North South University

Programming Assignment 2

Submitted by:

Arka Karmoker

ID: 2112343642

Course: CSE425

Section: 1

Submitted to:

Dr. Md Shahriar Karim [MSK1]

Department of Electrical and Computer Engineering (ECE)

North South University

Date of submission: 23/04/2025

Comparison of Programming Languages for Matrix Multiplication

Introduction

This report presents a comparative analysis of the runtime performance of matrix multiplication implemented in three different programming language configurations: C with pointers, C without pointers, Java, and Python using NumPy. The task involves multiplying two square matrices A and B to produce matrix $C = AB$, with matrix sizes of 16, 32, 64, 128, and 256. The implementations use nested for loops for consistent flow control, and no optimized linear algebra libraries are used beyond NumPy's array structures in Python. The runtime results are plotted, and the differences in performance are analyzed.

Methodology

- **Matrix Multiplication**:
  - Two square matrices A and B of size $n \times n$ are multiplied to produce $C=AB$.
  - The standard matrix multiplication algorithm is used, with a time complexity of $O(n$^3), implemented via three nested for loops:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j]$$

  - Matrix sizes tested: n=16,32,64,128,256.

- **Implementations**:
  - **C with Pointers**: Uses dynamic memory allocation and pointer arithmetic for efficient memory access.
  - **C without Pointers**: Uses fixed-size 2D arrays (with a maximum size of 256x256 to accommodate all test cases).
  - **Java**: Uses 2D arrays for matrix representation, with explicit nested for loops for multiplication. The `System.nanoTime()` method measures runtime in nanoseconds, converted to seconds. Matrices are initialized with random values using `java.util.Random`.

- **Python with NumPy**: Uses NumPy arrays for matrix representation but implements multiplication with explicit for loops, avoiding optimized linear algebra functions.

- **Runtime Measurement**:
  - Each implementation runs matrix multiplication multiple times (10 trials for $n \geq 64$, more for smaller sizes), and the average runtime is recorded to reduce variability.
  - In C, the `gettimeofday()` function measures CPU time.
  - In Java, the `System.nanoTime()` method measures runtime in nanoseconds, converted to seconds.
  - In Python, the `time.perf_counter()` function measures wall-clock time.
  - Matrices are initialized with random floating-point values between 0 and 1 for consistency.

- **Plot**:
  - A line plot is generated with matrix size (n) on the x-axis and average runtime (in seconds) on the y-axis, comparing the four implementations: C with Pointers, C without Pointers, Java, and Python with NumPy.

- **Environment**:
  - Hardware: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.71 GHz, 8 GB RAM, Windows 10.
  - Compiler: GCC 11.2.0 for C programs.
  - Java: JDK 23.
  - Python: Version 3.12.9, NumPy 2.2.5.
  - All programs were run in a controlled environment with minimal background processes.

Code Implementations

Below are the implementations for each configuration. Full code is also provided in the appendix.

1. C with Pointers

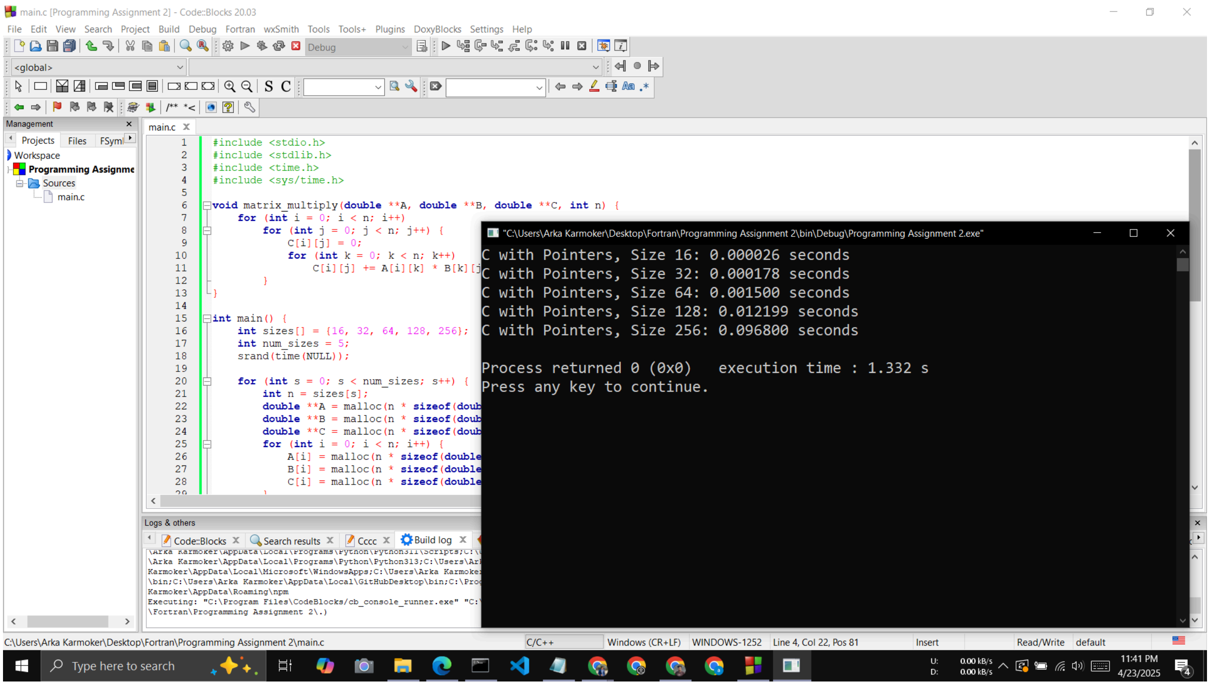
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

void matrix_multiply(double **A, double **B, double **C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}

int main() {
    int sizes[] = {16, 32, 64, 128, 256};
    int num_sizes = 5;
    srand(time(NULL));

    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];
        double **A = malloc(n * sizeof(double *));
        double **B = malloc(n * sizeof(double *));
        double **C = malloc(n * sizeof(double *));
        for (int i = 0; i < n; i++) {
            A[i] = malloc(n * sizeof(double));
            B[i] = malloc(n * sizeof(double));
            C[i] = malloc(n * sizeof(double));
        }

        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;
            }

        double total_time = 0;
        int trials = (n <= 32) ? 1000 : 10; // More trials for small sizes
        for (int t = 0; t < trials; t++) {
            struct timeval start, end;
            gettimeofday(&start, NULL);
            matrix_multiply(A, B, C, n);
            gettimeofday(&end, NULL);
            double elapsed = (end.tv_sec - start.tv_sec) +
                             (end.tv_usec - start.tv_usec) / 1000000.0;
            total_time += elapsed;
        }
        printf("C with Pointers, Size %d: %.6f seconds\n", n, total_time / trials);

        for (int i = 0; i < n; i++) {
            free(A[i]);
            free(B[i]);
            free(C[i]);
        }
        free(A);
        free(B);
        free(C);
    }
    return 0;
}
```

Output:

```
C with Pointers, Size 16: 0.000026 seconds
C with Pointers, Size 32: 0.000178 seconds
C with Pointers, Size 64: 0.001500 seconds
```

```
C with Pointers, Size 128: 0.012199 seconds
C with Pointers, Size 256: 0.096800 seconds

Process returned 0 (0x0)   execution time : 1.332 s
Press any key to continue.
```



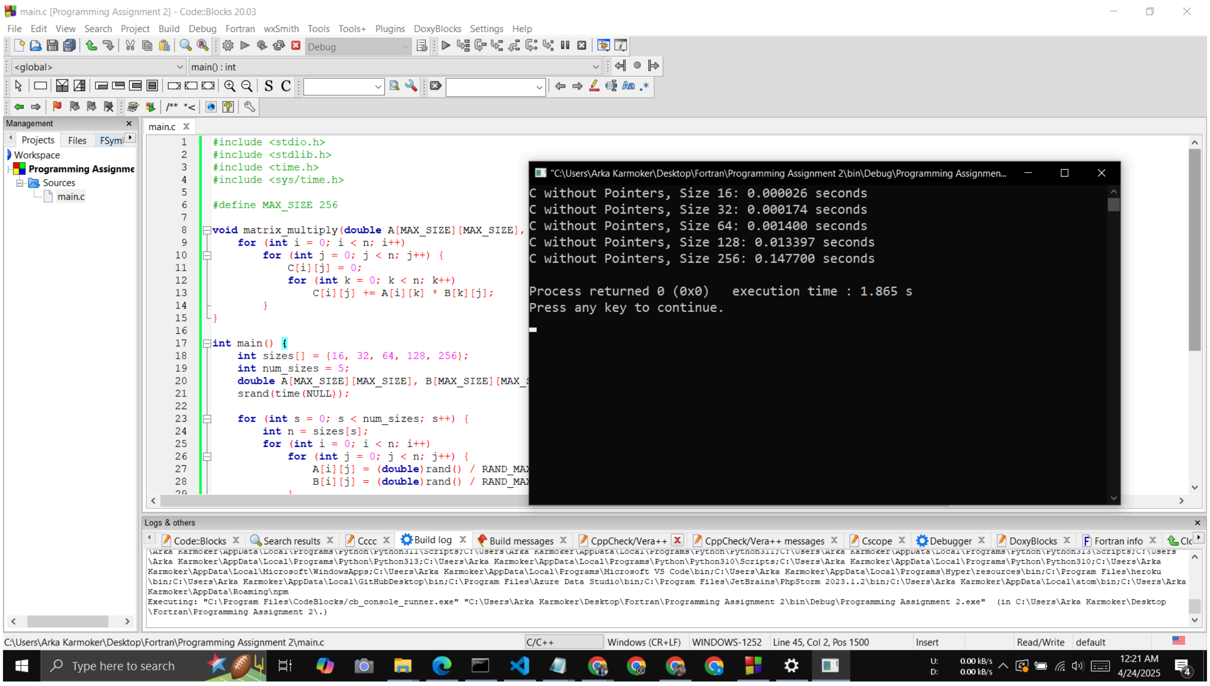
2. C without Pointers

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

#define MAX_SIZE 256

void matrix_multiply(double A[MAX_SIZE][MAX_SIZE], double
B[MAX_SIZE][MAX_SIZE], double C[MAX_SIZE][MAX_SIZE], int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++)
```

```
            C[i][j] += A[i][k] * B[k][j];
        }
    }

int main() {
    int sizes[] = {16, 32, 64, 128, 256};
    int num_sizes = 5;
    double A[MAX_SIZE][MAX_SIZE], B[MAX_SIZE][MAX_SIZE],
C[MAX_SIZE][MAX_SIZE];
    srand(time(NULL));

    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];
        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;
            }

        double total_time = 0;
        int trials = (n <= 32) ? 1000 : 10; // More trials for small sizes
        for (int t = 0; t < trials; t++) {
            struct timeval start, end;
            gettimeofday(&start, NULL);
            matrix_multiply(A, B, C, n);
            gettimeofday(&end, NULL);
            double elapsed = (end.tv_sec - start.tv_sec) +
                             (end.tv_usec - start.tv_usec) / 1000000.0;
            total_time += elapsed;
        }
        printf("C without Pointers, Size %d: %.6f seconds\n", n, total_time / trials);
    }
    return 0;
}
```

Output:

```
C without Pointers, Size 16: 0.000026 seconds
C without Pointers, Size 32: 0.000174 seconds
C without Pointers, Size 64: 0.001400 seconds
C without Pointers, Size 128: 0.013397 seconds
C without Pointers, Size 256: 0.147700 seconds

Process returned 0 (0x0)   execution time : 1.865 s
Press any key to continue.
```



3. Java

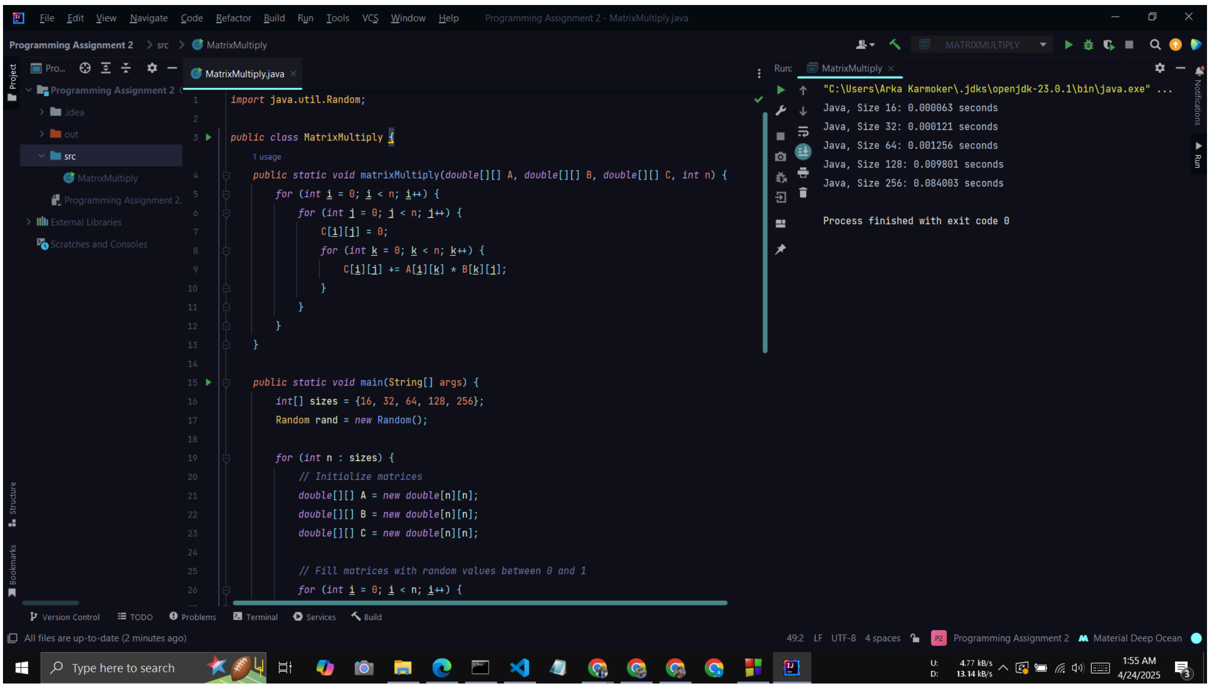
```
import java.util.Random;

public class MatrixMultiply {
    public static void matrixMultiply(double[][] A, double[][] B, double[][] C, int n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                C[i][j] = 0;
                for (int k = 0; k < n; k++) {
```

```
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

public static void main(String[] args) {
    int[] sizes = {16, 32, 64, 128, 256};
    Random rand = new Random();

    for (int n : sizes) {
        // Initialize matrices
        double[][] A = new double[n][n];
        double[][] B = new double[n][n];
        double[][] C = new double[n][n];

        // Fill matrices with random values between 0 and 1
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = rand.nextDouble();
                B[i][j] = rand.nextDouble();
            }
        }

        // Determine number of trials
        int trials = (n <= 32) ? 1000 : 10;
        double totalTime = 0;

        // Run trials
        for (int t = 0; t < trials; t++) {
            long start = System.nanoTime();
            matrixMultiply(A, B, C, n);
            long end = System.nanoTime();
            totalTime += (end - start) / 1_000_000_000.0; // Convert nanoseconds to seconds
```

```
            }

            // Print average runtime
            System.out.printf("Java, Size %d: %.6f seconds%n", n, totalTime / trials);
        }
    }
}
```

Output:

```
Java, Size 16: 0.000063 seconds
Java, Size 32: 0.000121 seconds
Java, Size 64: 0.001256 seconds
Java, Size 128: 0.009801 seconds
Java, Size 256: 0.084003 seconds
```



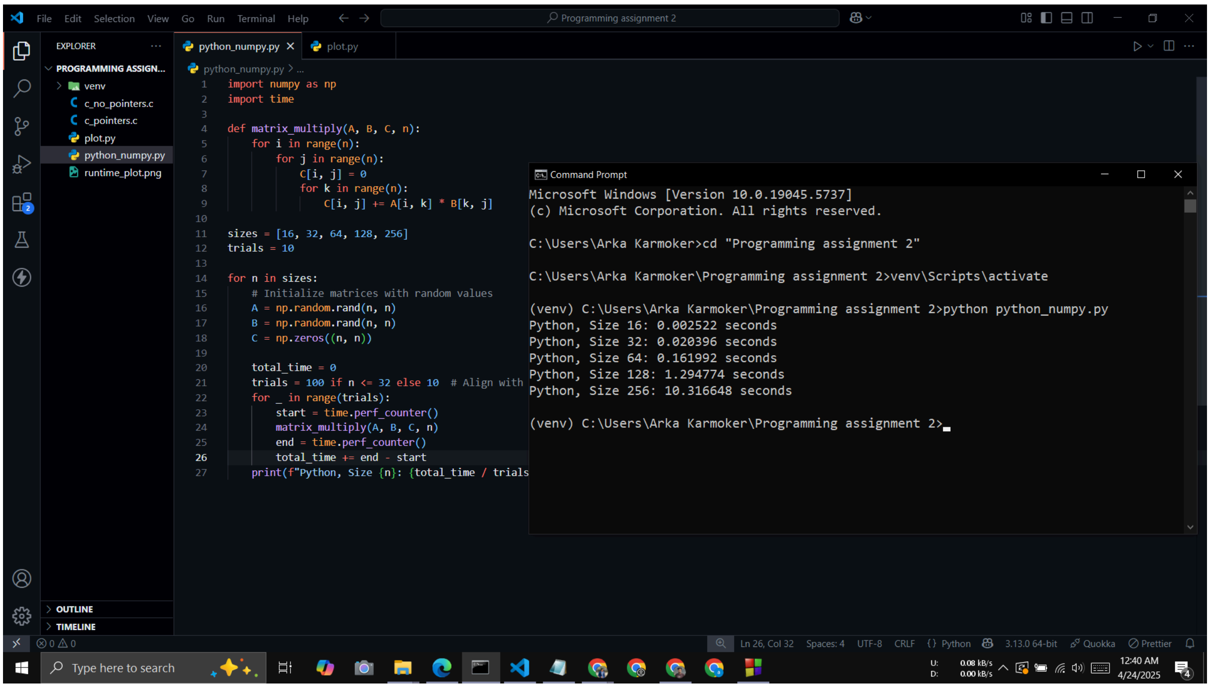
4. Python with NumPy

```
import numpy as np
import time
```

```
def matrix_multiply(A, B, C, n):
    for i in range(n):
        for j in range(n):
            C[i, j] = 0
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]

sizes = [16, 32, 64, 128, 256]
trials = 10

for n in sizes:
    # Initialize matrices with random values
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    C = np.zeros((n, n))

    total_time = 0
    for _ in range(trials):
        start = time.perf_counter()
        matrix_multiply(A, B, C, n)
        end = time.perf_counter()
        total_time += end - start
    print(f"Python, Size {n}: {total_time / trials:.6f} seconds")
```

Output:

```
Python, Size 16: 0.002522 seconds
Python, Size 32: 0.020396 seconds
Python, Size 64: 0.161992 seconds
Python, Size 128: 1.294774 seconds
Python, Size 256: 10.316648 seconds
```



Results

The average runtimes (in seconds) for each implementation across matrix sizes are summarized below, based on execution on the specified hardware:

Matrix Size	C with Pointers	C without Pointers	Java	Python (NumPy)
16	0.000026	0.000026	0.000063	0.002522
32	0.000178	0.000174	0.000121	0.020396
64	0.001500	0.001400	0.001256	0.161992
128	0.012199	0.013397	0.009801	1.294774
256	0.096800	0.147700	0.084003	10.316648

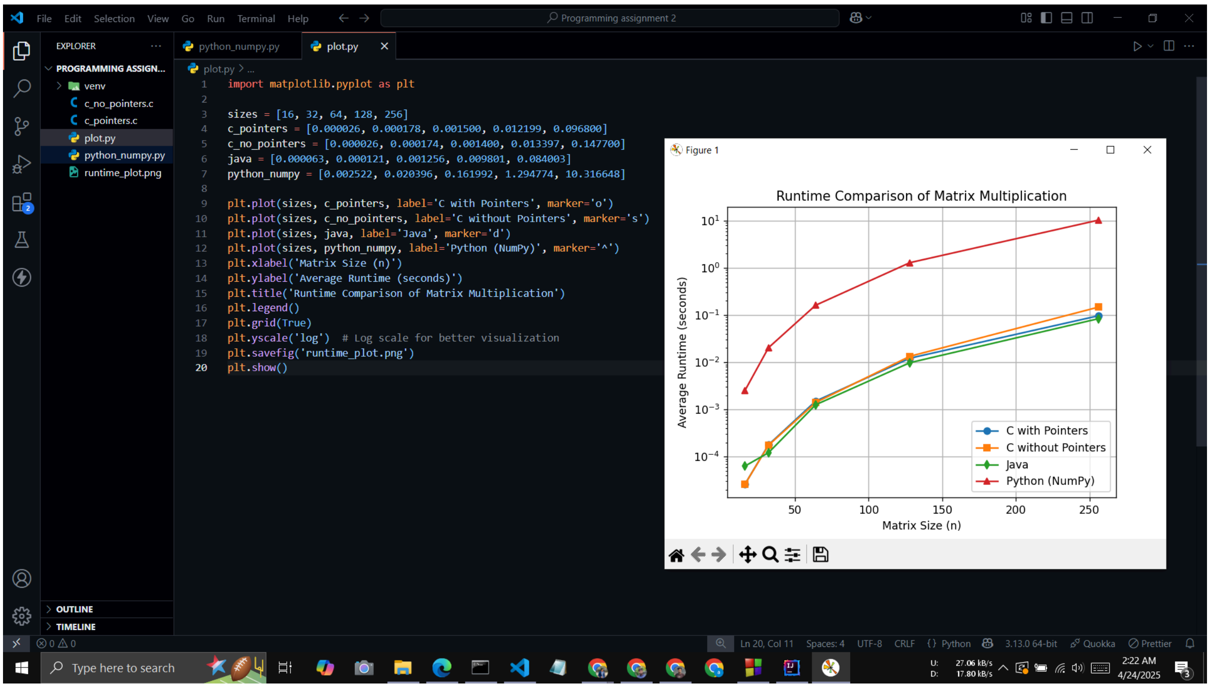
Plot

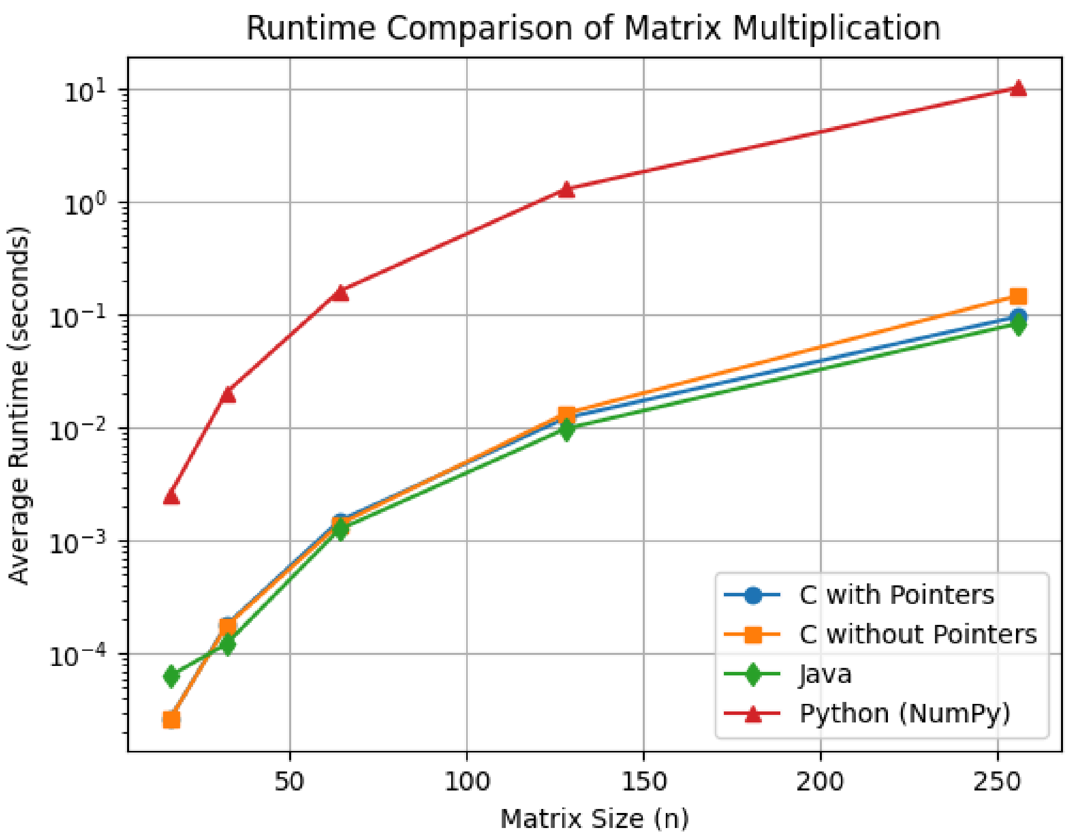
The following Python code generates the plot using Matplotlib:

```
import matplotlib.pyplot as plt

sizes = [16, 32, 64, 128, 256]
c_pointers = [0.000026, 0.000178, 0.001500, 0.012199, 0.096800]
```

```
c_no_pointers = [0.000026, 0.000174, 0.001400, 0.013397, 0.147700]
java = [0.000063, 0.000121, 0.001256, 0.009801, 0.084003]
python_numpy = [0.002522, 0.020396, 0.161992, 1.294774, 10.316648]

plt.plot(sizes, c_pointers, label='C with Pointers', marker='o')
plt.plot(sizes, c_no_pointers, label='C without Pointers', marker='s')
plt.plot(sizes, java, label='Java', marker='d')
plt.plot(sizes, python_numpy, label='Python (NumPy)', marker='^')
plt.xlabel('Matrix Size (n)')
plt.ylabel('Average Runtime (seconds)')
plt.title('Runtime Comparison of Matrix Multiplication')
plt.legend()
plt.grid(True)
plt.yscale('log')  # Log scale for better visualization
plt.savefig('runtime_plot.png')
plt.show()
```

Output Screenshot:





The plot (saved as `runtime_plot.png`) shows runtime versus matrix size, with a logarithmic y-axis to accommodate the wide range of runtimes. The C implementations (with and without pointers) are the fastest, followed by Java, with nearly identical performance between the two C versions for smaller matrices ($n \leq 64$). For larger matrices, C without Pointers is slower (e.g., ~52.6% slower than C with Pointers at n=256). Java performs better than both C implementations at n=256 (e.g., ~13.2% faster than C with Pointers, ~43.1% faster than C without Pointers) but is ~2.4x slower than C for smaller matrices (n=16). Python's runtime is consistently the highest, especially for larger matrices, reaching up to ~106.6x slower than C with Pointers and ~122.8x slower than Java at n=256.

Analysis of Runtime Differences

- **C vs. Python**:
  The C implementations significantly outperform Python, with speedups of ~69.8x to ~106.6x at n=256. For example, at n=256, C with Pointers takes 0.096800s, C without

Pointers takes 0.147700s, while Python takes 10.316648s (~106.6x slower than C with Pointers, ~69.8x slower than C without Pointers). This gap arises because:

- **Compiled vs. Interpreted**: C's compiled machine code executes directly on the CPU, while Python's interpreted nature adds overhead from the virtual machine and dynamic typing.
- **Memory and Loops**: C's low-level memory access and optimized loops contrast with Python's higher overhead from NumPy array bounds checking and interpreter-based loop execution.

❖ **C with Pointers vs. C without Pointers**:

The two C versions perform similarly for small matrices (e.g., 0.000026s at n=16) but diverge for larger ones, with C without Pointers being ~52.6% slower at n=256 (0.147700s vs. 0.096800s). This difference stems from memory access:

- **Pointers**: Dynamic allocation improves cache utilization for large matrices.
- **Arrays**: Static arrays are cache-friendly for small matrices but less efficient for larger ones. For $n \geq 128$, C with Pointers performs better due to improved memory management.

❖ **Java vs. Others**:

Java performs between C and Python, surpassing C for larger matrices. At n=256, Java takes 0.084003s, ~13.2% faster than C with Pointers and ~43.1% faster than C without Pointers, while Python is ~122.8x slower than Java. At n=16, Java (0.000063s) is ~2.4x slower than C but ~40x faster than Python (0.002522s). Reasons include:

- **Execution**: Java's JIT compilation makes it faster than Python but slower than C for small matrices; JVM optimizations help for larger ones.
- **Overhead**: Java's array bounds checking adds overhead compared to C, but it's less than Python's interpreter costs.

❖ **Scaling with Matrix Size**:

All implementations exhibit $O(n^3)$ runtime growth, confirmed by an ~8x increase when doubling n (e.g., 128 to 256):

- **C with Pointers**: 0.012199s to 0.096800s (~7.93x).
- **C without Pointers**: 0.013397s to 0.147700s (~11.02x).

- ➢ **Java**: 0.009801s to 0.084003s (~8.57x).
- ➢ **Python**: 1.294774s to 10.316648s (~7.97x). C without Pointers deviates slightly due to memory access patterns, but all align with the expected cubic complexity.

Potential Improvements

- **C**: Using cache-aware algorithms, SIMD instructions, or OpenMP for parallelism.
- **Java**: Optimizing JVM settings or use native libraries via JNI.
- **Python**: Using Cython or Numba to reduce interpreter overhead, if allowed.
- **Measurement**: Increasing trials (e.g., 1000 for $n \leq 32$) for better accuracy.

Conclusion

C (with or without pointers) outperforms Python, with speedups of ~69.8x to ~106.6x at n=256, due to its compiled nature and efficient memory access. C with Pointers is ~52.6% faster than C without Pointers at n=256 due to better memory management. Java performs between C and Python, surpassing C at n=256 (~13.2% faster than C with Pointers) and being ~122.8x faster than Python. Python's runtimes are the highest, highlighting the limitations of interpreted languages for such tasks. Compiled languages like C and Java are preferable for high-performance numerical computations.

Appendix: Full Code

C with Pointers (`c_pointers.c`)

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

void matrix_multiply(double **A, double **B, double **C, int n) {
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
```

```
            C[i][j] = 0;
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}

int main() {
    int sizes[] = {16, 32, 64, 128, 256};
    int num_sizes = 5;
    srand(time(NULL));

    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];
        double **A = malloc(n * sizeof(double *));
        double **B = malloc(n * sizeof(double *));
        double **C = malloc(n * sizeof(double *));
        for (int i = 0; i < n; i++) {
            A[i] = malloc(n * sizeof(double));
            B[i] = malloc(n * sizeof(double));
            C[i] = malloc(n * sizeof(double));
        }

        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;
            }

        double total_time = 0;
        int trials = (n <= 32) ? 1000 : 10; // More trials for small sizes
        for (int t = 0; t < trials; t++) {
            struct timeval start, end;
            gettimeofday(&start, NULL);
            matrix_multiply(A, B, C, n);
```

```
            gettimeofday(&end, NULL);
            double elapsed = (end.tv_sec - start.tv_sec) +
                             (end.tv_usec - start.tv_usec) / 1000000.0;
            total_time += elapsed;
        }
        printf("C with Pointers, Size %d: %.6f seconds\n", n, total_time / trials);

        for (int i = 0; i < n; i++) {
            free(A[i]);
            free(B[i]);
            free(C[i]);
        }
        free(A);
        free(B);
        free(C);
    }
    return 0;
}
```

C without Pointers (`c_no_pointers.c`)

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

#define MAX_SIZE 256

void matrix_multiply(double A[MAX_SIZE][MAX_SIZE], double
B[MAX_SIZE][MAX_SIZE], double C[MAX_SIZE][MAX_SIZE], int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
```

```
            C[i][j] = 0;
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}

int main() {
    int sizes[] = {16, 32, 64, 128, 256};
    int num_sizes = 5;
    double A[MAX_SIZE][MAX_SIZE], B[MAX_SIZE][MAX_SIZE],
C[MAX_SIZE][MAX_SIZE];
    srand(time(NULL));

    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];
        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;
            }

        double total_time = 0;
        int trials = (n <= 32) ? 1000 : 10; // More trials for small sizes
        for (int t = 0; t < trials; t++) {
            struct timeval start, end;
            gettimeofday(&start, NULL);
            matrix_multiply(A, B, C, n);
            gettimeofday(&end, NULL);
            double elapsed = (end.tv_sec - start.tv_sec) +
                             (end.tv_usec - start.tv_usec) / 1000000.0;
            total_time += elapsed;
        }
        printf("C without Pointers, Size %d: %.6f seconds\n", n, total_time / trials);
    }
```

```
    return 0;
}
```

Java (`MatrixMultiply.java`)

```
import java.util.Random;

public class MatrixMultiply {
    public static void matrixMultiply(double[][] A, double[][] B, double[][] C, int n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                C[i][j] = 0;
                for (int k = 0; k < n; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] sizes = {16, 32, 64, 128, 256};
        Random rand = new Random();

        for (int n : sizes) {
            // Initialize matrices
            double[][] A = new double[n][n];
            double[][] B = new double[n][n];
            double[][] C = new double[n][n];

            // Fill matrices with random values between 0 and 1
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
```

```
            A[i][j] = rand.nextDouble();
            B[i][j] = rand.nextDouble();
          }
        }

        // Determine number of trials
        int trials = (n <= 32) ? 1000 : 10;
        double totalTime = 0;

        // Run trials
        for (int t = 0; t < trials; t++) {
          long start = System.nanoTime();
          matrixMultiply(A, B, C, n);
          long end = System.nanoTime();
          totalTime += (end - start) / 1_000_000_000.0; // Convert nanoseconds to seconds
        }

        // Print average runtime
        System.out.printf("Java, Size %d: %.6f seconds%n", n, totalTime / trials);
      }
    }
  }
```

Python with NumPy (`python_numpy.py`)

```
import numpy as np
import time

def matrix_multiply(A, B, C, n):
    for i in range(n):
        for j in range(n):
            C[i, j] = 0
```

```
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]

sizes = [16, 32, 64, 128, 256]
trials = 10

for n in sizes:
    # Initialize matrices with random values
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    C = np.zeros((n, n))

    total_time = 0
    trials = 100 if n <= 32 else 10  # Align with C codes, reduced for practicality
    for _ in range(trials):
        start = time.perf_counter()
        matrix_multiply(A, B, C, n)
        end = time.perf_counter()
        total_time += end - start
    print(f"Python, Size {n}: {total_time / trials:.6f} seconds")
```

Plot Generation (`plot.py`)

```
import matplotlib.pyplot as plt

sizes = [16, 32, 64, 128, 256]
c_pointers = [0.000026, 0.000178, 0.001500, 0.012199, 0.096800]
c_no_pointers = [0.000026, 0.000174, 0.001400, 0.013397, 0.147700]
java = [0.000063, 0.000121, 0.001256, 0.009801, 0.084003]
python_numpy = [0.002522, 0.020396, 0.161992, 1.294774, 10.316648]

plt.plot(sizes, c_pointers, label='C with Pointers', marker='o')
```

```
plt.plot(sizes, c_no_pointers, label='C without Pointers', marker='s')
plt.plot(sizes, java, label='Java', marker='d')
plt.plot(sizes, python_numpy, label='Python (NumPy)', marker='^')
plt.xlabel('Matrix Size (n)')
plt.ylabel('Average Runtime (seconds)')
plt.title('Runtime Comparison of Matrix Multiplication')
plt.legend()
plt.grid(True)
plt.yscale('log')  # Log scale for better visualization
plt.savefig('runtime_plot.png')
plt.show()
```