Comparative Study on Matrix Multiplication Efficiency Across Java, Python, and C

Owen Urdaneta Morales

Abstract

Matrix processing and calculation are critical tasks in computational programming. Matrices provide an optimal structure for managing large datasets and executing complex computations, although their computational demands can be substantial. As a result, continuous efforts are made to optimize matrix operations, aiming to minimize both resource consumption and processing time.

This paper presents a detailed comparative study on the efficiency of matrix multiplication across three widely used programming languages: Java, Python, and C. To evaluate performance across varying matrix sizes (256x256, 512x512, and 1024x1024), we employed specialized benchmarking tools tailored to each language: JMH for Java, Pytest for Python, and Perf for C. The results demonstrate notable disparities in execution times between the languages, with Python showing significantly slower performance compared to Java and C. Conversely, C exhibited superior efficiency, outperforming both Python and Java, and proving to be the most effective language for matrix computations in this comparison.

1 Introduction

In the Big Data field, we often work with large amounts of information. When working on a small scale, having inefficient code is not usually a problem. However, this becomes an issue when the task at hand involves large datasets, as it translates into longer processing times. Time that results in non-productivity and escalates further as the size of the matrix increases.

Benchmarks are one of the best tools to address this issue. They not only provide detailed information about the execution time of our code but also allow us to control the resource consumption that the execution demands from the machine. This enables us to better identify potential areas of the code that can be optimized.

Matrices are a common structure for storing information since they can hold large quantities of data and provide a good way to organize massive datasets. However, while they are efficient for data organization, this does not mean that they are easy to process for a computer.

Matrix multiplications can be a costly operation, especially when executed multiple times across large matrices. To minimize this non-productive time, it

is crucial to know different approaches and techniques, but most importantly, to identify the best tools for the task.

This paper seeks to answer the question of which of the programming languages mentioned, Java, Python and C, is the most efficient when working with matrix structures of varying sizes. We will first work with a 256x256 matrix, followed by a 512x512 matrix, and finally, a 1024x1024 matrix.

2 Method

To address the problem of matrix multiplication and determine the time each programming language takes to complete this task, we will use the different benchmarking tools that each language provides. We will present these tools individually in order to offer a clear and comprehensive comparison of their performance:

2.1 Java Benchmarking

For the development of this experiment in Java, IntelliJ IDEA was used as the development environment, with Maven as the project's build system, since the benchmarking tool we are using, JMH (Java Microbenchmarking Harness), is part of the OpenJDK project. Being that also the very same reason why it was necessary to add the following dependency to the project's `pom.xml` file in order to utilize it.

```
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>1.35</version>
</dependency>
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
    <version>1.35</version>
    <scope>provided</scope>
</dependency>
```

Once the dependency was added to the project's `pom.xml`, a class was created to perform the benchmark for matrix multiplication. In addition to the JMH dependencies, it was necessary to import the `Random` library from `java.util` and the `TimeUnit` library from `java.util.concurrent`. The `Random` class was used to initialize two matrices, **A** and **B**, with randomized values, while the `TimeUnit` class was employed to configure the output of the benchmarking process.

```
import org.openjdk.jmh.annotations.*;

import java.util.Random;
import java.util.concurrent.TimeUnit;
```

```java
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Benchmark)
@Warmup(iterations = 10, time = 1, timeUnit = TimeUnit.
    SECONDS)
@Measurement(iterations = 10, time = 1, timeUnit = TimeUnit.
    SECONDS)
@Fork(5)
public class MatrixBenchmark {

    // Parametrized matrix size
    @Param({"256", "512", "1024"})
    public int n;

    double[][] a;
    double[][] b;
    double[][] c;

    @Setup(Level.Trial)
    public void setup() {
        // Initialize matrices with random values
        a = new double[n][n];
        b = new double[n][n];
        c = new double[n][n];

        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;
            }
        }
    }

    @Benchmark
    public void testMatrixMultiplication() {
        // Perform matrix multiplication
        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];
                }
            }
        }
    }
}
```

As shown in the provided code, annotations play a crucial role in configuring the specific details of benchmarking in JMH. Below is an explanation of the key

annotations used:

- **@BenchmarkMode**: This annotation sets the mode in which the performance of the code will be measured during the benchmark. JMH offers several modes such as `Throughput` (measuring operations per unit of time), `AverageTime` (measuring the average time per operation), among others. In this case, it is configured to `Mode.AverageTime`, which means the benchmark will measure the average time it takes to execute an operation.
- **@OutputTimeUnit**: This annotation specifies the time unit in which the benchmark results will be displayed. Here, `TimeUnit.MILLISECONDS` is used, meaning the measured time will be expressed in milliseconds. Other options include `TimeUnit.SECONDS`, `TimeUnit.MICROSECONDS`, etc.
- **@State**: This annotation defines the scope in which the state is shared between different invocations of the benchmark. In this example, `Scope.Benchmark` is used, meaning the same state (data and configurations) is shared across all iterations of the benchmark. Other scope options include `Thread` (one state per thread) or `Group` (state shared among a group of threads).

Within the class, the **@Param** annotation functions as a parameterizer, allowing benchmarking of different values without needing to modify or recompile the code. This is especially useful when performing tests on matrices of various sizes. For instance, to test matrices of different dimensions, simply adjust this parameter to run tests on matrices of sizes `256`, `512`, or `1024`.

The **@Setup** annotation ensures that any necessary configuration, such as object creation or data initialization, is complete and ready before the performance measurement begins. In this case, the use of `Level.Trial` indicates that this method will be executed before each trial, where a trial refers to a complete execution of the benchmark test. It is important to note that the time spent on initialization is excluded from the benchmark, ensuring that only the time spent on the actual operation is measured.

Finally, the **@Benchmark** annotation identifies the method to be tested for performance. In this example, the method annotated with `@Benchmark` performs the matrix multiplication, and JMH will measure the time it takes to complete this operation for different matrix dimensions.

2.2 Python Benchmarking

In the case of Python, the PyCharm environment was used for development. To perform benchmarking in Python, the `pytest-benchmark` library is employed. Before running the tests, it is essential to ensure that the required library is installed. This can be achieved by executing the following command in the console:

```
pip install pytest-benchmark
```

Once the library is installed, you can benchmark a function by creating a test file, for example, `test_benchmark.py`, which contains a function specifically for testing.

```
import random
import pytest


# Matrix multiplication function
def matrix_multiply(A, B, n):
    C = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C


# Parametrize the test function to run with different matrix
    sizes
@pytest.mark.parametrize("n", [256, 512, 1024])
def test_matrix_multiply(benchmark, n):

    # Generate random matrices A and B of size n x n
    A = [[random.random() for _ in range(n)] for _ in range(
        n)]
    B = [[random.random() for _ in range(n)] for _ in range(
        n)]

    # Run the benchmark on the matrix_multiply function
    result = benchmark(matrix_multiply, A, B, n)

    assert result is not None
```

Just like in Java, we import `random` to initialize the matrix values, as well as importing the `pytest` library itself. With the decorator `@pytest.mark.parametrize`, we can define different values for `n` to perform multiple benchmarks within the same execution.

2.3 C Benchmarking

For the tests in C, an Ubuntu virtual machine was used. Benchmarking in C is simpler compared to the previous cases. Instead of creating a separate test script for benchmarking, we simply need to compile the same code that will be benchmarked. In this case, the code in question is the matrix multiplication implementation.

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

```
#define n 1024
double a[n][n];
double b[n][n];
double 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("%0.6f\n",diff);
}
```

The main challenge here is the need to modify the code manually to test different matrix sizes, as opposed to parameterizing the tests as was done in Java and Python.

To perform benchmarking in C, the `perf` tool is required, so it is essential to ensure it is installed. This can be done with the following command:

```
sudo apt-get install linux-tools-common linux-tools-generic
    linux-tools-$(uname -r)
```

You also need to ensure that the `gcc`, `make`, and `perl` libraries are installed. To do so, run the following command:

```
sudo apt-get install gcc make perl
```

Once installed, and assuming you are in the same directory as the C code, you can compile the code in release mode to measure real-world performance. Assuming the C file is named `Matrix.c`, the executable would be generated with the following command:

```
gcc -O2 -o matrix Matrix.c
```

It is not necessary for the executable to share the same name as the source code. The user can assign any name to the output executable as they prefer.

It is possible that during the benchmark running it may not display results due to restricted access to performance monitoring and observability. This is controlled by the `perf_event_paranoid` parameter, which needs to be modified to allow monitoring.

To enable the use of almost all events for all users, you need to set this value to `-1` by executing the following command:

```
sudo sh -c 'echo -1 > /proc/sys/kernel/perf_event_paranoid'
```

3 Experiment

3.1 Java

Once the code has been correctly implemented, we adjust the values of `@Warmup`, `@Measurement`, and `@Fork` to ensure the proper execution of the benchmarking.

```
import org.openjdk.jmh.annotations.*;

import java.util.Random;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Benchmark)
@Warmup(iterations = 10, time = 1, timeUnit = TimeUnit.
    SECONDS)
@Measurement(iterations = 10, time = 1, timeUnit = TimeUnit.
    SECONDS)
@Fork(5)
public class MatrixBenchmark {

    // Parametrized matrix size
    @Param({"256", "512", "1024"})
    public int n;

    double[][] a;
    double[][] b;
    double[][] c;

    @Setup(Level.Trial)
    public void setup() {
        // Initialize matrices with random values
        a = new double[n][n];
        b = new double[n][n];
        c = new double[n][n];

        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;
            }
        }
    }

    @Benchmark
    public void testMatrixMultiplication() {
        // Perform matrix multiplication
        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];
                }
            }
        }
    }
}
```

- **@Warmup**: The `@Warmup` annotation is used to perform a series of "warm-up" iterations before actual performance measurements begin. These warm-up iterations allow the JVM (Java Virtual Machine) to reach a stable state, especially with regards to code optimization by the Just-In-Time (JIT) compiler. This is crucial because without proper warm-up, the initial iterations might produce artificially high execution times due to resource initialization, such as JIT compilation, which do not reflect the true performance of the code.

  In this case, the configuration `@Warmup(iterations = 10, time = 1, timeUnit = TimeUnit.SECONDS)` means that 10 warm-up iterations will be performed, each lasting 1 second.

- **@Measurement**: The `@Measurement` annotation specifies how many iterations will be used to measure performance after the warm-up period has completed. During these iterations, the actual execution time of the code is measured. This provides reliable and repeatable results about the code's performance under optimal conditions, after the JVM has reached a stable state following the warm-up.

  In this case, the configuration `@Measurement(iterations = 10, time = 1, timeUnit = TimeUnit.SECONDS)` means that 10 measurement iterations will be performed, each lasting 1 second, to evaluate performance.

- **@Fork**: The `@Fork` annotation controls how many times the JVM is restarted to run the benchmark. Each fork is a separate execution in

a new JVM instance. Using multiple forks is important because it ensures that the benchmark results are not affected by the previous state of the JVM, such as JIT optimizations performed during previous runs. By executing multiple forks, the benchmark can provide more reliable and consistent results, as it mitigates the effects of JVM-specific optimizations during a single run.

In this case, the configuration `@Fork(5)` means that the benchmark will be run 5 times in separate JVM instances, ensuring that the results are not biased by prior runs.

These are the obtained results for each matrix on milliseconds per operation:

Matrix Size	**256x256**
Average Time (ms/op)	15.157
Min Time (ms/op)	14.741
Max Time (ms/op)	15.579
Std Dev	0.179
Confidence Interval (99.9%)	[15.068, 15.246]

Matrix Size	**512x512**
Average Time (ms/op)	144.977
Min Time (ms/op)	142.846
Max Time (ms/op)	148.436
Std Dev	1.209
Confidence Interval (99.9%)	[144.379, 145.576]

Matrix Size	**1024x1024**
Average Time (ms/op)	1728.477
Min Time (ms/op)	1505.678
Max Time (ms/op)	2064.605
Std Dev	170.066
Confidence Interval (99.9%)	[1644.287, 1812.666]

Table 1: Benchmark Results for Matrix Multiplication in Java (256x256, 512x512, 1024x1024)

The benchmarking results for matrix multiplication in Java reveal a notable increase in computation time as the matrix size grows. With the smallest matrix size (256x256), the average execution time was 15.157 milliseconds, reflecting efficient performance with minimal variance, as shown by the small standard deviation. This suggests that Java handles smaller matrix sizes relatively well and consistently.

As the matrix size increases to 512x512, the average time required jumps to 144.977 milliseconds. This significant increase demonstrates the heavier computational load imposed by larger matrices. Despite this, the results remain

relatively stable, albeit with a slight increase in variance, as seen by the larger standard deviation compared to the smaller matrix.

When the matrix size expands further to 1024x1024, the average execution time rises dramatically to 1,728.477 milliseconds, with a noticeably wider range between minimum and maximum times. This highlights the computational strain that such large matrices place on the system, as well as the inherent fluctuations in performance.

These preliminary findings suggest that Java's performance, while stable for smaller matrices, is increasingly affected by matrix size. It also raises questions about how well this performance compares to other languages like Python and C, which will be explored in subsequent sections. The results indicate that further analysis is required to determine the suitability of Java for large-scale matrix operations and whether optimizations or alternative approaches are necessary for handling bigger datasets efficiently.

3.2 Python

Assuming the necessary libraries are installed and the test file is appropriately named, the benchmark in Python can be executed by running the following command in the console:

```
pytest test_benchmark.py --benchmark-only
```

This command will execute the test file `test_benchmark.py`, specifically focusing on the benchmarks defined within the code. The `--benchmark-only` flag ensures that pytest runs only the benchmarking tests, ignoring any other potential test cases that may be present in the same file.

You can adjust parameters globally from the command line:

```
pytest --benchmark-min-rounds=10 --benchmark-max-time=1.0
```

These are the obtained results for each matrix on seconds per operation:
Preliminary Observations:

- **Scalability and Time Performance:** The results show a significant increase in execution time as the matrix size grows. Specifically, the execution time for the 512 x 512 matrix is approximately 8.65 times greater than the 256 x 256 matrix, and the 1024 x 1024 matrix takes around 80 times longer. This indicates that Python's matrix multiplication scales poorly with larger matrix sizes, as the increase in time is not linear but rather grows exponentially as the matrix dimensions increase. This pattern suggests that as matrix sizes increase, the computational complexity involved in the matrix multiplication grows significantly, which is particularly taxing in an interpreted language like Python. Larger matrices demand more resources not just in terms of processing power but also memory usage, leading to slower processing times. Such results highlight the limitations of Python's inherent design for intensive computational

Table 2: Python Benchmarking Results for Matrix Multiplication

Matrix Size	**256x256**
Min (s)	1.6570
Max (s)	1.7270
Mean (s)	1.6841
StdDev (s)	0.0283
Median (s)	1.6852
OPS	0.5938

Matrix Size	**512x512**
Min (s)	14.3861
Max (s)	14.7407
Mean (s)	14.5666
StdDev (s)	0.1312
Median (s)	14.5760
OPS	0.0687

Matrix Size	**1024x1024**
Min (s)	134.9562
Max (s)	135.1299
Mean (s)	135.0116
StdDev (s)	0.0685
Median (s)	134.9899
OPS	0.0074

tasks and its relative inefficiency when compared to compiled languages like Java or C for these kinds of operations.

- **Operations Per Second (OPS):** The operations per second (OPS) decrease as matrix size increases, which aligns with the expectations given the higher computational demands. For the 1024 x 1024 matrix, the OPS is considerably low (0.0074), suggesting a steep decline in efficiency for larger matrices.
- **Comparison with Java:** When comparing these results with the Java benchmarking outcomes, Python exhibits longer execution times across all matrix sizes. For the 1024 x 1024 matrix, in particular, Python's performance appears much slower than Java's. These initial results suggest that Python may be less efficient for handling larger matrices in comparison to Java, although further analysis of memory usage and runtime behavior would provide a more complete understanding of the differences.

3.3 C

Assuming the necessary libraries are properly installed and the ‘perf‘ permissions have been adjusted to allow performance monitoring, the benchmarking can be executed by running the following command:

```
perf stat ./matrix
```

Where `matrix` is the executable of the code that you want to benchmark. These are the obtained results:

Table 3: Performance Results for Matrix Multiplication in C

Metric	**Value**
Task-Clock (msec)	2.96
CPU Utilization (%)	1.166
Context Switches	5
CPU Migrations	0
Page Faults	445
Time Elapsed (sec)	0.002535406
User Time (sec)	0.001100000
System Time (sec)	0.002200000

Metric	**Value**
Task-Clock (msec)	10.29
CPU Utilization (%)	0.837
Context Switches	5
CPU Migrations	0
Page Faults	1,597
Time Elapsed (sec)	0.012285740
User Time (sec)	0.004291000
System Time (sec)	0.006436000

Metric	**Value**
Task-Clock (msec)	39.20
CPU Utilization (%)	0.954
Context Switches	7
CPU Migrations	0
Page Faults	6,206
Time Elapsed (sec)	0.041074384
User Time (sec)	0.029391000
System Time (sec)	0.010134000

The benchmarking results for matrix multiplication in C show a progressive increase in execution time as the matrix size grows. For the 256x256 matrix, the

execution time is relatively fast, taking about 0.0025 seconds. CPU utilization is efficient, with a utilization rate of approximately 1.166%. The system shows minimal overhead with a low number of context switches and page faults, indicating that this operation doesn't heavily strain system resources for smaller matrices.

As the matrix size increases to 512x512, the execution time rises to 0.012 seconds, which is roughly five times longer than for the smaller matrix. Interestingly, CPU utilization drops slightly to 0.837%, suggesting that while the task becomes more computationally intensive, the system is still not fully using available CPU resources. Additionally, there is an increase in the number of page faults, which could be an early sign that memory management begins to play a more prominent role as matrix sizes grow.

With the largest matrix, 1024x1024, the execution time increases further to 0.041 seconds, which is about three times longer than for the 512x512 matrix. The CPU utilization rises again to 0.954%, with a notable spike in page faults (6,206). This reflects the additional demand larger matrices place on system resources, particularly memory management and access times. The scaling pattern here suggests that as the matrix size grows, both CPU and memory demand increase, but not always proportionally.

Comparing these results to the performance observed in Python and Java, C shows faster execution times across all matrix sizes. However, the difference in execution time between the three languages becomes more pronounced as matrix size increases. Python, for example, experiences a much sharper increase in execution time, reflecting the higher computational cost of working in an interpreted language. Java, while slower than C, performs better than Python but has its own overhead due to the Java Virtual Machine (JVM).

These observations suggest that C may have an advantage in terms of raw speed, particularly for computationally intensive tasks like matrix multiplication. However, further investigation into aspects like ease of development, memory usage, and potential for parallelization would be useful for a more comprehensive understanding of the trade-offs involved between these languages. The results open up a broader discussion on the balance between performance and other factors such as flexibility and development complexity.

4 Conclusion

This study presents a comparative analysis of matrix multiplication performance across three programming languages: Java, Python, and C. The benchmarking was conducted on matrices of varying sizes (256x256, 512x512, and 1024x1024), with each language using its own specialized benchmarking tools: JMH for Java, Pytest for Python, and Perf for C. The results provide insight into the strengths and weaknesses of each language in handling computationally intensive tasks like matrix multiplication.

C, as expected, consistently exhibited the best performance across all matrix sizes. This result is not surprising, as C is a compiled language known for its

low-level control over memory and execution, allowing it to execute operations much faster than higher-level languages. For the largest matrix (1024x1024), C's execution time was notably shorter than both Java and Python, demonstrating its capability to handle large-scale computations efficiently. The low overhead, as indicated by the minimal context switches and page faults, further highlights C's effectiveness in managing both CPU and memory resources.

Java showed a relatively strong performance, although it was slower than C. The use of the Java Virtual Machine (JVM) introduces some overhead, but Java's just-in-time (JIT) compilation helps mitigate these effects. For mid-sized matrices (512x512), Java performed reasonably well, although as matrix size grew, the overhead of the JVM became more noticeable. Nonetheless, Java provides a good balance between performance and development ease, making it a suitable option for applications where execution speed is important but not as critical as in real-time systems.

Python, on the other hand, exhibited the slowest performance across all matrix sizes. The interpreted nature of Python, combined with the lack of low-level control over memory management, makes it less suited for intensive computational tasks like matrix multiplication. For the 1024x1024 matrix, Python's performance lagged significantly behind both Java and C, with an exponential increase in execution time as matrix size grew. Despite this, Python remains a highly versatile and accessible language, especially for rapid prototyping, and is supported by a vast ecosystem of libraries that can be leveraged to overcome some of its inherent performance limitations.

In terms of scalability, all three languages demonstrated an increase in execution time as the matrix size grew. However, the rate at which performance degraded differed significantly. C showed the most linear scaling, maintaining relatively consistent efficiency even as matrix size increased. Java, while slower, still scaled in a predictable manner. Python, however, scaled poorly, with performance diminishing rapidly as matrix size grew.

In conclusion, C is the clear winner in terms of raw performance, especially for large-scale matrix operations. Java strikes a balance between performance and flexibility, offering faster execution than Python but with the added benefits of a robust development environment. Python, while slower, remains an excellent choice for applications where development speed and ease of use are prioritized over execution speed. Ultimately, the choice of language should depend on the specific requirements of the task at hand—whether raw speed, development ease, or scalability is the primary concern. Further studies could explore additional optimizations within each language, such as multi-threading or GPU acceleration, to improve matrix multiplication performance.

5 Future Work

While this study provided a comprehensive benchmarking of matrix multiplication across three different programming languages, there are several avenues for future work that could significantly enhance both the performance and appli-

cability of the current code. One of the most immediate areas of improvement would be the implementation of parallel computing techniques. Matrix multiplication is inherently parallelizable, as each element of the resulting matrix can be computed independently. Leveraging this property by distributing the workload across multiple CPU cores or threads could greatly reduce computation time, especially for larger matrices. Both Java and C offer robust libraries and frameworks for multi-threaded programming, and future versions of this study could explore their use to maximize hardware utilization.

In Python, where performance tends to lag behind due to its interpreted nature, significant gains could be made by integrating optimized numerical libraries such as NumPy, which is built on highly efficient C and Fortran code. Further enhancements could involve utilizing GPU acceleration with libraries like CuPy, which offloads matrix operations to a graphics processing unit (GPU), a hardware component designed for parallel computation. These tools are specifically designed for high-performance numerical tasks and would likely produce drastic improvements in execution times, especially for large-scale matrix operations.

Another promising direction is the exploration of more advanced matrix multiplication algorithms. The classical approach used in this study, with a time complexity of $O(n^3)$, could be replaced with Strassen's algorithm or other algorithms that reduce this complexity. Strassen's algorithm, for instance, reduces the number of multiplications needed and has a time complexity of approximately $O(n^{2.81})$. While this may not seem like a dramatic improvement for small matrices, it could provide significant time savings when dealing with very large datasets.

Optimizing memory usage is another critical area for future improvement. Large matrices not only require substantial computational resources but also significant amounts of memory. Techniques such as blocking (or tiling), which divides the matrix into smaller submatrices that fit into the CPU cache, could help reduce memory access times and improve overall performance. This could be particularly beneficial for languages like C, where memory management is more direct and can be manually fine-tuned for better results.

Additionally, future work could involve running these benchmarks on a variety of hardware platforms. The current study was conducted on standard hardware, but testing on high-performance computing clusters or systems equipped with dedicated GPUs could offer valuable insights into how these optimizations scale across different environments. This would also allow us to explore how matrix multiplication performance is affected by hardware factors such as cache size, memory bandwidth, and core count.

Finally, incorporating real-world applications into future benchmarks could provide a clearer picture of how these optimizations perform in practice. Matrix multiplication is used in fields ranging from scientific computing to machine learning, and testing with application-specific datasets could yield more practical insights.

All the code and data used in this study are available on GitHub for further extension and optimization. Contributions to the project, whether in the form of performance improvements, bug fixes, or new features, are wel-

come and encouraged. The repository can be accessed at the following link: https://github.com/Yurazu-n/BigData_Individual.