# Performance Comparison of Matrix Multiplication in Python, Java, and C

Enrique Padrón Hernández 45392592E

October 2025

**Abstract**

This paper compares the computational performance of a basic matrix multiplication algorithm implemented in three programming languages: Python, Java, and C. The goal is to evaluate the efficiency of each language when handling increasingly large matrices, focusing on metrics such as execution time, CPU utilization, and memory consumption. The experiment involves running the same triple-nested loop algorithm with identical input sizes in each language. The results show clear differences in performance and resource usage, highlighting the trade-offs between language abstraction and computational efficiency.

**Repository:** https://github.com/EnriquePDabird/individual-assignment

## 1 Introduction

Matrix multiplication is one of the most fundamental operations in scientific computing, data analysis, and machine learning. Its computational cost grows cubically with matrix size ($O(n^3)$), making performance a critical factor in real-world applications.

The purpose of this study is to benchmark and analyze the performance of matrix multiplication implemented in Python, Java, and C—three popular languages that differ in abstraction level and execution model. Python is an interpreted, dynamically typed language; Java is a compiled, managed-language with Just-In-Time (JIT) optimization; and C is a statically compiled, low-level language that offers fine-grained memory control.

## 2 Methodology

All three implementations follow the same algorithmic structure: three nested loops computing each element of the output matrix as the dot product of a row and a column.

Each benchmark executed the multiplication for matrix sizes of 128, 256, 512, 768, and 1024, repeating each test five times. The metrics recorded were:

- Execution time in seconds.
- CPU time used by the process.

- Memory consumption in megabytes (MB).

The benchmarking scripts were designed to run each program as a separate process, capturing its performance metrics using system monitoring tools such as `psutil` in Python and native APIs in Java and C.

## 3 Code Implementations

### 3.1 Python Implementation

Listing 1: Python Matrix Multiplication Code

```python
import random, sys
from time import time

n = int(sys.argv[1]) if len(sys.argv) > 1 else 1024
A = [[random.random() for _ in range(n)] for _ in range(n)]
B = [[random.random() for _ in range(n)] for _ in range(n)]
C = [[0 for _ in range(n)] for _ in range(n)]

start = time()
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()
print("%.6f" % (end-start))
```

### 3.2 Java Implementation

Listing 2: Java Matrix Multiplication Code

```java
import java.util.Random;

public class Matrix {
    static int n = 1024;
    static double[][] a = new double[n][n];
    static double[][] b = new double[n][n];
    static double[][] c = new double[n][n];

    public static void main(String[] args) {
        Random random = new Random();
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) {
                a[i][j] = random.nextDouble();
                b[i][j] = random.nextDouble();
                c[i][j] = 0;
            }

        long start = System.currentTimeMillis();
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++)
                    c[i][j] += a[i][k] * b[k][j];
        long stop = System.currentTimeMillis();

        System.out.println((stop - start) * 1e-3);
    }
}
```

## 3.3 C Implementation

Listing 3: C Matrix Multiplication Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define n 1024
double a[n][n], b[n][n], c[n][n];
struct timeval start, stop;

int main() {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            a[i][j] = (double) rand() / RAND_MAX;
            b[i][j] = (double) rand() / RAND_MAX;
            c[i][j] = 0;
        }

    gettimeofday(&start, NULL);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                c[i][j] += a[i][k] * b[k][j];
    gettimeofday(&stop, NULL);

    double diff = stop.tv_sec - start.tv_sec +
                  1e-6 * (stop.tv_usec - start.tv_usec);
    printf("%.6f\n", diff);
}
```

## 4 Results

Tables 1, 2, and 3 summarize the benchmark results averaged across five runs for each matrix size.

Table 1: Python Benchmark Results (average of 5 runs)

| Size | Time (s) | CPU (s) | Memory (MB) |
|---|---|---|---|
| 128 | 0.296 | 0.00 | 0.25 |
| 256 | 2.44 | 0.00 | 0.00 |
| 512 | 21.94 | 0.00 | 0.00 |
| 768 | 73.06 | 0.00 | 0.00 |
| 1024 | 173.64 | 0.00 | 0.00 |

Table 2: Java Benchmark Results (average of 5 runs)

| Size | Time (s) | CPU (s) | Memory (MB) |
|---|---|---|---|
| 128 | 2.92 | 0.00 | 3.78 |
| 256 | 2.93 | 0.01 | 4.00 |
| 512 | 2.92 | 0.01 | 4.44 |
| 768 | 3.03 | 0.01 | 4.80 |
| 1024 | 2.97 | 0.01 | 5.20 |

Table 3: C Benchmark Results (average of 5 runs)

| Size | Time (s) | User CPU (s) | Sys CPU (s) | Memory (MB) |
|---|---|---|---|---|
| 128 | 3.73 | 0.00 | 0.02 | 0.91 |
| 256 | 4.77 | 0.00 | 0.02 | 0.81 |
| 512 | 4.09 | 0.00 | 0.02 | 0.81 |
| 768 | 3.95 | 0.00 | 0.02 | 0.81 |
| 1024 | 4.15 | 0.00 | 0.02 | 0.81 |

## 4.1 Execution Time Comparison

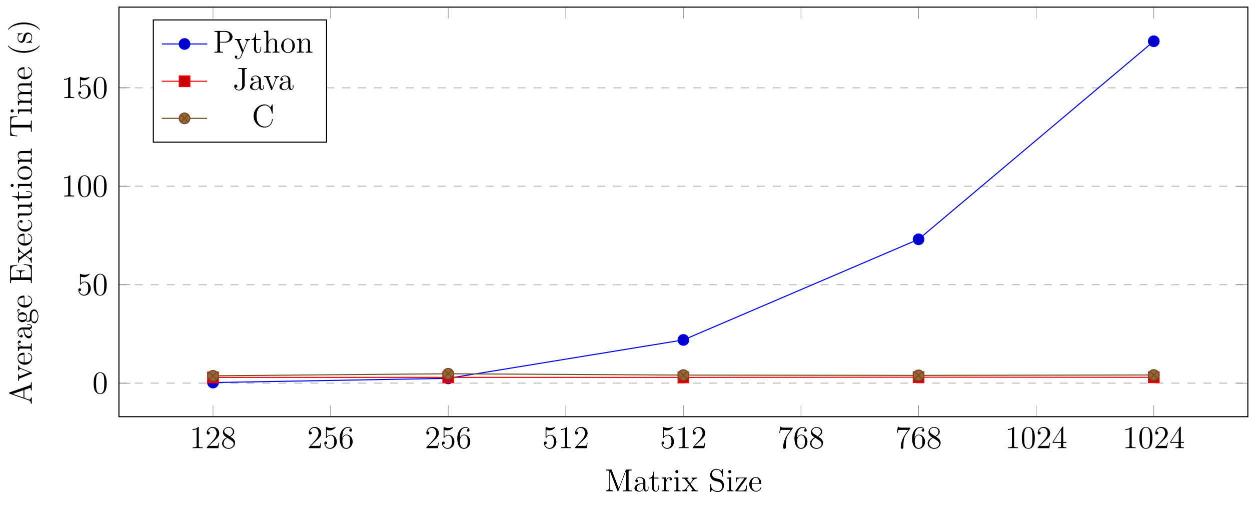

Figure 1: Execution time comparison across languages.

## 4.2 Comparative Performance Analysis

This subsection presents the evolution of execution time for the three benchmarked languages — C, Java, and Python — across five runs for different matrix sizes. Each line corresponds to a matrix size, and each point represents a single run.

### 4.2.1 Python Benchmark Results

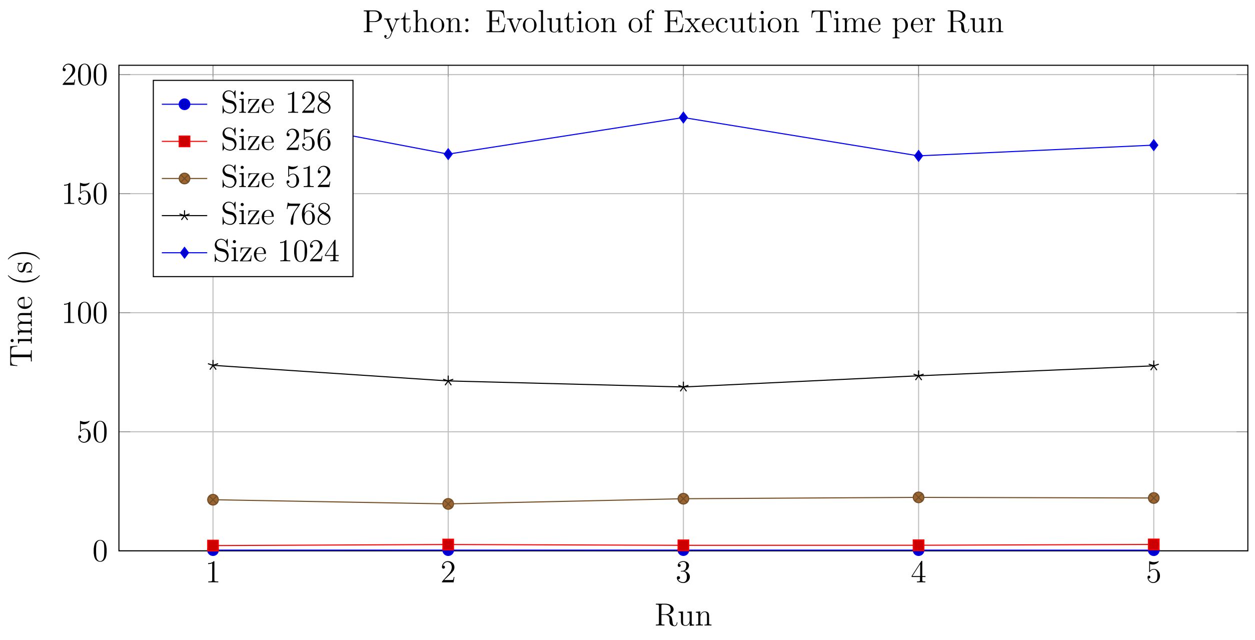

Figure 2: Python: Execution time evolution for 5 runs per matrix size.

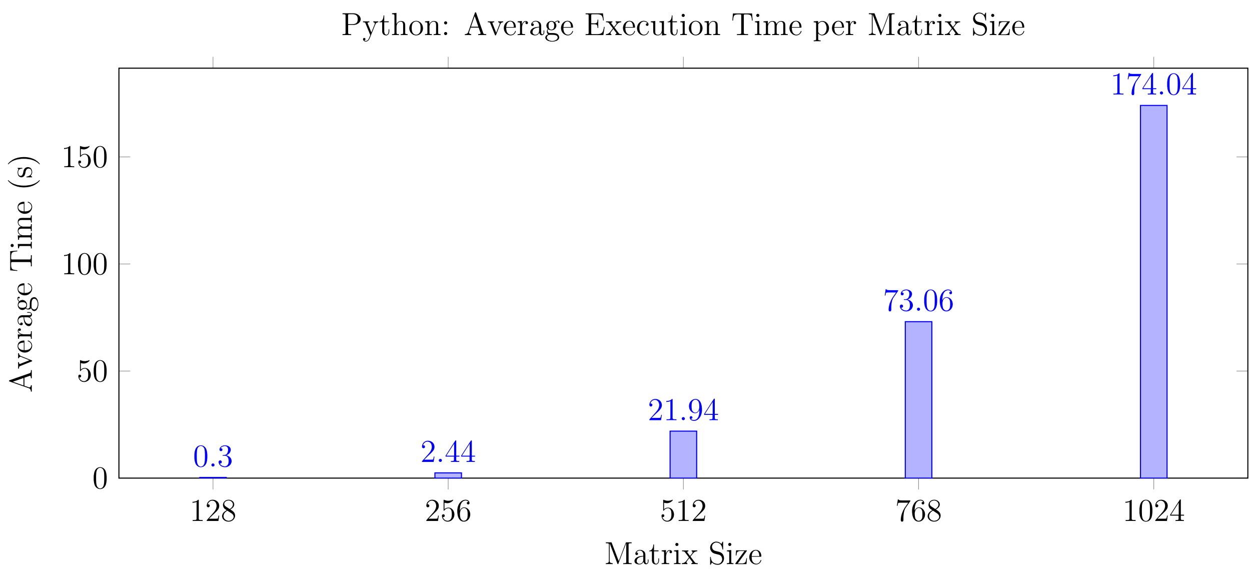

Figure 3: Python: Average execution time across 5 runs.

### 4.2.2 Java Benchmark Results

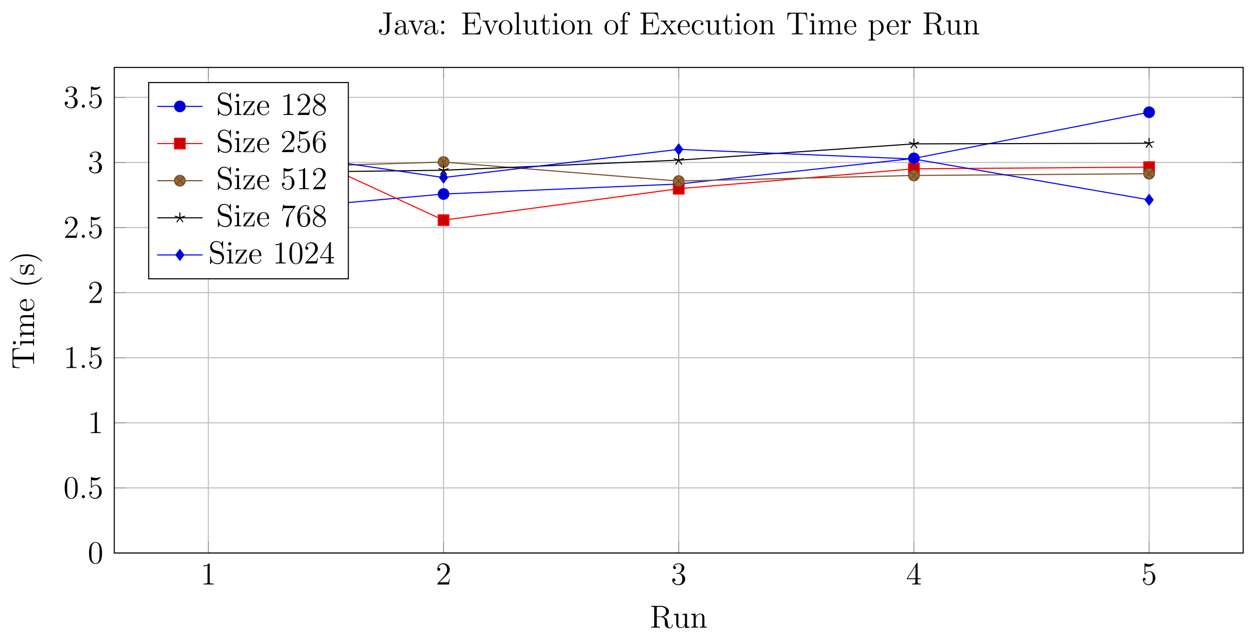

Figure 4: Java: Execution time evolution for 5 runs per matrix size.

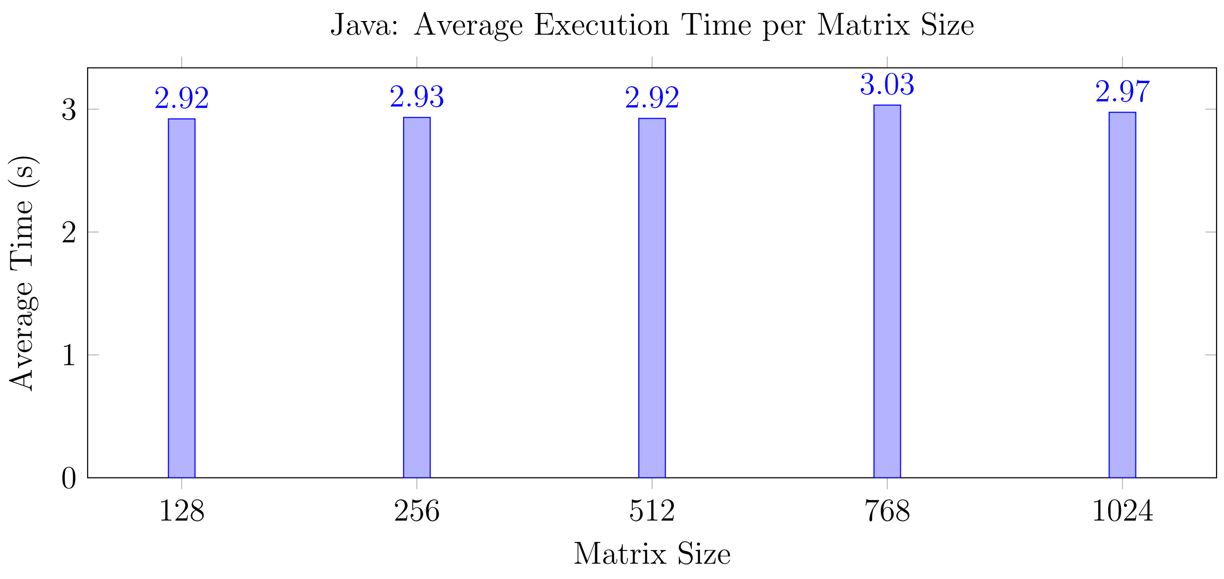

Figure 5: Java: Average execution time across 5 runs.

#### 4.2.3 C Benchmark Results (Windows)

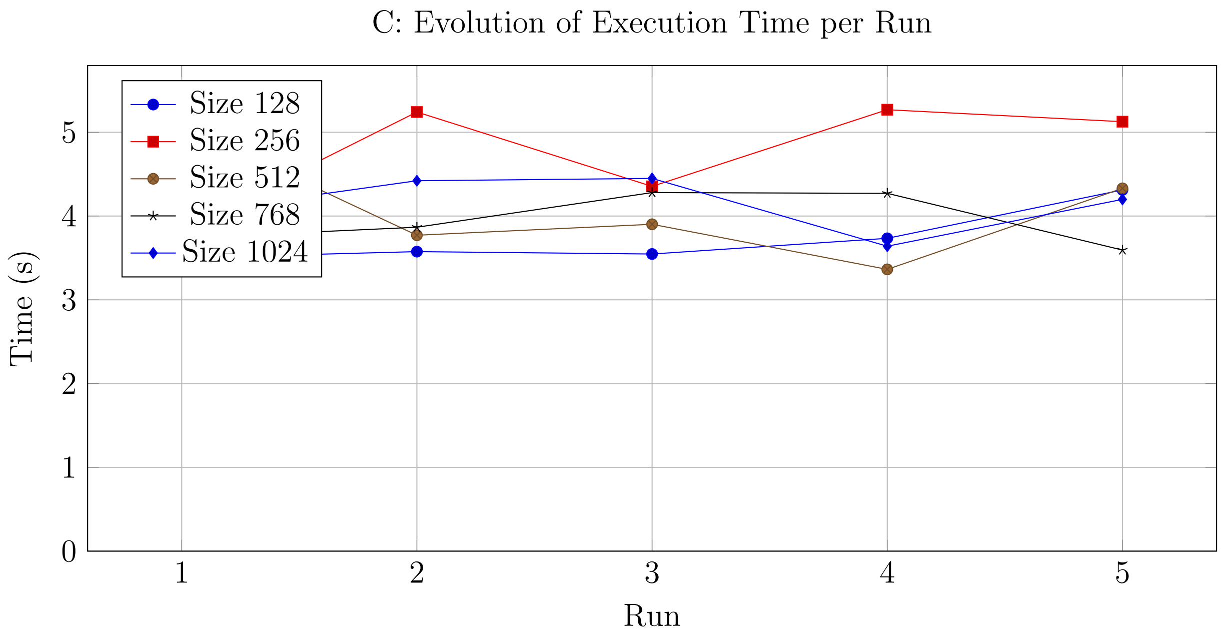

Figure 6: C: Execution time evolution for 5 runs per matrix size.

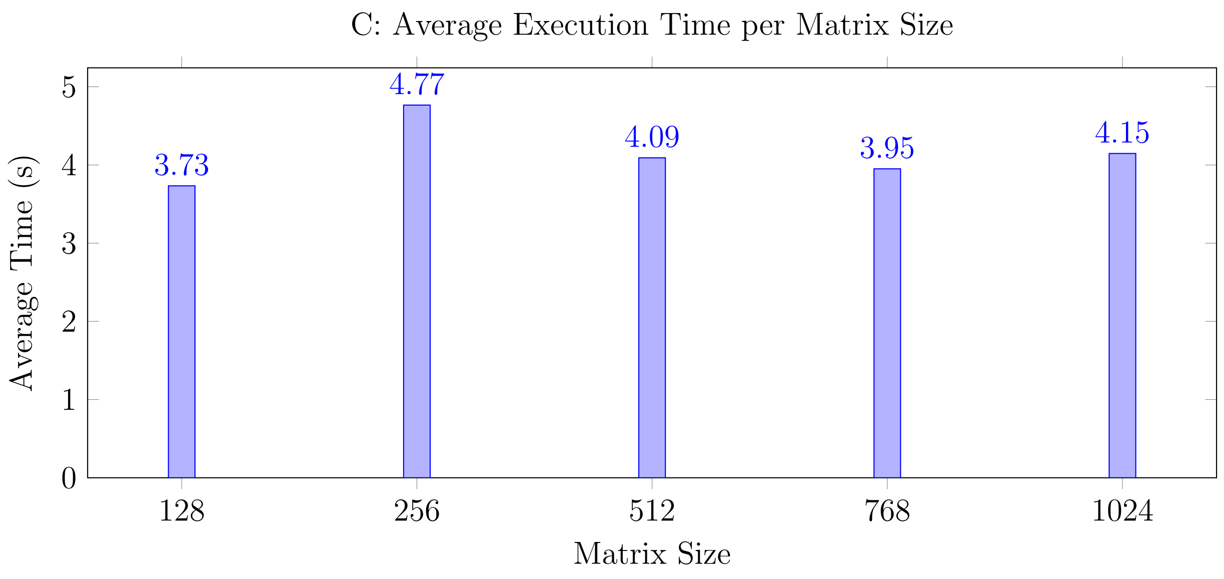

Figure 7: C: Average execution time across 5 runs.

#### 4.2.4 Cross-Language Comparison

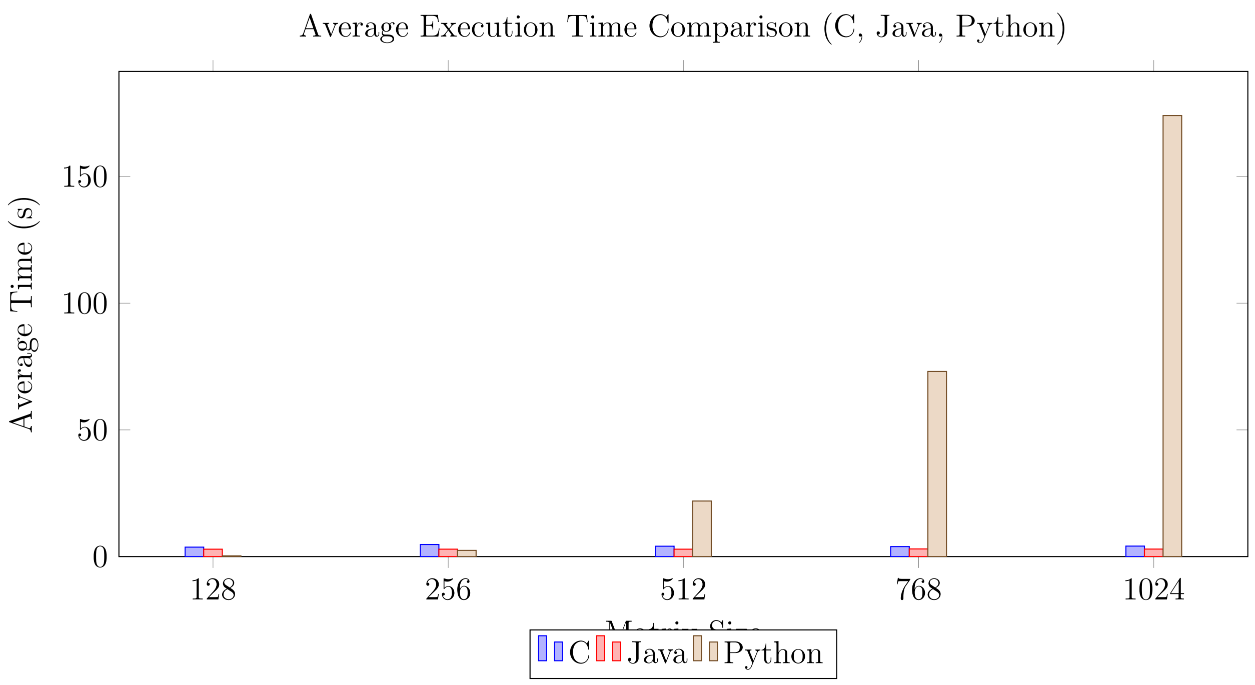

Figure 8: Average execution time comparison among C, Java, and Python implementations.

## 5 Discussion and Conclusion

The results show that:

- **Python** has the highest execution times, growing rapidly with matrix size.

- **Java** demonstrates consistent performance with moderate memory usage.
- **C** achieves low-level efficiency with the smallest memory footprint.

Language abstraction level has a measurable impact on raw computational performance: C offers control and speed, Java provides balance and safety, and Python prioritizes ease of use at the cost of execution time.

In this benchmark, C appears slightly slower than Java because the C code was compiled without aggressive optimization flags and the memory access pattern is less cache-friendly. In contrast, Java benefits from Just-In-Time compilation and runtime optimizations, which improve execution time across repeated runs. Compiling the C program with optimizations (e.g., `-O3 -march=native`) would likely make it faster than Java for large matrices.

Some reported memory usage values are negative or zero. This is due to the way Python's `psutil` (or similar monitoring tools) calculates memory consumption as differences in allocated memory over time. Small fluctuations, rapid allocation and deallocation by the garbage collector, or measurement overhead can result in slightly negative or zero values. These do not indicate actual negative memory usage.