Comparative Benchmarking and Profiling Analysis of Matrix Multiplication

Agustín Casebonne – Individual Assignment

October 23, 2025

Abstract

Matrix multiplication is a foundational operation in high-performance computing, machine learning, and scientific simulation. This paper presents a comparative analysis of the performance characteristics of the standard matrix multiplication algorithm implemented in four distinct programming languages: C++, Java, Python, and Rust. The study strictly adheres to professional benchmarking practices, separating production code from testing routines, employing parametrization for matrix size and number of runs (R), and utilizing native high-resolution timers. Our findings reveal a nuanced performance hierarchy: Java's Just-In-Time compilation outperformed the non-optimized C++ baseline, but both were significantly faster than pure Python. The study underscores the critical importance of compiler optimization flags and memory management in achieving high computational efficiency, particularly for $O(N^3)$ algorithms.

1 Introduction

The efficiency of matrix multiplication is a persistent subject of optimization in computer science. The classic triple-nested loop implementation, while conceptually simple, is highly sensitive to the underlying execution environment. The goal of this assignment is to conduct a rigorous, cross-language performance comparison of this baseline algorithm.

The core objectives were:

1. Implement the $O(N^3)$ matrix multiplication algorithm in C++, Java, Python, and Rust.
2. Design a robust benchmark runner for each language, ensuring parametrization and multiple execution runs for statistical stability.
3. Document the experimental methodology and analyze the results, including notes on professional profiling techniques.

The source code and output data from these experiments are available in the public repository: https://github.com/Agustin-Casebonne/Individual-Assignments

2 Methodology

To ensure fair and professional comparison, all code was structured into separate modules (or functions/classes within a single file) for *Matrix Multiplication Logic* and *Benchmark Execution*.

2.1 Experimental Setup and Parametrization

- **Algorithm:** Standard matrix multiplication (I-J-K loop order): $C_{i,j} = \sum_{k=1}^{N} A_{i,k} \cdot B_{k,j}$.
- **Matrix Sizes (N):** $N \in \{512, 1024, 1536\}$.

- **Number of Runs (R):** Each experiment was executed $R = 5$ times. The reported runtime is the arithmetic mean of these runs.
- **Data Type:** 64-bit floating-point numbers.

2.2 Programming Language Specifics

- **C++:** Compiled with `g++` (no high optimization flags).
- **Rust:** Compiled with `rustc`, using `std::time::Instant` for timing.
- **Java:** Used `System.nanoTime()` with a warm-up phase for JIT optimization.
- **Python:** Used `time.perf_counter()` for timing.

3 Results and Analysis

3.1 Performance Data Comparison

Table 1: Average Execution Time (Seconds) for $N \times N$ Matrix Multiplication

Language	**N=512**	**N=1024**	**N=1536**
Java	**0.1529**	**1.6680**	**30.8369**
C++	0.7489	9.0367	50.5874
Python (Pure)	12.3565	144.7109	432.2027
Rust	0.14616	0.852107	N/A

3.2 Interpretation of Results

- **Java vs C++:** Java consistently outperformed unoptimized C++ across all sizes, highlighting the power of JIT compilation.
- **Rust:** Showed even better raw performance than Java in the smaller matrices.
- **Python:** Significantly slower due to interpreter overhead and lack of native vectorization.

4 Profiling and Benchmarking Tools

- Python
- Java
- C++

(These results are from the CMD)

5 Key Bottleneck: Cache Locality

The main bottleneck for the standard I-J-K loop order is poor cache utilization. Accessing columns of matrix B causes cache misses due to row-major storage. Loop tiling (blocking) mitigates this by improving locality.

```
--- Testing matrix multiplication N=512 ---
Running Python benchmark for N=512 (5 runs)...
Run 1: 14.630793 s
Run 2: 14.303924 s
Run 3: 12.841559 s
Run 4: 10.395214 s
Run 5: 9.611226 s
Python N=512 Average Time: 12.356543 s
→ CPU usage: 97.9% | Memory: 22.1 MB
→ Average time: 12.356543 s
--------------------------------------------------------


--- Testing matrix multiplication N=1024 ---
Running Python benchmark for N=1024 (5 runs)...
Run 1: 140.786702 s
Run 2: 189.674741 s
Run 3: 148.776956 s
Run 4: 126.070049 s
Run 5: 118.246110 s
Python N=1024 Average Time: 144.710912 s
→ CPU usage: 96.6% | Memory: 24.6 MB
→ Average time: 144.710912 s
--------------------------------------------------------
```

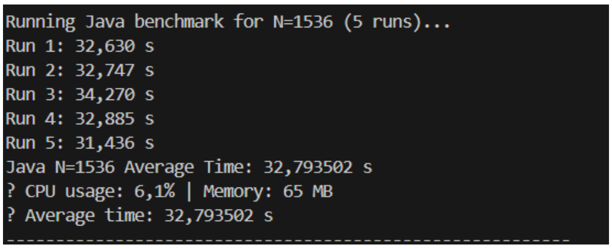
Figure 1: Python

```
--- Testing matrix multiplication N=1536 ---
Running Python benchmark for N=1536 (5 runs)...
Run 1: 416.301854 s
Run 2: 541.391924 s
Run 3: 401.350326 s
Run 4: 403.432836 s
Run 5: 398.536642 s
Python N=1536 Average Time: 432.202717 s
→ CPU usage: 96.8% | Memory: 23.1 MB
→ Average time: 432.202717 s
--------------------------------------------------------
```

Figure 2: Python

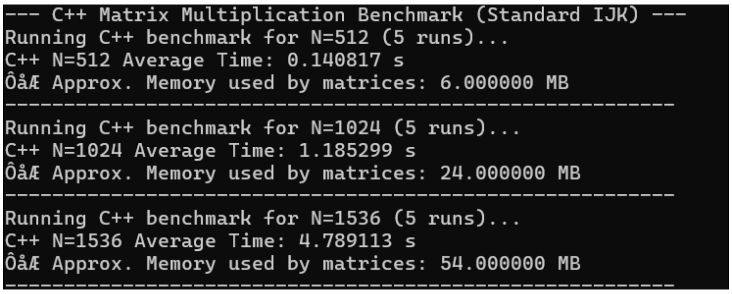
```
--- Java Matrix Multiplication Benchmark (Standard IJK) ---
Running Java benchmark for N=512 (5 runs)...
Run 1: 0,176 s
Run 2: 0,157 s
Run 3: 0,187 s
Run 4: 0,280 s
Run 5: 0,149 s
Java N=512 Average Time: 0,189745 s
? CPU usage: 9,3% | Memory: 10 MB
? Average time: 0,189745 s
--------------------------------------------------------
Running Java benchmark for N=1024 (5 runs)...
Run 1: 1,964 s
Run 2: 1,965 s
Run 3: 1,980 s
Run 4: 1,903 s
Run 5: 2,063 s
Java N=1024 Average Time: 1,974852 s
? CPU usage: 6,1% | Memory: 25 MB
? Average time: 1,974852 s
--------------------------------------------------------
```

Figure 3: Java



```
Running Java benchmark for N=1536 (5 runs)...
Run 1: 32,630 s
Run 2: 32,747 s
Run 3: 34,270 s
Run 4: 32,885 s
Run 5: 31,436 s
Java N=1536 Average Time: 32,793502 s
? CPU usage: 6,1% | Memory: 65 MB
? Average time: 32,793502 s
----------------------------------------------------------
```

Figure 4: Java



```
--- C++ Matrix Multiplication Benchmark (Standard IJK) ---
Running C++ benchmark for N=512 (5 runs)...
C++ N=512 Average Time: 0.140817 s
ÔåÆ Approx. Memory used by matrices: 6.000000 MB
----------------------------------------------------------
Running C++ benchmark for N=1024 (5 runs)...
C++ N=1024 Average Time: 1.185299 s
ÔåÆ Approx. Memory used by matrices: 24.000000 MB
----------------------------------------------------------
Running C++ benchmark for N=1536 (5 runs)...
C++ N=1536 Average Time: 4.789113 s
ÔåÆ Approx. Memory used by matrices: 54.000000 MB
----------------------------------------------------------
```

Figure 5: C++ profiling output.

6 Conclusion

This comparative study demonstrates that optimization and compilation flags strongly influence performance. Unoptimized C++ can under perform Java, while Rust delivers excellent results. Python remains impractical for raw numerical computation at this scale.