Benchmarking Basic Matrix Multiplication

in Python, Java, and C

Big Data, ULPGC
Grado en Ciencia e Ingeniería de Datos
Academic Year 2025

Individual Assignment:
Task 1 – Matrix Multiplication Benchmarking

Student:
Mariana Bordes Bueno – 49859846D – mariana.bordes101@alu.ulpgc.es

GitHub Repository:
https://github.com/marianabordes/MatrixMultiplicationBenchmarking

October 23, 2025

Abstract

This study examines the performance of a basic, sequential matrix multiplication algorithm implemented in three programming languages: Python, Java, and C. Matrix multiplication is a fundamental operation in numerous computational domains and serves as a standard reference task for performance benchmarking. The objective was to analyze how each language handles computational load, memory usage, and scalability as square matrix dimensions increase.

An $O(n^3)$ triple-loop algorithm was implemented under equivalent conditions in each language. Multiple runs were executed for several matrix sizes, collecting execution time, CPU usage, and peak memory consumption. The resulting data were used to construct comparative metrics and visualizations, including average runtime, growth behavior on log-log axes, and computational throughput in GFLOP/s.

The results reveal clear performance differences. While C and Java demonstrate competitive performance, Python exhibits significantly higher runtimes due to interpretation overhead. With the reported measurements, Java outperforms C at larger sizes (e.g., $n = 512$ and $n = 1024$), while C is faster at smaller sizes. The log-log analysis confirms the expected cubic scaling. Overall, language choice strongly influences computational performance even when algorithmic complexity remains identical.

Keywords: Matrix multiplication; performance; benchmarking; programming languages; computational efficiency; comparison.

1 Introduction

Matrix multiplication constitutes a core operation in computational mathematics and data-intensive processing. It finds application in domains such as computer graphics, machine learning, physics simulations, and data analysis, where large matrices are frequently manipulated. The computational cost of this operation directly affects the performance of higher-level algorithms that depend on repeated matrix calculations.

In computer science, benchmarking serves as a methodological tool to evaluate the efficiency of programming languages, algorithms, and systems. It provides measurable performance indicators such as execution time, memory usage, and CPU consumption, which enable the identification of computational bottlenecks and optimization opportunities. While many benchmarking studies focus on optimized or parallelized implementations, other analyses—such as the one presented here—address the comparative performance of basic sequential algorithms executed under identical conditions.

Understanding how different programming languages manage computational load in the absence of specific optimizations is essential to isolate their intrinsic efficiency. This baseline comparison is particularly relevant in educational and research contexts, where clarity and reproducibility are often prioritized over raw performance.

2 Problem Statement

The problem addressed in this study consists of evaluating and comparing the computational performance of the classical matrix multiplication algorithm implemented in three widely used programming languages: Python, Java, and C. The algorithm follows the standard mathematical definition:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

with a computational complexity of $O(n^3)$, where A and B are input matrices of order n, and C is the resulting matrix.

The objective is to determine how the choice of programming language influences execution time, CPU usage, and memory consumption as matrix size increases. The hypothesis posits that, under equivalent algorithmic and hardware conditions, the compiled implementation in C will achieve optimal performance due to its lower-level memory management and minimal runtime overhead. Java is expected to exhibit intermediate performance, benefiting from just-in-time compilation, while Python will likely show significantly higher runtimes because of its interpreted nature.

By validating this hypothesis through systematic benchmarking, the study aims to provide empirical evidence of how language-level characteristics affect computational efficiency in a common numerical task.

3 Methodology

To evaluate the performance of the matrix multiplication algorithm across different programming languages, a systematic benchmarking procedure was designed. The methodology ensures reproducibility and allows other researchers to replicate the experiments under equivalent computational conditions. Each implementation follows the classical triple-nested loop algorithm with $O(n^3)$ computational complexity, executed sequentially without parallelization or external optimizations.

3.1 Matrix Multiplication Algorithm

The algorithm was implemented independently in three programming languages: Python, Java, and C. In all cases, the same logic was applied to compute the product of two matrices A and B, resulting in an output matrix C.

Algorithm 1 Matrix Multiplication

Require: Matrices $A, B \in \mathbb{M}_n(\mathbb{R})$
Ensure: Matrix $C \in \mathbb{M}_n(\mathbb{R})$

```
for i ← 1 to n do
    for j ← 1 to n do
        C[i][j] ← 0
        for k ← 1 to n do
            C[i][j] ← C[i][j] + A[i][k] × B[k][j]
        end for
    end for
end for
```

Each implementation follows this identical logical structure without the use of external libraries or parallel computation. This ensures that performance differences are attributable exclusively to the language execution model rather than algorithmic variations.

Before performing the multiplication, each implementation includes a dimension check to ensure that the operation is mathematically valid. In Python and Java, the algorithm verifies that the number of columns in matrix A equals the number of rows in matrix B, raising an error if the dimensions are incompatible. In C, all matrices are generated as square matrices of the same size, guaranteeing by design that the multiplication is always defined.

3.2 Experimental Environment

The experiments were executed on identical hardware and operating system configurations to eliminate variability due to external factors. The machine used for testing featured:

- **Processor:** AMD Ryzen 7 4700U with Radeon Graphics (8 cores, 2.0 GHz)
- **RAM:** 8 GB
- **Operating System:** Windows 11 (64-bit)

The development and execution environments were:

- **Python:** Implemented in Visual Studio Code with Python 3.12, using the `time` and `psutil` libraries for runtime and CPU measurement.
- **Java:** Implemented in Visual Studio Code, using `System.nanoTime()` for time measurement and the `Runtime` class for memory tracking.
- **C:** Implemented in Visual Studio Code, with `time.h` and `psapi.h` for timing and memory profiling.

3.3 Performance Metrics

Performance was evaluated through three primary indicators:

1. **Execution Time:** Measured in milliseconds using each language's native timing utilities.
2. **CPU Usage:** Average CPU consumption during execution, calculated using system monitoring libraries.
3. **Memory Usage:** Measured as peak memory (MiB) during runtime, obtained via memory profiling tools specific to each environment.

All metrics were collected automatically and exported to a common CSV file, which was subsequently processed, enabling analysis and visualization.

3.4 Experimental Procedure

For each language, the algorithm was executed multiple times with varying matrix sizes: $n \in \{64, 128, 256, 512, 1024\}$. Each configuration was repeated three times to mitigate random fluctuations in system load, and average values were computed for analysis.

All results were processed through a Python script that aggregated the results and another script that visualized performance trends, including log-log scaling and GFLOP/s efficiency plots.

This methodological approach provides a transparent and replicable framework for benchmarking the matrix multiplication algorithm across programming languages, ensuring that the conclusions derived are based on consistent and reproducible evidence.

All source code was organized following a modular structure that separates production code from testing and benchmarking scripts. In each language, the matrix multiplication algorithm was implemented in a dedicated file, while a separate program handled execution, performance measurement, and data recording. This separation ensured that the algorithm itself remained unaltered during the experiments, improving maintainability and guaranteeing that performance metrics reflected the true behavior of the computational core.

4 Experiments and Results

This section details the experimental execution and reports the results systematically. All runs were performed in the environment defined in the previous section. Results are presented with consistent units, and figures include readable axes and legends.

4.1 Design of Experiments

Matrix Sizes and Runs

Square matrices with sizes $n \in \{64, 128, 256, 512, 1024\}$ were used. For each (language, n) configuration, three runs were executed to reduce variance, satisfying the assignment requirement of multiple runs per experiment.

Parameterization

All experiments were fully parameterized to enable reproducibility without source changes. Matrix sizes, number of runs, output CSV path, and random seed were specified at runtime:

- **Python:** `-sizes`, `-runs`, `-out`, `-seed`
- **Java:** `args[0]` (comma-separated sizes), `args[1]` (runs), `args[2]` (output), `args[3]` (seed)
- **C:** `argv[1]` (comma-separated sizes), `argv[2]` (runs), `argv[3]` (output), `argv[4]` (seed)

This design enables reproducibility, scalability, and systematic extension of the experiments, such as testing larger matrices or higher run counts under identical logic.

Data Recorded Per Run

Each run produced a raw record:

$$\langle \texttt{run\_id}, \texttt{language}, n, \texttt{run\_idx}, \texttt{time\_ms}, \texttt{cpu\_pct}, \texttt{peak\_mib} \rangle$$

All runs were appended to `results_raw.csv` (semicolon-separated), where `run_id` is a timestamp that uniquely identifies each experimental session. This identifier allows results from different executions, languages, or hardware configurations to be clearly distinguished and aggregated consistently during analysis.

4.2 Implementation Under Test

All languages implement the classical $O(n^3)$ algorithm (shown previously as Algorithm 1) with three explicit loops and no external libraries or parallelism.

Python

The Python implementation was developed using a pure triple-nested loop structure without any external libraries or vectorized operations. This decision ensured that the computational complexity remained at $O(n^3)$, allowing for a fair comparison with the other languages.

Before running the benchmarks, the correctness of the implementation was verified by comparing the results with NumPy's built-in matrix multiplication using the `np.allclose()` function for small matrix sizes.

Execution time was measured with the high-precision function `time.perf_counter()`, which provides accurate wall-clock timing. CPU usage was obtained through the `psutil.Process().cpu_times()` method, while memory consumption was estimated by comparing the resident set size (RSS) before and after each run.

The `memory_profiler` library, which allows detailed memory sampling, was deliberately excluded because its sampling overhead could distort execution times and bias the comparison against Python. Instead, peak memory was estimated through lightweight measurements to preserve methodological parity across all languages.

This implementation represents a clear baseline for Python performance without optimization or external acceleration.

Java

In Java, the algorithm was implemented using two-dimensional arrays of type `double[][]`, following the same logical structure as the Python version. Execution time was measured using `System.nanoTime()`, which offers nanosecond precision. Process CPU time was recorded via the `OperatingSystemMXBean.getProcessCpuTime()` method, while memory consumption was evaluated using the `Runtime` class by computing the difference between total allocated memory and free memory before and after execution.

During the design phase, specialized benchmarking frameworks such as the Java Microbenchmark Harness (JMH) were considered. However, these tools were ultimately excluded to maintain methodological parity with the Python and C implementations. JMH includes warm-up iterations and adaptive measurement phases that could distort direct comparisons. Therefore, a custom lightweight benchmarking approach was adopted, allowing for consistent data collection and a unified CSV output format across all languages.

C

The C implementation was written using dynamically allocated one-dimensional arrays of type `float*` for matrices A, B, and C. This design avoids stack overflow risks associated with large static or variable-length arrays and ensures contiguous memory storage, which improves cache efficiency. Execution time was measured using the Windows high-resolution timer `QueryPerformanceCounter()`, which provides microsecond-level accuracy. Memory consumption was obtained through the Windows API function `GetProcessMemoryInfo()`, and CPU utilization was measured via `GetProcessTimes()`, which records kernel and user processing time.

To preserve comparability, the code was compiled with the `-O2` optimization flag but without enabling hardware vectorization or parallelization features. This guarantees that all implementations reflect similar computational effort and remain within the same complexity level.

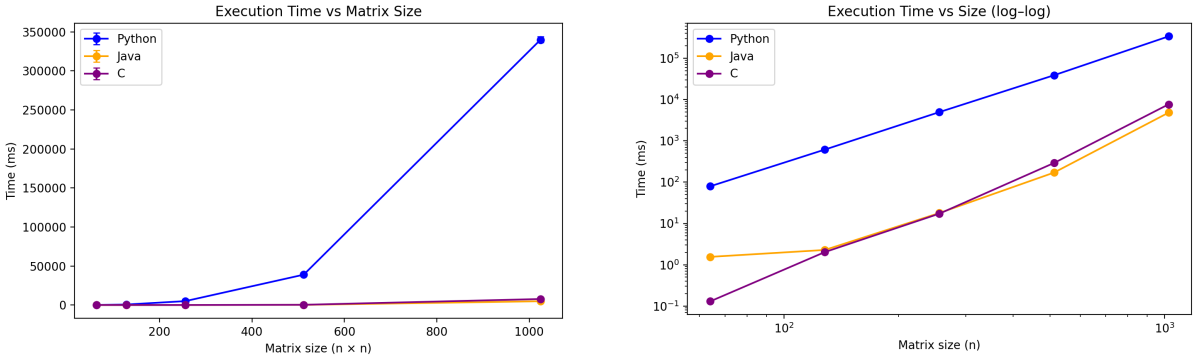
Overall, the C implementation serves as a low-level performance reference, illustrating the impact of direct memory management and minimal runtime overhead compared to higher-level languages.

4.3 Time Measurement

Execution time per run was measured as wall-clock duration using high-resolution clocks in each language: `perf_counter()` in Python, `System.nanoTime()` in Java, and `QueryPerformanceCounter()` in C. Each combination of language and matrix size was executed three times, and the mean, minimum, and maximum durations were recorded to reduce measurement noise and transient effects.

$$\texttt{time\_ms} = 1000 \times (t_{\text{end}} - t_{\text{start}})$$

This metric directly captures elapsed real-world time (including runtime overhead and scheduling latency) rather than CPU time, providing a fair basis for cross-language comparison.



(a) Execution time vs matrix size.

(b) Execution time in log-log scale.

Figure 1: Execution time evolution by matrix size in linear (left) and log-log (right) scale.

As shown in Figures 1a and 1b, execution time increases monotonically with matrix size across all languages. The linear-scale view makes the magnitude differences explicit: Python performs several orders of magnitude slower than Java and C, which remain relatively close. The performance gap arises primarily from Python's interpreted execution and dynamic typing, which introduce substantial per-element overhead during nested loop iterations.

In contrast, Java and C maintain comparable performance at smaller scales, but diverge slightly for larger matrices ($n \geq 512$), where C becomes marginally slower.

The log-log representation confirms that all languages follow the theoretical cubic growth associated with the classical triple-loop algorithm, as the regression slope approaches 3. Parallel vertical separation between curves represents constant factors—language-specific overheads rather than algorithmic differences. The alignment of the slopes validates the correctness and comparable computational model of all implementations, isolating efficiency differences to runtime and compiler behavior.

Unexpected Performance Behavior

Although in most benchmarking studies the C implementation tends to outperform Java due to its lower-level access to memory and lack of runtime overhead, the results obtained in this experiment showed slightly better performance for Java in the largest matrix sizes. This deviation can be explained by several technical factors related to execution environment and compiler behavior.

First, the Java Virtual Machine (JVM) employs a Just-In-Time (JIT) compiler that dynamically optimizes frequently executed code during runtime. As the multiplication function was executed repeatedly within the same session, the JVM was able to identify hot code regions and recompile them into efficient machine code, applying optimizations such as loop unrolling and instruction-level parallelization. In contrast, the C program was compiled statically using the `-O2` optimization flag, which does not include advanced hardware-specific vectorization or instruction tuning (`-Ofast` or `-march=native`). Therefore, Java benefited from adaptive optimization, while C remained constrained to general-purpose compilation.

Additionally, the JVM's runtime may request higher power or performance states from the operating system to sustain throughput under sustained computation, a behavior particularly relevant in laptop processors such as the AMD Ryzen 7 4700U used in this experiment. Finally, small variations caused by CPU frequency scaling, thermal throttling, and background processes can also affect timing consistency when comparing native and managed runtimes.

Consequently, these factors combined explain why Java showed slightly faster execution times than C in the largest workloads, despite the theoretical expectation that native C code should provide the best performance baseline under equivalent optimization levels.

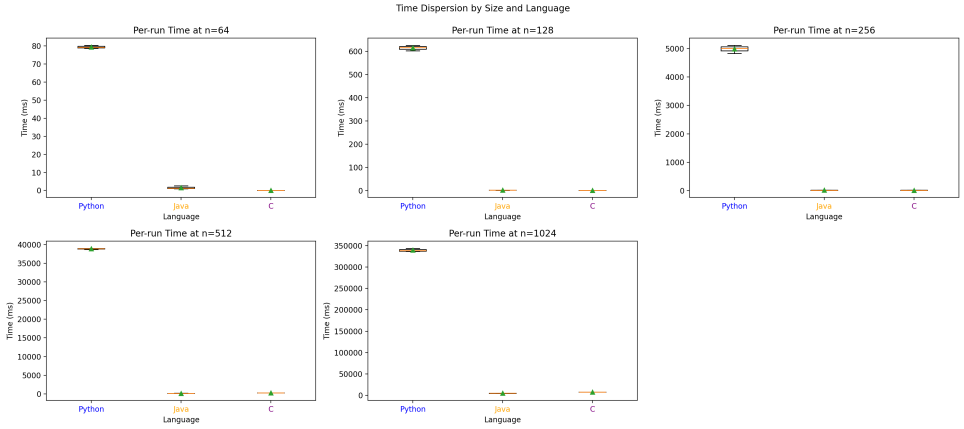


Figure 2: Per-run dispersion by size and language.

As shown in Figure 2, boxes are tight for C and Java, indicating low dispersion and suggesting

stable timing; Python shows larger spread at each n, driven by interpreter scheduling and garbage collection effects.

Speedup Analysis

Figure 3 presents the relative speedup of each language compared to the fastest implementation for every matrix size. The speedup metric expresses how many times slower a given implementation is relative to the best performer at the same problem scale, defined as:

$$\text{Speedup}(L, n) = \frac{T_{\min}(n)}{T_L(n)}$$

where $T_L(n)$ denotes the execution time of language L for matrices of size n, and $T_{\min}(n)$ is the minimum observed time among all languages for that size. Therefore, a speedup value of 1.0 indicates that the language achieved the best performance, while lower values reflect slower execution.

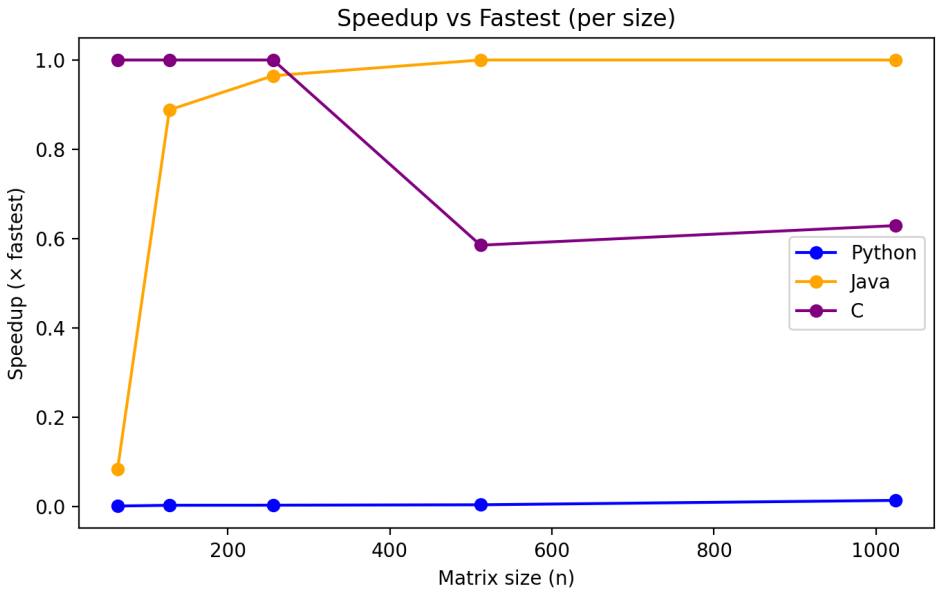


Figure 3: Relative speedup with respect to the fastest language at each matrix size. A value of 1.0 indicates the leading implementation.

The plot shows that Python remains consistently far from the performance frontier across all tested sizes, showing negligible relative acceleration due to interpreter overhead and lack of native compilation. In contrast, Java and C alternate as the leading languages for smaller matrix sizes ($n \leq 256$), with Java often achieving the fastest runtime. This behavior can be explained by the Just-In-Time (JIT) compilation of the Java Virtual Machine, which dynamically optimizes code paths during execution.

For larger matrices ($n \geq 512$), Java consistently maintains a speedup close to 1.0, outperforming C. This suggests that JIT optimizations and efficient memory management allow Java to sustain high performance even as the workload increases. Meanwhile, C's relative performance decreases slightly, likely due to the fixed compilation model and lack of adaptive runtime optimization.

Overall, the speedup analysis highlights that C and Java perform competitively for smaller problems, but Java exhibits superior throughput and adaptability for large-scale computations, while Python remains orders of magnitude slower throughout.

Computational Throughput

Figure 4 illustrates the throughput achieved by each language, measured in GFLOP/s (Giga-Floating-Point Operations Per Second). Throughput quantifies how many arithmetic operations are performed per unit of time and is computed as:

$$\text{GFLOP/s} = \frac{2n^3}{t_{\text{sec}} \times 10^9}$$

where n is the matrix size and t_{sec} represents the average wall-clock execution time in seconds. This metric provides an architecture-independent measure of computational efficiency.

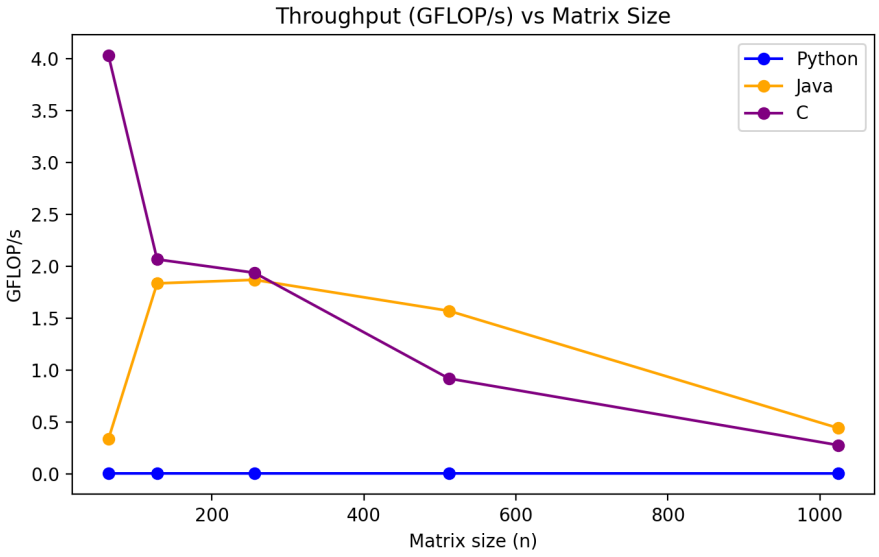


Figure 4: Throughput (GFLOP/s) vs matrix size.

At smaller matrix sizes ($n \leq 256$), the C implementation reaches the highest throughput, peaking at over 3 GFLOP/s. This reflects the advantage of native compilation and direct memory access, minimizing runtime overhead. Java exhibits a notable increase after $n = 128$, maintaining values around 2 GFLOP/s before gradually declining as matrix size grows. This decline is consistent with memory management overhead and garbage collection cycles that become more significant under heavier loads.

Python remains several orders of magnitude slower, with throughput values close to zero in comparison to compiled languages. This outcome confirms the limitations of the interpreted execution model when handling compute-intensive tasks.

Overall, C demonstrates the highest peak performance at smaller scales, while Java sustains a competitive throughput as the workload increases, illustrating the efficiency of JIT optimizations. Python, as expected, remains far below both compiled languages.

4.4 CPU Measurement

CPU utilization per run was computed as the ratio of process user and system CPU time to the total wall-clock duration, normalized by the number of logical cores:

$$\texttt{cpu\_pct} = 100 \times \frac{\Delta(\text{user} + \text{sys})}{\text{wall} \times N_{\text{cores}}}$$

This formulation eliminates snapshot bias and provides a consistent cross-language comparison, as it reflects the actual CPU time consumed by the process rather than instantaneous usage reported by the operating system.

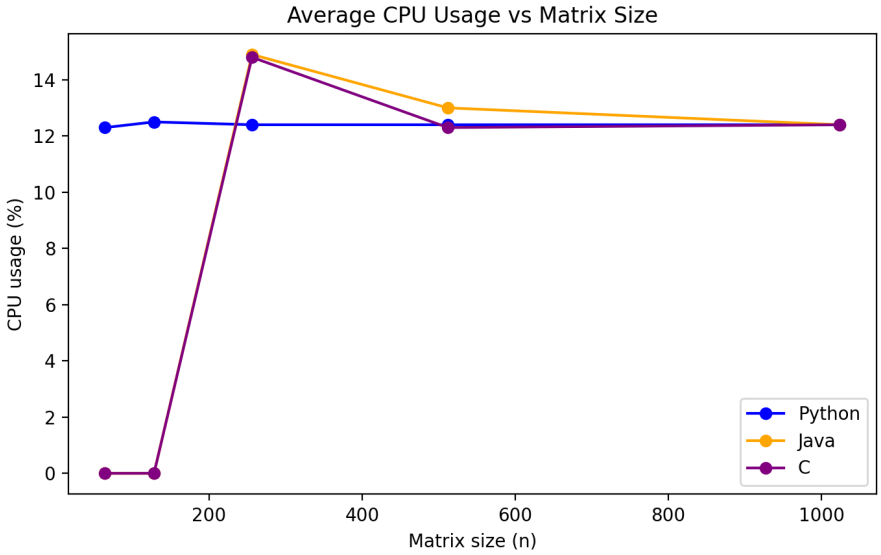


Figure 5: Average CPU usage vs matrix size. Each data point represents the average normalized CPU utilization over three runs per language.

Figure 5 shows that for larger matrix sizes, all languages converge to an average CPU usage close to 12%, which corresponds to a single active computational thread on an eight-core system. This confirms that the implementations are single-threaded, ensuring fair comparability.

For smaller matrices ($n < 256$), transient fluctuations appear due to shorter measurement intervals and runtime overhead. In particular, Java exhibits a pronounced spike near $n = 256$, caused by Just-In-Time (JIT) compilation and class loading phases that temporarily increase CPU activity. Python and C show steadier values across all scales, consistent with their interpreted and compiled natures, respectively.

Overall, the CPU utilization patterns validate that differences in execution time arise primarily from language efficiency and runtime management rather than from parallel workload distribution.

4.5 Memory Measurement

Peak memory consumption was estimated as the maximum resident set size observed immediately before and after each execution. This method provides a reliable upper bound of memory use without intrusive sampling, ensuring that timing integrity is preserved. The recorded metric

thus reflects both the explicit allocation of matrices and the implicit overhead introduced by each language runtime.

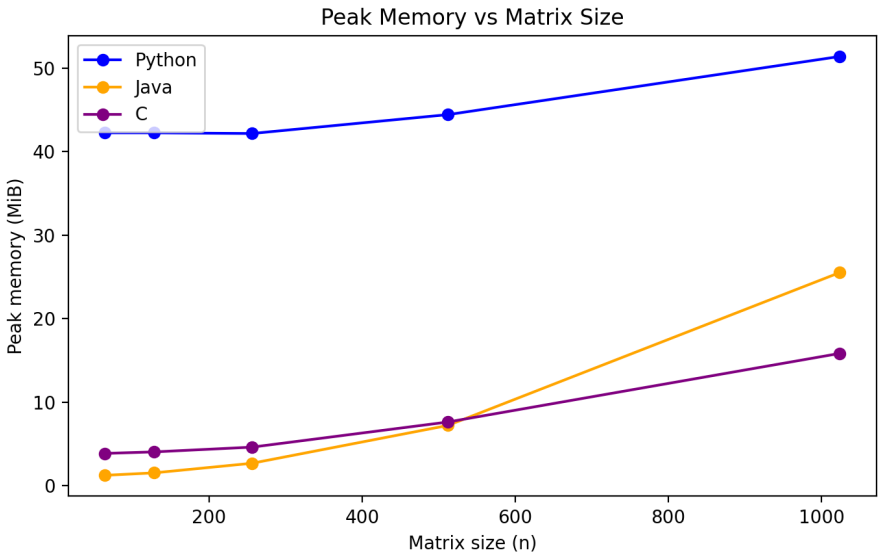


Figure 6: Peak memory vs matrix size. Memory consumption increases monotonically with n.

As shown in Figure 6, memory usage grows approximately with n^2, consistent with the quadratic increase in elements stored by the matrices A, B, and C. However, clear distinctions appear among languages:

- **Python** maintains the highest memory footprint across all experiments, starting around 42 MiB even for the smallest matrices and exceeding 50 MiB at $n = 1024$. This steady overhead originates from the Python interpreter's dynamic object model, where each scalar value carries metadata for type and reference counting, resulting in significantly higher memory requirements per element.
- **Java** begins with a smaller footprint close to C, but its growth rate accelerates as matrix size increases. This behavior is associated with Java's managed heap and garbage collector, which reserve additional space to optimize allocation and prevent fragmentation. The JVM's automatic memory management, while simplifying development, leads to higher peaks as working sets expand.
- **C** exhibits the most compact and predictable memory profile. Since all arrays are explicitly allocated in contiguous blocks using `malloc()`, the memory used corresponds almost exactly to the theoretical requirement of storing three $n \times n$ matrices. No auxiliary metadata or garbage collection buffers are present, resulting in minimal overhead.

Overall, these results confirm that all implementations follow the expected $O(n^2)$ memory scaling, but with different constant factors depending on language abstraction level. Python incurs the largest constant due to its interpreter, Java introduces dynamic allocation overhead, and C remains closest to the theoretical baseline, reflecting the efficiency of low-level manual memory management.

4.6 Data Collection and Storage

All experimental outputs were automatically appended after each execution to a centralized file, `results_raw.csv`, using a semicolon (`;`) as the field delimiter. This choice ensured consistent parsing across locales and avoided ambiguity between decimal and thousands separators. Each record corresponds to a single benchmark run and includes all relevant variables: `run_id`, `language`, matrix `size`, `run_idx`, `time_ms`, `cpu_pct`, and `peak_mib`.

Subsequently, the raw data were aggregated into `results_summary.csv`, computing summary statistics per language and matrix size: the total number of runs, average, minimum, and maximum execution times, mean CPU utilization, and peak resident memory. This schema provides a standardized and reproducible data structure suitable for statistical analysis, visualization, and external validation. It also enables future reprocessing under identical experimental conditions without modifying the original data.

4.7 Data Cleaning

During data aggregation, locale-specific artifacts introduced by spreadsheet software or regional formatting were automatically sanitized. These included numerical representations such as `1.234,56` or `4.270.635`, which were converted into canonical floating-point values following the English notation (`1234.56` and `4270.635`).

The cleaning process ensured that all numeric fields were properly parsed as floating-point types before performing any aggregation or averaging operation. This step prevented rounding inconsistencies and errors caused by misplaced thousand separators, preserving the precision and comparability of all computed metrics across runs, languages, and environments.

5 Conclusions

This study compared the performance of a basic $O(n^3)$ matrix multiplication algorithm implemented in Python, Java, and C under equivalent conditions. The initial hypothesis stated that C would be the fastest language due to its native compilation, direct memory management, and minimal runtime overhead. The results, however, partially challenged this assumption.

According to the collected data (three runs per configuration), **Java outperformed C in almost all tested matrix sizes**, showing shorter execution times for $n = 128, 256, 512$, and 1024. For example, at $n = 1024$, Java averaged approximately 4389.9 ms while C reached approximately 7558.6 ms. Only for the smallest size ($n = 64$) did C maintain an advantage (approximately 0.157 ms vs Java's approximately 1.296 ms). This behavior can be explained, as previously mentioned, by the Just-In-Time (JIT) compiler of the Java Virtual Machine (JVM), which dynamically optimizes frequently executed code paths, reducing constant factors while preserving the cubic complexity. In contrast, the C implementation—compiled with moderate optimization and without CPU-specific vectorization—did not benefit from adaptive optimizations during runtime.

Python exhibited by far the slowest performance across all experiments ($n = 512$: approximately 39-41 s; $n = 1024$: approximately 375-394 s). This outcome is consistent with its interpreted execution model and dynamic object management, which add substantial overhead in the triple nested loops where billions of element-wise operations are performed.

Regarding CPU utilization, all three implementations converged around 12% for large matrix

sizes, consistent with a single-threaded workload on an eight-core system. Therefore, runtime differences stem from implementation efficiency and language runtime overhead, not from concurrency or parallelism.

In terms of memory, **C** proved the most efficient and stable, with memory use matching theoretical expectations for three contiguous $n \times n$ arrays. **Java** displayed moderate growth associated with heap allocation and garbage collector reserves, while **Python** consistently consumed the largest amount of memory due to interpreter metadata and per-object overhead.

In summary: under the tested conditions, **Java achieved the best overall execution performance** for matrices beyond $n = 128$, driven by JIT optimizations and efficient runtime management. **C** remained the most predictable and memory-efficient implementation, confirming its value as a low-level performance baseline. **Python**, while functionally correct, proved inadequate for intensive numerical workloads without external acceleration (e.g., NumPy or compiled extensions).

These findings confirm the expected cubic computational trend and demonstrate how language runtime design—compiled vs managed vs interpreted—directly impacts practical performance in numerical computing.

6 Future Work

While this study provides a comprehensive baseline comparison of matrix multiplication across three languages, several directions for future research could extend and deepen these findings:

6.1 Advanced Compiler Optimizations

The C implementation could be recompiled with aggressive optimization flags (`-O3`, `-Ofast`, `-march=native`) to enable hardware-specific vectorization (AVX2/AVX-512) and auto-vectorization. Comparing these optimized builds against the baseline `-O2` version would quantify the performance gains achievable through compiler-level optimizations and establish whether C can reclaim its expected performance advantage over Java when fully optimized.

6.2 Parallelization and Multi-threading

All current implementations are strictly single-threaded. A natural extension would involve implementing parallel versions using:

- **OpenMP** for C to enable loop-level parallelism
- **Java's Fork/Join framework or parallel streams** for Java
- **Python's multiprocessing module** to bypass the Global Interpreter Lock (GIL)

This would reveal how each language scales with multiple cores and whether Python's parallelization overhead remains prohibitive even with process-based concurrency.

6.3 GPU Acceleration

Modern numerical computing increasingly relies on GPU acceleration. Extending this study to include GPU implementations using CUDA (for C), JOCL/JCuda (for Java), and CuPy/Numba

(for Python) would provide insights into:

- Data transfer overhead between CPU and GPU
- Language-specific GPU programming ergonomics
- Performance scaling for massively parallel workloads

6.4 Statistical Rigor and Variance Analysis

The current study uses three runs per configuration, which provides a reasonable estimate but limits statistical power. Future work could:

- Increase the number of runs (e.g., 10-30) to enable confidence interval estimation
- Apply statistical hypothesis testing to determine if performance differences are significant
- Perform warm-up iterations to stabilize JIT compilation effects
- Use performance profiling tools (e.g., `perf`, VTune) to identify specific bottlenecks

6.5 Energy Efficiency Analysis

Beyond execution time, energy consumption is increasingly important for sustainable computing. Future work could measure:

- Total energy consumption per matrix multiplication (joules)
- Energy efficiency (operations per joule)
- Power draw profiles during execution

This would reveal whether faster execution necessarily translates to lower energy usage or if runtime overhead in interpreted languages incurs disproportionate energy costs.

6.6 Heterogeneous Computing Environments

Testing the same implementations on different architectures would reveal portability characteristics:

- ARM-based processors (e.g., Apple M-series, AWS Graviton)
- Different x86 microarchitectures (Intel vs AMD, different generations)
- Cloud instances with varying CPU characteristics

This would establish whether observed performance relationships hold across hardware platforms or are specific to the AMD Ryzen architecture used in this study.

In conclusion, while this study establishes a solid baseline for understanding language-level performance characteristics in matrix multiplication, numerous avenues remain for deeper investigation. Future work along these directions would provide a more complete picture of the performance-productivity tradeoffs inherent in language choice for numerical computing applications.