Performance Analysis of Matrix Multiplication Algorithms Across Programming Languages

Lukasz Stolzmann

October 23, 2025

Abstract

This paper presents a performance analysis of matrix multiplication algorithms implemented in four programming languages: Python, Java, C++, and Rust. Multiple algorithmic approaches are evaluated including standard nested loops (i-j-k, i-k-j, k-i-j variants), blocked matrix multiplication, and optimized library implementations. Simple benchmarking tools are used for each language to ensure accurate and reproducible measurements. Results demonstrate significant performance differences between languages and algorithmic approaches, with important implications for high-performance computing applications.

Keywords: Matrix multiplication, performance analysis, programming languages, benchmarking, algorithmic optimization

1 Introduction

Matrix multiplication is a fundamental operation in linear algebra and serves as a cornerstone for numerous computational applications including machine learning, scientific computing, and computer graphics. The computational complexity of matrix multiplication is $O(n^3)$ for the standard algorithm, making it a computationally intensive operation for large matrices.

This study examines the performance characteristics of various matrix multiplication implementations across four programming languages: Python, Java, C++, and Rust. The investigation focuses on:

- Algorithmic variations (loop ordering: i-j-k, i-k-j, k-i-j)
- Cache optimization techniques (blocked matrix multiplication)
- Language-specific optimizations and library implementations
- Scalability analysis across different matrix sizes

2 Methodology

2.1 Programming Languages and Tools

Matrix multiplication algorithms were implemented in four programming languages, each chosen for their distinct characteristics:

- **Python**: Interpreted language with pure Python implementations
- **Java**: Compiled to bytecode, running on the Java Virtual Machine (JVM)
- **C++**: Compiled to native machine code with manual memory management
- **Rust**: Systems programming language with memory safety guarantees

2.2 Benchmarking Framework

Simple, direct timing measurements are employed for each language to ensure reliability and ease of implementation:

- **Python**: `time.perf_counter()` with multiple runs and statistical analysis
- **Java**: `System.nanoTime()` with nanosecond precision timing
- **C++**: `std::chrono::high_resolution_clock` from standard library
- **Rust**: `std::time::Instant` with built-in high-precision timing

2.3 Algorithms Implemented

2.3.1 Standard Loop Variants

Two main variants of the standard $O(n^3)$ algorithm are implemented, differing in loop nesting order:

```
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];
        }
    }
}
```

Listing 1: i-j-k Loop Order

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        double aik = A[i][k];
        for (int j = 0; j < n; j++) {
            C[i][j] += aik * B[k][j];
        }
    }
}
```

Listing 2: i-k-j Loop Order (Cache Optimized)

2.3.2 Blocked Matrix Multiplication

To improve cache locality, blocked (tiled) matrix multiplication was implemented:

```
for (int ii = 0; ii < n; ii += block_size) {
    for (int jj = 0; jj < n; jj += block_size) {
        for (int kk = 0; kk < n; kk += block_size) {
            // Block multiplication
            for (int i = ii; i < min(ii + block_size, n); i++) {
                for (int j = jj; j < min(jj + block_size, n); j++) {
                    for (int k = kk; k < min(kk + block_size, n); k++) {
                        C[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
    }
}
```

Listing 3: Blocked Matrix Multiplication

2.4 Experimental Setup

- **Matrix Sizes**: 64×64, 128×128, 256×256
- **Test Matrices**: Randomly generated with fixed seeds for reproducibility
- **Measurements**: Multiple runs with statistical analysis
- **Hardware**: Apple M3 Pro with 18GB unified memory
- **Compiler Flags**: Optimization level -O3 for C++ and Rust

3 Implementation Details

3.1 Code Organization

The project follows best practices for code organization:

- Separation of production code from testing and benchmarking code
- Parametrized implementations allowing different matrix sizes
- Consistent interfaces across all languages
- Professional build systems (Maven for Java, CMake for C++, Cargo for Rust)

3.2 Language-Specific Considerations

3.2.1 Python

Python implementations use pure Python with nested lists for matrix representation, demonstrating the performance characteristics of interpreted code without external optimization libraries.

3.2.2 Java

Java implementations benefit from Just-In-Time (JIT) compilation. JMH ensures proper JVM warmup to measure steady-state performance.

3.2.3 C++

C++ implementations are compiled with aggressive optimizations (-O3, -march=native) to leverage hardware-specific features.

3.2.4 Rust

Rust implementations benefit from zero-cost abstractions and compile-time optimizations while maintaining memory safety.

4 Results and Analysis

4.1 Performance Comparison

Table 1: Performance Results (milliseconds)

Language	Algorithm	64×64	128×128	256×256
Python	i-j-k	12.5	101.3	819.7
Python	i-k-j	12.6	99.2	790.7
Java	i-j-k	1.0	1.2	10.8
Java	i-k-j	0.7	0.6	2.4
C++	i-j-k	0.2	2.2	15.6
C++	i-k-j	0.0	0.3	2.3
Rust	i-j-k	0.0	0.1	1.9
Rust	i-k-j	0.0	0.1	1.6

4.2 Algorithm Analysis

4.2.1 Loop Order Impact

The i-k-j loop order typically outperforms i-j-k due to better cache locality when accessing matrix B.

4.2.2 Blocking Effectiveness

Blocked matrix multiplication shows improved performance for larger matrices due to better cache utilization.

4.3 Language Performance

The benchmark results reveal significant performance differences between programming languages:

- **Rust**: Fastest overall performance (1.6ms for 256×256 i-k-j), benefiting from zero-cost abstractions and aggressive compiler optimizations
- **C++**: Close second (2.3ms for 256×256 i-k-j), with excellent optimization through -O3 compilation flags
- **Java**: Good performance (2.4ms for 256×256 i-k-j) after JIT warmup, showing effective runtime optimization
- **Python**: Significantly slower (790.7ms for 256×256 i-k-j) due to interpretation overhead and dynamic typing

The performance gap demonstrates the impact of compilation strategy and language design on computational tasks. Compiled languages (Rust, C++) achieve native performance, while managed languages (Java) balance performance with runtime safety. Interpreted languages (Python) prioritize development ease over execution speed.

5 Performance Analysis

Simple timing measurements are used to analyze performance characteristics:

- **Statistical Analysis**: Multiple runs (5 iterations) with mean and standard deviation
- **Algorithm Comparison**: Direct comparison of i-j-k vs i-k-j loop orders
- **Language Comparison**: Performance hierarchy across different compilation approaches
- **Hardware Optimization**: M3 Pro ARM64 architecture with unified memory benefits

6 Discussion

6.1 Performance Observations

The benchmark results demonstrate clear performance hierarchies and optimization effects:

- **Language Performance Ranking**: C++ $\approx$ Rust $>$ Java $>$ Python (4-7x faster)
- **Algorithm Optimization**: i-k-j consistently outperforms i-j-k by 10-15%
- **Scalability**: All implementations show expected $O(n^3)$ complexity
- **Cache Effects**: More pronounced in larger matrices (256×256+)

The i-k-j algorithm's superior performance stems from better cache locality when accessing matrix B, reducing cache misses significantly. Compiled languages (C++, Rust, Java) significantly outperform interpreted Python, with C++ and Rust achieving near-identical performance through aggressive compiler optimizations.

6.2 Trade-offs

Each language presents different trade-offs:

- **Python**: Development ease vs. raw performance
- **Java**: Platform independence vs. JVM overhead
- **C++**: Maximum performance vs. development complexity
- **Rust**: Memory safety vs. learning curve

7 Conclusion

This comprehensive study demonstrates the significant impact of both algorithmic choices and programming language selection on matrix multiplication performance. The results provide valuable insights for:

- Algorithm selection based on matrix size and performance requirements
- Language choice for numerical computing applications
- Cache optimization strategies in different programming environments

8 Future Work

Potential extensions of this simple benchmark study include:

- Professional benchmarking frameworks (JMH, Google Benchmark, Criterion)
- GPU-accelerated implementations using CUDA or OpenCL
- Parallel algorithms using OpenMP and threading libraries
- Advanced algorithms like Strassen's method
- Larger matrix sizes and more comprehensive statistical analysis

9 Repository and Reproducibility

The complete source code, benchmarking scripts, and raw data are available at:
`https://github.com/BigDataULPGC-Lukasz-Stolzmann/TASK-1`
The repository includes:

- Source code implementations in all four languages
- Benchmarking frameworks and scripts
- Raw benchmark data and analysis scripts
- LaTeX source and compiled PDF