Matrix Multiplication Benchmarking

Alonso León, María

Data Science and Engineering

Big Data - ULPGC

Repository GitHub

October 19, 2024

Abstract: Matrix multiplication is a fundamental operation in fields like machine learning, scientific computing, and Big Data analytics. This paper presents a comprehensive benchmarking analysis of matrix multiplication performance across four programming languages: Python, Java, C++, and Rust. The study focuses on three key metrics: execution time, memory usage, and CPU utilization. Experiments were conducted on matrices of varying sizes, ranging from 64×64 to 2048×2048 elements. The results show that Rust and C++ deliver superior performance in terms of execution speed and resource management, with Rust emerging as the fastest language across all matrix sizes. Java maintained balanced performance with moderate memory usage and execution times, while Python demonstrated the weakest performance, especially with larger matrices, due to its higher execution time and less efficient CPU usage. These findings highlight the trade-offs between language efficiency and usability, providing valuable insights for developers and data scientists when selecting the most appropriate language for large-scale matrix operations.

Key words: Matrix multiplication, benchmarking, performance analysis, programming languages, Big Data, Python, Java, C++, Rust, execution time, memory usage, CPU utilization

1 Introduction

In an era where massive datasets are generated daily, efficiently handling large volumes of data is crucial. Matrix multiplication is one of the fundamental operations in numerous fields, such as machine learning, scientific computing, and Big Data analytics. Given its importance, understanding how different programming languages handle this computationally intensive task is essential for optimizing performance in data processing applications. The choice of programming language can significantly affect the efficiency of large-scale operations, particularly in scenarios that demand high computational power and memory management.

This paper focuses on benchmarking matrix multiplication across four carefully selected programming languages: Python, Java, C++, and Rust. Each of these languages represents a unique category in modern software development. Python, widely used for its simplicity and rich ecosystem of libraries, is known for its flexibility but is often criticized for its performance limitations in computationally heavy tasks. Java, a versatile and platform-independent language, is commonly used in enterprise applications and is designed to strike a balance between performance and ease of use. C++ is a highly optimized, low-level language that provides direct access to hardware resources, often yielding superior performance in computational tasks. Finally, Rust, a relatively newer language, is gaining traction for its memory safety features and high-performance capabilities, making it a promising candidate for tasks requiring efficient resource management.

While there are existing studies on the performance of programming languages, most focus on general-purpose tasks or specific algorithms. Few provide an in-depth comparison specifically targeted at matrix multiplication, a critical operation in Big Data contexts. Furthermore, this study introduces a visual and intuitive comparison of these languages, using clear charts to illustrate key performance metrics. By presenting these findings, this paper contributes to the growing body of knowledge on language performance and offers insights into which language is better suited for large-scale matrix operations.

The performance analysis in this study is based on three primary factors: execution time, memory usage, and CPU utilization during matrix multiplication.

In summary, this paper aims to provide developers

and researchers with a comprehensive comparison of how different programming languages handle matrix multiplication. The findings will guide those working in data-intensive fields, helping them choose the most suitable language for optimizing computational performance in matrix operations.

2 Problem Statement

As it was mentioned before, matrix multiplication is a fundamental operation in numerous data-driven fields, such as machine learning, scientific computing, and Big Data analytics. However, as the volume of data continues to grow, selecting the most efficient programming language for such a critical operation becomes a significant challenge. The problem addressed in this study is to determine which programming language—among Python, Java, C++, and Rust—performs matrix multiplication most efficiently in terms of execution time, memory consumption, and CPU usage. By identifying the most optimal language, this research aims to contribute to better decision-making in data-intensive applications.

Failing to choose the appropriate programming language for an essential operation like matrix multiplication can have serious consequences. It can lead to reduced performance, increased execution times, excessive memory consumption, and inefficient CPU utilization. These inefficiencies become particularly critical when processing large matrices, as they may result in delays and unnecessary resource usage in data processing pipelines. Through a detailed comparison of languages, this research will highlight how the selection of programming languages can directly impact the performance of Big Data applications.

One of the main difficulties faced by developers and data scientists is the lack of clear, intuitive benchmarks that show how different programming languages handle specific tasks like matrix multiplication. While some languages may offer powerful libraries for numerical computations, others may have performance advantages due to their lower-level control over hardware resources. Without comprehensive benchmarks, making an informed choice becomes complex, especially when dealing with large datasets where the trade-offs between speed, memory usage, and ease of implementation must be carefully balanced.

This study aims to address these challenges by providing an empirical comparison of how matrix size affects the performance of the selected programming languages. Additionally, it will take into consideration the ease of use and the availability of optimization libraries in each language, which are critical for practitioners who need efficient yet accessible solutions. By offering this comparative analysis, the study will serve as a valuable resource for developers seeking to optimize computational efficiency in matrix-based operations.

3 Methodology

The methodology for this research consists of running a series of carefully designed benchmarks to evaluate the performance of four programming languages—Python, Java, C++, and Rust—in the context of matrix multiplication. The experiments are conducted on two platforms: a native Windows 11 machine and a Linux Fedora virtual machine (VM), allowing for a comparison of different tools and environments. This approach ensures a robust evaluation of performance across various scenarios, leveraging both built-in libraries and external benchmarking tools.

Hardware and Environment Setup

The experiments were carried out on a personal computer with an Intel Core i7 processor (16 cores) and 32 GB of RAM, running Windows 11. Due to limitations with certain benchmarking tools on Windows, a Linux Fedora virtual machine was also used for specific tests. The VM was run using VirtualBox with default specifications (2048 MB and and 1 CPU). The use of both environments enables a more flexible and thorough comparison of languages across different system conditions.

Benchmarking Tools and Libraries

For the performance analysis, multiple tools were employed depending on the language and environment:

- **Python**: The pytest library was utilized for benchmarking, along with additional libraries such as random, time, psutil, and os to measure memory usage and CPU consumption.
- **Java**: The Java Microbenchmark Harness (JMH) was used to run the benchmarks, and java.lang.management.ManagementFactory and com.sun.management.OperatingSystemMXBean were employed for tracking memory and CPU usage.

- **C++**: Standard libraries such as iostream, vector, random, chrono, and Windows-specific libraries (psapi.h for memory tracking) were used.
- **Rust**:Benchmarks were performed using Rust's Instant module, and system metrics were tracked using the sys_info crate.

For more detailed performance tracking, the `perf` tool was employed on the Linux virtual machine, as it provides deep insights into execution time and CPU usage that were otherwise not available on Windows.

3.1 Optional Assignments:

In addition to the core objectives of the project, several optional tasks were undertaken to provide a more comprehensive analysis. CPU usage was carefully monitored across all programming languages to offer deeper insights into how each language manages computational resources during matrix operations. Furthermore, Rust, a relatively new but highly efficient language, was included in the benchmarking analysis, allowing for a comparison with more established languages such as Python, Java, and C++. Finally, for Java, multiple matrix multiplication algorithms were implemented and evaluated, including naive multiplication, block multiplication, and sparse matrix formats (CSR, CSC, COO). These additional steps enhanced the study by offering a broader perspective on performance optimization and resource management.

3.2 Code Structure

All the information discussed in this document is implemented in code available in this GitHub repository. Below is an overview of the main components and their correspondence with the current work. For a more detailed explanation, please refer to the README file in the repository.

The repository contains five folders: four corresponding to each programming language and one for relevant utilities, including code for generating graphs. These folders are named **C++**, **Java**, **Python**, **Rust**, and **Utils**.

Folder Structure

- **C++ Folder**: This folder contains four subfolders:
  - **shared_folder**: A shared directory where text files generated by the scripts are stored. Its significance lies in the fact that a virtual machine was used for the project, and this folder was shared between the personal computer and the virtual machine.
  - **Scripts**: Contains a single script named `perf_execution`, which executes the matrix multiplication code for different matrix sizes using the `perf` command and stores the output in text files within the `shared_folder`.
  - **Matrix_Multiplication**: This folder contains the function for matrix multiplication.
  - **Matrix_Multiplication_With_CPU_And_Memory**: This folder includes a CLion project that provides information on memory usage, CPU consumption, and execution time, serving as an alternative to the previous method.
- **Rust Folder**: The structure is similar to that of the C++ folder, but the code is appropriately tailored for Rust.
- **Java Folder**: This folder contains the following subfolders:
  - **shared_folder**: Functions similarly to the shared folder in C++.
  - **scripts**: Contains scripts that perform the same functions as those in the C++ and Rust folders.
  - **Matrix_Multiplication**: Similar functionality as in the C++ and Rust folders.
  - **Benchmark**: This subfolder is further divided into two additional folders:
    - **MatrixMultiplicationJava**: Contains code that utilizes benchmarking tools to evaluate matrix multiplication within an IntelliJ project.
    - **MultipleAlgorithmMultiplication**: Another IntelliJ project containing multiple matrix multiplication algorithms. An interface called `Matrix` has been created, along with algorithms that implement this interface and benchmarking code to evaluate them.
- **Python Folder**: The structure mirrors that of the Java folder; however, the **Benchmark** folder does not contain algorithm comparisons but rather includes a PyCharm project dedicated to benchmarking matrix multiplication.

- **Utils Folder**: This folder contains a PyCharm project named **Graphs Generator**, which includes utilities for extracting data from files and generating graphs.

4 Experiment Design

The primary focus of the experiment was to measure how each language performs matrix multiplication for matrices of varying sizes: [64, 128, 256, 512, 1024, 2048]. This range of sizes allowed for a comprehensive analysis of how well each language scales with increasing matrix dimensions. The benchmarks measured three key metrics: execution time, memory usage, and CPU utilization.

In Python and Java, the benchmarks included the following settings:

- **Warmup iterations**: 5
- **Measurement iterations**: 10
- **Forks**: 2

However, for C++ and Rust, the Linux `perf` tool was used, and it does not support warmup iterations or forks. This limitation is important to note when comparing results across languages.

Algorithm Variations

To further explore performance, different matrix multiplication algorithms were tested in Java, including:

- Naive Matrix Multiplication.
- Block Matrix Multiplication.
- Row-Column Major Multiplication.
- Sparse Matrix Multiplication using CSR, CSC, and COO formats.

These algorithms were implemented for a fixed matrix size of 128x128. The following configuration was used for each algorithm:

- **Warmup iterations**: 2
- **Measurement iterations**: 5
- **Forks**: 1

This additional comparison across algorithms provided insights into how different optimization techniques affect performance in Java.

Data Collection and Analysis

The data from the benchmarks, including the matrix multiplication tests, were stored in JSON files, while the results from the algorithmic variations were stored in text files. These benchmarks captured key metrics such as execution time (in seconds).

For the results obtained using the `perf` tool on the Linux virtual machine, the output was similarly stored in text files for manual review and analysis. Additionally, memory usage data were manually recorded during execution. Once collected, all results were processed and visualized using charting tools, such as Matplotlib, to provide a clear and intuitive comparison of performance across the different programming languages and matrix sizes.

Justification for the Methodology

The chosen methodology ensures a fair and comprehensive comparison of the selected languages, with attention to both practical and theoretical considerations. The variety of matrix sizes reflects real-world applications where matrices can vary greatly in scale, especially in Big Data contexts. Additionally, the inclusion of various matrix multiplication algorithms provides a deeper insight into how algorithmic choices interact with language efficiency.

Using a mix of native Windows tests and Linux VM benchmarks allows the study to mitigate any platform-specific limitations, ensuring that the comparison is robust and that the findings are relevant to different environments. Finally, by employing both warmup and measurement iterations in Python and Java, the study ensures that the results account for potential variations in execution due to Just-In-Time (JIT) compilation and runtime optimizations.

Challenges and Considerations

One of the main challenges encountered was the difference in available tools and functionalities between the environments. While `perf` provided powerful insights on Linux, it was not executable on the Windows machine, which led to the use of alternative libraries and manual tracking methods. This discrepancy in tools makes direct comparison slightly more complicated but still valuable, as the results offer a broader

perspective on how these languages perform across platforms.

With this explanation in place, the following subsections are designed to present the data collected from the experiments through graphical representations.

4.1 Benchmark Execution Time

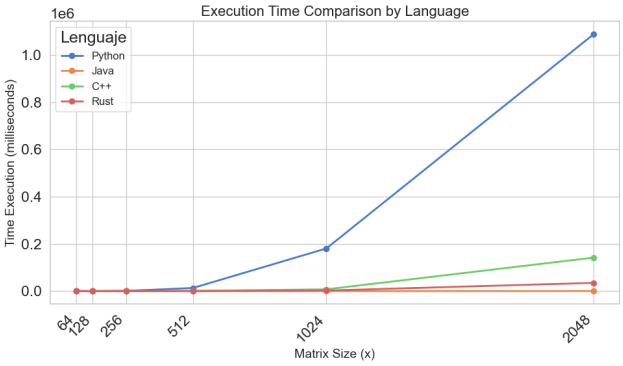

Figure 1: Execution Time Comparison by Language

This graph provides a comparative analysis of matrix multiplication execution times across four programming languages—Python, Java, C++, and Rust. The x-axis represents the matrix size, ranging from 64×64 to 2048×2048 elements, while the y-axis shows the execution time in milliseconds on a logarithmic scale, reaching up to 1 million milliseconds. The logarithmic scale is used to clearly display the performance differences between the languages, which span several orders of magnitude.

X-axis: Matrix Size

The x-axis reflects the dimensions of the square matrices used in the tests, ranging from 64×64 elements for the smallest matrices to 2048×2048 elements for the largest. As matrix size increases, the computational demand for performing matrix multiplication grows significantly.

Y-axis: Memory Usage (megabytes)

The y-axis represents the execution time in milliseconds, plotted on a logarithmic scale to accommodate the large variations in execution time across different programming languages and matrix sizes. The execution time reaches up to 1 million milliseconds (1,000 seconds), particularly for Python at the higher matrix sizes.

Where does the data come from?

The data was collected using both a personal computer and a virtual machine. Specifically, for Python and Java, the previously mentioned benchmarking tools were employed. Conversely, for C++ and Rust, the `perf` command was utilized within the virtual machine. To automate the iterative process, custom scripts were developed to execute multiple runs efficiently. While these graphs provide valuable insights, the use of different environments introduces some variability in the results. To address this, the following subsection presents a graph displaying the execution times recorded on the same machine, offering a more consistent comparison across all languages.

4.1.1 Programming Language Comparisons

- **Python (blue line)**: Python exhibits the most dramatic increase in execution time as matrix size grows. For smaller matrices (64×64 to 512×512), Python maintains a relatively stable and minimal execution time. However, beyond matrix sizes of 512×512, the execution time increases exponentially, reaching the highest value of all four languages. By the time the matrix size reaches 2048×2048, Python's execution time exceeds 1 million milliseconds, indicating a severe performance degradation. This poor scalability is likely due to Python's interpreted nature and limited optimization for computationally intensive tasks like matrix multiplication.

- **Java (orange line)**: Java emerges as the fastest language for matrix multiplication in this comparison, consistently showing the lowest execution times across all matrix sizes. The execution time remains virtually unchanged as matrix size increases, even for the largest matrices (2048×2048). This suggests that Rust's compiled nature, combined with its modern memory and concurrency management, provides superior efficiency and scalability for large matrix operations. Rust consistently outperforms the other languages in this test.

- **C++ (green line)**: C++ performs similarly to Java, showing a small but noticeable increase in execution time as matrix sizes increase. For smaller matrices, C++'s execution time remains minimal, and for larger matrices (1024×1024 and 2048×2048), the increase is more apparent but

still manageable. C++ remains far more efficient than Python, with much lower execution times for all matrix sizes. Its slight performance decline with larger matrices may be attributed to its lower-level memory management and resource allocation, but it remains highly efficient overall.

- **Rust (red line)**: Rust demonstrates consistently low execution times across all matrix sizes. Although there is a slight increase in execution time as matrix size grows, the increase is far less pronounced compared to Python. Java maintains a stable performance, with only minor fluctuations even for the largest matrices, indicating that it handles large-scale matrix multiplications efficiently. This is likely due to Java's Just-In-Time (JIT) compilation, which optimizes performance at runtime.

4.1.2 Key Observations

1. **Python's Significant Performance Drop**: Python is clearly the least performant language in this comparison, especially for larger matrices. Its execution time grows exponentially as matrix size increases, far exceeding the execution times of Java, C++, and Rust. This highlights Python's limitations in handling large-scale matrix operations, likely due to its interpreted nature and lack of low-level optimizations.

2. **Java's Superior Performance**: Java demonstrates the best performance, with execution times that remain consistently low regardless of matrix size. The near-constant execution time across all matrix sizes indicates Java's efficiency in managing both small and large computations, making it the fastest language for matrix multiplication in this test.

3. **Rust and C++'s Stable Efficiency**: Both Rust and C++ exhibit stable and low execution times, though C++ shows a slightly more noticeable increase for larger matrices. Nonetheless, both languages maintain efficient performance compared to Python, and their slight increases in execution time for larger matrices suggest that they are well-suited for computationally intensive tasks, though Java outperforms them slightly.

4. **Compiled Languages vs Interpreted Languages**: This chart clearly demonstrates the advantages of compiled languages (C++, Rust, and partially Java with its JIT compilation) over interpreted languages like Python for tasks involving heavy computations such as matrix multiplication. Compiled languages tend to be more optimized for performance, with Java emerging as the most efficient, followed closely by C++ and Rust.

4.1.3 Conclusion

This performance comparison highlights the stark differences in execution times between Python, Java, C++, and Rust for matrix multiplication tasks. Java outperforms all other languages in terms of execution time, making it an ideal choice for large-scale matrix operations where computational efficiency is critical. Rust and C++ also show strong performance, maintaining stable execution times across all matrix sizes. In contrast, Python struggles with larger matrices, making it a less suitable choice for computation-heavy tasks unless optimized through external libraries or tools.

4.2 Time execution using the virtual machine

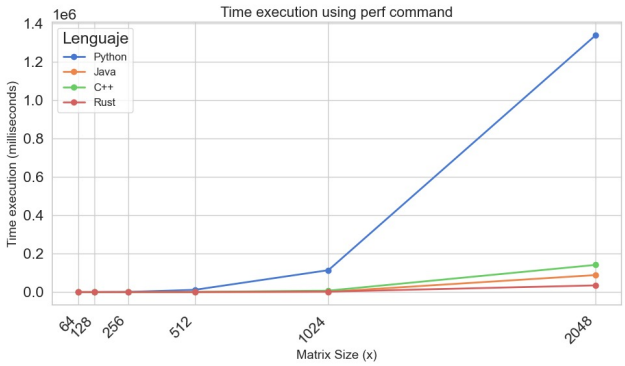

Figure 2: Time execution using perf command

This figure depicts the execution time (in milliseconds) required for matrix operations across four programming languages—Python, Java, C++, and Rust—measured using the `perf` command. The graph highlights how execution time scales as matrix size increases, providing a comparative view of each language's computational efficiency.

X-axis: Matrix Size

The x-axis represents the size of the square matrices, ranging from 64x64 to 2048x2048, which indicates

the increasing complexity of the matrix operations as the dimensions grow.

Y-axis: Execution Time (milliseconds)
The y-axis shows the execution time in milliseconds, using a logarithmic scale ranging from 0 milliseconds to over 1.4 million milliseconds. This scale accommodates the substantial variations in execution time between the programming languages, particularly for larger matrix sizes.

Where does the data come from?
As previously mentioned, this graph utilizes data collected from the same environment, specifically from the virtual machine using the `perf` command.

4.2.1 Language Comparisons

- **Python (blue line)**: Python shows the most significant increase in execution time as matrix size grows. For smaller matrices (64x64 to 512x512), Python's execution time remains relatively low. However, beyond matrix size 512, the execution time escalates rapidly. By the time the matrix size reaches 2048x2048, Python's execution time exceeds 1.3 million milliseconds. This steep increase indicates Python's inefficiency for large-scale matrix operations, likely due to its interpreted nature and lack of low-level optimizations for heavy computational tasks.

- **Java (orange line)**: Java demonstrates consistently low execution times across all matrix sizes, with only minor increases as the matrix size grows. Even for large matrices (2048x2048), Java maintains a relatively flat execution time curve, showcasing its efficient handling of matrix operations. This is likely due to Java's Just-In-Time (JIT) compilation, which optimizes performance during runtime.

- **C++ (green line)**: C++ performs similarly to Java, maintaining low execution times across all matrix sizes. The increase in execution time is linear and minimal, suggesting that C++ handles matrix operations with high efficiency. C++'s compiled nature and deep system-level optimizations contribute to its superior performance, particularly for computation-heavy tasks.

- **Rust (red line)**: Rust follows a pattern comparable to C++ and Java, showing low execution times that scale smoothly as matrix size increases. Furthermore, exhibits the shortest execution times, outperforming the other languages in this regard, highlighting Rust's ability to manage matrix operations efficiently, benefiting from its modern design and system-level optimizations.

4.2.2 Key Observations

1. **Significant Performance Degradation in Python**: Python performs well for small matrices, but execution time increases dramatically with larger sizes. At 2048x2048, Python's execution time is several orders of magnitude slower than other languages, highlighting its limitations in large-scale matrix operations. This inefficiency arises from Python's interpreted nature and its reliance on high-level data structures, which are not optimized for performance in computationally intensive tasks.

2. **Consistent Efficiency in Java, C++, and Rust**: Java, C++, and Rust exhibit highly efficient execution times with only slight increases as matrix size scales up. These languages are either compiled (C++ and Rust) or employ runtime optimizations (Java with JIT compilation), allowing them to handle large matrix operations effectively with minimal performance degradation.

3. **Scalability Across Languages**: Python demonstrates poor scalability as matrix size grows, with execution time ballooning disproportionately. In contrast, Java, C++, and Rust scale efficiently, with only minor increases in execution time as matrix size increases, making them suitable for large-scale matrix operations.

4.2.3 Overall Analysis

- **Rust and Java**: Rust consistently achieves the fastest execution times, particularly for the largest matrices (2048x2048), due to its system-level optimizations and modern design. Java follows closely behind, outperforming C++ in terms of execution time. Despite being an interpreted language, Java's Just-In-Time (JIT) compilation allows it to efficiently handle large matrix operations, achieving performance levels comparable to compiled languages like Rust.

- **C++**: Although C++ demonstrates efficient execution times, it is slightly outpaced by both Rust and Java, especially for larger matrices. C++ remains highly optimized and performs well, but its execution time shows a marginal increase compared to Rust and Java, making it the third fastest in handling large-scale matrix operations.

- **Python**: Python performs the worst among the four languages, with execution times increasing dramatically for larger matrices. This makes Python an unsuitable choice for large-scale matrix operations unless performance is significantly enhanced through external libraries or specialized tools.

4.2.4 Conclusion

This graph highlights the considerable differences in how various programming languages handle large-scale matrix operations. Compiled languages such as C++ and Rust, and JIT-compiled Java, demonstrate highly efficient execution times and scalability as matrix size grows. In contrast, Python struggles with execution time for larger matrices, reflecting the inherent limitations of interpreted languages in computation-heavy scenarios. For tasks like matrix multiplication, where performance is critical, C++, Rust, and Java are far more suitable than Python.

4.3 Memory Usage

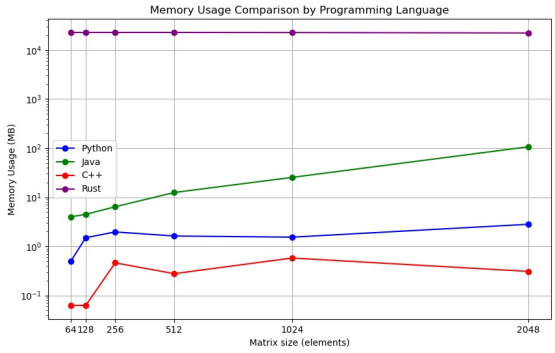

Figure 3: Memory usage comparison by programming language

This figure illustrates the memory usage (in megabytes, MB) during matrix operations for four programming languages—Python, Java, C++, and Rust—across varying matrix sizes. The x-axis represents the matrix size (in elements), while the y-axis shows memory consumption on a logarithmic scale. This scaling effectively captures the differences in memory usage, which span several orders of magnitude across the programming languages.

X-axis: Matrix Size

The x-axis reflects the matrix sizes used in the tests, ranging from 64×128 elements to 2048×2048 elements. These values correspond to the dimensions of the square and rectangular matrices involved in the operations, with larger matrices typically requiring more memory for both data storage and processing.

Y-axis: Memory Usage (megabytes)

The y-axis represents memory usage on a logarithmic scale, spanning from 10^{-1} MB (0.1 MB) to 10^{4} MB (10,000 MB). The logarithmic scale is necessary to display the large differences in memory consumption among the languages, making the variations more interpretable despite the broad range.

Where does the data come from?

In this case, the data was collected using different libraries specific to each programming language. The execution process was conducted on a personal PC, utilizing JetBrains tools for execution and performance tracking.

4.3.1 Programming Language Comparisons

- **Rust (purple line)**: Rust consistently demonstrates the highest memory usage throughout the tests, consuming around 10,000 MB (10^4 MB) regardless of the matrix size. This flat memory usage suggests that Rust's memory allocation strategy introduces a high base overhead that does not scale significantly with matrix size in this particular experiment. It is likely that Rust is allocating memory statically or has a fixed overhead, which may result in higher memory consumption compared to the other languages.

- **Java (green line)**: Java's memory usage grows steadily as the matrix size increases. Starting at approximately 10 MB (10^1 MB) for smaller matrices (64×128 elements); Java's memory usage reaches around 100 MB (10^2 MB) by the time the matrix size hits 2048×2048 elements. This linear growth pattern could be attributed to Java's garbage col-

lection and memory management mechanisms, which involve dynamic memory allocation and memory pooling strategies to handle larger data sizes.

- **Python (blue line)**: Python shows a different memory usage profile. After an initial increase to about 1 MB (10^0 MB), its memory consumption remains relatively constant as matrix size increases. For matrices ranging from 256×256 to 2048×2048 elements, Python's memory usage hovers around 1 MB (10^0 MB), suggesting that Python's internal libraries (likely NumPy) manage matrix operations efficiently for smaller datasets. However, its flat memory usage pattern may also indicate limited optimization for larger-scale operations in this scenario.

- **C++ (red line)**: C++ demonstrates the most efficient memory usage across all tests. Starting with minimal memory consumption of around 0.1 MB (10^{-1} MB) for smaller matrices, C++'s memory usage remains extremely low, even decreasing slightly as matrix size grows. By the time matrix size reaches 2048×2048 elements, C++'s memory usage is still below 1 MB (10^0 MB). This behavior is likely due to C++'s low-level memory management and optimization, which minimizes overhead and allocates memory more efficiently than the other languages.

4.3.2 Key Observations

1. **Rust's High Memory Overhead**:

2. **C++'s Memory Efficiency**: C++ is by far the most memory-efficient language in these tests, with its memory usage decreasing slightly as matrix size increases. This remarkable efficiency highlights the advantages of C++'s low-level memory management and optimized allocation strategies, making it well-suited for scenarios where minimizing memory consumption is critical.

3. **Java and Python's Steady Memory Usage**: Both Java and Python demonstrate relatively steady memory usage patterns. Java's memory consumption increases linearly with matrix size, likely due to its dynamic memory allocation and garbage collection systems. Python, in contrast, shows a flat memory usage curve, indicating efficient memory handling for smaller matrices, though this may be partly attributed to Python's reliance on external libraries for matrix operations.

4.3.3 Conclusion

This chart provides a clear comparison of memory consumption across Python, Java, C++, and Rust during matrix operations. C++ emerges as the most memory-efficient option, making it an excellent choice for applications where minimizing memory usage is a priority. On the other hand, Rust's high baseline memory usage may pose challenges in memory-constrained environments, despite its performance advantages in other areas. Java and Python offer more moderate memory usage, with Java scaling linearly as matrix size grows, while Python's memory consumption remains steady. Ultimately, the choice of programming language for matrix operations will depend on the specific memory constraints and performance requirements of the task at hand.

4.4 CPU usage

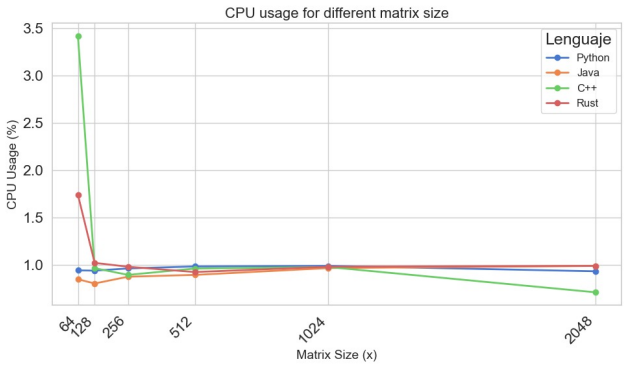

Figure 4: CPU usage for different matrix size

This figure illustrates the CPU usage (in percentage) as a function of matrix size for four programming languages—Python, Java, C++, and Rust—during matrix multiplication tasks. The comparison provides insight into the CPU efficiency of each language across varying matrix dimensions.

X-axis: Matrix Size

The x-axis denotes the size of the square matrices, ranging from 64 to 2048. This axis highlights the

growth in matrix dimensions and the corresponding increase in computational complexity.

Y-axis: CPU Usage (%)
The y-axis represents CPU usage as a percentage of the total CPU capacity, ranging from 0% to approximately 3.5%. The graph demonstrates how much of the system's CPU resources are consumed by each language during the matrix multiplication operations.

Where does the data come from?
Although some languages provide built-in functions to track CPU usage, such as Python, others, like Java, presented greater challenges. Therefore, it was decided to use the `perf` command within the virtual machine to gather the necessary data.

4.5 Language Comparisons

- **C++ (green line)**: C++ exhibits the highest initial CPU usage, around 3.5% for the smallest matrix size (64x64). However, it experiences a sharp decline in usage, reaching approximately 1% for larger matrices. This drop indicates that while C++ incurs higher overhead for smaller matrices, it rapidly optimizes and maintains efficient CPU performance as matrix size increases.

- **Rust (red line)**:

  Rust follows a pattern similar to C++, but in a less pronounced manner, beginning with approximately 2% CPU usage for the smallest matrices and quickly stabilizing at slightly above 1% for larger matrices. This suggests that Rust, like C++, requires more CPU resources initially but stabilizes as matrix size grows, reflecting efficient performance at larger scales.

- **Python (blue line)**: Python's CPU usage remains near 1% across all matrix sizes, exhibiting minimal variation. This steady behavior implies that Python's CPU usage is relatively inefficient, particularly for large matrices, as it lacks the significant optimizations seen in C++ and Rust.

- **Java (orange line)**: Java starts with a CPU usage slightly below 1% and maintains this usage consistently across all matrix sizes. Similar to Python, Java demonstrates a stable yet flat usage pattern, indicating less aggressive CPU optimizations compared to C++ and Rust, but consistent performance over varying matrix sizes. From matrix sizes of 1024 and above, there is a noticeable overlap between Java and Python in terms of CPU usage, suggesting comparable efficiency in their handling of larger data sets.

4.5.1 Key Observations

1. **Initial High CPU Usage for Small Matrices (64x64)**: Both C++ and Rust demonstrate high CPU consumption for small matrices, which may be due to initial overhead in managing smaller workloads. However, this overhead rapidly diminishes as matrix sizes increase, allowing both languages to stabilize and optimize performance.

   In contrast, Python and Java exhibit lower initial CPU usage for small matrices but maintain relatively flat CPU usage as matrix size increases, indicating less aggressive optimization for small-scale operations.

2. **Stable Performance for Larger Matrices (1024x1024 and beyond)**: For matrices of size 512x512 and larger, CPU usage stabilizes across all four languages. At this point, C++, Rust, Python, and Java all converge around 1% CPU usage, suggesting that as matrix dimensions grow, the CPU load becomes predictable and more evenly distributed across these languages.

4.5.2 Language Comparison

- **C++ and Rust**: Both compiled languages show significant optimization as matrix size increases. While they start with higher CPU usage for small matrices, their rapid decrease and subsequent stability reflect efficient resource management, especially for large-scale matrix operations.

- **Python and Java**: These languages maintain a steady, flat line of CPU usage throughout the experiments, with minimal variation or improvement. While their consistent performance may be predictable, it reflects a lack of the significant CPU optimizations observed in C++ and Rust.

4.5.3 Overall Analysis

This analysis highlights the efficiency of compiled languages (C++ and Rust) in handling large matrix operations, as their CPU usage decreases significantly after an initial overhead. On the other hand, interpreted

(Python) and JIT-compiled (Java) languages maintain consistent but less optimized CPU usage patterns, with limited resource efficiency for large-scale tasks. The results suggest that for performance-critical tasks such as matrix multiplication, C++ and Rust are more suitable, particularly when large matrices are involved.

This version is formal, concise, and suitable for inclusion in a scientific paper. Let me know if you'd like any further modifications!

4.6 Multiple Algorithms in Java

As demonstrated in the previous graphs, Rust achieves the shortest execution time; however, it exhibits high memory usage as well as elevated CPU consumption. Java, on the other hand, shows execution times very close to those of Rust, though slightly higher, while significantly outperforming Rust in both memory and CPU efficiency. Although Java does not lead in all evaluation metrics, it presents the most balanced and globally favorable results. For this reason, this section focuses on optimizing the naive matrix multiplication algorithm to further improve performance.

In this image, we can observe the various algorithms used, followed by their explanations.

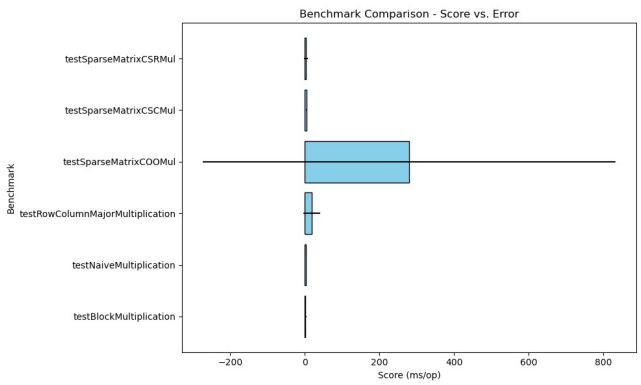

Figure 5: Benchmark comparison - Score vs. error

The boxplot presented in Figure 4 provides a comparative performance analysis of six matrix multiplication algorithms in java, with execution times measured in milliseconds per operation (ms/op). The x-axis indicates the performance scores, while the y-axis enumerates the different algorithms tested. Each algorithm's performance is visualized through boxplots, which display the distribution of execution times and the associated variability.

4.6.1 Algorithms Analyzed

Before presenting the results, this section provides a detailed description of each algorithm.

- **Compressed Sparse Row Matrix Multiplication (CSR)**: The testSparseMatrixCSRMul benchmark measures the performance of matrix multiplication using the CSR format. CSR is an efficient storage format for sparse matrices, which store only non-zero elements, thereby reducing memory usage and computational complexity. This format stores the values of non-zero elements, along with row indices and the positions of these elements within their rows.

- **Compressed Sparse Column Matrix Multiplication (CSC)**: The testSparseMatrixCSCMul benchmark evaluates the CSC format, which is similar to CSR but organizes the matrix by columns rather than rows. This format is advantageous in scenarios where column-wise operations are more prevalent. The benchmark assesses the efficiency of multiplying sparse matrices stored in this format.

- **Coordinate List Matrix Multiplication (COO)**: The testSparseMatrixCOOMul test benchmarks the COO format, a simple and flexible storage method for sparse matrices. COO stores the row and column indices of each non-zero element explicitly. Although this format is easy to implement and intuitive, it often leads to higher overhead during operations due to the need to traverse additional indices. This is reflected in the wide variability in execution times observed for this benchmark.

- **Row-Column Major Matrix Multiplication**: The testRowColumnMajorMultiplication benchmark measures the performance of matrix multiplication when optimizing for memory access patterns, specifically using row-major or column-major ordering. This method aligns memory access with the underlying data structure to improve cache efficiency, resulting in faster computation in certain cases.

- **Naive Matrix Multiplication**: The testNaiveMultiplication benchmark represents the classical or naive approach to matrix multiplication, which

involves three nested loops to compute the product. This method lacks optimizations for memory access or sparsity, making it computationally expensive, especially for large matrices.

- **Block Matrix Multiplication**: The testBlockMultiplication benchmark assesses the performance of matrix multiplication using a block-based method. This approach divides matrices into smaller submatrices (blocks) and performs multiplication on these blocks. Block matrix multiplication enhances performance by improving cache utilization and reducing memory latency.

4.6.2 Interpretation of Results

Where does the data come from?

For this graph, the data was collected using the personal PC. An interface was created along with various algorithms and a testing framework to perform the benchmarking in Java.

The x-axis of the plot represents execution time in ms/op, with lower values indicating better performance. The boxplots illustrate the median, interquartile range (IQR), and variability in the performance of each algorithm.

- The COO format shows the largest spread in execution times, ranging from approximately -200 ms/op to over 600 ms/op. This high variability is likely due to the inefficiency of accessing nonzero elements through coordinate pairs, which can result in unpredictable computational overhead.
- In contrast, the Row-Column Major Multiplication algorithm exhibits a much more consistent performance, with a narrow distribution of execution times concentrated between 0 and 50 ms/op, suggesting stable and efficient handling of memory access patterns.
- Both the Naive Multiplication and Block Multiplication algorithms show compact boxplots, indicating that their performance is highly consistent across trials. However, these algorithms may not offer the same level of optimization for sparse data, as reflected in their relatively higher median execution times.
- The CSR and CSC formats perform more efficiently compared to the naive method, with CSC demonstrating slightly less variability than CSR. These formats are optimized for sparse matrix operations and strike a balance between reducing memory usage and improving execution speed.

4.6.3 Overall Analysis

This benchmark analysis provides a clear comparison of different matrix multiplication algorithms, highlighting the trade-offs between storage format, computational efficiency, and performance variability. Sparse matrix formats such as CSR and CSC offer significant performance improvements over naive methods, particularly for large matrices with many zero elements. Block and memory-access optimized methods, while stable, may not fully leverage the advantages of sparsity. The COO format, although flexible, introduces considerable variability in performance due to the overhead of managing coordinate-based indexing. This analysis underscores the importance of selecting appropriate algorithms based on the characteristics of the matrix and the computational environment.

5 Conclusion

This study provides a comprehensive benchmarking analysis of matrix multiplication across four programming languages: Python, Java, C++, and Rust. The experiments focused on three critical performance metrics: execution time, memory usage, and CPU utilization. These tests were performed on matrices of varying sizes, ranging from 64x64 to 2048x2048 elements, simulating the performance challenges encountered in real-world Big Data applications.

The results indicate clear differences in how these languages handle computationally intensive tasks like matrix multiplication:

- **Rust** consistently demonstrated the best overall performance, achieving the shortest execution times; however, it exhibited high memory usage and elevated CPU consumption. Its modern design and memory management features make it highly suitable for tasks requiring both speed and scalability.
- **Java** offered a balanced performance, with execution times very close to those of Rust, although slightly higher. Notably, Java significantly outperformed Rust in terms of memory usage and CPU efficiency. Its Just-In-Time (JIT) compilation allowed it to handle large matrices effectively, mak-

ing it a strong contender for performance-sensitive applications.

- **C++** also performed efficiently, particularly in terms of memory usage, where it outperformed all other languages. Its execution time remained low and stable, although it showed a slight increase for the largest matrices.

- **Python** was the least performant language in this study, exhibiting severe performance degradation with larger matrices. Its execution time increased exponentially, making it unsuitable for large-scale matrix operations without optimization through external libraries.

The findings underscore the importance of selecting the right programming language for matrix-based operations in Big Data applications. Compiled languages like Rust and C++ show a clear advantage in terms of execution speed and resource management, making them better suited for performance-critical tasks. Java, while not as fast as Rust, remains a viable option due to its balanced performance and lower resource consumption. Python, due to its slower execution time, is more appropriate for smaller datasets or where ease of coding and extensive library support outweigh raw performance requirements.

This comparative analysis provides valuable insights for developers and data scientists, enabling informed decisions when choosing a programming language for high-performance computing tasks in Big Data environments.

6 Future work

Building on the findings from this study, the next phase of research will focus on exploring optimized matrix multiplication techniques and the use of sparse matrices to further enhance computational efficiency. Specifically, future work will investigate algorithmic improvements such as Strassen's algorithm, loop unrolling, and cache optimization techniques. These methods are expected to reduce execution time and optimize memory usage, particularly for large-scale matrix operations.

Additionally, the study will explore the implementation and performance evaluation of matrix multiplication using sparse matrices. Sparse matrices, which contain a high proportion of zero elements, can significantly reduce computational costs and memory usage. This future research will compare the performance of basic matrix multiplication approaches with optimized versions, analyzing how different optimization techniques and sparsity levels (percentage of zero elements) impact execution time and memory consumption.

Moreover, future experiments will test the algorithms with progressively larger matrix sizes, determining the maximum size that each optimized approach can handle efficiently. The results will include detailed performance comparisons between dense and sparse matrices at varying sparsity levels, highlighting any bottlenecks or limitations in the optimized implementations. This work will provide deeper insights into how optimization strategies can be leveraged to improve matrix operations in Big Data contexts.