and repeat the experiments.

Experimental Setup

I ran all tests on my own laptop, which has the following features:

- **Operating System**: Windows 11 Pro 64-bit, Version 26200.6901
- **CPU**: AMD Ryzen 5 6600H with Radeon Graphics
- **RAM**: 16 GB (2x8GB Samsung 4800MHz)
- **Motherboard**: 8A21 19.79
- **System BIOS**: F.26
- **C Compiler**: GCC 6.3.0 with flags `-O3 -march=native`
- **Python Interpreter**: Python 3.13.9 (with virtual environment)
- **Java**: Java 21.0.6 LTS (Java HotSpot 64-Bit Server VM)

All experiments were done using Visual Studio Code's built-in terminal, and I made sure to minimize background processes for consistent results.

Algorithms and Implementation

Matrix multiplication in each language was based on the classic triple loop method for multiplying two randomly generated square matrices. I made sure the code was modular and easy to adjust, so anyone can change the matrix size or number of repetitions for further testing.

Benchmarking and Parameters

For each matrix size, I ran the experiment five times and calculated the average, to reduce the effect of random system fluctuations. Both the matrix size and the number of runs were set as parameters in the code itself, making the experiments easy to repeat.

Use of Artificial Intelligence Tools

Because I was less familiar with C, I relied on artificial intelligence tools like ChatGPT and GitHub Copilot to help write and refine the C code and to set up the benchmarking routines. These tools were especially helpful for finding suitable libraries and programming patterns. I always tried to understand and adapt what was generated, so the code reflected my own learning and decisions.

Execution and Data Management

- **C**: Output directories are configured in the `Makefile` (lines 5-6) via `OUTPUT_DIR_UNIX` and `OUTPUT_DIR_WIN`. Data files containing benchmark results are automatically saved to the specified path.
- **Python**: The output directory for benchmarking and the directory for CSV data plotting are specified as command-line arguments in `basic_matrix_multiplier_benchmarking.py` and `plot_benchmarks.py`, respectively. All CSV files should be kept in a single folder for reliable analysis.
- **Java**: The output directory is specified as a command-line argument to BasicMatrixMultiplierBenchmarking.java.

Usage instructions, parameter configuration, and example commands are fully documented in the project README for reproducibility.

Dependencies and Tools

- **Python**: Benchmarking and analysis used the libraries `psutil`, `pandas`, and `matplotlib`.
- **Java**: Project dependencies (managed with Maven) included:

```
junit:junit:4.13.2 (test scope)
com.github.oshi:oshi-core:6.4.4
```

Data Collection and Analysis

All execution time and memory usage results were saved in CSV files. I checked the outputs to make sure the calculations were correct, and then used Python scripts to analyze and plot the data.

Best Practices and Limitations

To keep everything organized and reproducible, I recommend saving all experiment results in a single folder, using the exact compiler flags and library versions shown above, and following the setup steps in the README. Even though I tried to keep outside factors controlled, results might change slightly on different computers or operating systems.

4 Experiments and Results

To systematically compare the computational performance of matrix multiplication, I conducted benchmarks in C, Python, and Java using square matrices of sizes 64×64, 128×128, 256×256, 512×512, and 1024×1024. Each test was repeated five times, and the average execution time and real memory usage were recorded for each language.

Matrix Size	Python (s)	C (s)	Java (s)
64×64	0.02234	0.00035	0.00063
128×128	0.17305	0.00276	0.00137
256×256	1.38688	0.02352	0.01262
512×512	11.97746	0.19819	0.14803
1024×1024	146.97785	1.68503	1.20958

Table 1: Average execution times (seconds) for matrix multiplication in C, Python, and Java over five runs.

Table 1 summarizes the speed results. C is by far the fastest, with execution times increasing smoothly as matrix size grows. Java maintains low times for most sizes, but Python's execution time increases much more rapidly for bigger matrices. This is likely because C compiles directly to machine code and manages memory very efficiently, while Python is interpreted and relies heavily on its runtime environment, which adds significant overhead—especially as the amount of data grows. Java, being a compiled language with a virtual machine, manages to balance between efficiency and abstraction, which helps keep its performance stable for moderate sizes, though it still cannot match C for very large computations.

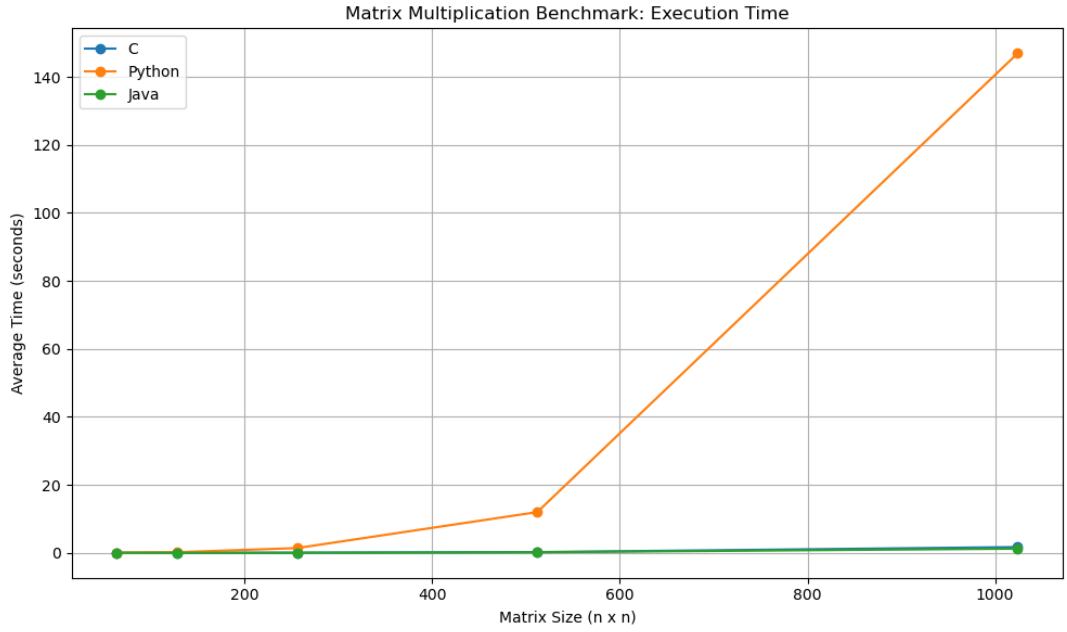

Figure 1: Average execution time for matrix multiplication as a function of matrix size ($n \times n$) for C, Python, and Java.

As shown in Figure 1, the gap between languages becomes much more noticeable as matrix size increases. Python's curve rises steeply, reflecting the impact of its interpreter and less optimized memory handling. Java's curve is flatter, possibly due to its compilation and effective memory management, but C remains the reference for speed, scaling even for the largest matrices.

Matrix Size	Python (MB)	C (MB)	Java (MB)
64×64	19.46	4.86	68.22
128×128	21.85	5.30	62.44
256×256	31.32	6.47	69.87
512×512	57.37	11.13	76.21
1024×1024	157.80	29.48	107.19

Table 2: Average real memory usage (MB) for matrix multiplication in C, Python, and Java over five runs.

Table 2 presents the memory usage results. C uses the least memory in all cases, with usage growing slowly as matrix size increases. Python and Java use much more memory, and for Java, the memory stays high even for smaller matrices. This likely happens because Java's virtual machine pre-allocates resources for garbage collection and object management, while Python's dynamic typing and memory model also add overhead. C, on the other hand, manages memory manually and only allocates what is strictly necessary.

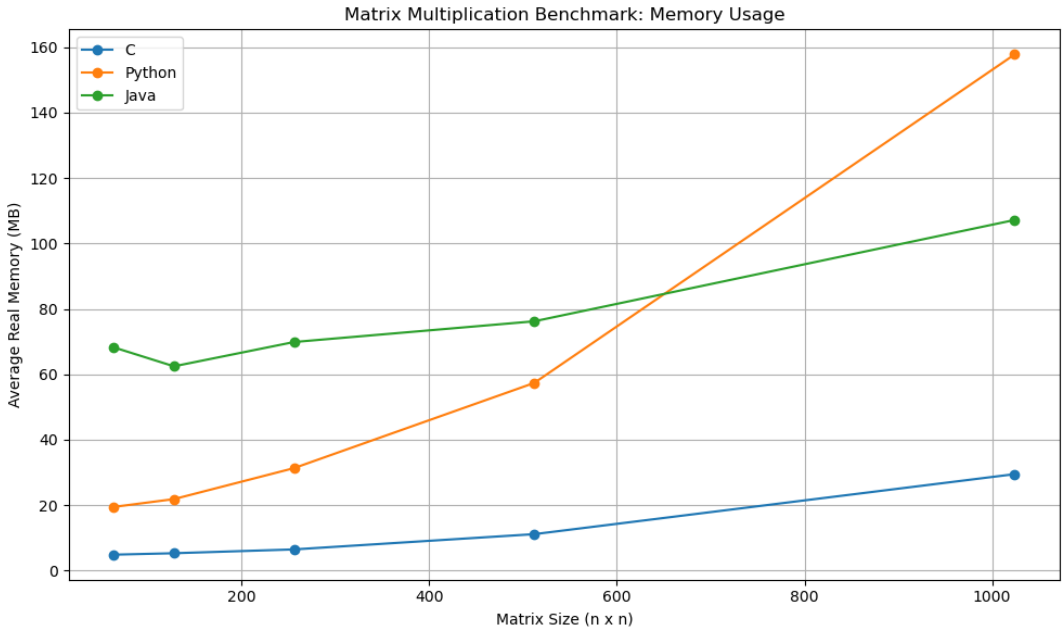

Figure 2: Average real memory usage during matrix multiplication for C, Python, and Java.

Figure 2 confirms these trends visually. The memory required by C grows moderately, while Python and Java both show higher usage, with Java's line consistently above the others. These results are in line with previous studies: lower-level languages like C are superior for heavy computation and resource management, while higher-level languages trade off some efficiency for ease of use.

In summary, C offers both speed and memory efficiency, Python is easy to use but struggles with large-scale performance, and Java provides a middle ground—efficient for moderate workloads, but with some extra memory cost. The choice of language for scientific computing tasks should consider both these performance factors and the practical needs of the application.

5 Conclusions

In conclusion, this study shows that the programming language chosen for matrix multiplication can greatly affect both speed and memory usage, especially as the problem gets larger. The results confirm that C is the best option for performance and efficiency, while Python, although very easy to use, is much slower and uses more memory when dealing with big calculations. Java sits in the middle, offering a good balance between ease of programming and reasonable performance.

This work is original in that it provides a practical and reproducible way to compare how different languages handle the same computational task. By organizing the code carefully, running fair and controlled experiments, and making all materials openly available, this study gives a useful reference for future research and helps others understand the strengths and weaknesses of each language in scientific computing. The findings highlight how important it is to choose the right tool for the job, especially in fields where performance really matters.

6 Future Work

In the future, this benchmarking approach could be applied to other types of problems, such as sorting, working with graphs, or calculations involving decimals, to see if the same patterns hold for different tasks. It would also be interesting to include more programming languages, like C++, Rust, or Julia, to compare how they perform in scientific computing.

One limitation of this study is that all the tests were done on a single computer and operating system. To make the results more general, it would be useful to repeat the experiments on different platforms, such as Linux or MacOS. Also, the current experiments used a basic way of multiplying matrices; future work could test more advanced methods, like optimized libraries or even using GPUs, to see how much faster things can get.

Finally, trying out bigger matrices and doing a deeper statistical analysis could help better understand how each language handles really large amounts of data and where performance issues might appear.