# Documentation
# Language Benchmark of matrix multiplication 2

Olga Kalisiak 00325481

November 9, 2024

Github repository: link

## Contents

# 1 Abstract

This project explores matrix multiplication performance across different approaches in Java and Python, focusing on standard, blocked, and sparse matrix multiplication. Matrix multiplication is crucial in areas like machine learning and scientific computing, where efficiency directly impacts performance, especially for large or sparse matrices. The Java tests included three methods: the Standard method, which computes all elements without optimization for sparsity; the Blocked method, which leverages CPU cache by processing matrices in smaller blocks; and the Sparse method, designed to skip zero elements, optimizing memory and processing time for high-sparsity matrices.

For the sparse matrix benchmark, I compared Java's sparse implementation against Python's handling of sparse matrices using SciPy. The matrix mc2depi from the TAMU SuiteSparse Matrix Collection was used in both languages to ensure consistency. Results indicated that Python outperformed Java in sparse matrix multiplication due to SciPy's optimized data structures and specialized functions. The Sparse method in Java was highly efficient for sparse matrices but underperformed for dense matrices due to its sparsity-specific optimizations. Conversely, the Blocked method provided stable, reliable performance across various sparsity levels, making it the most versatile choice in Java.

The findings highlight that while Python's SciPy library is ideal for sparse matrices due to its efficiency and low overhead, Java's flexibility can be beneficial, especially when using optimized methods tailored to specific matrix types. The project underscores the importance of choosing the appropriate language and method based on matrix properties, and suggests future research into parallelized and distributed matrix multiplication to further enhance performance.

# 2 Introduction

Matrix multiplication is a fundamental operation in data processing, commonly applied in areas like machine learning, computer graphics, and scientific computing. The choice of programming language and the method used for performing matrix multiplication can significantly impact performance, especially when dealing with large or sparse matrices. In this project, I compare three different approaches to matrix multiplication in Java—standard multiplication, blocked multiplication and sparse matrix multiplication—and contrast the performance of Java's sparse matrix multiplication with Python's sparse matrix handling capabilities.

To provide a practical comparison, I use the sparse matrix mc2depi from the SuiteSparse Matrix Collection in both Java and Python. By focusing on execution time, this project aims to evaluate the efficiency of different multiplication strategies in Java and compare them with Python's performance in handling sparse matrices.

# 3 Problem statement

Matrix multiplication is computationally expensive, and its performance can vary widely depending on how the algorithm is implemented and the language in which it is executed. In Java, multiple optimization techniques, such as blocked algorithms, offer potential improvements in performance, particularly for large matrices. Sparse matrix multiplication introduces additional complexity, and Python, with its specialized libraries, is known for efficient sparse matrix handling.

This project seeks to determine which method performs best for large-scale matrix multiplication, with a specific focus on sparse matrices, by comparing the execution times of various approaches in Java and Python. The goal is to assess how these languages and methods handle the computational challenges posed by matrix multiplication, particularly when working with sparse datasets like mc2depi.

# 4 Methodology

To benchmark the matrix multiplication methods, I conducted tests on a machine with the following specifications:

- Laptop model: Acer Nitro 5
- Processor: AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz
- Installed RAM: 16.0 GB (available: 15.4 GB)
- Product ID: 00328-00801-21601-AA448
- System Type: 64-bit operating system, x64 processor
- Operating system: Windows 11

Languages and compilers:

- Python (version 3.10.11, PyCharm 2024.2.3)
- Java (version 20.0.1, IntelliJ IDEA Community Edition 2024.2.2)

Each matrix multiplication method in Java was tested using standard, blocked and sparse matrix multiplication. For the Python benchmarks, I focused on sparse matrix multiplication for a direct comparison with Java's sparse method. I used the sparse matrix mc2depi from the SuiteSparse Matrix Collection to ensure consistency across tests.

To measure the performance of matrix multiplication, large matrices were multiplied in each experiment, and the execution time was recorded.

## 4.1 Experiment Setup

### 4.1.1 Programming Languages and Libraries

- Python: Sparse matrix multiplication was implemented using the scipy.sparse library and benchmarked with pytest in PyCharm.
- Java: Several matrix multiplication approaches were implemented, including standard multiplication, blocked multiplication and sparse matrix multiplication. These were benchmarked using the Java Microbenchmark Harness (JMH) in IntelliJ IDEA.

## 4.2 Performance Metrics

The key metric tracked in all tests was:

- Execution Time: The runtime of each matrix multiplication operation was measured in milliseconds using appropriate benchmarking tools for each language (JMH for Java, pytest-benchmark for Python).

## 4.3 Experimental Procedure

Matrices of various sizes were used to observe how performance scales with increasing matrix dimensions. The following sizes were tested: 32x32, 64x64, 128x128, 256x256, 512x512, 1024x1024, 2048x2048 and 4096x4096. The experiments were repeated multiple times to ensure consistency in results, and the average runtime was recorded for each method.

# 5 Experiments

Codes from these experiments are available in the repository: Github repository

## 5.1 Standard matrix multiplication

The results of the execution time experiment are presented in the Table 1 and in the chart Figure 1 below.

| Size | Performance Time [ms] | |
|---|---|---|
| | Sparsity 0% | Sparsity 90% |
| 32x32 | 0.022 | 0.021 |
| 64x64 | 0.173 | 0.175 |
| 128x128 | 1.478 | 1.494 |
| 256x256 | 15.627 | 16.643 |
| 512x512 | 176.051 | 173.297 |
| 1024x1024 | 3148.291 | 3265.453 |
| 2048x2048 | 54354.326 | 52433.304 |

Table 1: Execution Time for Standard Method with Different Sparsity Levels

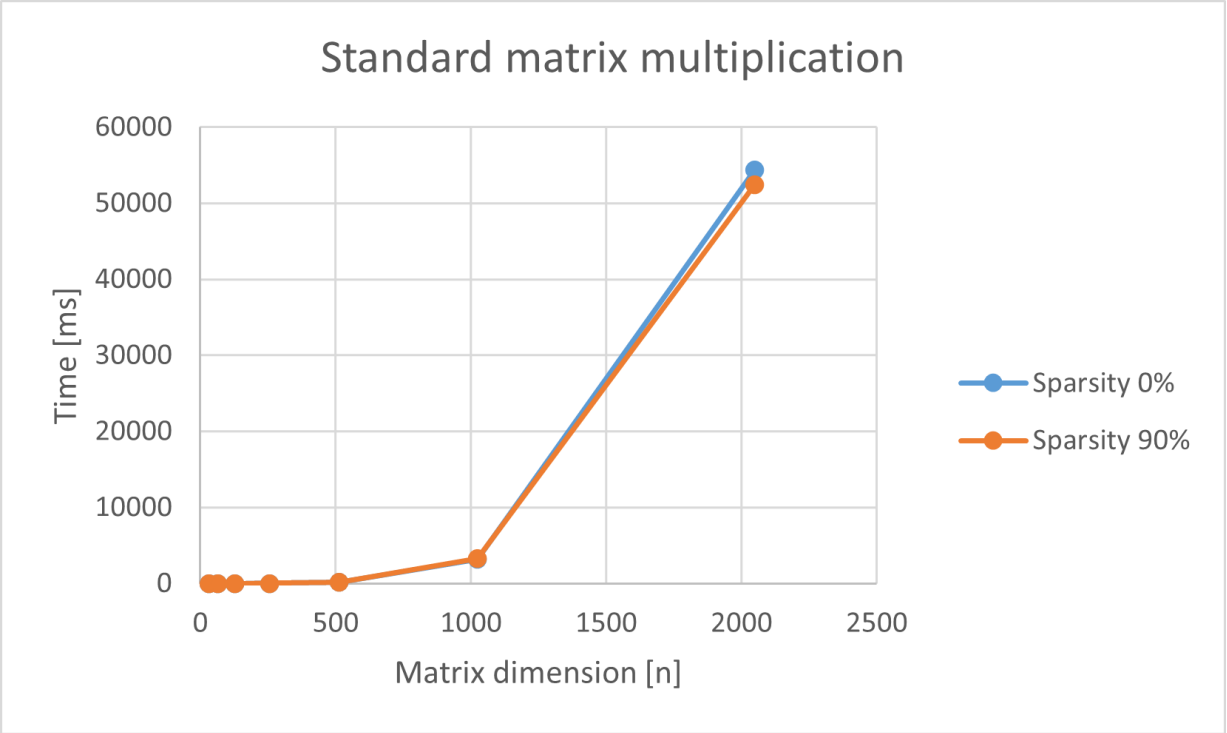


Figure 1: Standard Method - Computation Time vs Matrix Size (0% and 90% Sparsity)

### 5.1.1 Method description

The Standard method performs computations on all matrix elements without any optimizations for sparsity. This approach treats every element equally, regardless of its value, meaning that even zero elements in a sparse matrix are processed just like non-zero elements.

### 5.1.2 Results Analysis

For the Standard method, computation time increases consistently with matrix size, but it remains unaffected by the sparsity level. Since this method does not skip zero elements, it processes both sparse (90%) and dense (0%) matrices in the same way, leading to similar computation times across both sparsity levels.

## 5.2 Block matrix multiplication

The results of the execution time experiment are presented in the Table 2 and in the chart Figure 2 below.

| Size | Performance Time [ms] | |
|---|---|---|
| | Sparsity 0% | Sparsity 90% |
| 32x32 | 0.022 | 0.022 |
| 64x64 | 0.18 | 0.178 |
| 128x128 | 1.474 | 1.438 |
| 256x256 | 12.575 | 12.531 |
| 512x512 | 103.82 | 102.053 |
| 1024x1024 | 978.04 | 915.524 |
| 2048x2048 | 8212.039 | 8233.581 |
| 4096x4096 | 96138.899 | 83156.983 |

Table 2: Execution Time for Blocked Method with Different Sparsity Levels

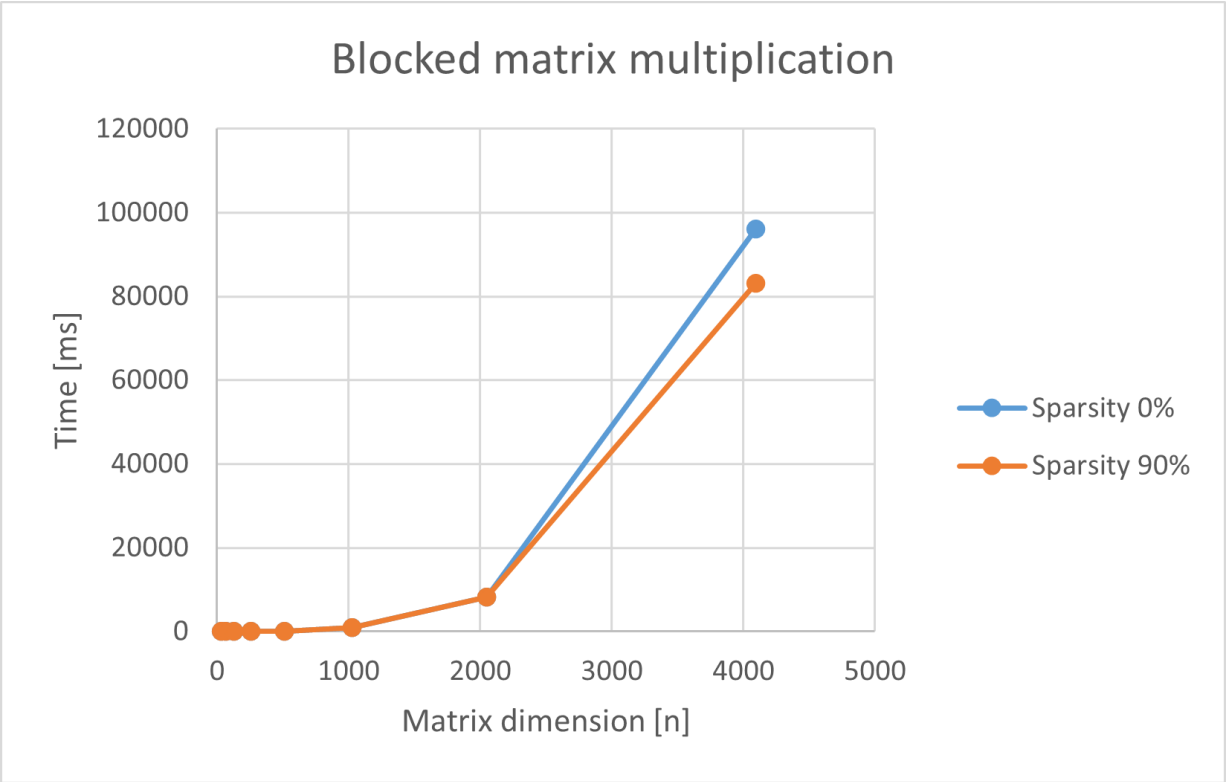


Figure 2: Blocked Method - Computation Time vs Matrix Size (0% and 90% Sparsity)

### 5.2.1 Method description

The Blocked method processes matrices in smaller blocks, which allows for more efficient use of the CPU's cache. By dividing the matrix into smaller blocks, it reduces the need for frequent memory access to the main RAM, especially beneficial for larger matrices. Each block is processed independently, which optimizes memory usage and processing efficiency.

### 5.2.2 Results Analysis

The Blocked method shows similar performance across both dense (0% sparsity) and sparse (90% sparsity) matrices, as expected. Any minor differences in computation time likely stem from system-level factors like caching or memory management, rather than the structure of the matrix itself. This stability makes the Blocked method a good choice for general-purpose matrix computations, especially for larger matrices where memory access patterns are more critical.

### 5.3 Sparse matrix multiplication

The results of the execution time experiment are presented in the Table 3 and in the chart Figure 3 below.

| Size | Performance Time [ms] | |
|---|---|---|
| | Sparsity 0% | Sparsity 90% |
| 32x32 | 1.185 | 0.011 |
| 64x64 | 9.886 | 0.117 |
| 128x128 | 78.041 | 1.166 |
| 256x256 | 704.748 | 9.781 |
| 512x512 | 6733.662 | 83.439 |
| 1024x1024 | 57778.207 | 1044.898 |
| 2048x2048 | 477175.371 | 10352.955 |
| 4096x4096 | - | 94680.812 |

Table 3: Execution Time for Sparse Method with Different Sparsity Levels

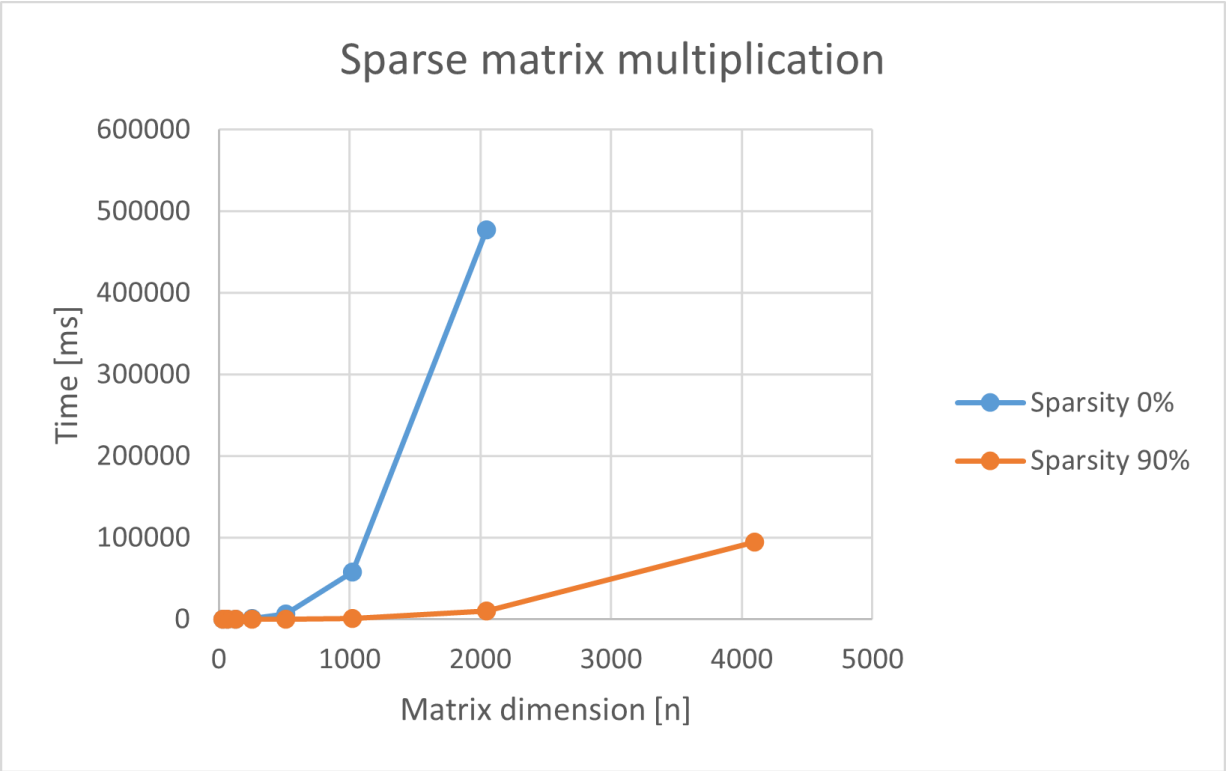


Figure 3: Sparse Method - Computation Time vs Matrix Size (0% and 90% Sparsity)

#### 5.3.1 Method description

The Sparse method is specifically optimized for sparse matrices by only storing and processing non-zero elements. This drastically reduces the amount of memory and computation required, as operations on zero elements are skipped entirely. It is especially effective for matrices with high sparsity (90%).

#### 5.3.2 Results Analysis

For matrices with high sparsity (90%), the Sparse method significantly accelerates computation times. Even for large sizes like 1024x1024, the computation time is much shorter compared to dense matrices (0% sparsity). The results highlight the Sparse method's efficiency with sparse matrices, though it loses performance for dense matrices. For sizes like 2048x2048 with 0% sparsity, the Sparse method shows very high computation times, indicating that it is specifically optimized for sparse matrices and not suitable for dense cases.

## 5.4 Comparison

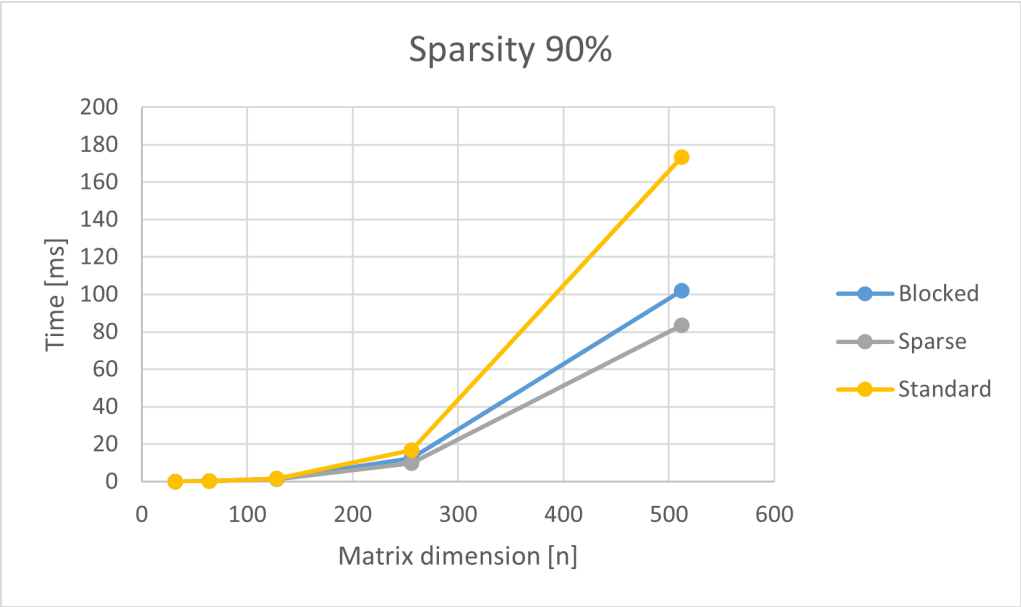


Figure 4: Comparison of Computation Time Across Methods (90% Sparsity)

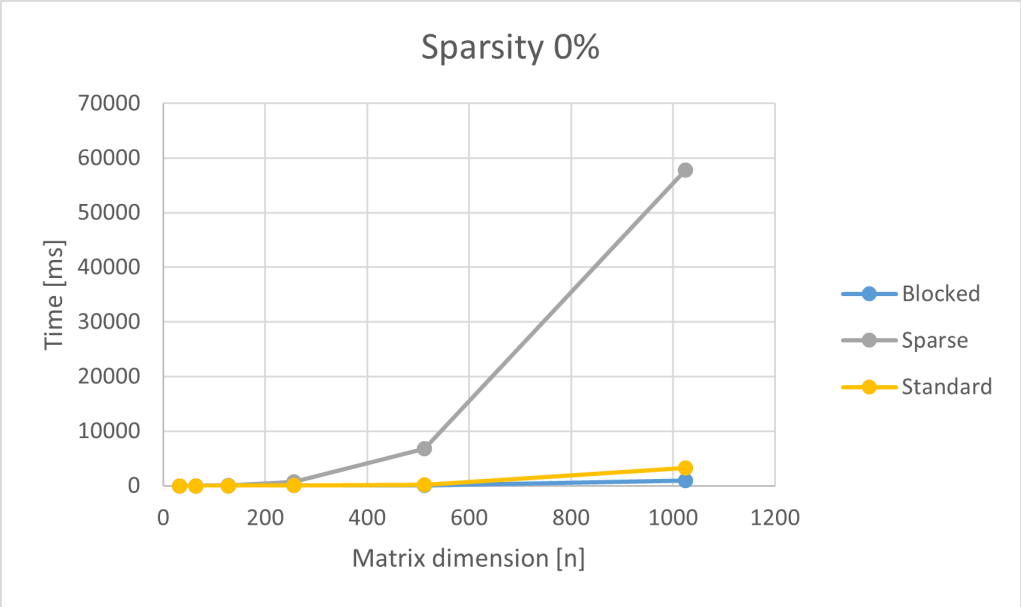


Figure 5: Comparison of Computation Time Across Methods (0% Sparsity)

- The Blocked method shows consistent performance regardless of matrix sparsity. This method focuses on optimizing memory access through blocking, making it a stable and reliable choice for both dense and sparse matrices, especially for large sizes.
- The Sparse method performs exceptionally well for high-sparsity matrices, where it can skip operations on zero elements, significantly reducing computation time. However, it performs poorly for low-sparsity (dense) matrices, where the overhead of its sparse-specific optimizations becomes a drawback. For dense matrices, the Sparse method ends up being the least efficient, as it lacks the straightforward processing of a non-sparsity-aware method.
- The Standard method shows uniform performance across different levels of sparsity but is inefficient for sparse matrices, as it processes every element, including zeroes. This makes it the least efficient choice for sparse data but reasonable for fully dense matrices.

### 5.5 mc2depi matrix

In this comparison, I benchmarked the performance of sparse matrix multiplication between Python and Java using the matrix mc2depi from the TAMU Sparse Matrix Collection (link). The goal was to evaluate the differences in execution time between the two languages when handling sparse matrices.

#### 5.5.1 Python implementation

The Python code uses SciPy to handle sparse matrices efficiently. Matrices are loaded from .mtx files using sio.mmread() and stored in the CSR format, optimized for fast matrix multiplication. The multiplication is performed with SciPy's highly optimized dot() method, which is written in lower-level languages for speed.

#### 5.5.2 Java implementation

The Java code handles sparse matrices manually, parsing .mtx files line by line and multiplying matrices without using specialized libraries. This custom handling leads to more overhead and slower performance compared to Python.

#### 5.5.3 Results

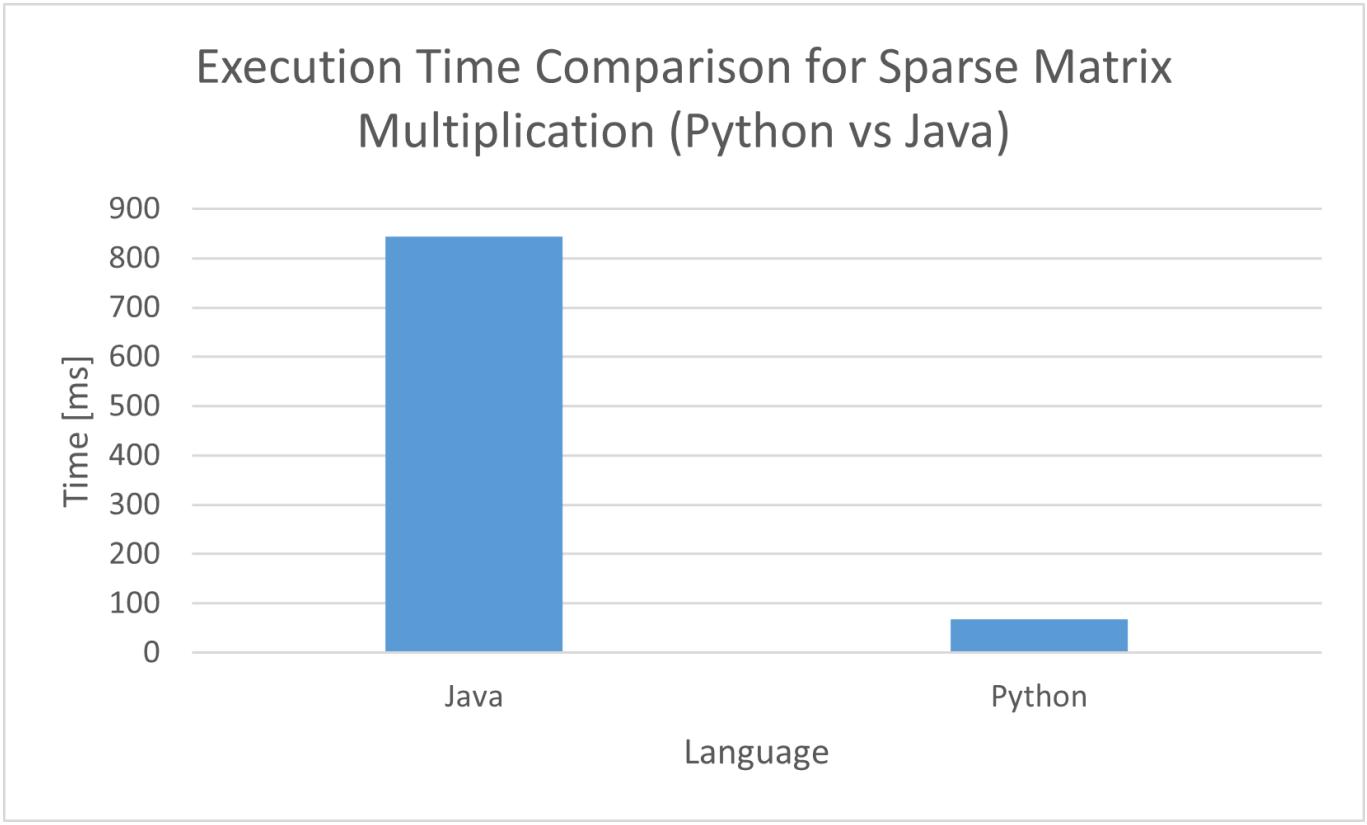


Figure 6: Execution Time Comparison for Sparse Matrix Multiplication (Python vs Java)

Python's multiplication took 67.9394 ms, while Java took 844.032 ms. This difference is due to Python's use of optimized sparse matrix handling, efficient data structures, and fast matrix loading.

# 6 Conclusion

In summary, for high-sparsity matrices, the Sparse method is by far the most efficient due to its ability to bypass zero elements. However, it should be avoided for dense matrices, where it is the slowest option. The Blocked method is a good all-around choice, especially for large matrices, as it is unaffected by matrix sparsity and provides stable, optimized performance. The Standard method, while straightforward, is generally inefficient for sparse matrices but performs adequately for dense matrices.

In comparing Python and Java for sparse matrix multiplication, Python demonstrated a clear advantage in handling high-sparsity matrices, primarily due to its use of the highly optimized SciPy library. SciPy's specialized data structures, like CSR, and efficient functions minimize overhead, making Python faster for sparse operations. Java, while flexible, requires more manual control over sparse data structures and lacks the same level of native optimization, leading to slower execution times. This distinction highlights Python's suitability for sparse matrix tasks, whereas Java may benefit from specialized libraries to close the performance gap in these scenarios.

# 7 Future work

Future experimentation will explore the following topics:

- Parallel (and Vectorized) Matrix Multiplication
- Distributed Matrix Multiplication