CS205 Project 2: Simple Matrix Multiplication

Yang Yanzhuo
12212726

INTRODUCTION

This project aims to compare the performance of two programs in C and Java, which are implemented for matrix multiplication, and explain the reason for any observed differences.

This report mainly tries to explain these aspects during the implement:

1. Analyze the differences between the implementation in `C` and `Java`;
2. Simply modify and analyze **the loop structure** for much faster speed;

CLAIM

1. All the algorithms in $O(n^3)$ time complexity: Since the **efficiency** and **analysis** are mainly focused, we ignore the others algorithms with lower time complexity but try to optimize the basic $O(n^3)$ one.

   > Here, we simply introduce some others algorithms with lower time complexity:
   >
   > - **Strassen Algorithm** in $O(n^{log_2 7}) \approx O(n^{2.807})$

> This efficiency is higher due to its recursive nature, which divides each matrix into four submatrices and recursively applies the same strategy, thus reducing the number of necessary multiplication operations.
>
> - **Coppersmith-Winograd Algorithm** in $O(n^{2.376})$
>
>   Through a series of recursive decompositions, the algorithm reduces the problem of matrix multiplication to these base cases, using the properties of rank and the structure of the matrices.

2. All the contents using ChatGPT 4.0 (OpenAI, 2024-03-31) are already cited.

3. Development Environment:

   - `Ubuntu 20.04.2 LTS x86_64` with gcc version `gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0`
   - C standard: `C11`
   - Java IDE Version: `IntelliJ IDEA 2023.3.4`
   - Java JDK Version: `Oracle OpenJDK version 21.0.2`

IMPLEMENTATION IN `C Language`

1. Initialize the matrices

```c
void createMatrix(float *m, int N) {
    for (int i = 0; i < N * N; i++)
        m[i] = (float)rand() / RAND_MAX;
}
```

2. Perform matrix multiplication using brute force

```
void multiplyMatrices(const float *a, const float*b, float*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 * N + j] += a[i * N + k] * b[k * N + j];
}
```

Here we mainly implement these functions with pointers, which allow the functions to directly access and manipulate data in memory **without copy the data itself**. This means that regardless of the size of the matrix, only **the memory address of the data is passed** to the `multiplyMatrices` function, not a complete copy of the data, significantly **saving memory space**.

Compared to copying data, direct data access via pointers can **reduce unnecessary memory operations**, enhancing the efficiency. This is particularly obvious when matrices is getting larger.

By marking the `a` and `b` pointers as `const`, the compiler prevents the code from modifying the data these pointers point to. This means the function **can not accidentally change the content** of the input matrices, preserving **data integrity**.

---

3. Modify and Analyze the Loop Structure

```
void multiplyMatrices(const float *a, const float*b, float*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 * N + j] += a[i * N + k] * b[k * N + j];
}
```

The original method is the most straightforward implementation, involving three direct loops. For each calculation of `c[i*N + j]`, it accumulates `a[i*N + k] * b[k*N + j]` within the inner loop. The issue with this approach is that it does not attempt to optimize memory access patterns, especially for accessing `b[k*N + j]`, which may lead to frequent cache misses due to accessing data across different columns in `b*`.

Thus we attempts to reduce write operations to `c` by accumulating the multiplication results into a temporary variable `sum` and then assigning the result to `c[i*N + j]`.

This method relative to the original method, does not fundamentally reduce the number of memory accesses and thus might not bring significant performance improvements.

```
void optimized_I(const float*a, const float *b, float *c, int N) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0;
            for (int k = 0; k < N; k++)
                sum += a[i * N + k] * b[k * N + j];
            c[i* N + j] = sum;
        }
}
```

On the basis of the former optimization ideas, we change the order of the original triple loop to reduce the repeated accesses to the `a` array.

For each `k`, `a[i*N + k]` is loaded once as a temporary variable `r` and can be used to multiply with all elements of the `k`th row of `b`. We improves data locality and memory access patterns. Especially when `b[k*N + j]` can be efficiently cached, it will significantly enhancing performance.

```c
void optimized_II(const float *a, const float *b, float *c, int N) {
    for (int i = 0; i < N; i++)
        for (int k = 0; k < N; k++) {
            float r = a[i * N + k];
            for (int j = 0; j < N; j++)
                c[i * N + j] += r * b[k * N + j];
        }
}
```

From the recursive algorithm, Strassen Algorithm, we can simplify it to get the original one opitimized.

`optimized_III` performs a block partitioning strategy, breaking down large matrices into smaller sub-matrices for computation. This method improves data locality in the CPU cache because the data blocks being processed are more likely to fit entirely into the cache. When processing a small block, the required data (from matrices A and B) is more likely to be in the cache, reducing the need for main memory access.

Depending on the size of the matrix (`N`), we can dynamically adjusts the block size (`BLOCK_SIZE`), optimizing performance across different scales of matrix multiplication calculations.

By employing OpenMP for parallel processing, the `#pragma omp parallel for` directive allows multiple processor cores to execute the outer loop in parallel, significantly speeding up the computation. Parallel processing enables the simultaneous calculation of multiple blocks on modern processors with multiple cores, further reducing the total computation time. (OpenAI, 2024-03-31)

```c
void optimized_III(const float *a, const float *b, float *c, int N) {
    int i, j, k, ii, jj, kk;
    int BLOCK_SIZE = 50;
    if (N > 1500) BLOCK_SIZE = 100;
    #pragma omp parallel for private(i, j, k, ii, jj, kk) shared(a, b, c, N)
    // This statement is coded by ChatGPT.
    for (ii = 0; ii < N; ii += BLOCK_SIZE)
        for (jj = 0; jj < N; jj += BLOCK_SIZE)
            for (kk = 0; kk < N; kk += BLOCK_SIZE)
                for (i = ii; i < ii + BLOCK_SIZE && i < N; i++)
                    for (k = kk; k < kk + BLOCK_SIZE && k < N; k++) {
                        float r = a[i * N + k];
                        for (j = jj; j < jj + BLOCK_SIZE && j < N; j++)
                            c[i * N + j] += r * b[k * N + j];
                    }
}
```

4. Implementation in `Main` function

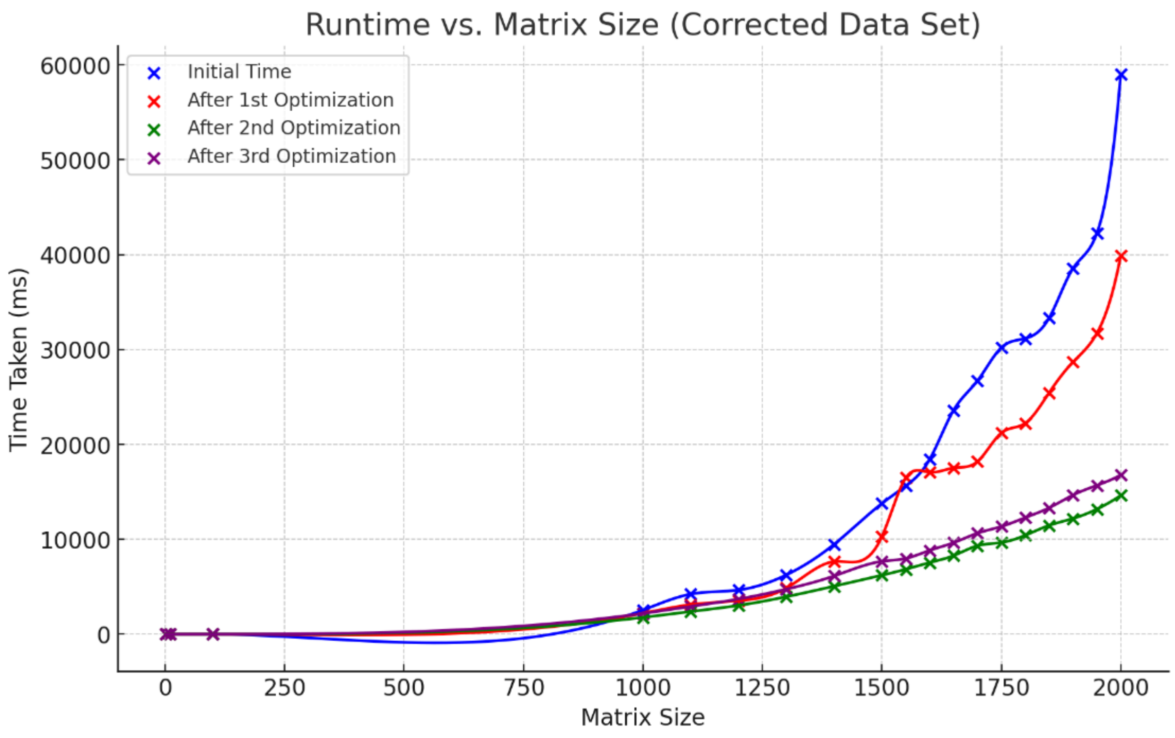
```
int main() {
    for (int i = 1; i <= 2000; ) {
        float *a = (float*)malloc(i * i * sizeof(float)), *b = (float*)malloc(i * i * sizeof(float)), *c = (float*)malloc(i * i * sizeof(float));
        createMatrix(a, i);
        createMatrix(b, i);
        createMatrix(a, i);
        clock_t startTime = clock();
        multiplyMatrices(a, b, c, i);
        clock_t endTime = clock();
        double timeCost = (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000;
        printf("The Size of Matrices: %d\nTime Taken: %f ms", i, timeCost);
        startTime = clock();
        optimized_I(a, b, c, i);
        endTime = clock();
        timeCost = (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000;
        printf("\nTime Taken after the first Optimization: %f ms", timeCost);
        startTime = clock();
        optimized_II(a, b, c, i);
        endTime = clock();
        timeCost = (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000;
        printf("\nTime Taken after the second Optimization: %f ms", timeCost);
        startTime = clock();
        optimized_III(a, b, c, i);
        endTime = clock();
        timeCost = (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000;
        printf("\nTime Taken after the third Optimization: %f ms", timeCost);
        printf("\n--------------------------------");
        if (i < 1000) i *= 10;
        if (i < 1500) i += 100;
        else i += 50;
        free(a);
        free(b);
        free(c);
    }
}
```

ANALYSIS

1. Visualization

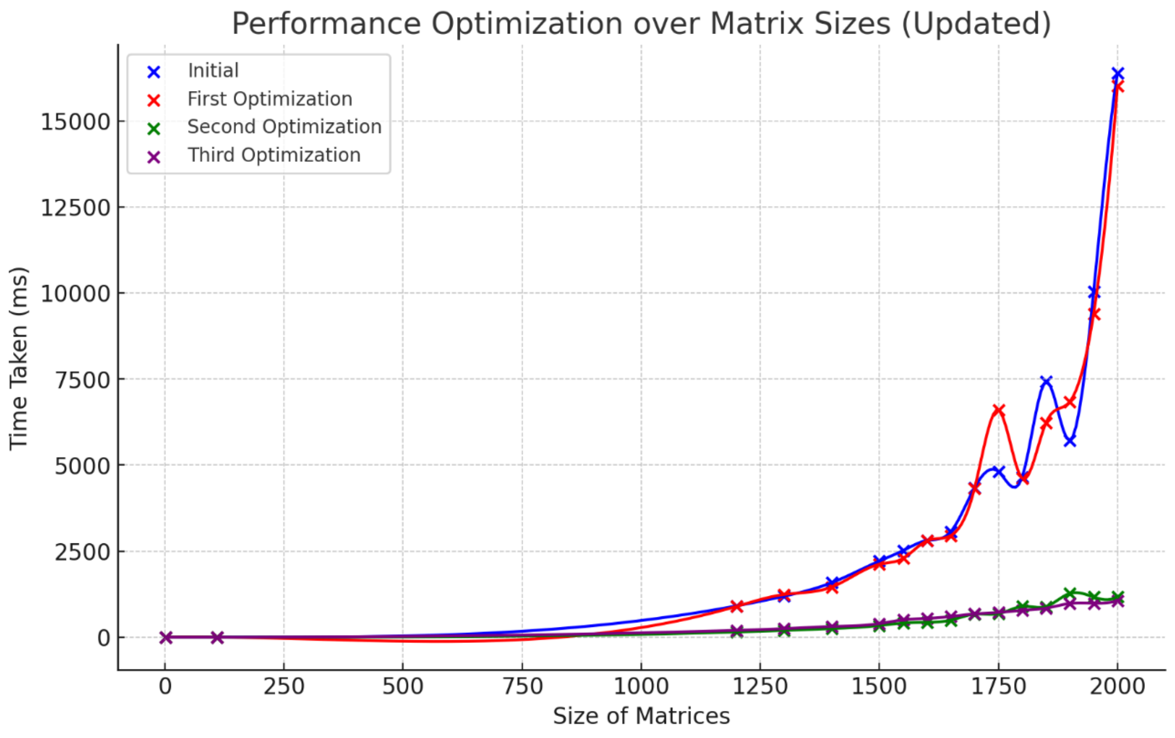
Compile and run the executable file directly and get a set of running time data. Draw a scatter plot and link it with smooth curves.

```
gcc MatrixMultiplication.c
./a.out
```



Then compile it with the option `-O3` and run the executable file:

```
gcc -O3 MatrixMultiplication.c
./a.out
```



2. Analysis

2.0 Quantitative Analysis

Let's begin with the quantitative analysis. We calculate the times of visiting and writing matrices and list this diagram:

	Original	**optimized_I**	**optimized_II**	**optimized_III**
Read Matrix A	N^3	N^3	N^2	N^2
Read Matrix B	N^3	N^3	N^3	N^3
Write Matrix C	N^3	N^2 *C is written when the most inner loop end*	N^3	N^3

2.1 Why such a slight change makes `optimized_II` much faster?

It seems that there are no difference between these optimized method.

So, why does such a simple modification of loop order can significantly improve the efficiency?

1. In the `optimized_II` method, the data accesses to matrix `B` are sequential and continuous. In the most inner loop, pointer `*b` moves to the adjacent next float type variable's address.

   But in `original` method, the ways to accesses matrix `B` differs. Pointer `*b` moves to the next `N` float type variables' address.

2. Modern CPUs have prefetch mechanisms that predict and load upcoming memory data. By accessing matrix B sequentially, optimized_II makes prefetching more effective, further reducing the processor's wait time for memory loads.
3. The structure of optimized_II is relatively simple and regular, making it easier for compilers to optimize the code, such as loop unrolling. These compiler-level optimizations further improve execution efficiency.

Although the total number of read and write operations in `optimized_II` has not decreased, by optimizing the memory access order continuously, it effectively reduces the CPU's wait time. This is the primary reason for its significant performance improvement. Particularly in processing large-scale matrix multiplication, this optimization is crucial.

2.2 Why there is no obvious difference between `optimized_II` and `optimized_III`?

1. `optimized_III` introduces parallel processing. However, the overhead associated with parallel computing (thread creation, destruction, and synchronization costs) might offset some of the performance gains, especially when `N` is relatively small and the workload per thread is not enough to cover these costs.
2. The block size (`BLOCK_SIZE`) in `optimized_III` impacts performance a lot. If `BLOCK_SIZE` does not match the CPU cache size well, the benefits of block partitioning might not be very pronounced.

The author of this project un-strictly adjusts the block size, without any powerful and precise algorithms to get the exactly numerical value of `BLOCK_SIZE`. But this optimization is theoretically feasible.

2.3 Why the `-O3` option runs much faster?

What is `` `-O3 `` option?

In C compilers, `-O3` represents one level of optimization, where "O" stands for Optimization, and the number 3 indicates the level of optimization. Compared to `-O2`, `-O1`, or no optimization (`-O0`), `-O3` provides more aggressive code optimization strategies, including but not limited to the following:

The compiler attempts to unroll iterations within loops to reduce the overhead of loop control and increase the efficiency of loop execution. The `-O3` optimization option can significantly improve the execution speed of a program but may also increase compilation time and, in some cases, result in larger program size.

However, aggressive optimization might introduce unpredictable behavior, especially in complex systems, so be careful when using `-O3`.

IMPLEMENTATION IN `Java` (Same Algorithm)

1. Initialize the matrices

```java
private static float[][] createMatrix(int N) {
    float[][] matrix = new float[N][N];
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            matrix[i][j] = (float) Math.random();
    return matrix;
}
```

2. Perform matrix multiplication using brute force

```java
private static void multiplyMatrices(float[][] A, float[][] B, float[][] 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];
}
```

3. Similarly Modify the Loop Structure

```java
private static void optimized_I(float[][] A, float[][] B, float[][] C, int N) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float r = 0;
            for (int k = 0; k < N; k++)
                r += A[i][k] * B[k][j];
            C[i][j] = r;
        }
}

private static void optimized_II(float[][] A, float[][] B, float[][] C, int N) {
    for (int i = 0; i < N; i++)
        for (int k = 0; k < N; k++) {
            float r = A[i][k];
            for (int j = 0; j < N; j++)
                C[i][j] += r * B[k][j];
        }
}

private static void optimized_III(float[][] A, float[][] B, float[][] C, int N) {
    int BLOCK_SIZE = 50;
    if (N > 1500) BLOCK_SIZE = 100;
    for (int ii = 0; ii < N; ii += BLOCK_SIZE)
        for (int kk = 0; kk < N; kk += BLOCK_SIZE)
            for (int i = ii; i < Math.min(ii + BLOCK_SIZE, N); i++)
                for (int k = kk; k < Math.min(kk + BLOCK_SIZE, N); k++) {
                    float r = A[i][k];
                    for (int j = 0; j < N; j++)
                        C[i][j] += r * B[k][j];
                }
}
```

---

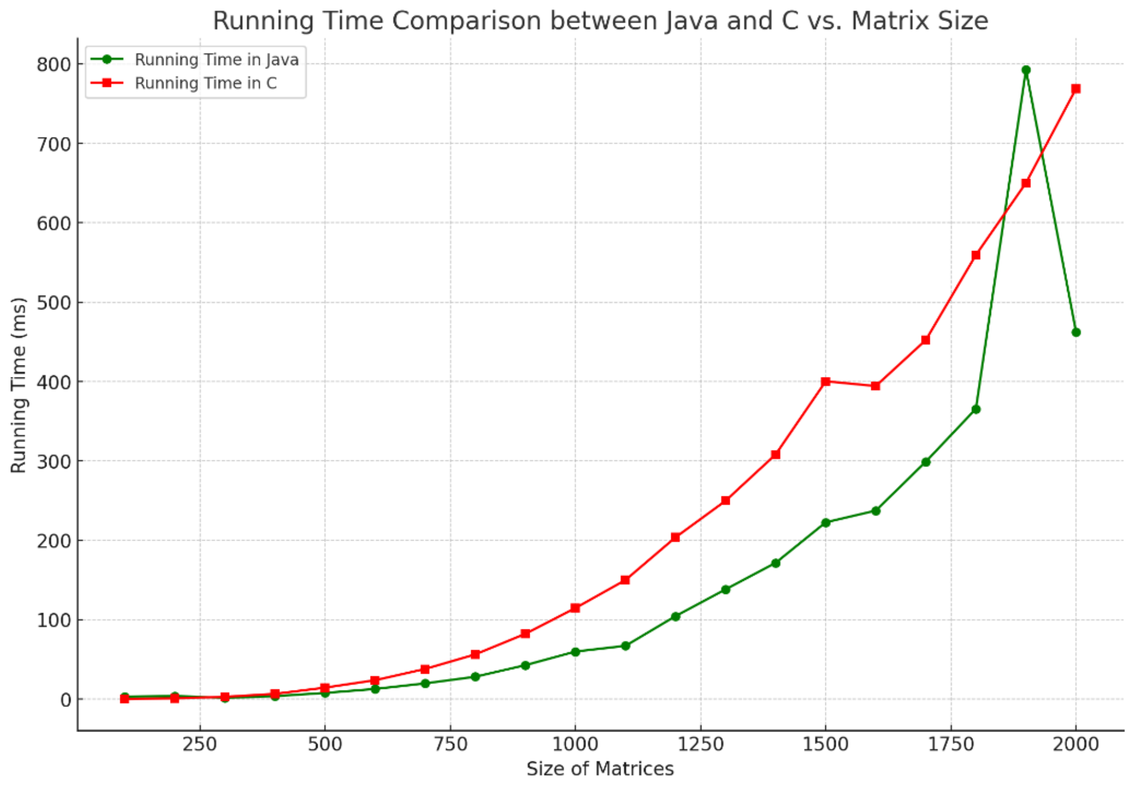
4. Implementation in `Main` function

```java
public static void main(String[] args) {
    for (int i = 0; i <= 2000; i += 100) {
        float[][] A = createMatrix(i);
        float[][] B = createMatrix(i);
        float[][] C = new float[i][i];
        System.out.println("The Size of Matrices: " + i);
        long startTime = System.currentTimeMillis();
        multiplyMatrices(A, B, C, i);
        long endTime = System.currentTimeMillis();
        System.out.println("Time Taken: " + (endTime - startTime) + " ms");
        startTime = System.nanoTime();
        optimized_I(A, B, C, i);
        endTime = System.nanoTime();
        System.out.println("Time Taken after the first Optimization: " + (endTime - startTime) / 1e6 + " ms");
        startTime = System.nanoTime();
        optimized_II(A, B, C, i);
        endTime = System.nanoTime();
        System.out.println("Time Taken after the second Optimization: " + (endTime - startTime) / 1e6 + " ms");
        startTime = System.nanoTime();
        optimized_III(A, B, C, i);
        endTime = System.nanoTime();
        System.out.println("Time Taken after the third Optimization: " + (endTime - startTime) / 1e6 + " ms");
        System.out.println("-------------------------------------------------\n");
    }
}
```

ANALYSIS (Compare With `C Language`)

1. Visualization

- Use `-O3` to compile and run the `C Language` program.



---

2. Analysis

Why did `Java` run faster than `C Language`?

1. **JIT**: The Java Virtual Machine (JVM) uses a Just-In-Time (JIT) compiler to optimize code at runtime. The JIT compiler can analyze the execution of the program and apply highly optimized code, including better loop unrolling.
2. **Garbage Collection**: Java's garbage collection (GC) mechanism manages memory allocation and deallocation, reducing the risk of memory leaks. Although GC introduces additional overhead, its impact is relatively minor for compute-intensive tasks like matrix multiplication.
3. **Test Environment and Conditions**: Performance test results are influenced by various factors, including but not limited to hardware configuration of the test environment, operating systems, compiler versions, runtime environments and so on. Different testing conditions might lead to different performance outcomes.

CONCLUSION

1. Limitations

- The calculation of `BLOCK_SIZE` in `optimized_II` is not precise enough, which may leads to unsatisfactory result;
- When a computer is running programs, the CPU is occupied by different other processes, resulting in inaccurate time measurement.

2. Experience and Insights

- I learned the difference between these two languages from it, including the ;
- I have gained a deeper understanding of memory, addresses, and other related knowledge in `C Language`;
- I learned many other knowledge such as compilation option `-O3`, OpenMP , CPU cache and so on.