TASK 1. Benchmarking the Naive Matrix Multiplication Algorithm

Joel Ojeda Santana

October 2025

Github Repository: `https://github.com/joelojeda/individual-assignment-1-BD`

Abstract

This report benchmarks the naive matrix multiplication algorithm implemented in C, Java and Python. The objective is to measure and compare two metrics: execution time and memory used, across a range of matrix dimensions. Results are presented as both tables and graphs that show the evolution of time and peak memory as the matrix dimension increases.

1. Introduction

The task is to implement the naive matrix multiplication ($C = A \times B$) with straightforward code in C, Java and Python, and benchmark these implementations across a range of matrix sizes to quantify differences in execution time and peak memory usage attributable to language and runtime characteristics.

Measured metrics: - Mean Execution Time per Run (s) - Total Memory Used (MiB)

2. Dataset (matrix sizes) and memory footprint

For the Naive Matrix Multiplication algorithm, three double-precision matrices are required, and each occupy $n^2 \cdot 8$ bytes in memory, so the total memory is:

$$\text{Memory bytes} = 3 \cdot n^2 \cdot 8$$

Table 1: Theoretical memory used according to matrix sizes

n	n^2	Memory (bytes)	Memory (MiB)
256	65,536	1,572,864	1.50
512	262,144	6,291,456	6.00
1024	1,048,576	25,165,824	24.00
2048	4,194,304	100,663,296	96.00
4096	16,777,216	402,653,184	384.00

We will see that these sizes are similar to the ones measured in the benchmarks.

3. Result tables and included plots

Python: Execution time and peak memory

Table 2: Python – Mean time per run and used memory

n	Mean Time per Run (s)	Used Memory (MiB)
128	0.121	0.38
256	1.249	1.50
512	10.646	6.12
1024	118.125	26.00

C: Execution time and peak memory

Table 3: C – Mean time per run and used memory

n	Mean Time per Run (s)	Used Memory (MiB)
128	0.010	0.38
256	0.099	1.50
512	0.812	6.00
1024	10.275	24.00

Java: Execution time and peak memory

Table 4: Java – Mean time per run and used memory

n	Mean Time per Run (s)	Used Memory (MiB)
128	0.004	0.00
256	0.027	1.46
512	0.213	6.00
1024	8.588	23.23

Measured time evolution

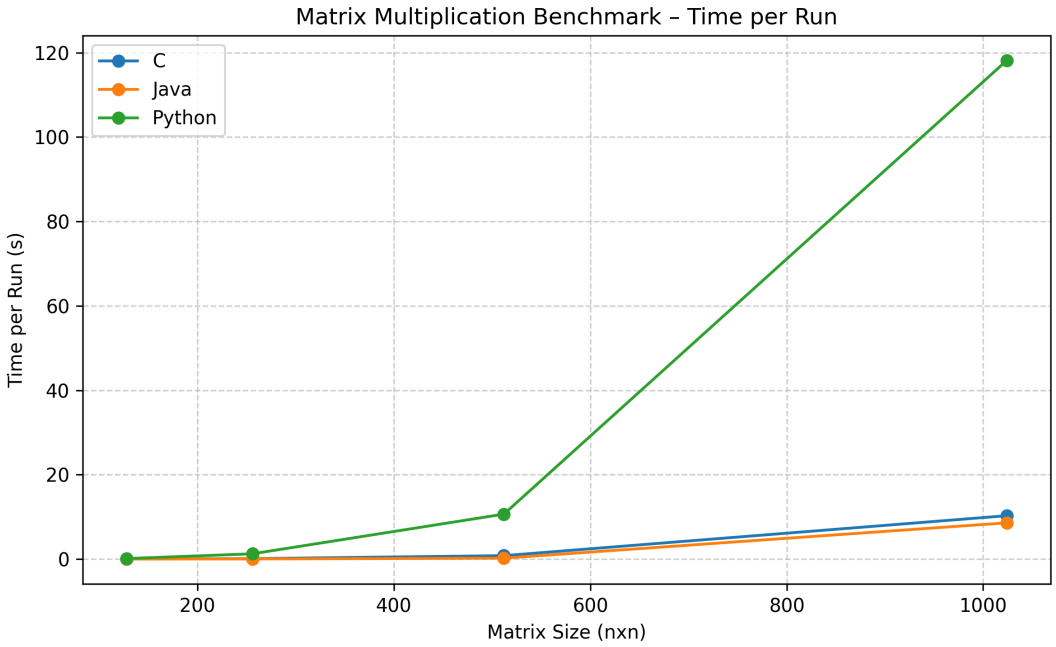

Figure 1: Time per run vs matrix dimension.

Measured peak memory evolution

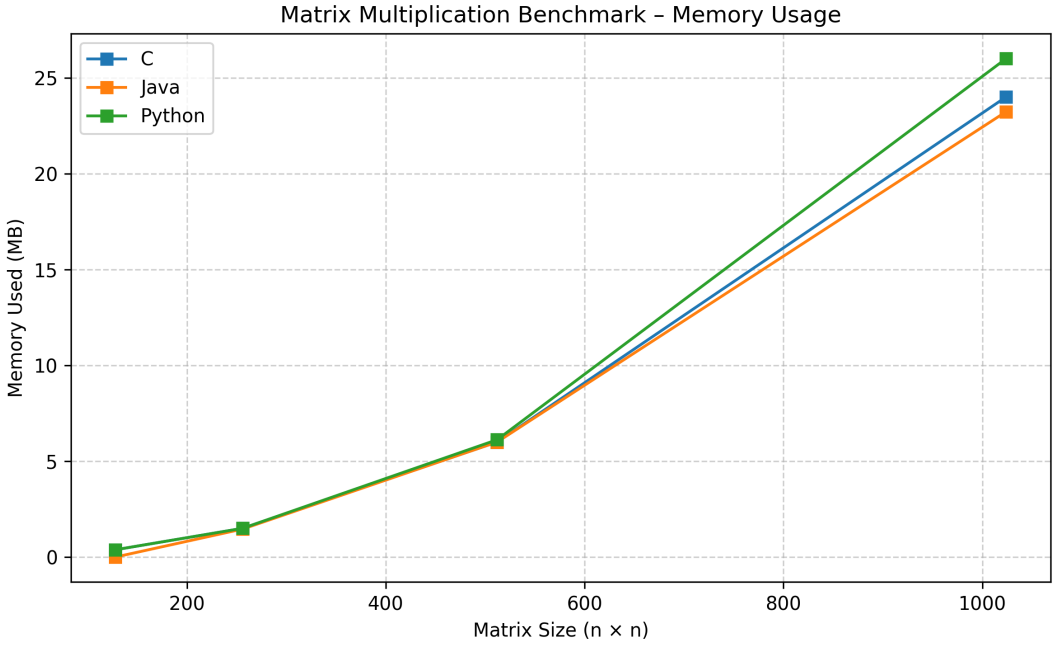

Figure 2: Memory used vs matrix dimension.

4. Analysis and interpretation

The measured runtimes generally follow the expected cubic growth of the naive triple-loop algorithm: increasing the matrix dimension produces large multiplicative increases in time consistent

with $O(n^3)$, although there is a big difference between python and the rest of languages, which was expected given that java also uses c for its compiler. The Python runtime could be improved using libraries like numpy, as it uses c instead of actual python.

In the other hand, memory measurements are follow the theoretical size that was calculated before. All three languages use similar memory, which is expected as we are implementing the same algorithm, although we can see that java and c are slightly more efficient than python in terms of memory too.

5. Conclusions and recommendations

The experiments confirm that the naive triple-loop implementation scales roughly as $O(n^3)$ and that language/runtime choice substantially affects absolute performance. The C implementation is the best choice for raw speed. Java is a good option too. Pure Python loops are suitable for prototyping but are not appropriate for applications that require fast performance.

This algorithm can be improved many ways. Reordering the loops can reduce cache misses by ensuring that data is accessed in a cache-friendly manner, allowing the processor to reuse elements already stored in memory. This effect can be further enhanced through the blocking (tiling) technique, which divides the matrices into smaller sub-blocks that fit within the CPU cache, enabling repeated use of data during intermediate computations and significantly boosting performance for large matrices. Additionally, the inherently independent nature of matrix multiplication makes it well-suited for parallelization, where the computation of different rows or blocks of the result matrix can be distributed across multiple CPU cores to achieve substantial reductions in execution time.