Benchmarking

Wojciech Dróżdż

October 6, 2024

Abstract

This study benchmarks the efficiency of matrix multiplication in three popular programming languages - *Java*, *Python*, and *C* - to guide decisions in high-performance computing contexts. Instead of focusing on advanced algorithms, the evaluation here uses a straightforward matrix multiplication method, considering variances in dataset size. Results indicate that the performance of these programming languages significantly varies according to the size of the dataset being processed. Python showed inferior performance handling larger matrix sizes, despite its overall popularity for user-friendly coding. Java provided considerably better performance in comparison, even with larger dataset sizes, an advantage that can likely be attributed to its advanced just-in-time compilation. Interestingly, *C*, known for its efficiency, performed slightly inferior to Java, possibly highlighting advancements in Java's runtime environment. It should be noted that the efficiency of these languages can be influenced by various factors such as specific tasks, implementations, and certain hidden language features. This study can serve as a fundamental guide when selecting languages for tasks necessitating efficient, rapid computations. Future explorations could focus on comparisons involving additional languages or different operations.

The code for this paper can be found in the GitHub repository

1 Introduction

Efficiency holds significant importance in high-performance computing, particularly when undertaking data-extensive tasks. A common operation in many applications, such as computer graphics and machine learning, is matrix multiplication—an operation that demands optimal efficiency. This paper examines how various programming languages (*Java*, *Python* and *C*) and different dataset sizes affect the performance of a standard matrix multiplication algorithm. *C* is appreciated for its speed, *Python* for its simplicity and robustness, *Java* for its memory management. This paper offers a nuanced examination of these languages' efficiency by employing a classical matrix multiplication method and studying the effect varying dataset sizes have on each one's performance. These insights will provide data for selecting an appropriate language when dealing with mathematically-intensive tasks.

2 Methodology

2.1 Hardware

All of the tests were performed on a 2023 16 inch MacBook Pro. Relevant specifications:

- Apple M3 Pro - 4 GHz 12-core CPU with 6 performance cores and 6 efficiency cores
- 18GB RAM
- 1024GB of internal storage

In order to preserve consistency all of the tests were performed while the device was plugged into a power source.

2.2 Interpreters and Compilers

- *Python* - Python 3.12.1
- *Java* - Java 17 with JetBrains runtime
- *C* - gcc compiler with *-O3* compilation argument

2.3 Benchmarking Tools

For *Python*, the *PyTest* library was utilized due to its robust and scalable testing capabilities. In the *Java* environment, the *Java Microbenchmark Harness (JHM)*, part of the *OpenJDK* project, was chosen for its microbenchmarking features. Owing to a lack of compatible software for ARM-chip-based MacBooks, a custom timing solution was employed for benchmarking *C*.

2.4 Benchmarking Methodology

The Benchmarks measure only the time it takes to multiply the matricies, all of the memory is pre-allocated before running the multiplication method. The time of memory allocation is not measured. Testing for each language was started with 10 warmup iterations before full testing, allowing for just-in-time compilations and appropriate cache optimizations. This was followed by 10 testing iterations, during which the shortest, longest, and average execution times were recorded.

3 Experiments

3.1 Python

The *Python* language was tested up to a *300x300* matrix size. Beyond that size the time of computation was too large for any practical applications.

Name (time in ms)	Min	Max	Mean
test_multiply_matrices[1]	0.000533	0.000542	0.000538
test_multiply_matrices[5]	0.025708	0.025717	0.025713
test_multiply_matrices[10]	0.203708	0.208483	0.206096
test_multiply_matrices[50]	26.023417	26.190758	26.107088
test_multiply_matrices[100]	212.143492	212.739783	212.441637
test_multiply_matrices[150]	711.709033	712.062725	711.885879
test_multiply_matrices[200]	1685.305342	1688.160000	1686.732671
test_multiply_matrices[300]	5893.233308	5946.454175	5919.843742

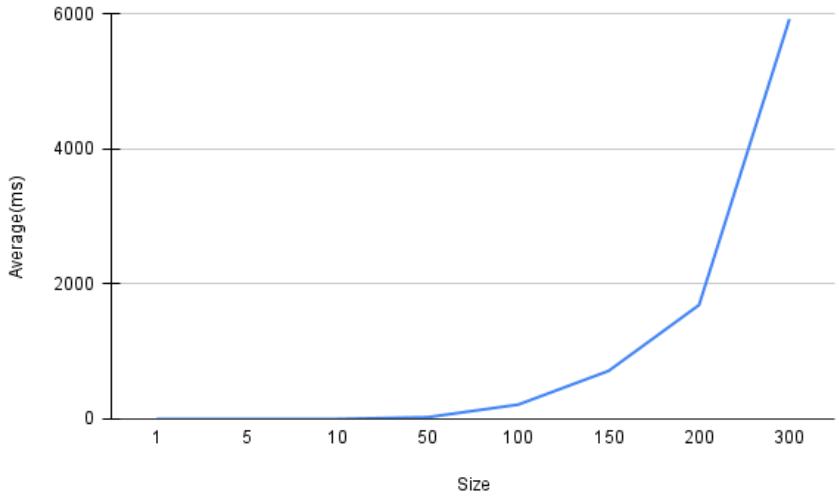

Figure 1: The average excecution times for Python

As expected the graph follows the familiar n^3 curve. For example for sizes *100x100* and *200x200* the width of the matrix increases only two times, but the execution time is eight times longer.

The standard deviation is really small between the tests and doesn't go beyond values, which could be considered a margin of error.

3.2 Java

Due to much better performance of the *JVM* the matrix size was increased up to *1500x1500*.

Benchmark	(size)	Mode	Cnt	Score	Error	Units
Main.multiplyMatrices	1	sample	276815	10		ms/op
Main.multiplyMatrices	5	sample	226653	10		ms/op
Main.multiplyMatrices	10	sample	285439	0.001 ±	0.001	ms/op
Main.multiplyMatrices	50	sample	158656	0.063 ±	0.001	ms/op
Main.multiplyMatrices	100	sample	18807	0.534 ±	0.001	ms/op
Main.multiplyMatrices	200	sample	2061	4.882 ±	0.006	ms/op
Main.multiplyMatrices	300	sample	570	17.725 ±	0.024	ms/op
Main.multiplyMatrices	500	sample	120	87.275 ±	0.669	ms/op
Main.multiplyMatrices	1000	sample	20	920.859 ±	20.387	ms/op
Main.multiplyMatrices	1500	sample	10	4064.281 ±	16.373	ms/op

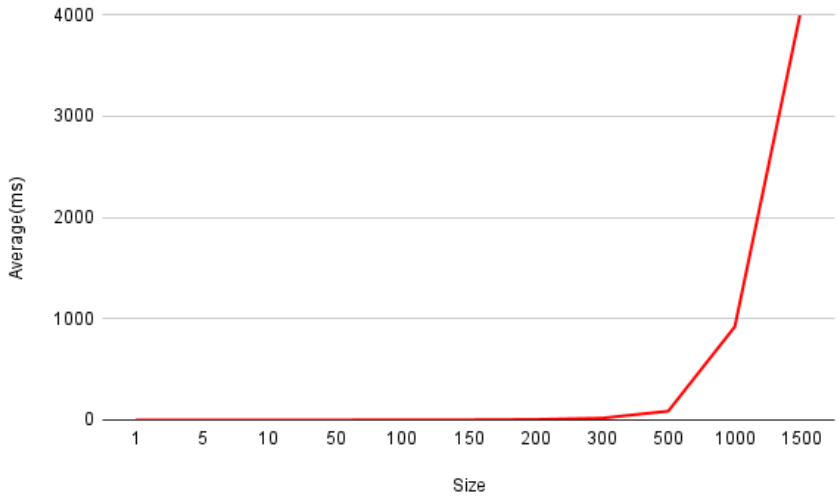

Figure 2: The average excecution times for Java

Similar to *Python* the curve resembles the n^3 curve, although the execution times are over two orders of magnitude shorter, which allows for much more complex calculations.

3.3 C

The following results were archived for the *C* language:

```
Size: 1, Min time: 0 ms, Max time: 0.000042 ms, Avg time: 0.000012 ms, Median time: 0 ms
Size: 5, Min time: 0.000125 ms, Max time: 0.000334 ms, Avg time: 0.000208 ms, Median time: 0.000187 ms
Size: 10, Min time: 0.000625 ms, Max time: 0.000875 ms, Avg time: 0.000733 ms, Median time: 0.000708 ms
Size: 50, Min time: 0.108834 ms, Max time: 0.109334 ms, Avg time: 0.109083 ms, Median time: 0.109062 ms
Size: 100, Min time: 0.927 ms, Max time: 1.156542 ms, Avg time: 1.049191 ms, Median time: 1.031688 ms
Size: 150, Min time: 2.803833 ms, Max time: 3.665292 ms, Avg time: 3.149391 ms, Median time: 3.113396 ms
Size: 200, Min time: 6.301417 ms, Max time: 7.149250 ms, Avg time: 6.539216 ms, Median time: 6.362249 ms
Size: 500, Min time: 114.875209 ms, Max time: 117.697625 ms, Avg time: 115.395454 ms, Median time: 115.026479 ms
Size: 1000, Min time: 980.939250 ms, Max time: 1163.346042 ms, Avg time: 1056.438320 ms, Median time: 1041.766646 ms
Size: 1500, Min time: 6825.023666 ms, Max time: 7226.068375 ms, Avg time: 7032.894891 ms, Median time: 6993.858604 ms
```

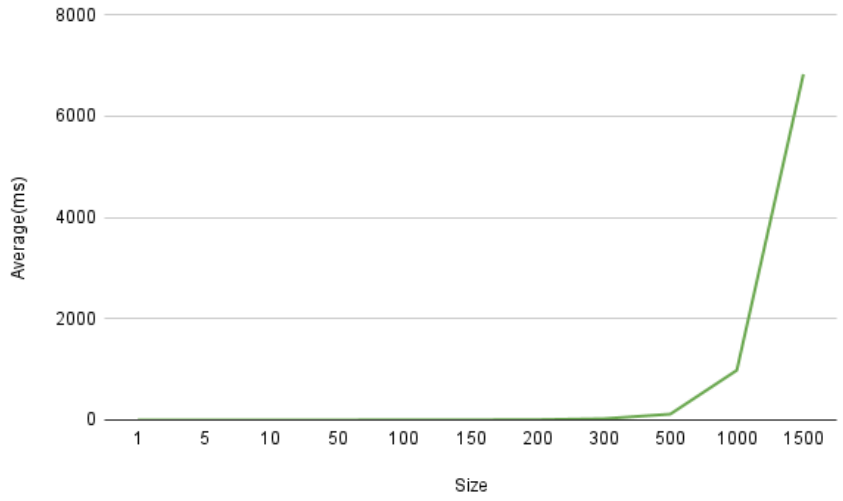

Figure 3: The average excecution times for C

Suprisingly the *C* language performed worse than *Java*, this is most likely, because the *JVM* implements more optimizations than the *-O3* optimization level of the *gcc* compiler.

4 Conclusion

Study focused on comparing the performance of three programming languages — *Python*, *Java*, and *C* — when implementing matrix multiplication, a standard computation in high-performance computing. All languages were evaluated on varying dataset sizes to produce robust results.

Python, generally known for its simplicity and user-friendliness, unfortunately, lacks in performance when faced with larger matrix sizes. Despite its powerful toolkits for scientific computation, its relatively slower execution speed makes it less preferred for applications where quick computation is vital.

Java, contrary to *Python*, showed substantial performance, even for larger matrix sizes. This can be attributed to Java's advanced *just-in-time (JIT)* compilation and several runtime optimizations, which results in highly efficient machine code.

Surprisingly, *C*, known for its efficiency and speed, performed slightly worse than *Java* in this experiment. Although slightly counterintuitive, this might highlight the advanced optimizations done by the *Java* runtime environment. However, keep in mind that C's granular control over hardware and memory management may still make it the language of choice when maximized performance is the primary concern.

This study also revealed that the dataset size considerably impacts the execution speed, which is expected according to the theory of matrix multiplication. However, when extrapolating these results, it's important to note that language performance may vary greatly depending on the specific task, the implementation, and several hidden language features.

This study can be considered a guideline in choosing the language for high-performance computing. Choice of language should take into account multiple factors such as task complexity, dataset size, and the need for rapid and efficient computation. Further studies could include more languages or various operations to deepen our understanding of performance trade-offs in high-performance computing.

5 Future work

The matrix multiplication problem can be improved a lot. This study measures the speed in simplest of approaches. There are numerous improvements that can be implemented. A faster calculation algorithm could be used for the calculations, a multi-threaded solution can be relatively easily implemented, moreover a GPU is optimal for matrix operation, the CPU was used in this experiment. Moreover there are some language specific improvements that could be used in this experiment. For *Python* the *Numpy* library could be used to store and perform the operations on the arrays, which should allow a dramatic performance improvement. Moreover there are multiple techniques of speeding up the C execution (for example the *register* keywoard), which are beyond the scope of this paper and would likely significantly speed up the execution.