C/C++ Programming
Project 2: Simple Matrix Multiplication

Liyuan Zhang, Department of EEE
Student ID: 12012724

March 31,2024

1 BACKGROUND

Matrix multiplication is one of the most fundamental arithmetic operations, which can be conducted and boosted by computer. Matrices are essentially correspondent to linear maps, and the definition of matrix multiplication is based on the very nature of matrix multiplication which is the composition of linear maps. Suppose there is a $m \times n$ matrix A and a $n \times p$ matrix B, the multiplication of A and B denoted by C is:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Suppose the time complexity of calculating each element of the multiplication is $T_{element}$ and the time complexity of calculating the whole multiplied matrix is T_{whole}:

$$T_{element} = O(n)$$

$$T_{whole} = \sum_{i=1}^{m} \sum_{j=1}^{p} T_{element} = mp \cdot T_{element} \approx O(n^3)$$

The analysis of time complexity indicates that as the size of multiplier matrix doubles, the consumed time should be increased by 8 times. The operations involved in a matrix multiplication are addition and multiplication of floating-point numbers, while technically, the consumed time of CPU to perform additions and multiplications are different. So some blocked-matrix-based computational optimization algorithms such as the *Strassen* and *Winograd* algorithms boost the computation by replacing some multiplications into additions, and reduce the time complexity from $O(n^3)$ to $O(n^{2.8})$ and $O(n^{2.3})$ respectively. In this project, we do not adopt the optimized algorithms because we mainly focus on the measurement and comparison of performance of the multiplication programs implemented by C and JAVA.

2 TASKS

- Implement a program for matrix multiplication in C and JAVA where the multiplication can be wrapped as a function. The datatype of all computations are set to be **float**.
- Measure the time of computation. Observe the increasing pattern of time with respect to the size of matrices.
- Compare the performance of the two programs in C and JAVA and explain the reasons for any observed differences.

- Perform additional comparisons and analysis to identify any interesting insights.

3 CODES

3.1 C

The file structure is as followed:

includes
- matrix.h
- multiply.h

data
- data files in .txt format

main.c
matrix_multiply.c

In matrix.h, we defined a *struct* named *Matrix* having member variable *row*, *col* and *data*:

Matrix.h

```
1 struct Matrix
2 {
3         int row;
4         int col;
5         float * data ;
6 };
```

The *row* and *col* specify the number of rows and columns of a matrix, and the float pointer *data* specifies the address of the head of a float array where the values are stored.

In multiply.h, the function of matrix multiplication is declared:

multiply.h

```
1 struct Matrix naive_multiply(struct Matrix A, struct Matrix B);
```

The function naive_multiply have two parameters: Matrix A and B, and returns a Matrix.

In matrix_mul_naive.c, we further implemented the function naive_multiply:

matrix_mul_naive.c

```
1  #include<stdio.h>
2  #include<math.h>
3  #include"./include/matrix.h"
4  #include"./include/multiply.h"
5  #include<stdlib.h>
6  #define EXIT_SUCCESS 0
7  #define EXIT_FAILURE 1
8
9  struct Matrix naive_multiply(struct Matrix A, struct Matrix B)
10 {
11         int rowA = A.row;
12         int colA = A.col;
13         int rowB = B.row;
14         int colB = B.col;
15         if(!(rowA==colA&&rowB==colB))
16         {
17                 printf("Matrix not square\n");
18                 exit(EXIT_FAILURE);
19         }
20         if(!(rowA==rowB&&colA==colB))
21         {
22                 printf("Dimensions not equal\n");
23                 exit(EXIT_FAILURE);
24         }
25         int m = rowA;
26         struct Matrix C;
27         C.row = rowA;
28         C.col = colA;
29
30         for(int i=0;i<sizeof(C.data)/sizeof(float);i++)
31         {
32                 C.data[i] = 0;
33         }
34
35         for(int i=0;i<m;i++)
36         for(int j=0;j<m;j++)
37         for(int k=0;k<m;k++)
38         {
39                 C.data[i*m + j] += A.data[i*m + k] * B.data[k*m + j];
40         }
41
42         return C;
43 }
```

Line 1-3 are to include standard libraries in C and line 4-5 are to include the .h files. Line 5-6 are to make a macro definition of two status of exit() used when input matrices are invalid. When the function gets the two input matrices A and B,

it first make an decision whether the number of columns in A equals to the number of rows in B. If so, the two input matrices can be multiplied and the program continues. If not, the two input matrices cannot be multiplied and the program will be terminated by exit(EXIT_FAILURE) and print error message. Line 25-30 are to declare and initialize a Matrix C as the container. Line 30-33 set all elements in the storage memory space to be 0 (If not do so, the program will accumulate the value each time the function is invoked.). Line 35-43 is the i-j-k for-loop to implement matrix multiplication, which is the essential part of the program.

In main.c, we defined several functions correlated to the I/O of matrices:

- read_matrix: allow the user to insert matrices from the console.
- read_matrix_from_file: the program will read values from a .txt file and assign to a Matrix struct.
- print_matrix: print the values of a matrix to the console.

Then, the main function invokes the I/O functions and the function of matrix multiplication, to do the test.

main.c

```
1  #include<stdio.h>
2  #include<math.h>
3  #include"./include/matrix.h"
4  #include"./include/multiply.h"
5
6  void print_matrix(struct Matrix mat)
7  {
8          for(int i=0;i<mat.row;i++)
9          {
10                 for(int j=0;j<mat.col;j++)
11                 {
12                         printf("%.2f ",mat.data[i*mat.row + j]);
13                 }
14                 printf("\n");
15         }
16         printf("\n");
17 }
18
19 struct Matrix read_matrix(struct Matrix mat)
20 {
21         printf("the size of the matrix is:\n");
22         scanf("%d %d",&mat.row,&mat.col);
23         printf("please insert the elements of the matrix:\n");
24         printf("%d %d\n",mat.row,mat.col);
25
```

```
26          for(int i=0;i<mat.row;i++)
27          for(int j=0;j<mat.col;j++)
28          {
29                  scanf("%f",&mat.data[i*mat.row + j]);
30          }
31          return mat;
32
33  }
34
35  struct Matrix read_matrix_from_file(struct Matrix  mat, const char* dir)
36  {
37          FILE * file = NULL;
38          file = fopen(dir, "r");
39          fscanf(file,"%d",&mat.row);
40          fscanf(file,"%d",&mat.col);
41
42          for(int i=0;i<mat.row;i++)
43          {
44          for(int j=0;j<mat.col;j++)
45          {
46          fscanf(file,"%f",&mat.data[i*mat.row+j]);
47          }
48          }
49          fclose(file);
50          return mat;
51  }
52  int main()
53  {
54          struct Matrix A,B,C;
55          A = read_matrix_from_file(A,"./data/a_100.txt");
56          print_matrix(A);
57          B = read_matrix_from_file(B,"./data/b_100.txt");
58          print_matrix(B);
59          for(int i =0;i<10;i++)
60          {
61                  C = naive_multiply(A, B);
62                  printf("The result is:\n");
63          }
64          print_matrix(C);
65  }
```

In function read_matrix_from_file, we used FILE * to represent a object of file, and used fopen, fscan, fclose to read data from file. In the main function, we designate the directory of the data file, read the data and assign it to matrix A and B. Then we made multiple times of matrix multiplication. In the end, we print the result of the multiplication to verify its correctness.

The above is the complete demonstration of the C codes. The JAVA codes will

be demonstrated in the next section.

3.2 JAVA

The file structure is:

data
– data files in .txt format
Main.java
MatrixIO.java
MatrixMultiplyNaive.java

In MatrixMultiply.java, we first declared and defined a class named Matrix consisting of three members: row, col and data. And we defined the function for matrix multiplication. (To better showcase the codes in LaTeX, the codes are modified and do not strictly obey the standard code style.)

MatrixMultiply.java

```
1  public class MatrixMultiplyNaive {
2  Matrix multiplyNaive(Matrix A, Matrix B)
3  {
4  Matrix C = new Matrix();
5  if(A.col != B.row)
6  {
7  System.out.println("Invalid size of input matrices.");
8  return C;
9  }
10 C.row = A.row;
11 C.col = B.col;
12 C.data = new float[C.row*C.col];
13 for(int i=0;i<A.row;i++)
14 {
15 for(int j=0;j<B.col;j++)
16 {
17 for(int k=0;k<A.col;k++)
18 {
19 C.data[i*A.row + j] += A.data[i*A.row + k] * B.data[k*B.row + j];
20 }
21 }
22 }
23 return C;
24 }
25 }
26 class Matrix
27 {
28         int row;
```

```
29          int col;
30          float [] data;
31  }
```

In MatrixIO.java, same as in C-version program, we defined some functions to input and output matrices.

MatrixIO.java

```
1   import java.util.Scanner;
2   public  class MatrixIO {
3   public void printMatrix(Matrix mat)
4   {
5   for(int i=0;i<mat.row;i++)
6   {
7   for(int j=0;j<mat.col;j++)
8   {
9   System.out.print(mat.data[i*mat.row + j]+" ");
10  }
11  System.out.println();
12  }
13  System.out.println();
14  }
15  public Matrix readMatrix(Scanner scn)
16  {
17  System.out.println("size of the matrix is:");
18  int m = scn.nextInt();
19  int n = scn.nextInt();
20  Matrix mat = new Matrix();
21  mat.row = m;
22  mat.col = n;
23  mat.data = new float[m*n];
24  if(m<=0||n<=0)
25  {
26  System.out.println("size must be greater than zero.");
27  return mat;
28  }
29  for(int i=0;i<m;i++)
30  {
31  for(int j=0;j<m;j++)
32  {
33  mat.data[i*mat.row + j] = scn.nextFloat();
34  }}
35  return mat;
36  }}
```

In Main.java, we invoked all the functions we declared and test the program.

Main.java

```java
import java.util.Scanner;
import java.io.File;
public class Main {
public static void main(String args[]) {
File file_a = new File("./data/a_100_float.txt");
File file_b = new File("./data/b_100_float.txt");
try
{
long startTime=System.currentTimeMillis();
Scanner scn_a = new Scanner(file_a);
Scanner scn_b = new Scanner(file_b);
MatrixIO io = new MatrixIO();
MatrixMultiplyNaive mul = new MatrixMultiplyNaive();
Matrix A = io.readMatrix(scn_a);
Matrix B = io.readMatrix(scn_b);
io.printMatrix(A);
io.printMatrix(B);
for(int i=0;i<10;i++)
{Matrix C = mul.multiplyNaive(A, B);io.printMatrix(C);}
long endTime=System.currentTimeMillis();
System.out.println("running time is:"+(endTime-startTime)+"ms");
}catch(Exception e){System.out.println("File not found error");}
}}
```

4 TEST

4.1 Test Corpus

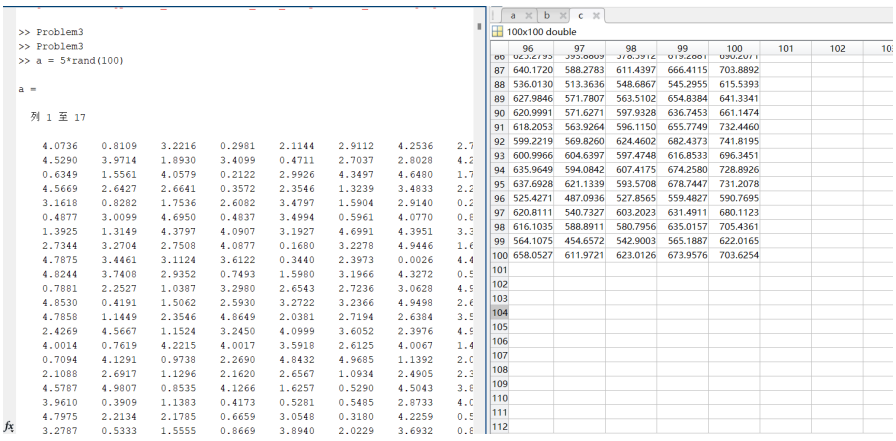


Figure 1: Testing data generated from MATLAB

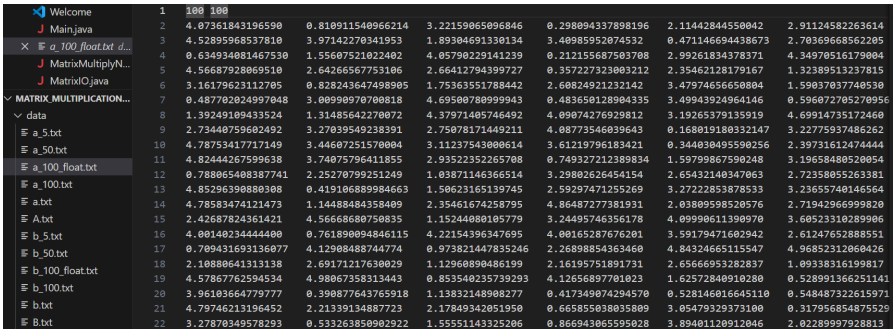
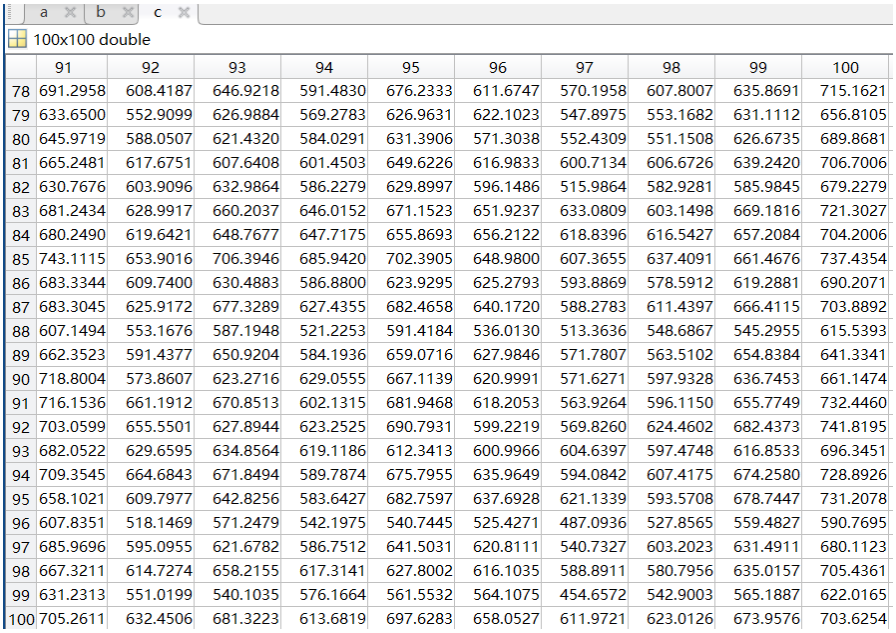


Figure 2: Testing data

The test corpus or dataset is generated from MATLAB. The numbers in the matrices are randomly generated by function rand() in MATLAB. We have matrices of different sizes (10,20,50,100), and different shapes (square or non-square). All the data is stored in folder *data* in form of .txt files.

4.2 Correctness



	91	92	93	94	95	96	97	98	99	100
78	691.2958	608.4187	646.9218	591.4830	676.2333	611.6747	570.1958	607.8007	635.8691	715.1621
79	633.6500	552.9099	626.9884	569.2783	626.9631	622.1023	547.8975	553.1682	631.1112	656.8105
80	645.9719	588.0507	621.4320	584.0291	631.3906	571.3038	552.4309	551.1508	626.6735	689.8681
81	665.2481	617.6751	607.6408	601.4503	649.6226	616.9833	600.7134	606.6726	639.2420	706.7006
82	630.7676	603.9096	632.9864	586.2279	629.8997	596.1486	515.9864	582.9281	585.9845	679.2279
83	681.2434	628.9917	660.2037	646.0152	671.1523	651.9237	633.0809	603.1498	669.1816	721.3027
84	680.2490	619.6421	648.7677	647.7175	655.8693	656.2122	618.8396	616.5427	657.2084	704.2006
85	743.1115	653.9016	706.3946	685.9420	702.3905	648.9800	607.3655	637.4091	661.4676	737.4354
86	683.3344	609.7400	630.4883	586.8800	623.9295	625.2793	593.8869	578.5912	619.2881	690.2071
87	683.3045	625.9172	677.3289	627.4355	682.4658	640.1720	588.2783	611.4397	666.4115	703.8892
88	607.1494	553.1676	587.1948	521.2253	591.4184	536.0130	513.3636	548.6867	545.2955	615.5393
89	662.3523	591.4377	650.9204	584.1936	659.0716	627.9846	571.7807	563.5102	654.8384	641.3341
90	718.8004	573.8607	623.2716	629.0555	667.1139	620.9991	571.6271	597.9328	636.7453	661.1474
91	716.1536	661.1912	670.8513	602.1315	681.9468	618.2053	563.9264	596.1150	655.7749	732.4460
92	703.0599	655.5501	627.8944	623.2525	690.7931	599.2219	569.8260	624.4602	682.4373	741.8195
93	682.0522	629.6595	634.8564	619.1186	612.3413	600.9966	604.6397	597.4748	616.8533	696.3451
94	709.3545	664.6843	671.8494	589.7874	675.7955	635.9649	594.0842	607.4175	674.2580	728.8926
95	658.1021	609.7977	642.8256	583.6427	682.7597	637.6928	621.1339	593.5708	678.7447	731.2078
96	607.8351	518.1469	571.2479	542.1975	540.7445	525.4271	487.0936	527.8565	559.4827	590.7695
97	685.9696	595.0955	621.6782	586.7512	641.5031	620.8111	540.7327	603.2023	631.4911	680.1123
98	667.3211	614.7274	658.2155	617.3141	627.8002	616.1035	588.8911	580.7956	635.0157	705.4361
99	631.2313	551.0199	540.1035	576.1664	561.5532	564.1075	454.6572	542.9003	565.1887	622.0165
100	705.2611	632.4506	681.3223	613.6819	697.6283	658.0527	611.9721	623.0126	673.9576	703.6254

Figure 3: MATLAB result for size 100 matrix multiplication

To test the correctness, we will refer to the result calculated by MATLAB. If the results computed by the program we wrote in C and JAVA are consistent with that by MATLAB, then our program can perform correctly.

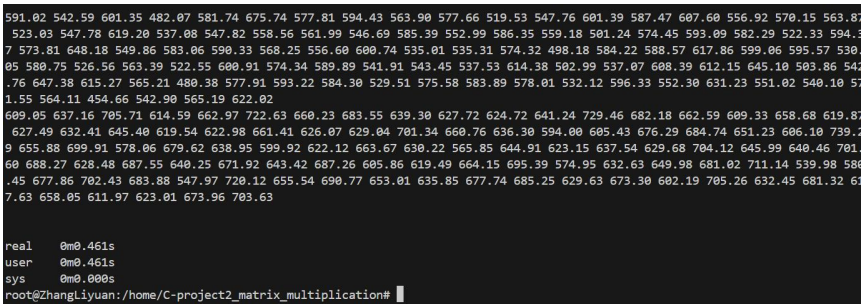


Figure 4: C program result for size 100 matrix multiplication

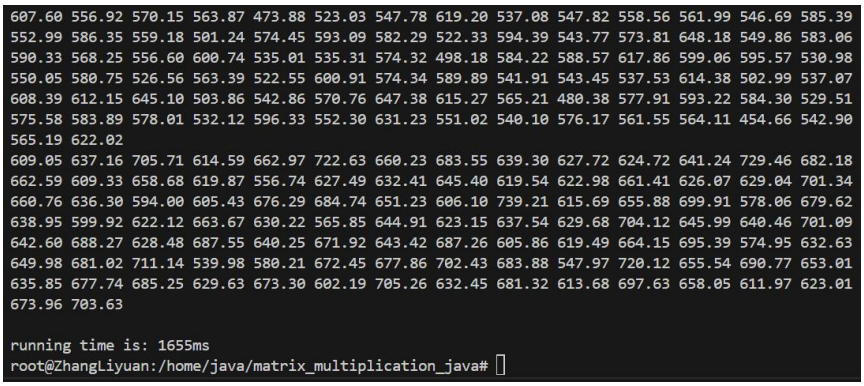


Figure 5: Java program result for size 100 matrix multiplication

We can see that the results both in C and JAVA are consistent with the MATLAB result, so we can basically affirm that our program performs correctly.

4.3 Runtime Comparison

To measure the running time in C, we simply use

```
time ./a.out
```

when we run the executable file. This command will measure the CPU time, the user time and the system time of the program. CPU time is the duration of the kernel of CPU processes instructions, which do not include the time of I/O. The user time is the real time measurement, including the whole process of the program. Here, we take the user time as the running time of the program.

And also to better measure the speed of the program, when we compile the .c file by gcc, we use

```
gcc -c main.c matrix_mul_naive.c -O3
```

to activate the automatic optimization in the compiler.

To measure the running time in JAVA, we use

```
long startTime = System.currentTimeMills();
// process
long endTime = System,currentTimeMills();
System.out.println("The running time is:"+(endTime-startTime)+"ms");
```

We test the C and JAVA programs by computing the 5000 times of multiplication of matrices with size of 10,20,50,100,200, respectively, to get significant results, and measure the running time of C and JAVA programs. The result is shown below:

We found that C program has a faster overall speed than JAVA program, and the running time of C program is about 2 times of JAVA program, which is within our expectation that C generally runs faster than JAVA **due to its language features that are close to hardware layer, while the objective-oriented and some other high-level features of JAVA drag down its speed.** Also, we found that the running time for both JAVA and C increases by approximately 8 times when the input matrix size doubles (see the plot below), which roughly verifies the time complexity of this naive three-for-loop algorithm is $O(n^3)$

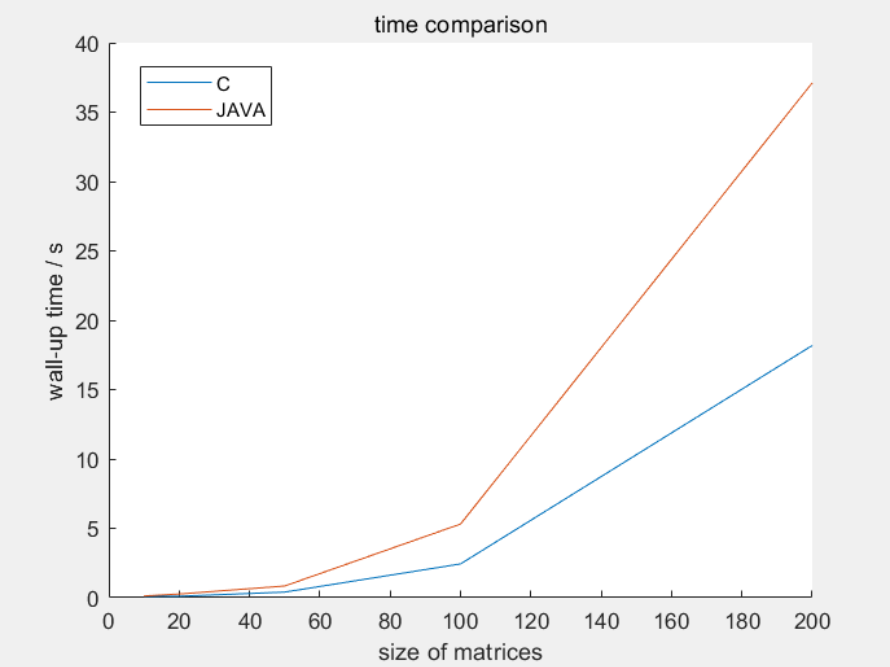


Figure 6: Time comparison between C and JAVA program