Language Benchmark of Matrix Multiplication

Richard Raatz: L0NC8J5G

https://github.com/RGR-repo-cloud/ULPGC_BigData/tree/main/Task_1

Big Data (40386) - 3. year

Universidad de Las Palmas de Gran Canaria

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

List of Tables

1 Introduction

In the context of Big Data computational methods capable of handling massive datasets play a central role. The sheer volume, velocity, and variety of data necessitate highly efficient algorithms and frameworks to extract meaningful insights [1]. One such operation that is fundamental to countless scientific, data processing, and engineering tasks, including machine learning [2], graph analytics, and numerical simulation, is matrix multiplication. Its standard $O(n^3)$ algorithm is therefore a critical and frequently used benchmark for accessing raw computational performance [3].

This paper assesses the performance and efficiency of three dominant programming languages, namely *Python*, *Java*, and *C*, concerning the standard $O(n^3)$ matrix multiplication implementation. While the theoretical complexity remains constant across these environments, the empirical execution time and memory usage differ significantly. These performance discrepancies are mainly attributed to underlying differences in compiler optimization, memory management, and interpreter overhead [4]. These low-level architectural and implementation details have a substantial impact on the wall-clock time, particularly in data-intensive tasks where the performance gap between optimized compiled code and interpreted environments becomes pronounced [5].

The remainder of this paper provides a short overview of the implementation of the suite utilized for the benchmarking, then the benchmarking methodology is described, and finally, the results are presented.

2 Architecture and Implementation

This benchmark project employs a multi-language comparative architecture to evaluate computational efficiency across Python, Java, and C implementations using identical $O(n^3)$ algorithms. The system architecture follows a strict separation of concerns principle, dividing production code from testing infrastructure across all three language implementations. The production layer consists of core matrix multiplication modules that implement the fundamental triple-nested loop algorithm, alongside matrix utility modules. The testing and benchmarking layer comprises dedicated benchmark runners that handle performance measurement, memory profiling, and CSV data output. There is a clear structural seperation between the different languages so that each language can be individually assessed. In order to compare them the architecture incorporates a universal orchestration system that coordinates cross-language execution, followed by a centralized analytics pipeline including statistical aggregation and comprehensive visualization generation. This modular, language-agnostic design ensures algorithmic consistency while enabling fair performance comparisons, with standardized CSV output facilitating reproducible big data analysis workflows.

3 Benchmarking Methodology

The benchmark methodology employs a rigorous multi-run statistical approach designed to ensure measurement reliability and eliminate performance anomalies across heterogeneous programming environments. Each matrix multiplication test executes five independent runs per matrix size configuration (ranging from 64x64 to 1024×1024 elements) to capture statistical variance and provide robust performance metrics. For precise time and memory measurements language specific mechanisms are employed. Each benchmark iteration follows a controlled execution protocol: matrices are freshly generated per run using random number generators and memory baselines are established before algorithm execution. The methodology captures comprehensive performance metrics including execution time (seconds), memory consumption (MB), and derives statistical summaries (mean, minimum, maximum, standard deviation) through dedicated utility modules.
The system used for benchmarking is charakterized by:

- CPU: AMD Ryzen 7 4700U (8 cores and threads, 2.0 GHz - 4.1 GHz)
- RAM: 16 GB DDR4 (3200 MHz)
- OS: Ubuntu 22.04.5 LTS
- Python version: 3.10.12
- Java version: 11.0.28
- gcc version: 11.4.0

4 Results

Table 1: Matrix Multiplication Performance Comparison

Language	Matrix_Size	Avg_Time_Seconds	Min_Time_Seconds	Max_Time_Seconds	Std_Dev_Seconds	Avg_Memory_MB	Peak_Memory_MB
C	64	0.001614	0.001599	0.001642	0.000017	0.05	0.25
C	128	0.015182	0.015098	0.015342	0.000096	0.076	0.38
C	256	0.130748	0.130222	0.131379	0.000417	1.024	1.12
C	512	1.095654	1.072856	1.110345	0.015616	5.5	5.5
C	1024	9.234707	9.003220	9.753174	0.308796	23.5	23.5
Java	64	0.011544	0.001909	0.030526	0.010992	0.128	0.16
Java	128	0.016306	0.016042	0.016952	0.000388	0.43	0.43
Java	256	0.163176	0.158583	0.168517	0.003671	1.566	1.6
Java	512	1.622254	1.537117	1.741855	0.074514	6.306	6.49
Java	1024	17.008471	16.905218	17.192113	0.120079	24.438	24.46
Python	64	0.046289	0.045339	0.049704	0.001911	0.22	0.62
Python	128	0.314368	0.296633	0.380408	0.036926	0.802	1.89
Python	256	2.891140	2.861304	2.971989	0.045600	2.858	5.2
Python	512	30.816077	29.899132	34.070624	1.821212	11.21	18.99
Python	1024	457.161065	309.290326	1037.965786	324.700815	40.224	70.52

This cross-language benchmark evaluates a uniform $O(n^3)$ matrix multiplication implemented in C, Java, and Python over five runs per matrix size (64–1024). The overall picture shows a stable

performance hierarchy with C consistently fastest, Java within a small constant factor of C, and Python orders of magnitude slower at larger sizes. At 1024×1024, the average execution times are 9.23 s (C), 17.01 s (Java, 1.8× slower), and 457.16 s (Python, 49× slower), and Python exhibits substantial dispersion at this scale (Std Dev 324.70 s), indicative of runtime volatility. Memory usage follows the expected $O(n^2)$ growth—dominated by the three matrices in play—with averages near 23.5 MB (C), 24.4 MB (Java), and 40.2 MB (Python) at 1024×1024. Taken together, the results quantify runtime overhead and object-model costs rather than algorithmic differences, since the algorithm is held constant.

Figure-level detail strengthens these conclusions.

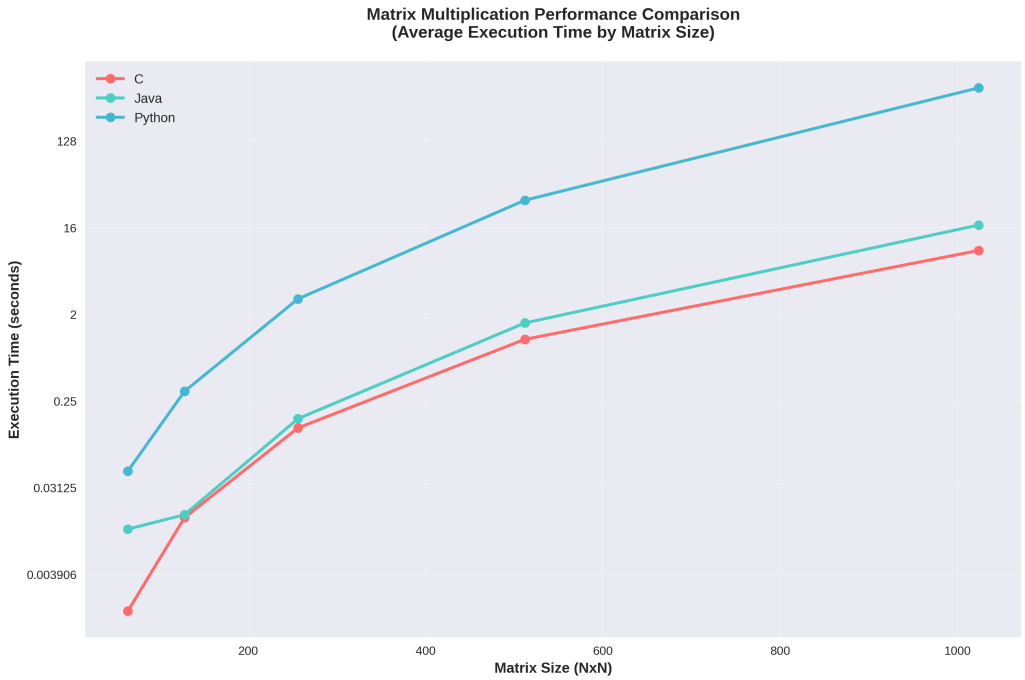

Figure 1: Performance comparison concerning execution time

In 1 (execution time vs size; log_2 Y-axis), the three curves diverge predictably as size increases: C forms the lower envelope, Java tracks closely with a near-parallel slope, and Python rises steeply, especially beyond 256, reflecting interpreter overhead in tight nested loops.

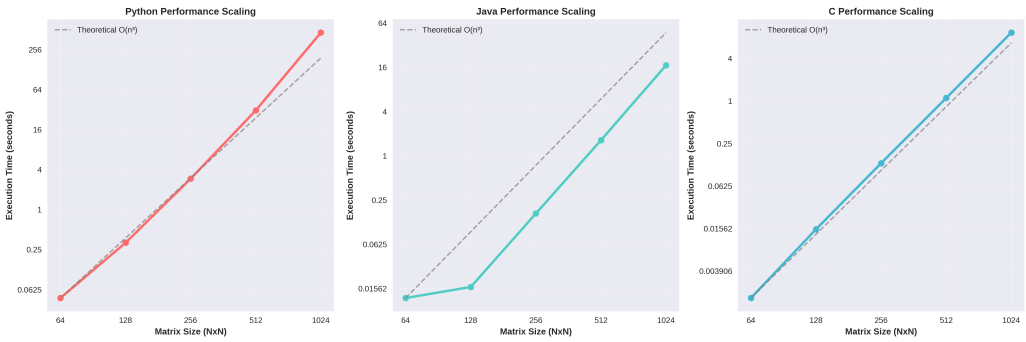

Figure 2: Scaling comparison concerning execution time

Figure 2 ($log_2 - log_2$ axes) places empirical timings against the theoretical $O(n^3)$ trend; the language-specific lines display near-linear slopes, corroborating cubic growth and similar asymptotic behavior across implementations. Only Java shows a bigger deviation from that trend between the sizes 64x64 and 128x128 where the execution time stays more constant.

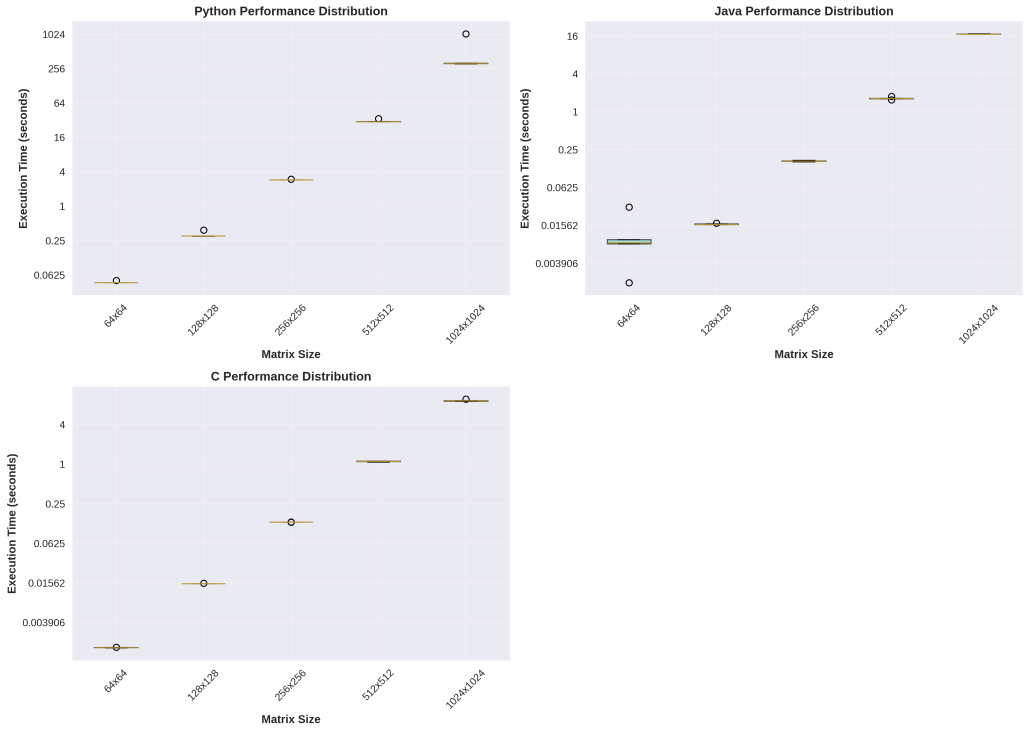

Figure 3: Statistical summary of execution times

The box plots in 3 (log_2 Y-axis) show that C's dispersion remains tight across sizes, Java's variability is modest and scale-stable, while Python's interquartile range and whiskers broaden at 512 and 1024, consistent with scheduler effects, garbage collection pauses, and cache pressure at large n.

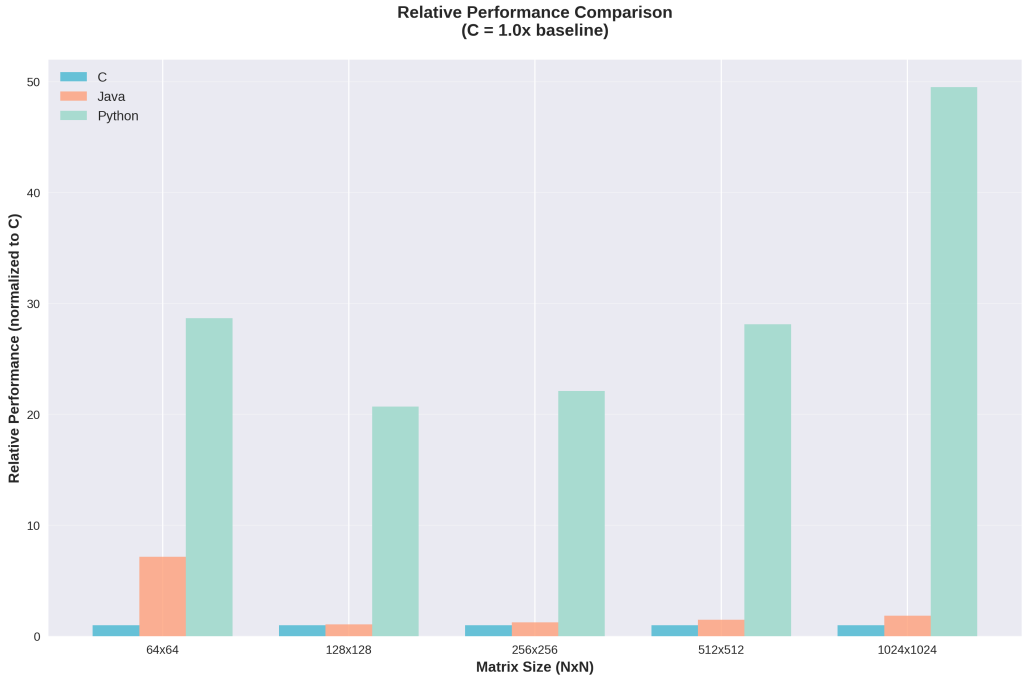

Figure 4: Relative performance concerning execution time normalized with C as the baseline

Complementing these, figure 4 (linear scale) makes the constant-factor gaps explicit: Java typically remains within 1.2×–1.9× of C (1.8× at 1024), while Python spans 20×–50× depending on size (49× at 1024), which aligns with expectations for a pure-Python triple loop absent vectorization.

Memory behavior presents a coherent secondary narrative.

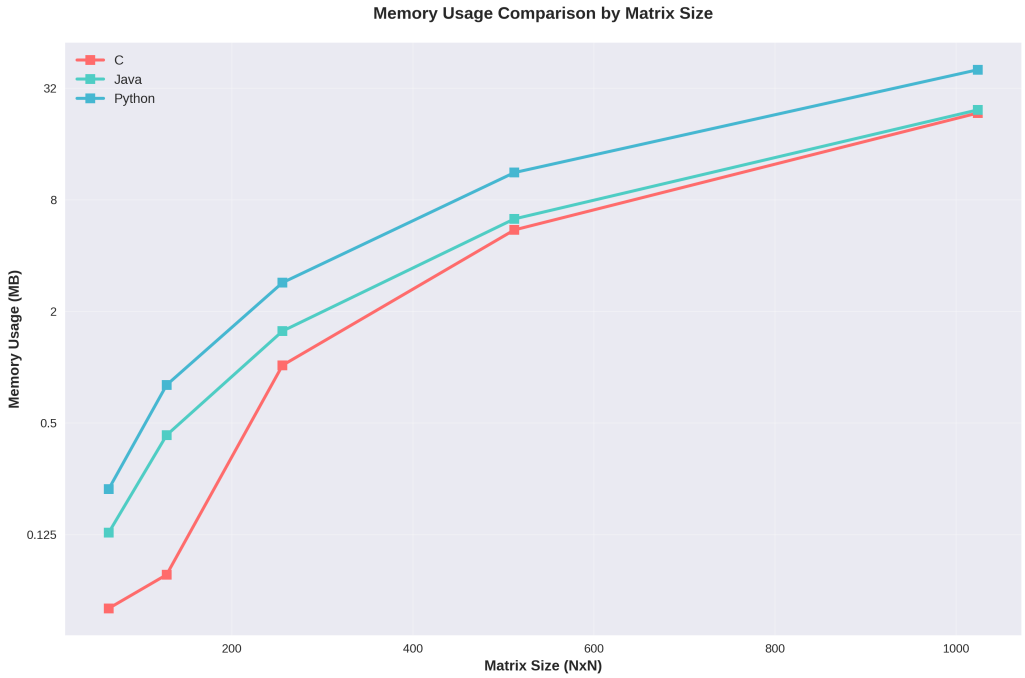

Figure 5: Performance comparison concerning memory usage

In figure 5 (log_2 Y-axis), average memory rises smoothly with size, with Python consistently above C and Java due to higher per-object and interpreter overheads.

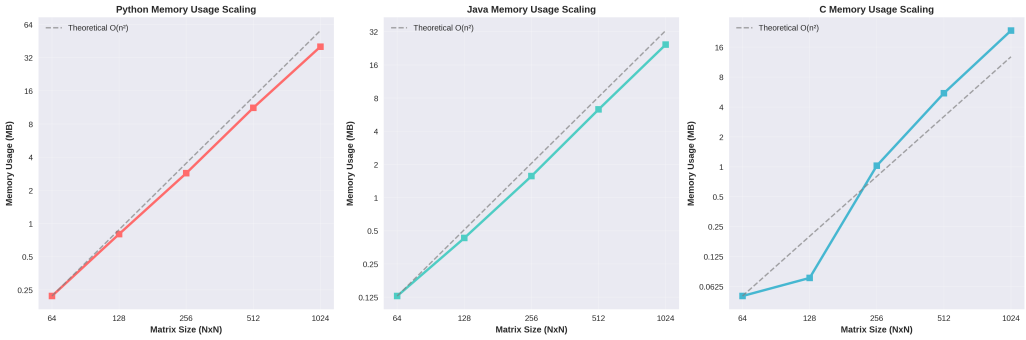

Figure 6: Scaling comparison concerning memory usage

The $log_2 - log_2$ plot in 6 is close to linear across languages, aligning with $O(n^2)$ storage; the plotted theoretical guide tracks observed slopes well, with small constant-factor differences between languages. C shows the largest deviation from constantly scaled memory usage.

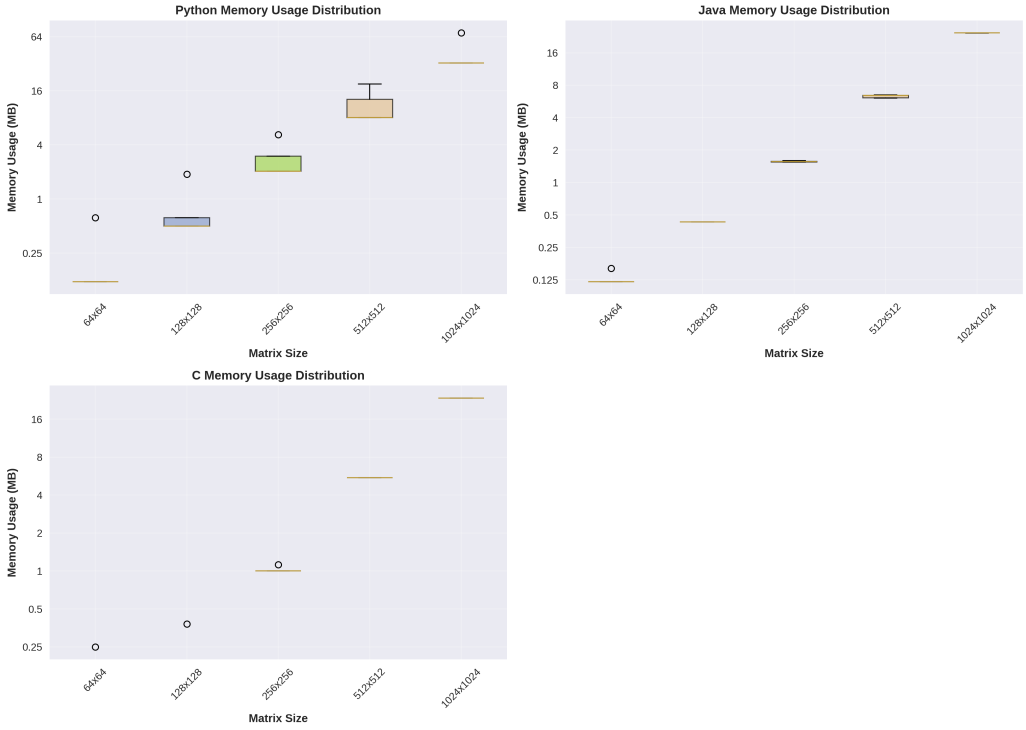

Figure 7: Stastical summary of memory usage

The box plots in figure 7 (log_2 Y-axis) show limited variance overall—consistent with deterministic allocation patterns—though Python's distributions sit higher at every size.

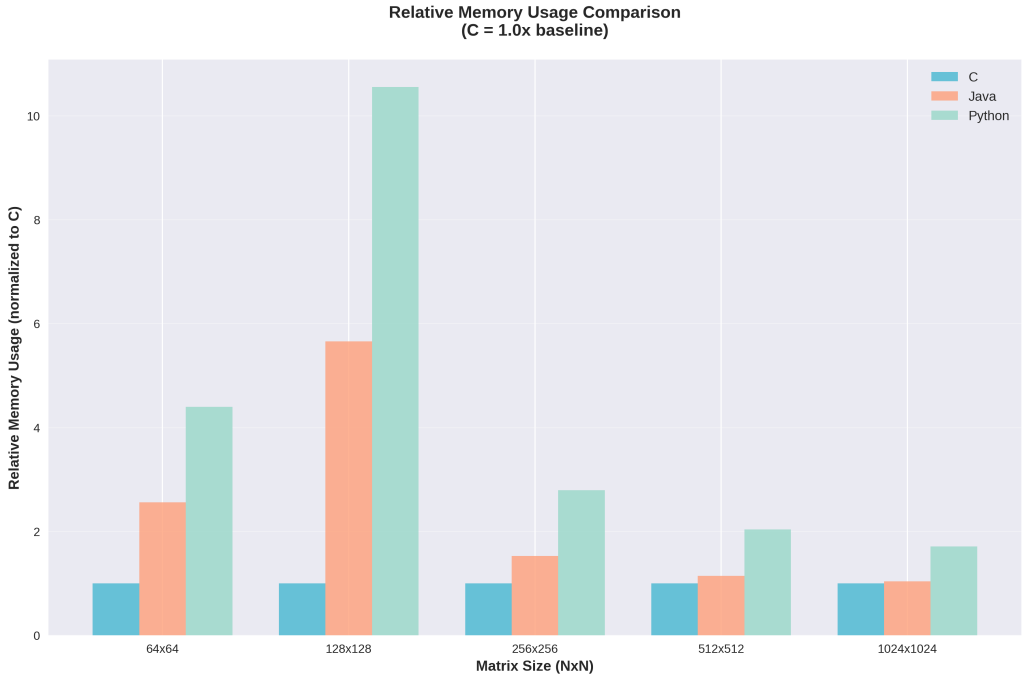

Figure 8: Relative memory usage normalized with C as the baseline

Finally, figure 8 (linear scale) highlights that Java is typically within 1.0–1.1× of C, while Python requires 1.5–1.8× at 1024, reinforcing the constant-factor overhead picture for memory as well.

Interpreting these results for big data workloads, the figures collectively indicate that naïve, non-optimized matrix multiplication is dominated by runtime characteristics [6]. C is fastest because it's ahead-of-time compiled over contiguous primitives with no interpreter or garbage collector, letting the compiler optimize loops and use SIMD; at 1024×1024 it averages 9.23 s. Java is close (17.01 s, 1.8× slower) thanks to HotSpot JIT (inlining, bounds-check elimination) but still pays warm-up/garbage collector and arrays-of-arrays hurt cache locality [7]. Pure-Python loops are far slower (457 s, 49× C) because every inner-loop step runs through the interpreter over lists-of-lists of boxed floats with ref-count/GC overhead [5]. Memory grows $O(n^2)$ for all; at 1024×1024 it's 23.5 MB (C), 24.4 MB (Java), 40.2 MB (Python).

5 Conclusion

This project presented a cross-language benchmark of a naïve $O(n^3)$ matrix multiplication across C, Java, and Python. The results consistently show C as the fastest and most stable, Java within a modest constant factor, and pure-Python orders of magnitude slower at larger matrix sizes; memory usage follows the expected $O(n^2)$ trend with Python incurring higher constant-factor overhead. The results show a coherent narrative that the dominant differences arise from runtime and language-level overhead rather than algorithmic choices, given the shared implementation [7]. However, the study's scope is bounded by single-threaded baselines, the absence of vendor-optimized kernels, and the use of a ba-

sic triple-loop algorithm [6]; therefore, the reported numbers quantify language/runtime costs rather than the attainable peak performance on modern systems. Practically, production pipelines should avoid interpreted inner loops and prefer optimized numerical libraries (BLAS/MKL via NumPy) [6], algorithmic improvements (blocking/tiling, cache-aware variants, Strassen/Winograd where appropriate) [3], and parallel hardware (multithreading, SIMD, GPUs) [2]. The study's limitations—single-threaded baselines, no vendor-optimized kernels, and a fixed cubic algorithm—mean the results primarily isolate language/runtime overheads rather than the ceiling of achievable performance. Future work should evaluate optimized and parallel implementations and expand the analysis to distributed settings to reflect real-world big data pipelines more comprehensively [1].

List of sources

[1] A. McAfee et al. “Big data: The management revolution”. In: *Harvard Bus Rev* 90 (Jan. 2012), pp. 61–67.

[2] Jeffrey Heaton. “Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning: The MIT Press, 2016, 800 pp, ISBN: 0262035618”. In: *Genetic Programming and Evolvable Machines* 19 (Oct. 2017). DOI: 10.1007/s10710-017-9314-z.

[3] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. “The LINPACK Benchmark: past, present and future”. In: *Concurrency and Computation: Practice and Experience* 15 (Aug. 2003), pp. 803–820. DOI: 10.1002/cpe.728.

[4] Karan Vishwakarma. “Java vs Python vs C++ comparison – coding and performance”. In: *Medium* (2024). URL: https://medium.com/@karanvishwakarma/java-python-c-comparison-coding-and-numerical-b25122698930.

[5] Seongho Ahn, Joohyun Kim, and Jaesun Kim. “Programming language comparison for scientific computing: C, C#, Java, and Python”. In: *2015 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE. 2015, pp. 1164–1167. DOI: 10.1109/ICTC.2015.7354674.

[6] Charles R. Severance and Kevin Dowd. *High Performance Computing*. OpenStax CNX, 2015.

[7] Peter Sestoft. “Numeric performance in C, C# and Java”. In: *IT University of Copenhagen* (2012). URL: http://www.itu.dk/~sestoft/papers/numericperformance.pdf.