Basic Matrix Multiplication in Different Languages

Elisa-Marie Breeze Rebstock

October 2024

Abstract

This document presents an analysis of the performance of a Matrix Multiplication Algorithm, comparing three different programming languages: Python, Java and C. The objective is to evaluate the efficiency of each language by making use of Benchmark tools and other tools to evaluate the memory usage and execution times, as the matrix increases in size. The methodology includes clean designs and maintainability of the code by separating the implementation of the matrix multiplication algorithm and the Benchmark code. The experiments reveal that Python has the worst performance of the three languages, Java performs quite a lot better than Python, but C demonstrated the best efficiency overall. With this analysis, it is shown how different programming languages optimize the matrix multiplication, and it demonstrates which programming language is the right one for computationally intensive tasks.

1 Introduction

The idea of the assignment is to compare the performance of basic matrix multiplication, implementing the algorithm in Python, Java and C, to find out which programming language is more efficient, which will be analyzed with Benchmark tools.

With this, it is going to be visible how each programming language is different and how their specific optimization and features influence the results.

Matrix multiplication becomes an intensive task as the matrix sizes increase, so using Benchmark tools to see how each language reacts to increasing the size is going to tell us which is more efficient, thanks to different performance metrics like execution time and memory usage.

In this paper, the first thing shown will be the methodology used to benchmark, with an insight to the specific codes. Following this, there will be a series of experiments with their respective results, that will show us how the performance of each programming language differs with the increasing matrix sizes, which will also be discussed.

2 Methodology

Link to the GitHub Repository of this assignment:
https://github.com/ElisaBreeze/BigData.git

Firstly, I implemented the code of the Matrix Multiplication algorithm in the different languages, and then to ensure a clean design and easy maintainability I implemented the benchmark testing code separately.

The code to the matrix multiplication algorithm and the basic Benchmark methods were provided by the teachers, so I used that as a base, making some changes so that it worked as wanted. The Benchmark strategies given by the teachers only analyzed the execution times, so I researched on how to measure the memory usage, finding useful information on different programming websites and with a little help from AI and consulting options with different classmates, I managed to organize all the found information into one code that worked for my existing code, without having to change it too much, so that it would still stay simple and understandable.

2.0.1 Python

In Python, the Benchmark method was no problem, I just added a line of code into the Benchmark code, which converted the output data into milliseconds, so that all the tests would be in the same unit. On the other hand, to calculate the memory usage I used a package called Memory Profiler. By adding ”@profile” before the function, I was able to get information about the memory usage. So that it worked correctly, after the algorithm, I called the function so that when executing it, it would give back the information I was looking for.

To get the information I needed for the analysis, I executed it various times, to ensure that the data was consistent and there was no bias due to first time execution. To get the information needed, I used the following commands:

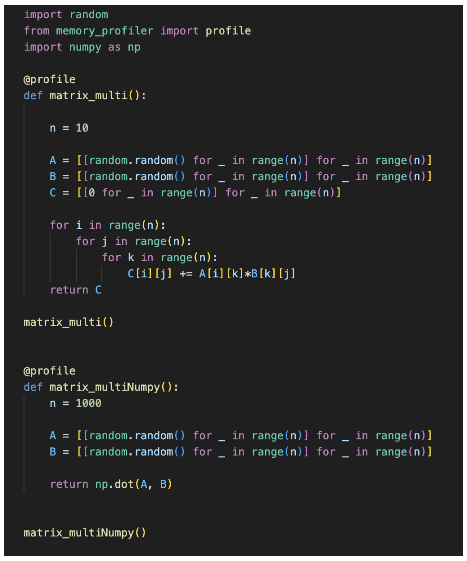
- ”pytest benchmarking.py –benchmark-only” for execution times
- ”python -m memory_profiler matrix_multiplication.py” for memory usage

It worked well, giving me all the analysis information I needed for the sizes 10 and 100, but for n=1000 it would not give out any information, as it seemed to be way to big for it to finish analyzing the memory usage. To solve this and manage to get information for n=1000, I decided to look for a way to optimize the code so that it would use less and execute in a more efficient way, making it possible to get the results I needed.

To optimize the Matrix Multiplication Algorithm, what I did was use the library Numpy, which includes a function that multiplies the matrix automatically, simply by executing this code: ”np.dot(a, b)”. I included this code in a new function called matrix_multiNumpy, so that I could do testing on both separately and compare them. I executed both versions for the sizes 10 and 100, giving me the same memory usage information; and to test size 1000, y commented the code line ”@profile” from the original matrix multiplication, so that it would only analyse the optimized version and give me the information I needed.

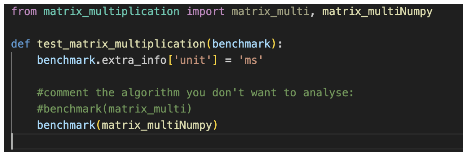
After that, I checked the execution times of the original and the optimized methods, and surprisingly the new execution times where much faster that the original algorithm, as you can see in the table in the results section below.

Here you can see the code in Python. The first picture shows the Matrix Multiplication code, and the second one, the Benchmark code:



```python
import random
from memory_profiler import profile
import numpy as np

@profile
def matrix_multi():

    n = 10

    A = [[random.random() for _ in range(n)] for _ in range(n)]
    B = [[random.random() for _ in range(n)] for _ in range(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

matrix_multi()


@profile
def matrix_multiNumpy():
    n = 1000

    A = [[random.random() for _ in range(n)] for _ in range(n)]
    B = [[random.random() for _ in range(n)] for _ in range(n)]

    return np.dot(A, B)


matrix_multiNumpy()
```

Figure 1: Matrix Multiplication Functions with the Memory Profiler.



```python
from matrix_multiplication import matrix_multi, matrix_multiNumpy

def test_matrix_multiplication(benchmark):
    benchmark.extra_info['unit'] = 'ms'

    #comment the algorithm you don't want to analyse:
    #benchmark(matrix_multi)
    benchmark(matrix_multiNumpy)
```

Figure 2: Benchmark Code.

2.0.2 Java

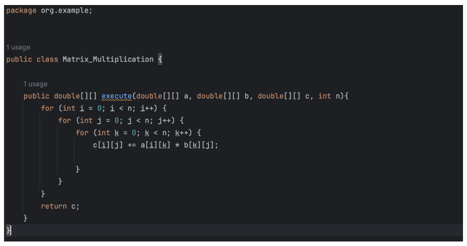
In java, I decided to implement a similar code to the example code given by the teachers, but I did a few changes that made the code look tidier and simpler: in the testing code, I added the solution matrix (c) to the setup code, as in that way, it was all in one place and more organized. I also passed on the size of the matrix as a value to the matrix multiplication function, as it was already initialized for the setup function. This way, the code is more optimized because it doesn't have to calculate the matrix size in the matrix multiplication code, but instead it's given to it automatically.

After a lot of researching and trying out codes, I found a way to calculate the memory usage in an efficient way that worked well:

I made use of Runtime functions, and calculated the memory by subtracting the free memory to the used memory, after the Benchmark execution of the Matrix Multiplication Algorithm. To show it only at the end of the Benchmark execution, I added a @TearDown function, which executes at the very end, and in it, I printed out the result of the calculation, so that it gives back the information looked for.

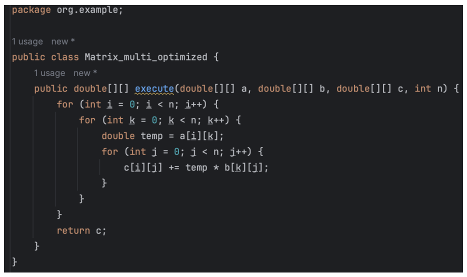
Once I did that, I decided to think about ways to optimize the code, and I remembered that in theory class the teacher showed us, how accessing the matrix affects the performance as well. That is why I tried a different method: I decided to change the order of accessing to a more sequential order by switching k and j and storing the value of the matrix a, to avoid multiple accesses to the same element.

In the first two pictures, you can see the different Matrix Multiplication algorithms, and the next two pictures are of the codes in Java for the Benchmark, where to execute the original function or the optimized, you just have to change the name in the benchmark to the desired one:



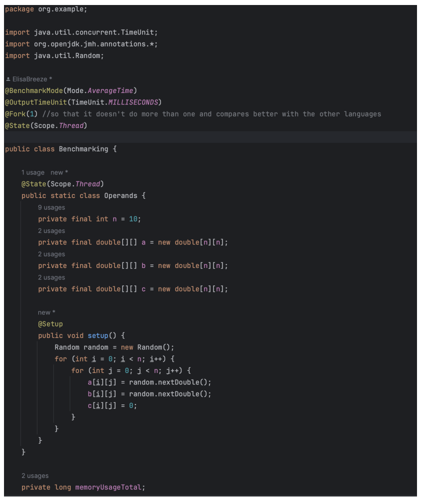
```java
package org.example;


1 usage
public class Matrix_Multiplication {

    1 usage
    public double[][] execute(double[][] a, double[][] b, double[][] c, int n){
        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];

                }
            }
        }
        return c;
    }
}
```

Figure 3: Matrix Multiplication Function.

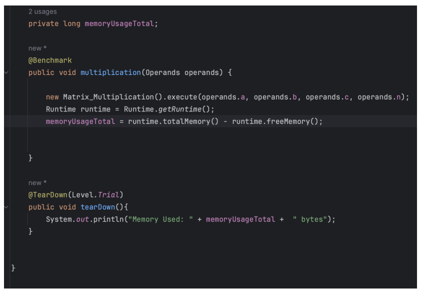


```java
package org.example;

1 usage   new *
public class Matrix_multi_optimized {
    1 usage   new *
    public double[][] execute(double[][] a, double[][] b, double[][] c, int n) {
        for (int i = 0; i < n; i++) {
            for (int k = 0; k < n; k++) {
                double temp = a[i][k];
                for (int j = 0; j < n; j++) {
                    c[i][j] += temp * b[k][j];
                }
            }
        }
        return c;
    }
}
```

Figure 4: Optimized Matrix Multiplication Function.



```
package org.example;

import java.util.concurrent.TimeUnit;
import org.openjdk.jmh.annotations.*;
import java.util.Random;

ElisaBreeze *
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@Fork(1) //so that it doesn't do more than one and compares better with the other languages
@State(Scope.Thread)

public class Benchmarking {

    1 usage   new *
    @State(Scope.Thread)
    public static class Operands {
        9 usages
        private final int n = 10;
        2 usages
        private final double[][] a = new double[n][n];
        2 usages
        private final double[][] b = new double[n][n];
        2 usages
        private final double[][] c = new double[n][n];

        new *
        @Setup
        public void setup() {
            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;
                }
            }
        }
    }

    2 usages
    private long memoryUsageTotal;
```



```
    2 usages
    private long memoryUsageTotal;

    new *
    @Benchmark
    public void multiplication(Operands operands) {

        new Matrix_Multiplication().execute(operands.a, operands.b, operands.c, operands.n);
        Runtime runtime = Runtime.getRuntime();
        memoryUsageTotal = runtime.totalMemory() - runtime.freeMemory();


    }

    new *
    @TearDown(Level.Trial)
    public void tearDown(){
        System.out.println("Memory Used: " + memoryUsageTotal +  " bytes");
    }


}
```

Figure 5: Benchmark Code.

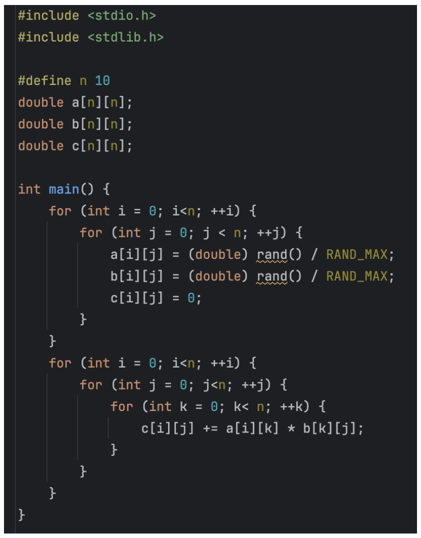
2.0.3 C

In C, only the code for the Matrix Multiplication had to be done, the Benchmark process could be done through the terminal.

To get the information about execution time, I used the command explained in the theory class, which is ”perf stat -e task-clock ./matrix”. And to find out the information about memory usage, I used the library time, and this command to get the necessary information: ”/usr/bin/time -v ./matrix”. I noticed that the execution times weren’t quite what I expected, because we learned in theory class that C should be faster than Java and Python, so I compiled the program in release mode -O2 optimization, which brought the metrics down to the expected level.

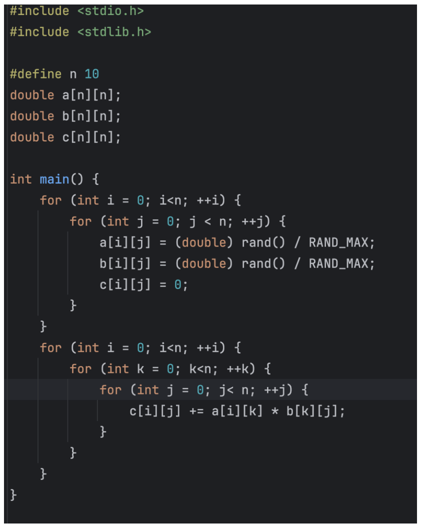
Once I had that, I decided to look for a way to optimize the code and bring the results of the tests down, which I did using the same method as with java, by changing the order of access to have a more sequential order, which I did by simply switching k and j, to avoid multiple accesses to the same element.

Here you can observe the Matrix Multiplication code in C and next to it the optimized code:



```c
#include <stdio.h>
#include <stdlib.h>

#define n 10
double a[n][n];
double b[n][n];
double c[n][n];

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;
        }
    }
    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];
            }
        }
    }
}
```

(a) Matrix Multiplication Code.



```c
#include <stdio.h>
#include <stdlib.h>

#define n 10
double a[n][n];
double b[n][n];
double c[n][n];

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;
        }
    }
    for (int i = 0; i<n; ++i) {
        for (int k = 0; k<n; ++k) {
            for (int j = 0; j< n; ++j) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

(b) Optimized Matrix Multiplication.

3 Experiments and Results

To analyze the different execution times and efficiency, I tested the code in each language, with various matrix sizes, starting with a smaller one, and increasing the size considerably, executing a few warm-up rounds to ensure there was no bias.

I started trying python with n = 10, n = 100 and n = 1000, then I did the same in Java, and finally the same in C in both the normal and the optimized code, and the units were all converted into the same unit for comparison purposes.

Each experiment was run three times, to ensure that the results are reliable and similar, and here you can see the final results of each test:

3.0.1 Python Results

In this case, it is important to explain both tables: In the first table, you can see the results of the original algorithm, and as you can see, the value for memory usage is missing for size 1000, which is due to the inefficient execution, making it impossible to get a result, as it does not seem finish (it was left executing for 9 hours and there was still no result). This is why I found a way to optimize the code, and analysed the parameters on this new code as well, which you can see in the second table:

Matrix Size	Execution time (ms)	Memory usage (MB)
10	50.1904	52.2
100	40416.7	52.7
1000	122356.4	X

Table 1: Benchmark test results for Python

Matrix Size	Execution time (ms)	Memory usage (MB)
10	3.14	52.1
100	208.576	52.7
1000	20611.3	161.9

Table 2: Benchmark test results for Python Optimized Method

3.0.2 Java Results

Here you can see the results of the original Algorithm and the optimized one:

Matrix Size	Execution time (ms)	Memory usage (MB)
10	0.003	10.88
100	1.314	9.76
1000	1963.124	25.70

Table 3: Benchmark test results for Java

Matrix Size	Execution time (ms)	Memory usage (MB)
10	0.001	11.32
100	0.192	10.89
1000	165.463	25.65

Table 4: Benchmark test results for Optimized Java Function.

3.0.3 C Results

Matrix Size	Execution time (ms)	Memory usage (MB)
10	0.0016	1.04
100	0.34	1.29
1000	586.4	23.92

Table 5: Benchmark test results for C Function

Matrix Size	Execution time (ms)	Memory usage (MB)
10	0.0001128	0.915
100	0.000858	1.16
1000	176.73	23.8

Table 6: Benchmark test results for Optimized C Function

3.1 Result Discussion

Once I had recollected all the results, I proceeded to create a graphic to visually see the differences, which I implemented in excel. This was a bit of a problem, because the values are very different, making it impossible to create a graphic that shows all the values. In the two of graphics shown, the first one shows clearly how python has the highest execution times, and in the next graphic I eliminated python to take a closer look at the values of Java and C, which were also not very interesting to represent, as the values between the different sizes change a lot.

Below you can see the two graphics, although they don't give a very good overview of the data, as there is such a difference between the sizes and languages:

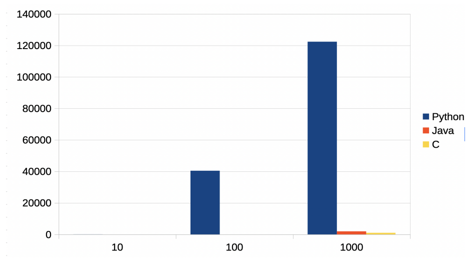


Figure 7: Python, Java and C execution Times.

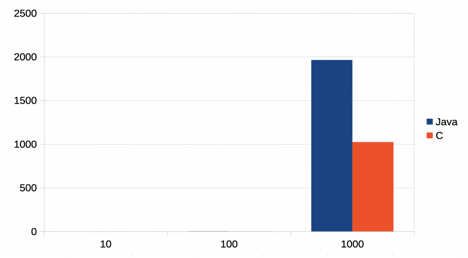
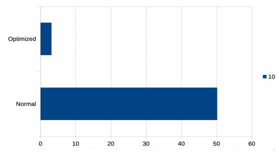
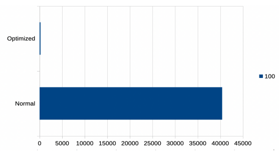


Figure 8: Java and C execution Times.

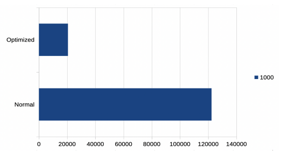
To represent the execution times in a more visually interesting way than the ones above, the next three graphics show the difference between the normal and the optimized codes, in Java, Python and C:



(a) Size 10

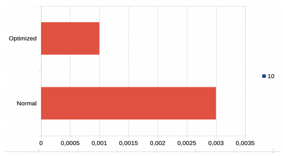


(b) Size 100

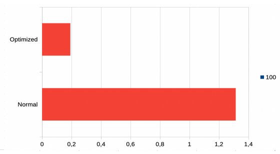


(c) Size 1000

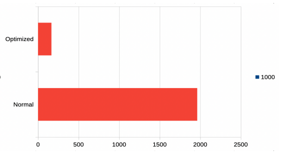
Figure 9: Python Comparison



(a) Size 10

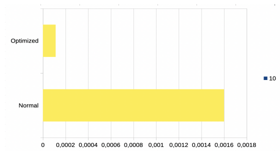


(b) Size 100

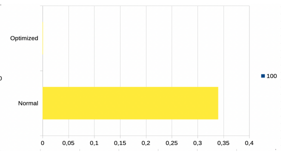


(c) Size 1000

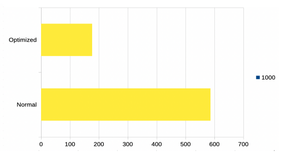
Figure 10: Java Comparison



(a) Size 10



(b) Size 100



(c) Size 1000

Figure 11: C Comparison

The next graphics represent the memory usage of the three programming languages in the normal matrix multiplication algorithm. I didn't create a graphic for the memory usage of the optimized codes, as the values hardly changed:

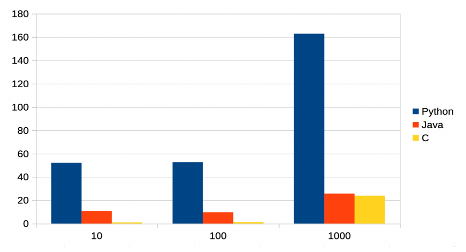


Figure 12: Memory Usage in Python, Java and C Functions.

As you can see, it is interesting to see the differences between the different languages, and also the differences within the same language when increasing the matrix size.

In general we can see that Python is the slowest, with Java being quite a lot faster, and C being even faster than Java. This tells us, that C has the best execution time, meaning the best performance in this aspect. In all three languages, we see an increase of the execution time when increasing the size of the matrix, which is completely normal and expected.

When looking at the memory usage, in Python it increases only slightly between 10 and 100, but then does a big jump when increasing the size to 1000, and what we can see in C is very similar, which is as expected, as the jump between 100 and 1000 is a lot bigger than between 10 and 100.

Analyzing the Java result, we see something very interesting: the memory usage of the size 100 decreases a little compared to size 10, but very similar. And then size 1000 it increases again quite a lot. As I found this slightly strange, I did about 10 runs, and every run had a slightly lower memory usage for size 100 than for size 10. And in the optimized code, the same thing happened, it was slightly below the size 10 value. I researched if this was normal, and my conclusion was that after so many runs and checking the code is right, this could definitely be right, and could have different reasons: Memory Management by the Operating System of my laptop, CPU Architecture and Cache Behavior, or even the Java Virtual Machine Implementation could have to do with it, under other options.

Comparing the memory usage between the different programming languages, we can perceive a similar pattern than the execution times: Python has the largest memory usage, with Java being in second place, and C has the least memory usage in all cases.

As you can see in the tables, there is also the data for the optimized algorithms, and they all have the same pattern: The optimized algorithms all made the execution times decrease by a lot, but the memory usage has stayed the same or decreased slightly, but hardly noticeable. With the new values, the results are the same: Python is the programming language with the worst performance, then Java in second place and C is the best of all.

4 Conclusion

In conclusion, the experiment shows the significant differences in performance when implementing the matrix multiplication algorithm in the different programming languages, demonstrating that C is the most efficient compared to Java and Python. Also, it is important to highlight that every code and implementation can be optimized with different strategies, from changing access orders to using libraries that help out, making the performance much better and more efficient. .

The results indicate that with the increase of matrix dimension, the execution time and memory usage also increases across all three languages. With this, we have demonstrated that C has the lowest execution times and also uses the least memory, which tells us that it is very suitable for computationally intensive tasks. On the other hand, Java has shown intermediate performance, and the Python results have made clear it has notably inferior performance, in both execution time and memory usage.

We can safely conclude that if we are working on a project that requires a high performance in intensive tasks, C is the best option for sure. Also, we can say that Java is a good option as well if the requirements don't fit well for C programming; and lastly python is definitely the least efficient choice in this context.