Matrix multiplication

Nicolas Trujillo Estévez
ULPGC

October 19, 2024

Abstract

This paper discusses the use of benchmarking to test execution time and memory usage through libraries, all focused on a general matrix multiplication code across various languages (Python, Java, and C) and multiple versions of the code. The methodology involved conducting all tests with the device freshly powered on to ensure fair conditions for all languages. First, the Python tests were performed in the PyCharm framework, followed by the Java tests in the IntelliJ framework, and finally, the C tests in a virtual machine running Linux software.

The results, while each language returns a very different range of outcomes, share common elements that allow us to compare the performance of each language and the versions of the code. The most significant metrics are the time in milliseconds per operation and memory allocation. We clearly observe a defeat, so to speak, for the Python language compared to the others. On the other hand, Java yields slightly better results than C, but they are very similar; this is due to the fact that, while C should be faster, it is running in a virtual machine, which affects its performance.

1 Introduction / Context / Motivation

The problem began, as inferred from the outset, with the need to evaluate performance regarding execution time and memory usage of matrix multiplication code in different languages, while optionally observing whether improvements can be achieved by making code variations. The main reason for this is the difference among these three languages, with Python being further from the machine compared to Java and C, and C being the closest. Furthermore, Python is dynamically interpreted, while Java and C are dynamically compiled. It is widely known that Python is generally slower; a general metric suggests it is approximately 10 to 100 times slower than C and 10 to 20 times slower than Java. Therefore, it is important to conduct such tests to determine in which situations a code performs better in which language.

2 Problem Statement

As a specific definition of the problem, we have three basic versions of a matrix multiplication code with a complexity of $O(n^3)$ in three different languages. As tools, we have those provided in a benchmarking practice for the course. The objectives are to compare the three codes using these tools, consider improvements for those codes, and finally compare the memory usage of the three codes.

3 Solution / Proposal / Method / Methodology

To begin with Python, I created several functions in a file, starting with "my$_{f}unction()$", $andsubsequentlytested$

To run this code, you first need to install `pytest-benchmark` to perform the performance test. Execute the command `pytest -s --benchmark-save=my_benchmark` in the terminal of

the framework you are using. This is similar to a practice available on campus, but you add the `-s` option so that the output of the code is printed on the screen. This is because, for the memory test, I used the `memory_profiler` library, which you can install in the same way, and I added a variable using `memory_usage` to store the memory test results and print them to the screen.

Afterward, I continued with Java, the most complicated of all due to the considerable time it took; therefore, I did not make optimizations and stuck with the general code. To give you an idea, the test with parameters included, using, for example, an optimization that divided the matrix into blocks, took around 5 hours. I created a file with the function and another for testing.

As I told you before, this was the most complicated language to test. First, you have to add the dependencies to your pom.xml: `org.openjdk.jmh jmh-core 1.35` and `org.openjdk.jmh jmh-generator-annprocess 1.35`; then you install the JMH plugin if you are using IntelliJ. After that, you add a special build to the pom.xml to make it work.

After that, you only need to execute these commands in the shell: `mvn clean install` and `java -jar target/mybenchmark.jar MyBenchmark -prof gc`; now you have a .jar file that you execute to test the benchmarks and memory usage. Of course, you must use the name of the jar that appears in your target.

Finally, with C, I created three files, one for each version of the matrix multiplication code, and tested them for different matrix sizes. To execute the code, once you install the necessary packages (perf and gcc), you compile the code and create an executable. Then you only need to use these commands in your shell: `perf stat ./my_program` to perform the performance test and `/usr/bin/time -v ./my_program` to evaluate memory usage; the `time` command is available in most Linux distributions.

It is worth mentioning that while all the tests yielded different results and cannot be easily compared between languages, there are several metrics that help compare, in this case, the base matrix multiplication code among its different versions in different languages. Moreover, in the case of Python and C, I was able to make comparisons between different versions of the code, which yielded many interesting results.

4 Experiments

Here, I present three tables that summarize the test results of the three languages for the base version of the code, making them easily comparable. Starting with Python, we observe the average time in milliseconds per operation for a total of 5 iterations, and, on the other hand, the maximum value from the memory usage list every 0.1 seconds in MB. In Java, we observe the average time per operation for a total of 25 iterations, as well as the memory usage in bytes per operation. Finally, in the C table, we observe two time units that are better explained together: in the case of the C code, there was a print statement that output the time for the matrix multiplication operation. That value corresponds to the time unit but implies higher computation costs reflected in the task-clock unit returned by the test. This is why, for small values of n up to 100, you see anomalous values that differ significantly between the two units, but beyond that, they stabilize and become more similar. Additionally, it's important to note that, as mentioned before, this code was executed on a virtual machine, which affects its real performance. On the other hand, there is another column with the maximum memory consumption value for the operation in KB.

Table 1: Python Results

n	Mean for 5 iterations (ms)	Biggest value of memory usage every 0.1 seconds (MB)
10	0.1103	77.5859375
50	10.6537	77.32421875
100	79.0397	78.08203125
200	603.0017	81.34375
500	9915.18	106.3828125
1024	104384.84	200.8828125

Table 2: Java Results

n	Time average for 25 iterations (ms/op)	Normalized Allocation Rate (B/op)
10	0.003	24004
50	0.126	24540
100	0.881	24039
200	6.760	24282
500	139.898	29883
1024	2985.114	128560

Table 3: C Results

n	task-clock for 1 iteration (ms)	time (ms/op)	Max Resident Set Size (kb)
10	0.49	0.001	1416
50	0.75	0.032	1572
100	1.25	0.337	1864
200	5.13	5.265	2508
500	59.45	49.485	7372
1024	3683.12	3775.855	26172

On the other hand, here I have the same tables but with more information so that you have the opportunity to analyze more variables if you are interested:

Table 4: Results of measurements in Python

n	Min (ms)	Max (ms)	Mean (ms)	StdDev (ms)	Median (ms)	IQR (ms)	OPS (Kops/s)	Rounds
10	0.1070	0.2391	0.1103	0.0035	0.1095	0.0018	9.07	6902
50	10.5111	11.1474	10.6537	0.1199	10.6491	0.1212	0.93	91
100	78.4027	80.5734	79.0397	0.6522	78.7478	0.0127	13.00	13
200	596.9171	606.0572	603.0017	3.5364	604.0239	3.1115	1.66	5
500	9723.23	10116.13	9915.18	149.5918	9895.18	213.5543	0.10	5
1024	103262.43	105634.44	104384.84	1070.4907	103875.28	1861.2545	0.01	5

Table 5: Results of measurements in Java

n	Mode	Count	Time (ms/op)	Error	Allocation Rate (MB/sec)	Normalized Allocation Rate (B/op)	G1 Eden Space Churn (MB/sec)	G1 Eden Space (B/op)
10	avgt	25	0.003	± 0.001	8.118	24004	8.221	24.305
50	avgt	25	0.126	± 0.001	0.176	24540	0.251	35.076
100	avgt	25	0.881	± 0.001	0.025	24039	-	-
200	avgt	25	6.760	± 0.016	0.003	24282	-	-
500	avgt	25	139.898	± 0.376	-	29883	-	-
1024	avgt	25	2985.114	± 28.254	-	128560	-	-

Table 6: Memory Results in C

n	CPU (%)	Time (ms)	Max Resident Set Size (kB)	Minor Page Faults	Page Size (bytes)
10	0%	0.001	1416	72	4096
50	0%	0.032	1572	85	4096
100	50%	0.337	1864	133	4096
200	50%	5.265	2508	308	4096
500	79%	49.485	7372	1536	4096
1024	93%	3775.855	26172	6220	4096

Table 7: Results of measurements in C

n	task-clock (msec)	context-switches	cpu-migrations	page-faults	time elapsed (sec)	user time (sec)	sys time (sec)
10	0.49	0	0	57	0.000193184	0.0	0.001004000
50	0.75	0	0	73	0.000186190	0.0	0.001174000
100	1.25	0	0	117	0.000606661	0.001705000	0.0
200	5.13	0	0	291	0.004315580	0.005539000	0.0
500	59.45	0	0	1522	0.059730923	0.056919000	0.001967000
1024	3683.12	0	0	6202	3.722402522	3.568677000	0.011730000

To continue and as a complement to the comparison between languages, here are the remaining tables with less summarized information that is useful for analyzing performance improvements between different versions of the same language:

4.1

Python Results:

4.1.1

Benchmark Results (n = 10)

Metric	**Description**
Min (ms)	Minimum time in milliseconds per iteration
Max (ms)	Maximum time in milliseconds per iteration
Mean (ms)	Average time in milliseconds per iteration
StdDev (ms)	Standard deviation in milliseconds
Median (ms)	Median in milliseconds
IQR (ms)	Interquartile range in milliseconds
OPS (Kops/s)	Operations per second (thousands)
Rounds	Total iterations

Table 9: Function Measurement Results

Function	**Min (ms)**	**Max (ms)**	**Mean (ms)**	**StdDev (ms)**	**Median (ms)**	**IQR (ms)**	**OPS (Kops/s)**	**Rounds**
test_my_function2	0.0039	0.0140	0.0043	0.0004	0.0044	0.0005	230.90	10320
test_my_function3	0.0045	0.0265	0.0050	0.0005	0.0051	0.0004	200.73	15106
test_my_function	0.1070	0.2391	0.1103	0.0035	0.1095	0.0018	9.07	6902
test_my_function4	0.1292	0.2452	0.1348	0.0065	0.1336	0.0013	7.42	6494
test_my_function5	0.2378	0.4013	0.2451	0.0121	0.2425	0.0038	4.08	1065

4.1.2

Memory Usage

- Memory usage for my_function with n=10: 77.5859375 MB

- Memory usage for my_function2 with n=10: 78.20703125 MB
- Memory usage for my_function3 with n=10: 79.00390625 MB
- Memory usage for my_function4 with n=10: 79.27734375 MB
- Memory usage for my_function5 with n=10: 79.62109375 MB

4.1.3

Benchmark Results (n = 50)

Metric	Description
Min (ms)	Minimum time in milliseconds per iteration
Max (ms)	Maximum time in milliseconds per iteration
Mean (ms)	Average time in milliseconds per iteration
StdDev (ms)	Standard deviation in milliseconds
Median (ms)	Median in milliseconds
IQR (ms)	Interquartile range in milliseconds
OPS (Kops/s)	Operations per second (thousands)
Rounds	Total iterations

Table 11: Function Measurement Results

Function	Min (ms)	Max (ms)	Mean (ms)	StdDev (ms)	Median (ms)	IQR (ms)	OPS (Kops/s)	Rounds
test_my_function2	0.0383	0.0716	0.0388	0.0012	0.0386	0.0002	25.75	7831
test_my_function3	0.0390	0.1275	0.0396	0.0018	0.0393	0.0003	25.27	9329
test_my_function5	0.4421	0.6194	0.4556	0.0135	0.4523	0.0067	2.19	1168
test_my_function4	10.4024	10.7481	10.4989	0.0602	10.4851	0.0611	0.95	93
test_my_function0	10.5111	11.1474	10.6537	0.1199	10.6491	0.1212	0.93	91

4.1.4

Memory Usage

- Memory usage for my_function with n=50: 77.32421875 MB
- Memory usage for my_function2 with n=50: 77.625 MB
- Memory usage for my_function3 with n=50: 78.1640625 MB
- Memory usage for my_function4 with n=50: 78.7265625 MB
- Memory usage for my_function5 with n=50: 79.09375 MB

4.1.5

Benchmark Results (n = 100)

Metric	Description
Min (ms)	Minimum time in milliseconds per iteration
Max (ms)	Maximum time in milliseconds per iteration
Mean (ms)	Average time in milliseconds per iteration
StdDev (ms)	Standard deviation in milliseconds
Median (ms)	Median in milliseconds

IQR (ms)	Interquartile range in milliseconds
OPS (Kops/s)	Operations per second (thousands)
Rounds	Total iterations

Table 13: Function Measurement Results

Function	Min (ms)	Max (ms)	Mean (ms)	StdDev (ms)	Median (ms)	IQR (ms)	OPS (Kops/s)	Rounds
test_my_function3	0.2242	0.3728	0.2596	0.0095	0.2577	0.0069	3.85	2383
test_my_function2	0.2471	0.3648	0.2597	0.0103	0.2574	0.0073	3.85	803
test_my_function5	1.1931	2.1518	1.4442	0.2469	1.3125	0.3700	0.69	523
test_my_function4	78.4027	80.5734	79.0397	0.6522	78.7478	0.8859	0.0127	13
test_my_function0	81.5918	84.9163	82.6614	0.9509	82.5145	1.2788	0.0121	13

4.1.6

Memory Usage

- Memory usage for my_function with n=100: 78.08203125 MB
- Memory usage for my_function2 with n=100: 79.15234375 MB
- Memory usage for my_function3 with n=100: 79.3046875 MB
- Memory usage for my_function4 with n=100: 80.3046875 MB
- Memory usage for my_function5 with n=100: 80.828125 MB

4.1.7

Benchmark Results (n = 200)

Metric	Description
Min (ms)	Minimum time in milliseconds per iteration
Max (ms)	Maximum time in milliseconds per iteration
Mean (ms)	Average time in milliseconds per iteration
StdDev (ms)	Standard deviation in milliseconds
Median (ms)	Median in milliseconds
IQR (ms)	Interquartile range in milliseconds
OPS (Kops/s)	Operations per second (thousands)
Rounds	Total iterations

Table 15: Function Measurement Results

Function	Min (ms)	Max (ms)	Mean (ms)	StdDev (ms)	Median (ms)	IQR (ms)	OPS (Kops/s)	Rounds
test_my_function2	0.8339	1.1420	0.9386	0.0461	0.9304	0.0488	1,065.40	1114
test_my_function3	0.8424	1.2993	0.9275	0.0436	0.9195	0.0427	1,078.15	837
test_my_function5	3.8658	5.5562	4.3611	0.3257	4.3034	0.5136	229.30	228
test_my_function4	596.9171	606.0572	603.0017	3.5364	604.0239	3,111.53	1.66	5
test_my_function0	640.7745	652.1618	646.2290	4.4754	647.3267	6,629.98	1.55	5

4.1.8

Memory Usage

- Memory usage for my_function with n=200: 81.34375 MB
- Memory usage for my_function2 with n=200: 81.37890625 MB

- Memory usage for my_function3 with n=200: 81.4375 MB
- Memory usage for my_function4 with n=200: 85.171875 MB
- Memory usage for my_function5 with n=200: 83.89453125 MB

4.1.9

Benchmark Results (n = 500)

Metric	Description
Min (ms)	Minimum time in milliseconds per iteration
Max (ms)	Maximum time in milliseconds per iteration
Mean (ms)	Average time in milliseconds per iteration
StdDev (ms)	Standard deviation in milliseconds
Median (ms)	Median in milliseconds
IQR (ms)	Interquartile range in milliseconds
OPS (Kops/s)	Operations per second (thousands)
Rounds	Total iterations

Table 17: Function Measurement Results (Long Time)

Function	Min (ms)	Max (ms)	Mean (ms)	StdDev (ms)	Median (ms)	IQR (ms)	OPS (Kops/s)	Rounds
test_my_function2	5.4468	7.5777	5.8776	0.3250	5.7953	0.3282	170.14	177
test_my_function3	5.4743	7.7198	5.9153	0.2419	5.8915	0.2221	169.05	164
test_my_function5	27.4608	29.4218	28.4789	0.4850	28.5057	0.7343	35.11	33
test_my_function4	9723.23	10116.13	9915.18	149.5918	9895.18	213.5543	0.10	5
test_my_function0	10557.59	10833.72	10666.69	102.9756	10636.21	107.4227	0.09	5

4.1.10

Memory Usage

- Memory usage for my_function with n=500: 106.3828125 MB
- Memory usage for my_function2 with n=500: 96.6484375 MB
- Memory usage for my_function3 with n=500: 96.70703125 MB
- Memory usage for my_function4 with n=500: 119.5703125 MB
- Memory usage for my_function5 with n=500: 103.84375 MB

4.1.11

Benchmark Results (n = 1024)

Metric	Description
Min (ms)	Minimum time in milliseconds per iteration
Max (ms)	Maximum time in milliseconds per iteration
Mean (ms)	Average time in milliseconds per iteration
StdDev (ms)	Standard deviation in milliseconds
Median (ms)	Median in milliseconds
IQR (ms)	Interquartile range in milliseconds

OPS (Kops/s)	Operations per second (thousands)
Rounds	Total iterations

Table 19: Function Measurement Results (Extreme Long Time)

Function	Min (ms)	Max (ms)	Mean (ms)	StdDev (ms)	Median (ms)	IQR (ms)	OPS (Kops/s)	Rounds
test_my_function2	25.1936	29.3659	26.4792	0.7494	26.5036	0.9876	37.77	39
test_my_function3	26.3879	31.1307	27.6497	1.0116	27.4518	0.7493	36.17	35
test_my_function5	126.0056	133.9618	129.6547	2.2462	129.5281	1.7157	7.71	8
test_my_function4	85212.54	87086.43	85924.73	827.2586	85554.33	1381.0959	0.01	5
test_my_function0	103262.43	105634.44	104384.84	1070.4907	103875.28	1861.2545	0.01	5

4.1.12

Memory Usage

- Memory usage for my_function with n=1024: 200.8828125 MB
- Memory usage for my_function2 with n=1024: 119.34765625 MB
- Memory usage for my_function3 with n=1024: 122.109375 MB
- Memory usage for my_function4 with n=1024: 251.39453125 MB
- Memory usage for my_function5 with n=1024: 149.10546875 MB

4.2

Results C:

4.2.1

Benchmark Results (V1)

Table 20: Matrix Multiplication Benchmark Results (V1)

Version	n	task-clock (msec)	context-switches	cpu-migrations	page-faults	time elapsed (sec)	user time (sec)	sys time (sec)
V1	10	0.49	0	0	57	0.000193184	0.0	0.001004000
V1	50	0.75	0	0	73	0.000186190	0.0	0.001174000
V1	100	1.25	0	0	117	0.000606661	0.001705000	0.0
V1	200	5.13	0	0	291	0.004315580	0.005539000	0.0
V1	500	59.45	0	0	1522	0.059730923	0.056919000	0.001967000
V1	1024	3683.12	0	0	6202	3.722402522	3.568677000	0.011730000

4.2.2

Memory Usage Results (V1)

Table 21: Memory and CPU Test Results (V1)

Version	n	CPU Usage (%)	Total Execution Time (sec)	Max Resident Set Size (kB)	Minor Page Faults	Page Size (bytes)
V1	10	0	0.000001	1416	72	4096
V1	50	0	0.000032	1572	85	4096
V1	100	50	0.000337	1864	133	4096
V1	200	50	0.005265	2508	308	4096
V1	500	79	0.049485	7372	1536	4096
V1	1024	93	3.775855	26172	6220	4096

4.2.3

Benchmark Results (V2)

Table 22: Matrix Multiplication Benchmark Results (V2)

Version	n	Task Clock (msec)	Context Switches	CPU Migrations	Page Faults	Time Elapsed (sec)	User Time (sec)	Sys Time (sec)
V2	10	0.50	0	0	58	0.000000421	0.0	0.0
V2	50	0.80	0	0	72	0.000334028	0.0	0.001203
V2	100	1.30	0	0	113	0.000715666	0.0	0.001788
V2	200	5.31	0	0	292	0.005297907	0.001966	0.003876
V2	500	59.05	0	0	1520	0.062273733	0.054333	0.003864
V2	1024	3798.08	0	0	6203	3.859689869	3.668088	0.014544

4.2.4

Memory and CPU Test Results (V2)

Table 23: Memory and CPU Test Results (V2)

Version	n	CPU Usage (%)	Total Execution Time (sec)	Max Resident Set Size (kB)	Minor Page Faults	Page Size (bytes)
V2	10	0	0.000000	1468	74	4096
V2	50	0	0.000029	1572	84	4096
V2	100	33	0.000262	1724	131	4096
V2	200	36	0.004779	2592	307	4096
V2	500	75	0.047531	7316	1538	4096
V2	1024	94	3.817186	26064	6218	4096

4.2.5

Benchmark Results (V3)

Table 24: Matrix Multiplication Benchmark Results (V3)

n	Task Clock (msec)	Context Switches	CPU Migrations	Page Faults	Time Elapsed (sec)	User Time (sec)	Sys Time (sec)	
V3	10	0.54	0	0	58	0.000270799	0.000612000	0.000612000
V3	50	0.70	0	0	73	0.000253867	0.0	0.001120000
V3	100	1.67	0	0	117	0.001392890	0.001041000	0.001033000
V3	200	5.23	0	0	292	0.004625670	0.005670000	0.0
V3	500	61.70	0	0	1523	0.064453849	0.055215000	0.005742000
V3	1024	3606.33	0	0	6203	3.702112890	3.463525000	0.026961000

4.2.6

Memory and CPU Test Results (V3)

Table 25: Memory and CPU Test Results (V3)

Version	n	CPU Usage (%)	Total Execution Time (sec)	Max Resident Set Size (kB)	Minor Page Faults	Page Size (bytes)
V3	10	0	0.000001	1568	72	4096
V3	50	0	0.000030	1656	88	4096
V3	100	25	0.000316	1740	131	4096
V3	200	41	0.005979	2512	307	4096
V3	500	80	0.047846	7424	1537	4096
V3	1024	95	3.709086	26176	6217	4096

5 Conclusion

First, as the most relevant conclusion, we can comment on the comparison of the different languages for the same version. At the beginning of the paper, we briefly explained that Python generally has the worst performance compared to Java and C, in that order. The results corroborate this; Python shows poorer results in the benchmark tests compared to Java and C. Although, at low dimensions, they behave similarly, as we increase the size of n, Python diverges further from the other languages. If we look at Tables 1, 2, and 3, particularly the time columns, we see that for $n = 1024$, Python takes approximately 10 seconds, while Java and C take about 3 seconds. Moreover, if we examine the memory data, we find that while Java and C

are significantly better in memory consumption, Java is much better than C. Specifically, Java uses about 24 kB with $n = 1024$, whereas C uses 26172 kB in that case. In summary, both languages outperform Python, and the anomalous results for C, which should be the best, are due, as I mentioned earlier, to the virtual machine and the presence of a "print" statement.

To conclude, the other conclusions are that, in the case of Python, the most optimized version of the code with the best results is the one that uses "numpy," as it consistently performs better across all values of n. In contrast, the original implementation, while performing acceptably for $n = 10$, is the worst by a considerable margin in the benchmark tests, although it shows acceptable memory usage except for $n = 1024$. In the case of C, both the original and modified versions behave similarly, with nothing particularly noteworthy.

These results are relevant as they reiterate what everyone already knows: Python generally has the worst performance, primarily because it is an interpreted language that is further removed from the machine.

6 Future Work

One possible improvement for this experiment is, of course, to create several versions of the code in Java, remove the print statement from the C code, parameterize more for Python and C, and, above all, interpret the results through graphs in another language, such as R.