Matrix Multiplication Benchmark

Comparative Performance Analysis

Course: Big Data - Task 1
Universidad de Las Palmas de Gran Canaria

Juan Diego González Noguera

Repository:
https://github.com/juand4569/BenchmarkMatrixMultiplication

October 23, 2024

Contents

1 Introduction

Matrix multiplication is fundamental in Big Data applications including machine learning, scientific computing, and data analytics. The naive algorithm exhibits $O(n^3)$ complexity, making it ideal for comparing language performance characteristics.

This study implements and benchmarks the standard triple-nested loop algorithm in Java, Python, and C across matrix sizes from 128×128 to 1024×1024, measuring execution time and memory usage.

Objectives:

- Implement identical $O(n^3)$ algorithms in three languages
- Measure execution time and memory consumption
- Analyze scalability and performance trade-offs
- Provide empirical data for language selection in Big Data contexts

2 Implementation

2.1 Java with JMH

Used Java Microbenchmark Harness (JMH) with JDK 21. Configuration: AverageTime mode, 3 warmup iterations, 5 measurement iterations. OSHI library added for memory profiling since JMH lacks native memory tracking.

2.2 Python with Custom Timing

Python 3.11 with native lists, `time.perf_counter()` for timing, and `tracemalloc` for memory. Initial pytest-benchmark attempts abandoned due to excessive framework overhead obscuring actual algorithm performance.

2.3 C with GCC

Compiled with GCC -O3 optimization. Cross-platform timing via QueryPerformanceCounter (Windows) and gettimeofday (Unix). Memory tracking with platform-specific APIs (PROCESS_MEMORY_COUNTERS / rusage).

3 Results

3.1 Execution Time

Figure 1 visualizes the performance hierarchy. The log-scale plot reveals C and Java tracking closely (near-linear parallel lines), while Python diverges exponentially. For 1024×1024: C (baseline) → Java (2.16× slower) → Python (135.7× slower than C).

The linear-scale view (Figure 2) emphasizes Python's dramatic slowdown: C and Java appear nearly flat while Python dominates. This visualization underscores why production numerical computing cannot use naive Python.

Size	Java (ms)	Python (ms)	C (ms)
128×128	1.62	165.95	2.41
256×256	18.57	1354.48	27.50
512×512	178.15	12426.36	175.19
1024×1024	4143.32	260247.13	1917.39

Table 1: Execution Time Comparison

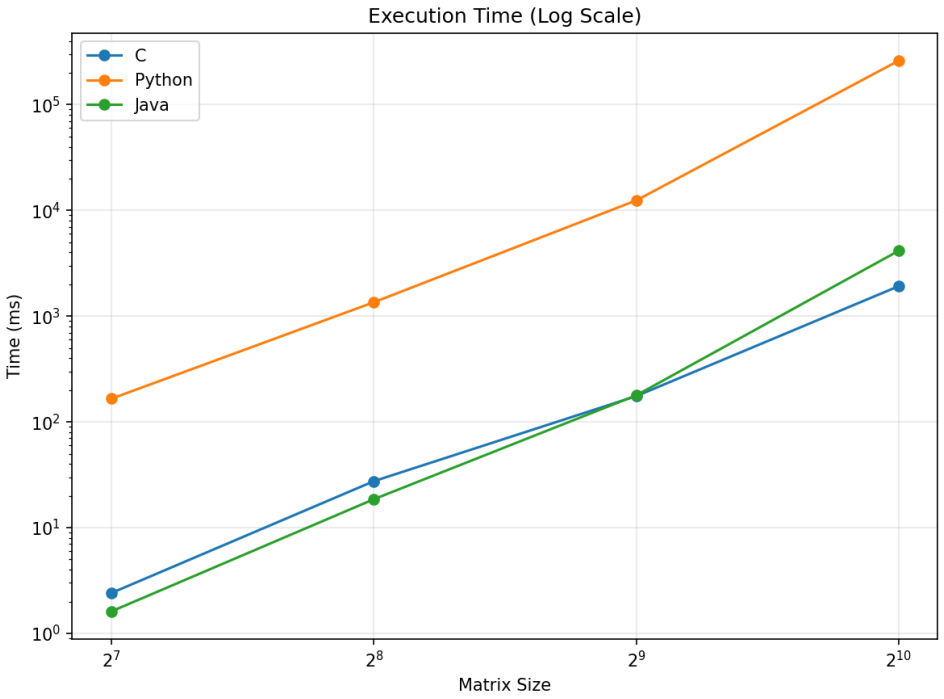

Figure 1: Execution Time (Log Scale) - Python's exponential overhead visible

3.2 Memory Usage

Size	Java (MB)	Python (MB)	C (MB)
128×128	187.44	88.82	9.87
256×256	195.11	93.72	9.87
512×512	219.19	116.49	9.87
1024×1024	251.36	208.77	25.74

Table 2: Peak Memory Usage

Figure 3 shows distinct memory profiles. C remains flat (10 MB) until 1024×1024, demonstrating minimal runtime overhead. Java exhibits high constant baseline (187 MB JVM) with moderate growth. Python starts lower but grows more steeply, reflecting per-object overhead in lists.

The flat C memory profile for sizes 512 suggests excellent cache locality and minimal allocator overhead. Java's linear growth from high baseline reflects JVM memory management. Python's steeper growth at 1024×1024 indicates list structure overhead

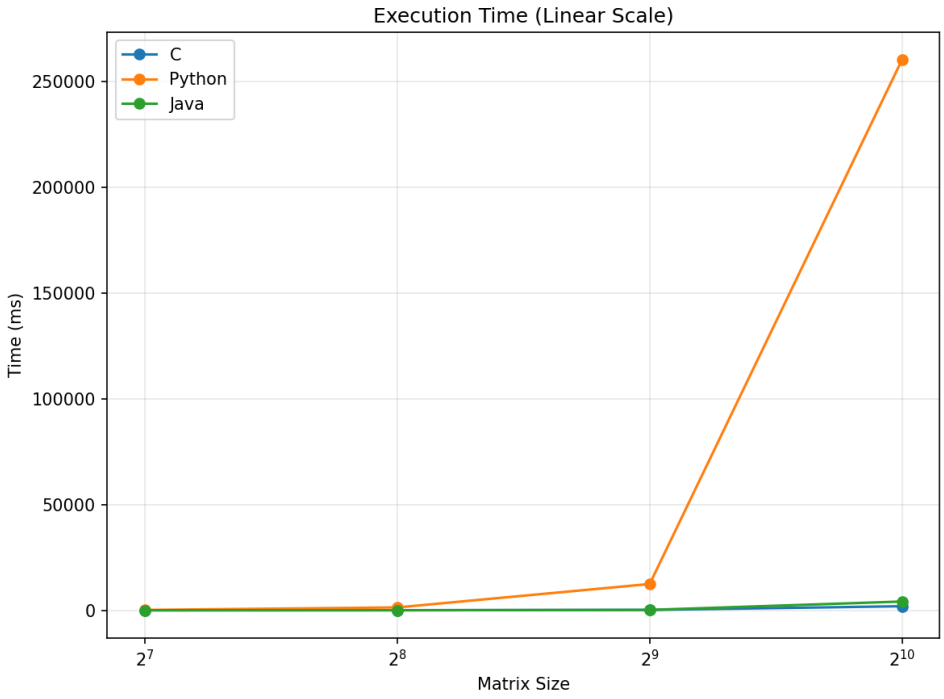

Figure 2: Execution Time (Linear Scale) - Python overhead dominates

accumulating with matrix size.

3.3 Scalability

Doubling matrix size should theoretically yield 8× slowdown ($2^3 = 8$). Observed ratios:

Transition	Java	Python	C
128→256	11.5×	8.2×	11.4×
256→512	9.6×	9.2×	6.4×
512→1024	23.3×	20.9×	10.9×

Table 3: Performance Scaling (Expected: 8×)

Deviations caused by cache effects, memory bandwidth limitations, and JIT optimization.

4 Analysis

4.1 Language Characteristics

Java: JIT compilation achieves near-C performance after warmup. High memory overhead from JVM runtime. Good balance between performance and productivity.

Python: Interpreter overhead and dynamic typing cause 100× slowdown. Excellent for prototyping but requires optimized libraries (NumPy) for production numerical work.

C: Minimal overhead, predictable performance, direct hardware access. Baseline for compiled language performance but requires manual memory management.

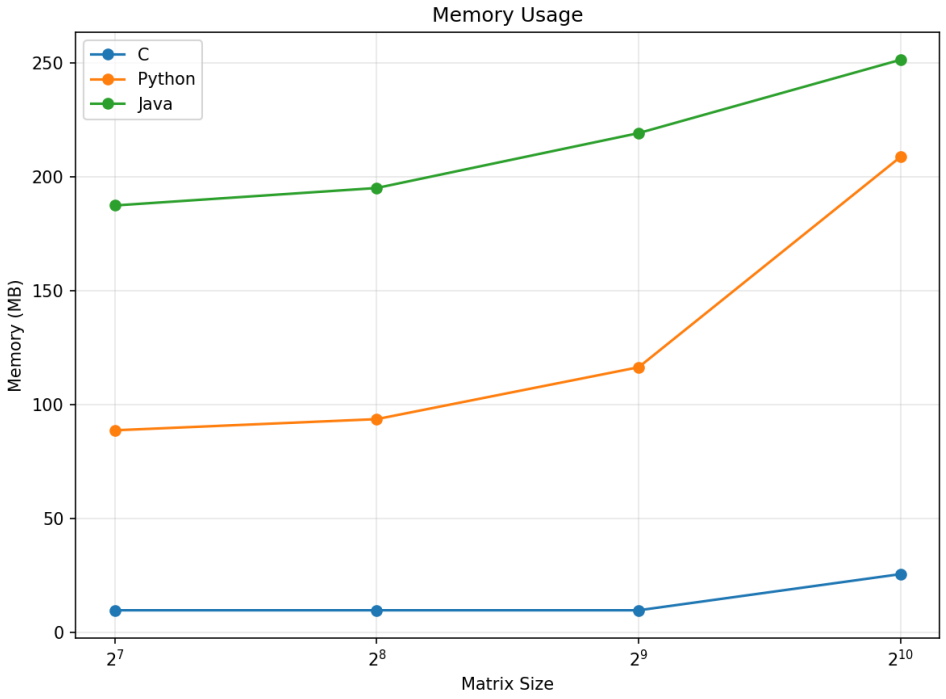

Figure 3: Memory Usage - C's efficiency vs Java's high baseline vs Python's growth

4.2 Cache Effects

Non-ideal scaling reveals memory hierarchy impact:

- Small matrices (less than or equal to 256): Fit in L2/L3 cache, minimal latency
- Medium matrices (512): Exceed L3 cache, increased memory traffic
- Large matrices (1024+): Memory-bound, DRAM latency dominant

Naive ijk loop order exhibits poor cache locality. Loop reordering (ikj) or blocking could improve performance 2-4×.

5 Conclusions

Key Findings:

1. Performance hierarchy: C, Java (2-3× slower), Python (60-135× slower)
2. Memory efficiency: C (minimal), Python (moderate), Java (high baseline)
3. All implementations scale as $O(n^3)$ with cache-induced deviations
4. Language selection requires balancing performance vs development velocity

Recommendations:

- **Java:** Production systems needing performance + maintainability, JVM ecosystem integration
- **Python:** Prototyping, data exploration, when using optimized libraries (NumPy)
- **C/C++:** Performance-critical components, HPC, building low-level libraries

5.1 Future Work

Algorithmic: Cache blocking, Strassen's algorithm, loop reordering for better locality.

Parallelization: Multi-threading, SIMD vectorization, GPU acceleration (50-100× potential speedup).

Libraries: Compare against optimized BLAS implementations (Intel MKL, OpenBLAS).

Extended benchmarks: Additional languages (Rust, Julia), sparse matrices, larger datasets, power consumption analysis.

5.2 Lessons Learned

Benchmarking requires statistical rigor (warmup, multiple iterations), awareness of tool limitations, and understanding that library selection often matters more than raw language performance. Production systems should use optimized libraries rather than naive implementations.

Acknowledgments

Completed for Big Data course at ULPGC. Development assistance from AI tools for memory measurement implementation, C syntax patterns, and documentation preparation.

A Tool Limitations

- Java JAR execution failed, used IntelliJ JMH plugin
- Manual CSV extraction for java due to jar not being able to work
- JMH lacks native memory profiling, required OSHI integration
- pytest-benchmark was very demanding