Basic Matrix Multiplication in Different Languages

Adonai Ojeda Martín

October 20th, 2024

Abstract

In this assignment, we compare the performance of a basic matrix multiplication algorithm implemented in three different programming languages: Python, Java, and C. The execution time, memory usage, and CPU efficiency are evaluated for different matrix sizes to analyze the efficiency of each language when handling matrix multiplications. This comparison helps to understand the trade-offs between high-level and low-level programming languages in computational tasks.

1 Introduction

Matrix multiplication plays a crucial role in various fields like data science, machine learning, and scientific computing. Efficiently implementing this algorithm is essential for applications dealing with large-scale data or requiring high computational performance.

This study evaluates a basic matrix multiplication algorithm implemented in Python, Java, and C. The tests are conducted using matrices of different sizes to assess execution time, memory consumption, and CPU usage. By analyzing the results, we can gain insight into the efficiency of each language when handling the same computational task, as well as the advantages and limitations of higher- and lower-level languages.

2 Implementations

2.1 Python

The implementation in Python was done in Visual Studio Code and is divided into two scripts to separate production code from test code. Python is a high-level language that offers simplicity and ease of use, but its performance can be slower compared to lower-level languages.

```
"matrix.py":

def matmul(A, B):
    n = len(A)
    C = [[0] * 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

"test_matrix.py":

import time
import numpy as np
from memory_profiler import memory_usage
import psutil
from matrix import matmul

def test_matmul(matrix_size, runs=3):
    A = np.random.randint(10, size=(matrix_size, matrix_size))
    B = np.random.randint(10, size=(matrix_size, matrix_size))

    times = []
    memory_used = []

    for _ in range(runs):
        mem_usage = memory_usage((matmul, (A, B)), max_usage=True)
        memory_used.append(mem_usage)

        start_time = time.time()
        matmul(A, B)
        end_time = time.time()

        times.append(end_time - start_time)

    avg_time = sum(times) / runs
    avg_memory = sum(memory_used) / runs
    cpu_usage = psutil.cpu_percent(interval=None)

    return avg_time, avg_memory, cpu_usage

if __name__ == "__main__":
    sizes = [10, 100, 200, 500]
    for size in sizes:
        avg_time, avg_memory, cpu_usage = test_matmul(size)
        print(f"Avg time for {size}x{size}: {avg_time:.6f} seconds")
        print(f"Avg memory usage for {size}x{size}: {avg_memory:.2f} MiB")
        print(f"CPU usage during execution for {size}x{size}: {cpu_usage:.2f}%")
```

2.2 Java

The implementation in Java was done in IntelliJ IDEA. Java, being a compiled language that runs on a virtual machine, strikes a balance between performance and portability. The implementation makes use of the ‘Random‘ class to initialize matrices and ‘System.currentTimeMillis‘ to measure execution time.

```
import java.util.Random;

public class Matrix {
    static int n = 1024;
    static double[][] a = new double[n][n];
    static double[][] b = new double[n][n];
    static double[][] c = new double[n][n];

    public static void main(String[] args) {
        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;
            }
        }

        long start = System.currentTimeMillis();
        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];
                }
            }
        }
        long stop = System.currentTimeMillis();
        System.out.println((stop-start) * 1e-3);
    }
}
```

2.3 C

The C implementation was developed in Visual Studio Code. C, being a low-level language, offers fine-grained control over memory and CPU usage, making it a top choice for performance-critical applications.

```
"matrix_multiplication.c"

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

void initialize_matrix(double **a, double **b, double **c, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            a[i][j] = (double) rand() / RAND_MAX;
            b[i][j] = (double) rand() / RAND_MAX;
            c[i][j] = 0;
        }
    }
}

void multiply_matrix(double **a, double **b, double **c, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            for (int k = 0; k < size; ++k) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

```
"benchmark_matrix.c"

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include "matrix_multiplication.c"

double get_execution_time(struct timeval start, struct timeval stop) {
    return (stop.tv_sec - start.tv_sec) * 1e3 + (stop.tv_usec - start.tv_usec) *
}

int main() {
    int sizes[] = {256, 512, 1024};
    int runs_per_experiment = 3;

    for (int s = 0; s < sizeof(sizes) / sizeof(sizes[0]); ++s) {
        int size = sizes[s];
        printf("Matrix Size: %d\n", size);

        double **a = malloc(size * sizeof(double *));
        double **b = malloc(size * sizeof(double *));
        double **c = malloc(size * sizeof(double *));
        for (int i = 0; i < size; ++i) {
            a[i] = malloc(size * sizeof(double));
            b[i] = malloc(size * sizeof(double));
            c[i] = malloc(size * sizeof(double));
        }

        double total_time = 0.0;

        for (int run = 1; run <= runs_per_experiment; ++run) {
            struct timeval start, stop;
            initialize_matrix(a, b, c, size);

            gettimeofday(&start, NULL);
            multiply_matrix(a, b, c, size);
            gettimeofday(&stop, NULL);

            double execution_time = get_execution_time(start, stop);
            total_time += execution_time;
            printf("    Execution Time (ms): %0.3f\n", execution_time);
        }

        double avg_time = total_time / runs_per_experiment;
        printf("  Average Execution Time for %dx%d: %0.3f ms\n\n", size, size, a

        for (int i = 0; i < size; ++i) {
            free(a[i]);
            free(b[i]);
            free(c[i]);
        }
        free(a);
        free(b);
        free(c);
    }

    return 0;
}
```

3 Results

3.1 Execution Time

The following table summarizes the average execution time for different matrix sizes across the three languages tested.

Table 1: Average Execution Time Results in Milliseconds

Matrix Size	Python	Java	C
256	12328.11	37.94	71.775
512	98490.25	402.01	664.891
1024	1087635.15	4573.43	9681.504

As seen, the Java implementation consistently outperforms both Python and C in terms of execution time, demonstrating the efficiency in heavy computation tasks.

3.2 Memory Usage

Python's and Java's memory usage was significantly higher, while C showed minimal memory usage in each iteration.

3.3 CPU Usage

The CPU utilization was also monitored for each language. Python, being an interpreted language, showed higher CPU usage compared to compiled languages like C and Java.

4 Conclusion

The comparative analysis highlights the trade-offs between different programming languages when implementing matrix multiplication. While Python offers simplicity, its performance is significantly slower than Java and C, making it less suitable for high-performance computing tasks. Java, on the other hand, proves to be the most efficient in terms of time but not in terms of memory consumption, C is still better.

5 References

- Python Documentation: `https://docs.python.org/3/`
- Java Documentation: `https://docs.oracle.com/javase/8/docs/`
- C Documentation: `https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html`
- Stack Overflow: `https://stackoverflow.com/`
- GitHub Repository: Language-Benchmark-of-matrix-multiplication