Language Benchmark of Naive Matrix Multiplication

Sergio Hernández González

October 23, 2025

Abstract

We evaluate a naive $O(n^3)$ matrix multiplication kernel across C, Java, and Python. All languages share the same algorithm and experimental protocol: multiple runs per size and consolidated logging to a single CSV. We report mean, variability, and memory behavior, discuss threats to validity, and provide scripts for full reproducibility.

1 Introduction

This report compares the runtime and memory behavior of the same triple-nested-loop matrix multiplication across three languages. We target clarity and reproducibility rather than peak performance.

2 Methodology

Algorithm. The implementation is the classic three-loop product $C = A \cdot B$ without blocking/tiling, SIMD, or multithreading.
Separation of concerns. Each language isolates production code (the multiplication routine) from benchmarking code (input generation, timing, memory sampling, CSV logging).
Parameters. We vary the matrix size n and repeat each experiment a fixed number of times (`runs`).
Timing. High-resolution clocks are used: `QueryPerformanceCounter` on Windows/C, `System.nanoTime()` in Java, and `time.perf_counter()` in Python.
Memory. We record process memory deltas (MB): Working Set (Windows) in C; heap usage via `Runtime` in Java; RSS via `psutil` in Python.
Data sink. All runs append to `data/results.csv` with schema: `language, matrix_size, run_index, elapsed_sec, memory_used_mb, timestamp_iso`.

3 Environment

Fill in CPU model, cores/threads, RAM, OS version, and toolchain versions (GCC/MinGW, Java, Python). Record power/performance profile and whether a laptop is on AC power.

4 Results

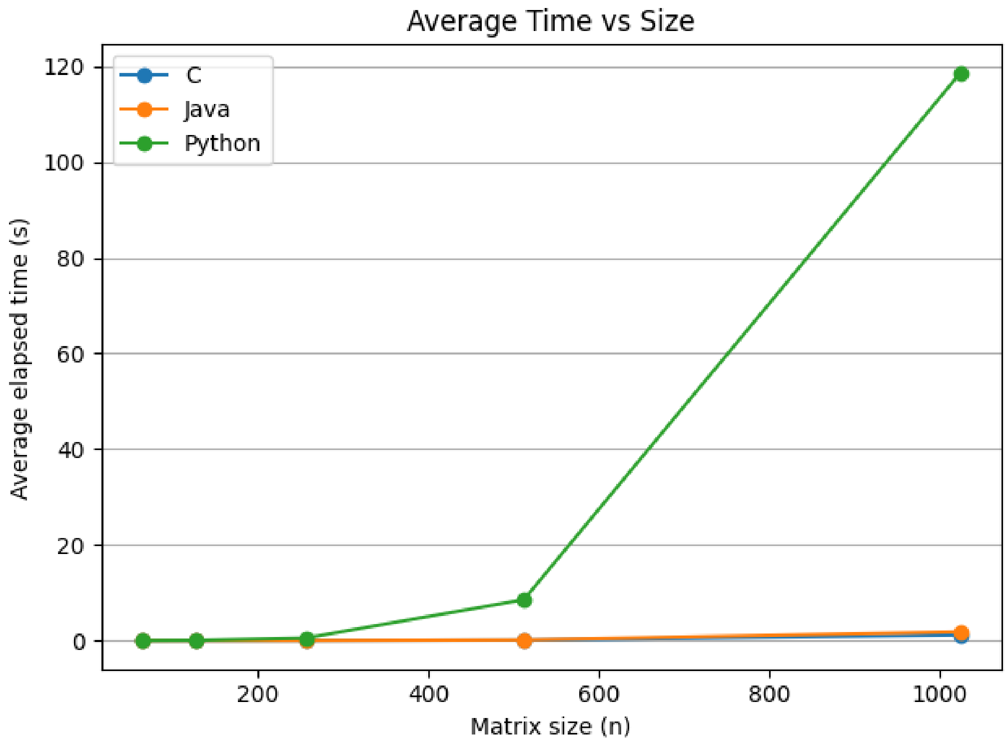

Figure 1: Average elapsed time by language and matrix size (markers show means over runs).

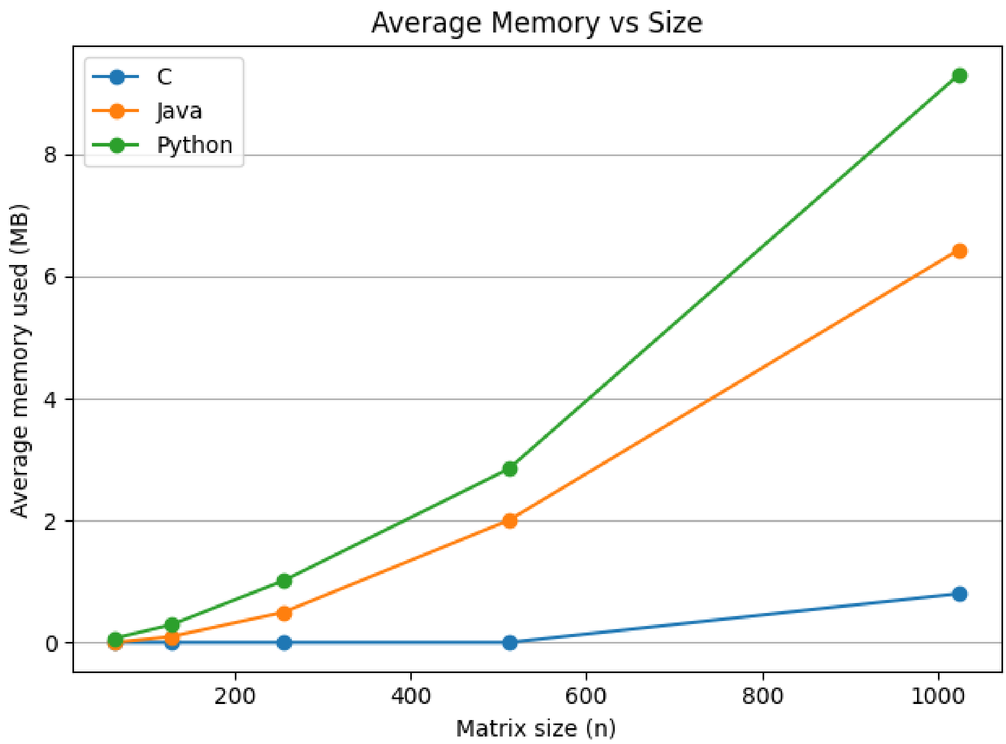

Figure 2: Average memory consumption by language and matrix size.

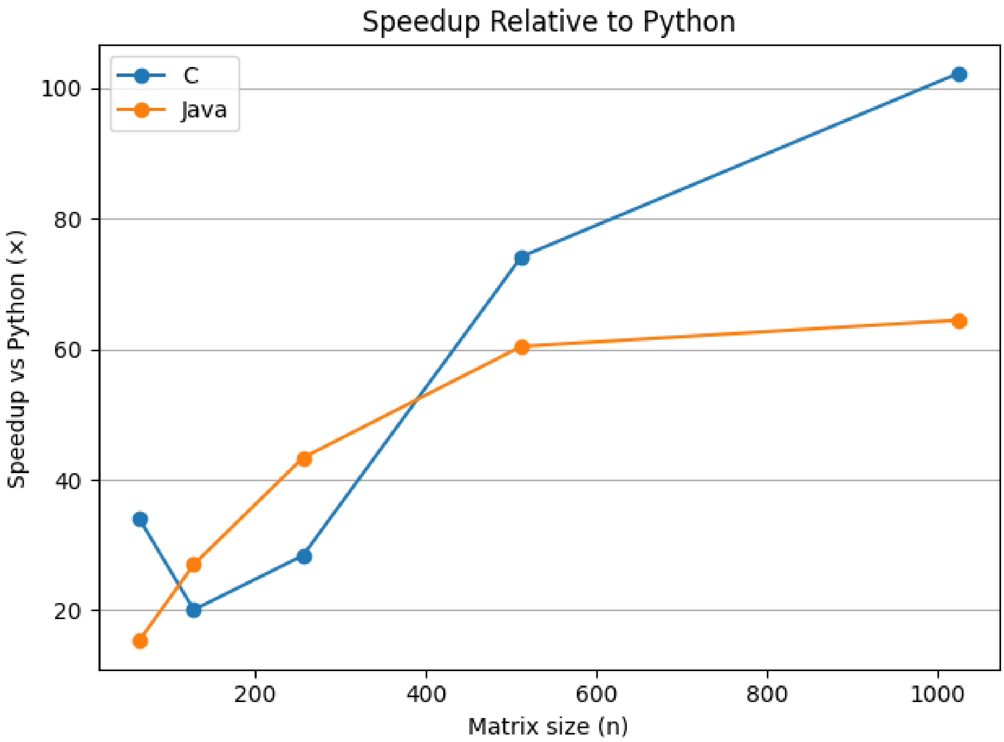

Figure 3: Speedup relative to Python (higher is better).

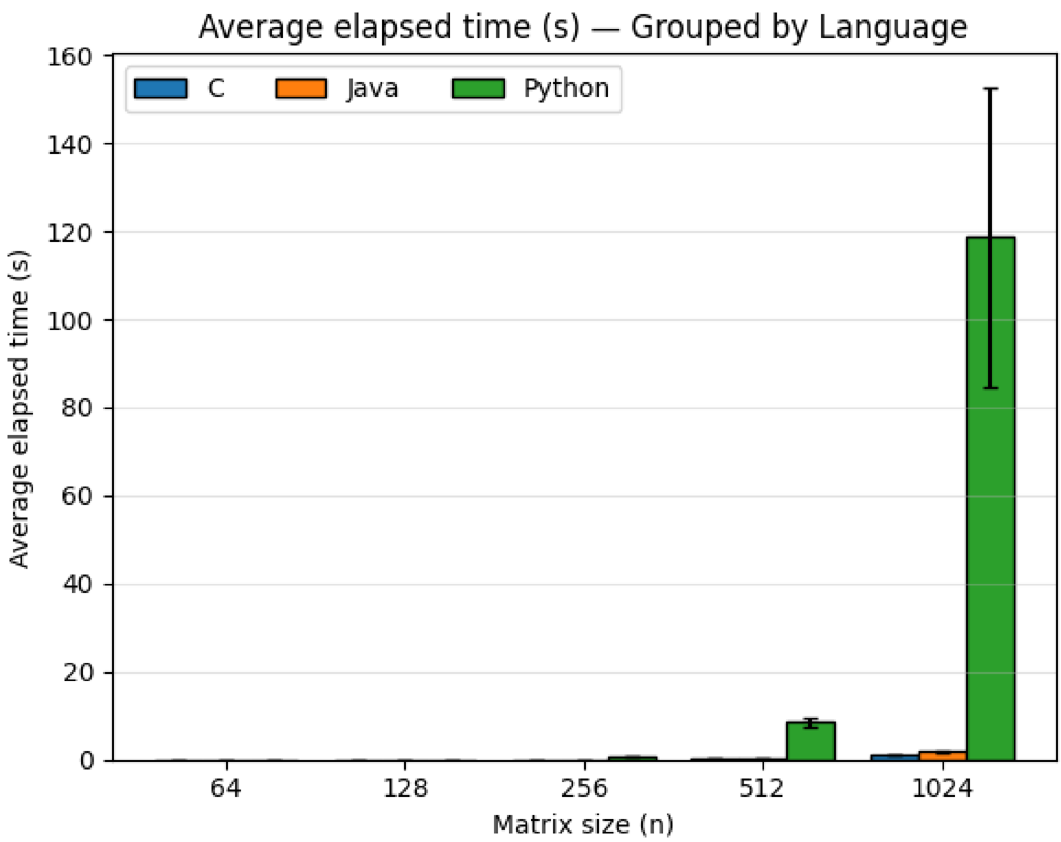

Figure 4: Average elapsed time with standard-deviation error bars (grouped by language and matrix size).

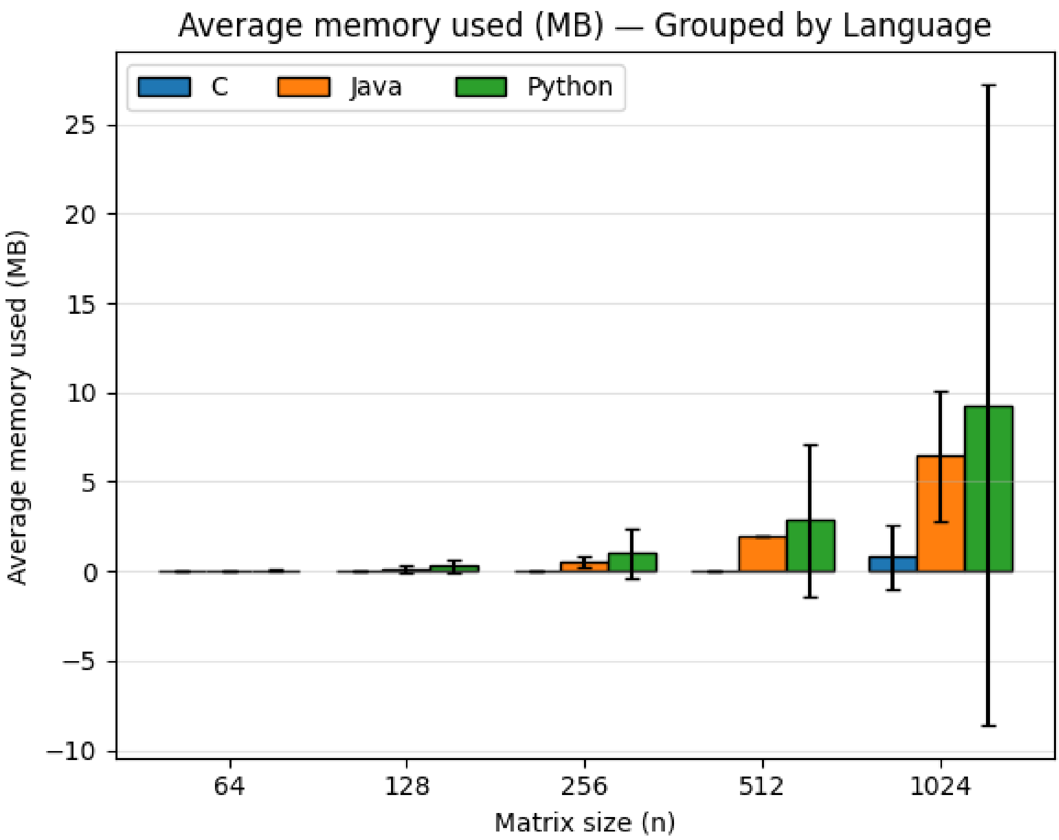

Figure 5: Average memory usage with standard-deviation error bars (grouped by language and matrix size).

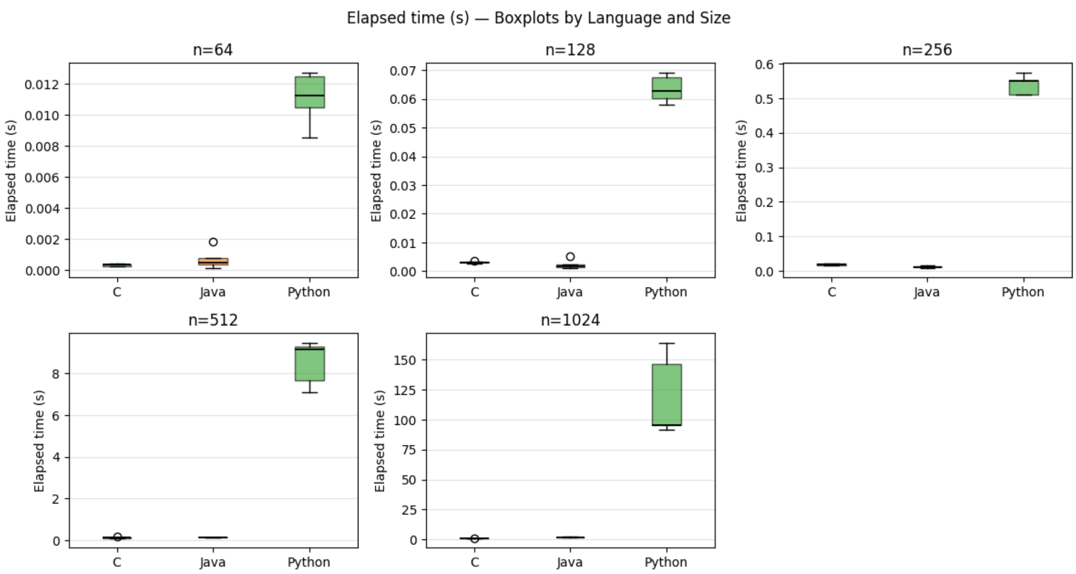

Figure 6: Elapsed time distributions (boxplots) by language across all matrix sizes.

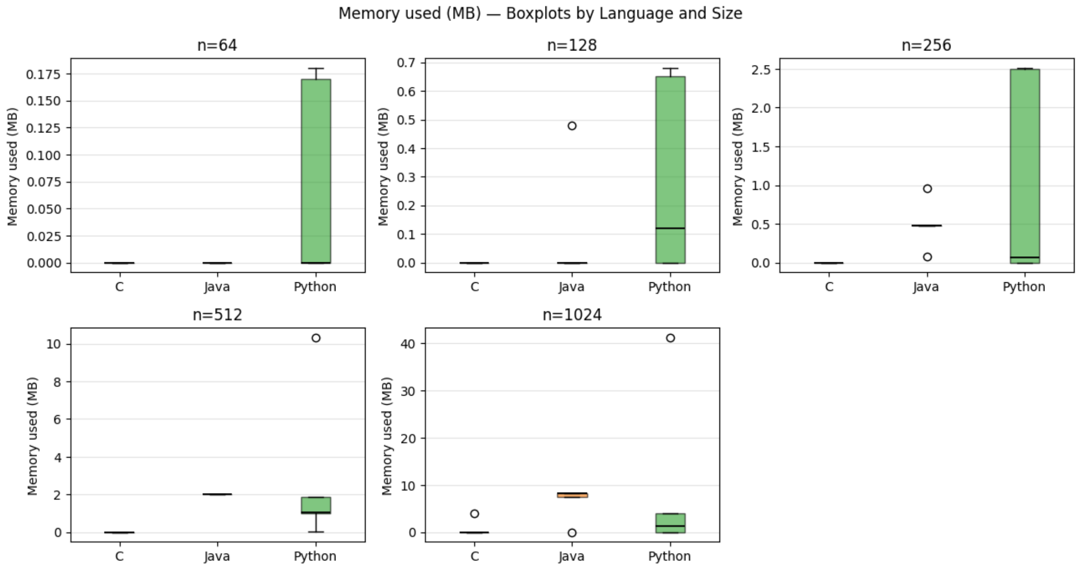

Figure 7: Memory usage distributions (boxplots) by language across all matrix sizes.

Table 1: Per-language summary statistics (mean ± std).

Language	Size	Runs	Avg time (s)	Avg mem (MB)	Note
C	64	5	0.000325 ± 0.000073	0.00 ± 0.00	
C	128	5	0.003167 ± 0.000330	0.00 ± 0.00	
C	256	5	0.018991 ± 0.003366	0.00 ± 0.00	
C	512	5	0.115088 ± 0.020415	0.00 ± 0.00	
C	1024	5	1.159879 ± 0.022769	0.80 ± 1.79	
Java	64	5	0.000723 ± 0.000661	0.00 ± 0.00	
Java	128	5	0.002353 ± 0.001642	0.10 ± 0.22	
Java	256	5	0.012428 ± 0.003182	0.50 ± 0.32	
Java	512	5	0.141220 ± 0.007584	2.00 ± 0.00	
Java	1024	5	1.840129 ± 0.191275	6.43 ± 3.62	
Python	64	5	0.011078 ± 0.001683	0.07 ± 0.10	
Python	128	5	0.063536 ± 0.004740	0.29 ± 0.35	
Python	256	5	0.538847 ± 0.028067	1.01 ± 1.36	
Python	512	5	8.535703 ± 1.071202	2.85 ± 4.23	
Python	1024	5	118.612292 ± 34.122409	9.30 ± 17.93	

5 Discussion

As expected, C generally leads on raw runtime due to its ahead-of-time compilation and minimal runtime overhead. Java follows closely after just-in-time warm-up. Pure-Python triple loops are substantially slower, but with vectorized libraries (NumPy/BLAS) Python can achieve performance competitive with C—left for future work. The observed memory deltas are modest because input matrices are reused and only the output matrix is written.

6 Threats to Validity

(1) JIT warm-up effects in Java can skew the first run. (2) Background processes, turbo/thermal throttling, and power profiles affect timings. (3) Measurement of "memory used" differs slightly

across platforms and APIs. We mitigate these threats by using multiple runs and reporting standard deviations.

7 Conclusion

The naive $O(n^3)$ kernel scales cubically across all languages. Results reflect language/runtime overhead rather than algorithmic differences. Future extensions include blocked/tiling variants, multithreaded versions, and BLAS-backed implementations to approach hardware limits.

Artifacts and Reproducibility

- GitHub repository: `https://github.com/seergiohrndz7/Individual-Assingment.git`.
- Unified CSV: `data/results.csv`.