Project 2

Jiawei Xiong SID:12011022

Introduction

This project aims to compare the performance of matrix multiplication between two popular programming languages, C and Java. Matrix multiplication is a fundamental operation in many scientific and engineering applications. We implemented matrix multiplication algorithms in both C and Java, and conducted experiments to measure the execution time for multiplying matrices of various sizes.

In C, we explored different optimization levels, including the highest optimization level -O3, to investigate its impact on performance. Additionally, we compiled the C code using various compilers to assess compiler optimizations.

The experiments involved multiplying matrices of different dimensions ranging from small to large sizes. We recorded the execution times for each multiplication operation and analyzed the results to compare the performance between C and Java implementations. The comparison considered factors such as execution speed, memory usage, and ease of implementation.

Implementation

C

Here is the multiply method:

```
void multiply(float *A, float *B, float *C, int a, int b, int c) {
    for (int i = 0; i < a; ++i) {
        for (int j = 0; j < c; ++j) {
            C[i*c + j] = 0;
            for (int k = 0; k < b; ++k) {
                C[i*c + j] += A[i*b + k] * B[k*c + j];
            }
        }
    }
}
```

This program conducts a performance analysis of matrix multiplication using C. Here's a brief explanation of the main steps:

1. **File Initialization**: The program initializes a file (`matrix_times.txt`) to store the results of the performance analysis.

```
int main() {
FILE *fp;
fp = fopen("matrix_times.txt", "w");
fprintf(fp, "sizeA\tsizeB\ttime\n");
if (fp == NULL) {
    printf("Error opening file!\n");
    return 1;}
```

2. **User Input Loop**: It enters into an infinite loop to repeatedly prompt the user to enter dimensions for square matrices. The user can type 'quit' to exit the loop.

```
char input[20];
    int a, b, c;
    while (1) {
        memset(input, 0, sizeof(input));
        printf("Enter dimensions (a b c) for square matrix where A is axb and B is bxc
(type 'quit' to exit): ");
        fgets(input, sizeof(input), stdin); // 读取一行输入
        input[strcspn(input, "\n")] = '\0';
        if (strcmp(input, "quit") == 0) { // 判断是否输入 "quit"
            break;
        }
        sscanf(input, "%d %d %d", &a, &b, &c); // 解析输入
```

3. **Matrix Initialization**: For each input size, it dynamically allocates memory for matrices A, B, and C. It then populates matrices A and B with random float values.

```
    float *A = (float *)malloc(a * b * sizeof(float));
    float *B = (float *)malloc(b * c * sizeof(float));
    float *C = (float *)malloc(a * c * sizeof(float));
```

4. **Matrix Multiplication and Time Measurement**: It conducts matrix multiplication between matrices A and B, storing the result in matrix C. The multiplication operation is performed 10 times for each input size. The clock is used to measure the time taken for the multiplication operation.

```
        srand(time(NULL));
        for (int i = 0; i < a * b; i++) {
            A[i] = rand() / (float)RAND_MAX;
        }
        for (int i = 0; i < b * c; i++) {
            B[i] = rand() / (float)RAND_MAX;
        }

        clock_t start, end;
        double total_time = 0.0;
        for (int i = 0; i < 10; i++) {
            start = clock();
            multiply(A, B, C, a, b, c);//计算
            end = clock();
```

```
            total_time += (double)(end - start) / CLOCKS_PER_SEC;
        }
```

5. **Average Time Calculation**: It calculates the average time taken for the matrix multiplication operation over the 10 iterations.
6. **Output**: The program prints the average time spent for matrix multiplication and writes this information, along with the input matrix sizes, to the output file (`matrix_times.txt`).
7. **Memory Deallocation**: After each iteration, memory allocated for matrices A, B, and C is freed to avoid memory leaks.

   ```
   free(A);
   free(B);
   free(C);
   ```

8. **File Closure**: Finally, the output file is closed.

here is the result for matrix multiplication:

size of matrix	Time(s)
10x10	0.000016
20x20	0.000094
50x50	0.000835
100x100	0.004170
150x150	0.008904
200x200	0.016353
250x250	0.029178
500x500	0.214845
750x750	1.136879
1000x1000	1.963570
1500x1500	8.323041
2000x2000	17.464851

And very interesting, if I use `gcc -O3 -o matrix matrix.c` to compile the file, I can get the time result:

```
avg time spent for matrix multiplication (C_750x750 = A_750x750 * B_750x750): 0.000001 seconds
Enter dimensions (a b c) for square matrix where A is axb and B is bxc (type 'quit' to exit): 1000 1000 1000
avg time spent for matrix multiplication (C_1000x1000 = A_1000x1000 * B_1000x1000): 0.000000 seconds
Enter dimensions (a b c) for square matrix where A is axb and B is bxc (type 'quit' to exit): 2000 2000 2000
avg time spent for matrix multiplication (C_2000x2000 = A_2000x2000 * B_2000x2000): 0.000001 seconds
Enter dimensions (a b c) for square matrix where A is axb and B is bxc (type 'quit' to exit): 10000 10000 10000
avg time spent for matrix multiplication (C_10000x10000 = A_10000x10000 * B_10000x10000): 0.000000 seconds
Enter dimensions (a b c) for square matrix where A is axb and B is bxc (type 'quit' to exit): 20000 20000 20000
avg time spent for matrix multiplication (C_20000x20000 = A_20000x20000 * B_20000x20000): 0.000000 seconds
```

It almost doesn't take any time even if the matrix is large.

Java

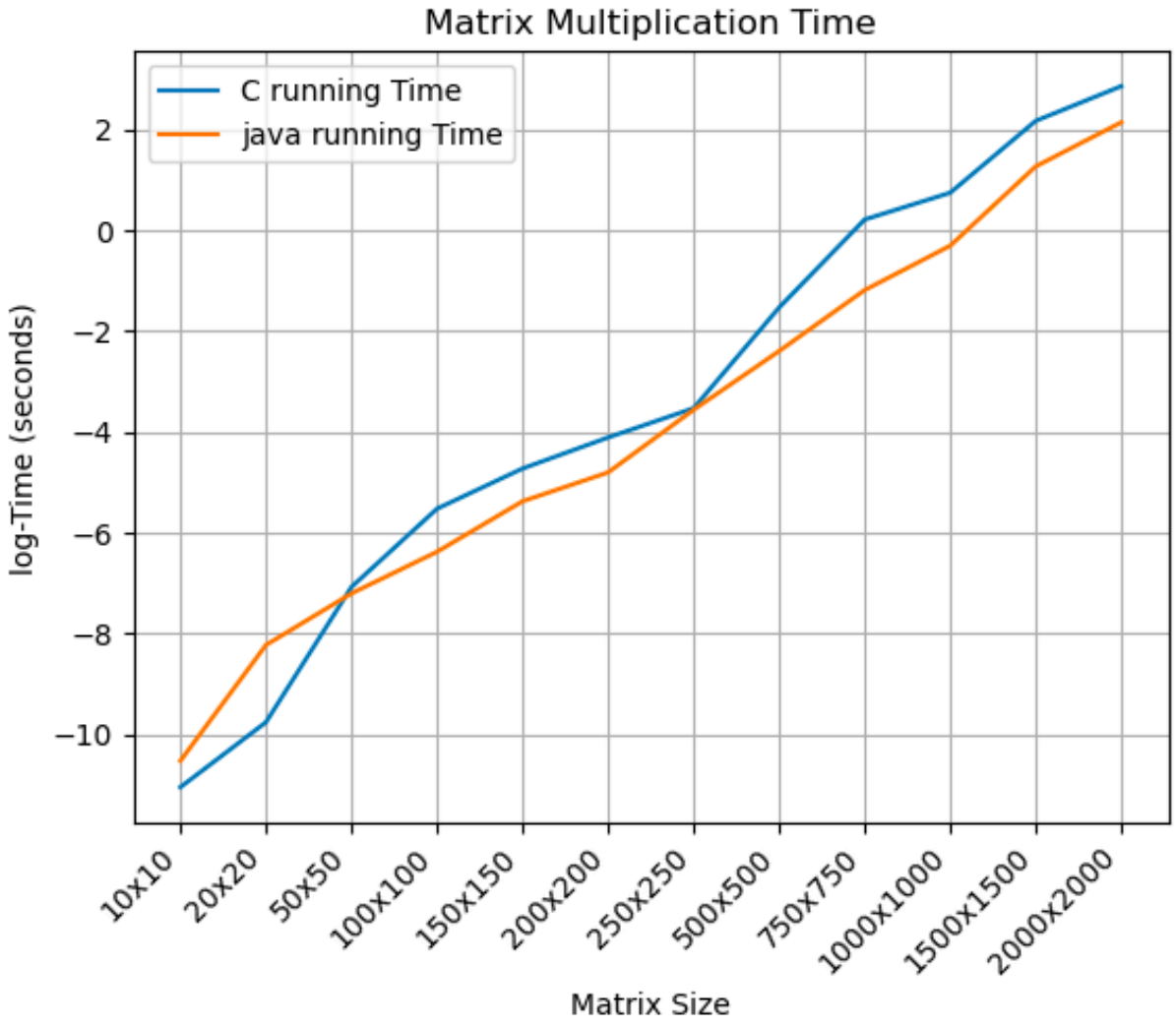
```
\\ the multiply method
public static void matrixMultiply(float[][] A, float[][] B, float[][] C) {
        int m = A.length;
        int n = A[0].length;
        int p = B[0].length;

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

and the implementation is almost the same as C's code. Details can be seen in source file, so I just skip to the result:

size of matrix	Time(s)
10x10	0.000027
20x20	0.000268
50x50	0.000745
100x100	0.001700
150x150	0.004629
200x200	0.008199
250x250	0.028388
500x500	0.090938
750x750	0.304465
1000x1000	0.733739
1500x1500	3.535861
2000x2000	8.474898

Comparison

Since the time using small matrix is very small, so I use **log(time)** to draw the plot, here is the result:



we can when using small matrixes, C is much faster than using Java. However, when the matrix becomes large, the Java seems outperform over C. For instance, 2000*2000, C takes 17.46s, but Java just take 8.47s. Note that I using average of 10 repetitions to represent the time, so the results is unlikely to be biased.

Utilization of ChatGPT

At the conclusion of the project, it's worth noting that ChatGPT was utilized for assistance in report generation and debugging. However, I wish to emphasize that the project was independently completed, with the assistance serving as a helpful resource throughout the process。