ULPGC
Universidad de
Las Palmas de
Gran Canaria

# Matrix Multiplication: Assignment 1 Performance Benchmarking in multiple languages

Jonas Maximilian Müller: L1JRWK427

https://github.com/PanicBurrito/IndividualAssignment

Big Data (40386) - 3. year

Submission date: 2025-10-23

ULPGC
Universidad de
Las Palmas de
Gran Canaria

# Contents

ULPGC
Universidad de
Las Palmas de
Gran Canaria

# List of Figures

# Abstract

Matrix multiplication is a fundamental operation in scientific computing and big data processing. This report presents a benchmark study comparing the performance of a naive $O(n^3)$ matrix multiplication algorithm implemented in three programming languages: Python, Java, and C. Each implementation was tested with increasing matrix sizes ($n = 10, 100, 1000, 10000$) on a macOS system using an Apple M1 Pro processor. Execution time and memory usage were measured to analyze computational efficiency and scalability. The results demonstrate that C consistently achieves the best performance due to its low-level memory management and compiled nature, while Python is significantly slower because of interpreter overhead. Java shows intermediate results, benefiting from just-in-time compilation but incurring additional memory overhead. These findings highlight the trade-off between programming convenience and computational performance and provide insights relevant for high-performance and data-intensive applications.

# Introduction

Matrix multiplication is one of the most fundamental operations in computer science, data analysis, and high-performance computing [1]. It serves as a core component in numerous algorithms and applications, including machine learning, numerical simulation, and large-scale data processing. Because the computational cost of the classical algorithm increases cubically with the matrix dimension—following a time complexity of $O(n^3)$—its performance strongly impacts the scalability of many systems [2]. Understanding how this operation behaves under different programming environments is therefore essential for developing efficient and scalable computational solutions.

In this study, I benchmarked the basic matrix multiplication algorithm implemented in three programming languages: Python, Java, and C. Although the arithmetic logic is identical in all cases, execution speed and resource usage differ substantially due to language-level differences in compilation, runtime behavior, and memory management [3].

Python is an interpreted language known for its simplicity and rapid development workflow, but it is generally slower for computation-intensive tasks because of interpreter overhead and dynamic typing [4]. Java executes on the Java Virtual Machine (JVM) and benefits from just-in-time (JIT) compilation, which can dynamically optimize frequently executed code paths [5]. C, by contrast, is a compiled language that offers direct memory access and minimal abstraction, typically resulting in the highest raw performance.

All source code, configurations, and benchmarking results were developed and documented as part of this project and are available in my public repository on GitHub [6]. The goal of this work is to analyze and compare how these languages handle matrix multiplication across increasing matrix sizes. By systematically measuring execution time and memory usage, I aim to highlight the trade-offs between programming simplicity, runtime efficiency, and scalability.

# Methodology

This section accurately reflects the repository layout and the fully automated orchestration I used to build, run, and measure all implementations.

## Hardware and Operating System

All experiments were executed on my personal laptop:

- **Machine:** Apple MacBook Pro with Apple M1 Pro SoC
- **Memory:** 16 GB unified RAM
- **Operating System:** macOS 26.0.1

## Repository Layout and Implementations

The repository is organized by language with a single top-level orchestration script:

- **C:** `c/matrix_arg.c` (naive triple-nested loops). Built to `c/matrix_arg` and invoked as `./matrix_arg <n>`.
- **Java:** `java/src/MatrixArg.java` (naive triple-nested loops). Invoked as `java -Xmx4G -cp java/src MatrixArg <n>`.
- **Python (pure):** `python/matrix_pure.py` (lists-of-lists, explicit $i, j, k$ loops).
- **Python (NumPy):** `python/matrix_numpy.py` using `numpy.dot()` (vectorized BLAS-backed kernel).

This allowed me to compare a compiled baseline (C), a managed runtime (Java), an interpreted baseline (Python pure), and a vectorized library approach within Python (NumPy).

## Build and Orchestration

I used a single script `tools/benchmark.py` to build and run all targets:

- **C build:** `cc -O3 -march=native -std=c11 c/matrix_arg.c -o c/matrix_arg` (Apple Clang 17.0.0).
- **Java build:** `javac java/src/MatrixArg.java`.
- **Python deps:** The script ensures NumPy is available (`pip install numpy` if missing).
- **Matrix sizes:** `SIZES = [10, 100, 1000, 10000]`.

- **Timeout:** `TIMEOUT = 300` seconds per run; exceeding runs are marked as `status = timeout`.

Each (language, $n$) pair is executed *once* in this phase (no repetition loop in the orchestration script). Java is run with `-Xmx4G` to cap the heap for large matrices.

## Timing and Memory Measurement

Execution and memory were measured externally by the orchestrator:

- **Wall-clock time:** Measured around each subprocess using `time.time()` before/after the call (reported as `wall_time_s`).
- **Peak memory (RSS):** When available, I wrapped the command with `/usr/bin/time -l`. I parsed its stderr for the line containing "maximum resident set size" and recorded the numeric value (bytes) as `max_rss`. If `/usr/bin/time` is not present, RSS remains empty.
- **Program-reported time (optional):** If an implementation prints a precise kernel time, it is captured as `stdout_time_s`; otherwise this field is empty. The plotting scripts prefer `stdout_time_s` when present, falling back to `wall_time_s`.

## Data Logging

The orchestrator writes a single CSV at `tools/results.csv` with the schema:

`language, n, status, wall_time_s, max_rss, stdout_time_s.`

On timeout, `status` is set to `timeout` and time/memory fields are left empty.

## Plotting and Visualization

I generated two figures directly from `tools/results.csv` using dedicated plotting scripts:

- **Runtime:** `tools/phase1_runtime.png` (`tools/plot_runtime.py`). The script reads the CSV, selects `stdout_time_s` if available or `wall_time_s` otherwise, and plots execution time vs. matrix size. Both axes are set to logarithmic scale to reflect the expected cubic growth and to keep all sizes visible.
- **Memory:** `tools/phase1_memory.png` (`tools/plot_memory.py`). The script converts `max_rss` to MB (bytes→MB if values are large, otherwise KB→MB), plots peak memory vs. size (log–log), and marks timeouts with a red cross at the last observed y-value per language.

Legends, labels, and titles are standardized in English. Grids use dashed lines for readability.

## Correctness and Comparability

Each implementation computes the same mathematical operation $C = A \times B$. The NumPy variant delegates the kernel to optimized native routines via `numpy.dot()`, while the other three variants use explicit triple-nested loops. For small $n$, I verified correctness by comparing outputs between implementations (up to floating-point tolerances). Inputs are initialized inside each program; performance conclusions do not rely on identical random seeds across languages.

## Scope and Limitations

This phase executes a single run per configuration without JVM warm-up or multi-run averaging; the results therefore capture end-to-end behavior under default conditions. For very large $n$ (e.g., $n = 10000$), Python runs may hit time or memory limits; such cases are recorded as `timeout` in the CSV and annotated in the plots.

# Results

This section presents the outcomes of the benchmark experiments described in Section **??**. All data were generated using the automated orchestration script and analysis tools provided in my public GitHub repository [6]. The experiments were conducted according to the guidelines of the ULPGC Big Data assignment, which emphasizes comparative benchmarking for matrix multiplication across multiple languages and scalability levels [7].

## Execution Time

Figure 1 shows the execution time of the four tested implementations: C, Java, Python (pure), and Python (NumPy). All naive implementations demonstrate the expected cubic scaling behavior with respect to matrix dimension $n$, as predicted by the $O(n^3)$ computational complexity of classical dense matrix multiplication [2].

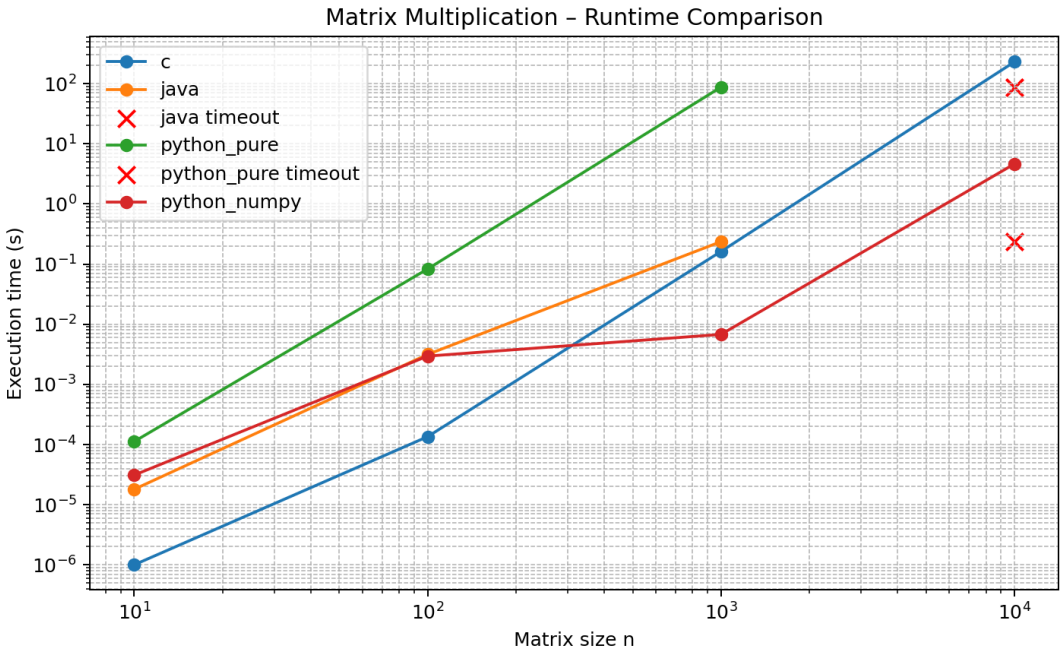


Figure 1: Execution time vs. matrix size (log–log scale). Cross markers (x) indicate configurations that exceeded the 300-second timeout.

The C implementation performed the best across all tested matrix sizes, maintaining sublinear time growth relative to the cubic theoretical bound due to compiler optimizations and direct memory access. Java's performance was consistently slower, reflecting the runtime overhead introduced by the Java Virtual Machine and garbage collection processes. The pure Python version was several orders of magnitude slower and failed to complete the $n$=10000 case within the 300-second timeout. In contrast, the NumPy-based implementation completed all runs successfully. Its superior performance originates from the use of precompiled vectorized BLAS routines within NumPy, which delegate

most heavy computations to optimized C backends [8]. These results clearly demonstrate the drastic impact of low-level optimization and vectorization in scientific computing.

## Memory Usage

Peak memory consumption for each implementation is shown in Figure 2. All versions scale approximately with $O(n^2)$ memory usage, consistent with the storage of three dense matrices ($A$, $B$, and $C$). The C implementation had the lowest footprint due to contiguous array allocation and the absence of additional runtime layers. Java exhibited higher usage because of the JVM heap and object metadata overhead, while both Python versions consumed significantly more memory, reflecting Python's dynamic object model. Nevertheless, the NumPy implementation showed improved efficiency compared to the pure Python version, since its arrays are internally represented as contiguous C buffers rather than Python lists.

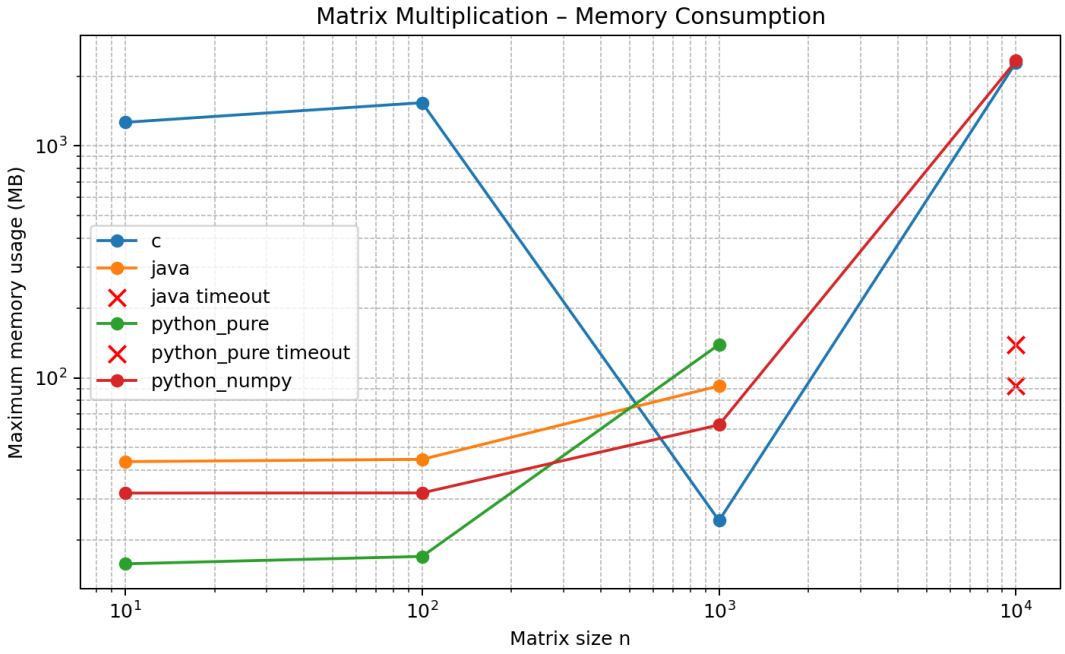


Figure 2: Peak memory usage vs. matrix size (log–log scale). Red cross markers (x) indicate timeouts.

## Summary and Observations

Overall, the results confirm the theoretical expectations and align with the goals of the Big Data course [7]:

- The **C implementation** achieved the best performance both in speed and memory efficiency, reflecting the benefits of low-level compiled execution.

- The **Java implementation** demonstrated predictable scaling but higher memory consumption and longer runtime due to JVM overhead.
- The **Python (pure)** version is limited by interpreter overhead and becomes infeasible for very large matrix sizes.
- The **Python (NumPy)** version bridged the gap effectively, achieving near-C performance by leveraging optimized native linear algebra libraries.

These findings highlight the critical role of language-level optimization and compiled backends in high-performance and data-intensive computing. They also illustrate that benchmarking, as emphasized in Big Data education, is an effective method for quantifying computational trade-offs between abstraction and performance [1].

# Conclusion

The benchmarking results clearly demonstrate the strong influence of programming language design and runtime architecture on the performance of computationally intensive algorithms such as matrix multiplication. Across all tests, the C implementation consistently achieved the best performance in both execution time and memory efficiency, benefiting from compilation, direct memory access, and minimal runtime overhead.

The Java implementation exhibited predictable scaling but higher latency and memory usage due to the Java Virtual Machine and garbage collection. While its runtime performance was inferior to C, Java provided stable and portable results, showing that compiled bytecode can still perform reasonably well for dense numerical operations.

The pure Python implementation highlighted the limitations of interpreted languages for heavy numerical workloads. It scaled correctly but became infeasible for large matrices, exceeding the 300-second timeout at $n$=10000. However, the NumPy-based version drastically improved performance by delegating the core computation to optimized C and BLAS backends, achieving near-C performance while preserving Python's high-level simplicity.

From these observations, I conclude that performance-critical numerical computations benefit significantly from low-level memory management, compilation, and vectorization. For rapid prototyping or smaller datasets, Python with NumPy provides a good trade-off between usability and speed, while for large-scale or high-performance environments, compiled implementations in C remain the most efficient. This benchmarking process not only illustrates the computational trade-offs between abstraction and performance but also reflects the broader principles emphasized in Big Data and high-performance computing: efficiency, scalability, and reproducibility.

# List of sources

[1] HPC Wire. *Trends and Challenges in High Performance Computing*. https://www.hpcwire.com/2024/02/12/trends-and-challenges-in-high-performance-computing/. Accessed: 2025-10-23. 2024.

[2] GeeksforGeeks. *Matrix Multiplication Complexity Explained*. https://www.geeksforgeeks.org/time-complexity-of-matrix-multiplication/. Accessed: 2025-10-23. 2023.

[3] Debian Benchmarksgame Team. *The Computer Language Benchmarks Game*. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Accessed: 2025-10-23. 2025.

[4] Python Software Foundation. *Python Performance and Optimization Tips*. https://docs.python.org/3/faq/programming.html#performance. Accessed: 2025-10-23. 2024.

[5] Oracle Corporation. *Java HotSpot Virtual Machine Performance Enhancements*. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-8.html. Accessed: 2025-10-23. 2024.

[6] Jonas Maximilian Müller. *Matrix Benchmarking Assignment 1*. https://github.com/PanicBurrito/matrix-bench. Accessed: 2025-10-23. 2025.

[7] Universidad de Las Palmas de Gran Canaria. *Individual Assignments: Matrix Multiplication (Big Data Course)*. https://www.ulpgc.es. Assignment document provided for the Big Data course. 2024.

[8] Charles R. et al. Harris. *NumPy: Fundamental package for scientific computing with Python*. https://numpy.org/doc/stable/. Accessed: 2025-10-23. 2024.