CS205 Mid-term Project Report

SID: 11910718 Name: 陆彦青

Part 1. Introduction & Features

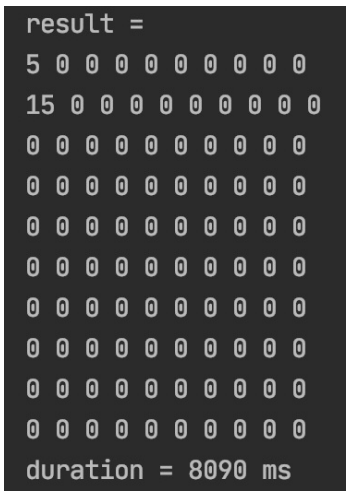
1. The main function of this project is to implement the multiplication of 2 matrices. The type of matrix elements is float.
2. Totally 4 versions of codes are updated. Compared with the first version, the latest is over **10** times faster. In the final version, the time used to multiply 2 matrices with 200M elements respectively is about **770ms**, which is closed to the cost of `cblas_sgemm` in **OpenBLAS**(300ms).
3. **AVX2,OpenMP, thread** are used to speed up the computation.
4. Using multi-thread in different places and different multi-thread functions will actually lead to different results and speeds, so I compare the difference in the final version.
5. One feature of this program is the use of **transposition matrix**. In the process of computing matrix multiplication, the dot product of matrix A's row and matrix B's column need to be done. However, arrays are stored in a 1-dimension space which means either the elements of row or column of a matrix can be arranged continuously. Thus, if we don't transpose matrix B, we have to use an extra for-loop to read the data in the columns of B. Then, one direct advantage of transposing B before the multiplication is that the extra for-loop is not needed in the most complex part of computation.
6. **Matrix inversion** is implemented in this project. I use the adjugate matrix to get the inverse matrix so the determinant is also acquired by using the recursion. This function can be used in matrix multiplication to get the original matrix if another original matrix and the result matrix are known (Both of them must be square matrices).
7. The input of matrix in a specific format is supported.

Part 2. Improvement

∗ *In this part, the matrices I used to test are the same, where m1 is 10*20000000 and m2 is 20000000*10. I use this 2 matrices because the result matrix will be 10*10 which is easy to present. Actually, the dfiff-erence of the scale of row and column will make difference in the result of costed time when the total elements of matrices are the same. The detailed comparison will be shown in the next part.*

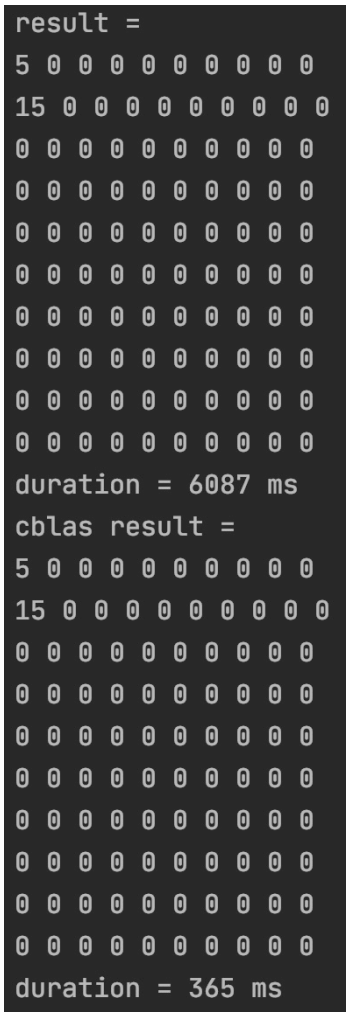
Version #0:

This is a elementary version which only uses some **AVX2** functions to improve the efficiency of CPU. The function can guarantee the accuracy but the speed is not fast.



Version #1:

In this version, `cblas_sgemm` in **OpenBLAS** is imported as a comparison. I edit the **CmakeList.txt** to assure that the compiler can link the library correctly. In addition, I optimize the code style to improve the readibility of the codes. Some long paragraphs of codes are packaged into single functions (for example, `init()`,`del()`,`showMatrix()`). As shown in the screenshot, the speed is a little faster, mainly because I replace the pass-by-value parameters of every functions with pass-by-reference.



Version #2:

The main change of this version is the strategy of multiplication. I transpose the second matrix before the computation and the reason has been explained in part 1. The result of acceleration is considerable.

```
result =
5 0 0 0 0 0 0 0 0 0
15 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
transpose duration = 820 ms
```

Including the time of transposition, the whole process is still faster than version 1. After using multi-thread, the process of transposition can be faster. But using it in the outer and inner loop would not cause much difference.

```
result =
5 0 0 0 0 0 0 0 0 0
15 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
transpose duration = 2237 ms
```

(a) not use multi-thread

```
result =
5 0 0 0 0 0 0 0 0 0
15 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
transpose duration = 1892 ms
```

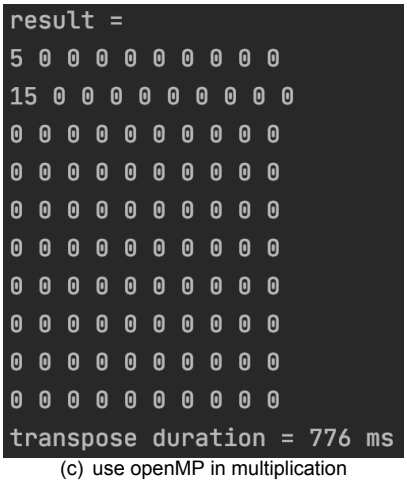
(b) use multi-thread

Version #3:

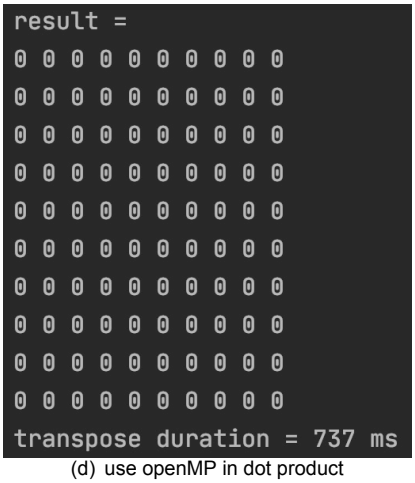
In this version, I try to use multi-thread to speed up the for loop. The libraries I try to use are **OpenMP** and **thread**, where the former is a cross-platform framework and the later is a standard c++11 library. I compare the difference of results and speeds when using **OpenMP** in different loops: the loop in multiplication and the loop in dot product. Since dot product is called in every step of multiplication, multi-thread can not be used in both of them at the same time.

Note that the result of figure(d) is wrong. It is because many threads write the same data in the same time so they conflict mutually. One strightforward solution is to create some arrays to store the datas that would make conflicts. However, as shown in figure(f), the total cost of time is far more than using multi-thread in the multiplication.

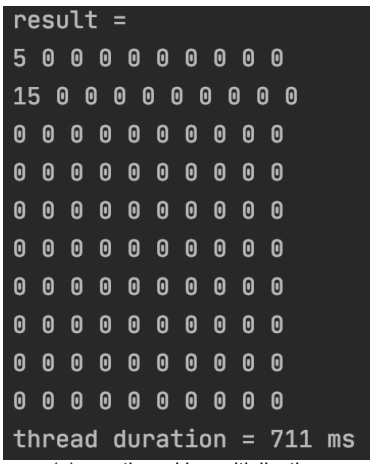
Figure (e) shows the result and time when using **thread**. Since the maximum number of threads my computer supports is 8, I divide the data into 8 parts to compute them respectively. The result is correct and the cost of time is nearly the same as **openMP** (the sequence of calls has been considered). However, the amount of extra code is far more than **openMP** (just 1 line). So from my perspective, **openMP** is a better choice when using multi-thread.



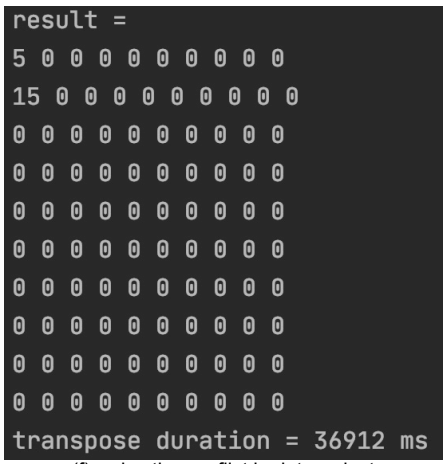
(c) use openMP in multiplication



(d) use openMP in dot product



(e) use thread in multiplication



(f) solve the conflict in dot product

Version #4:

3 new functions are implemented in this version.

1. **The input of matrix.** This function is not complex, so I just show the screenshot here.

```
Please input the rows and columns: 2 2
Please input the data:
1 2 3 4
1 2
3 4
```

2. **Get the determinant of a matrix.** I spread the matrix in the first column, then compute the corresponding cofactors (代数余子式) by using the recursion. At the begining I check whether the row and column are the same, it guarantees that the function will not easily make an error.

3. **Get the inverse of a matrix.** This is the most important function in this version. I implement it by compute the adjugate matrix and the determinant. Suppose A is a n*n invertible matrix, denote A^* as the adjugate matrix, then $A^{-1} = \frac{A^*}{|A|}$. Comparing with some other functions in the Internet, I only use 3 layers for-loop and the code is relatively concise. The main idea is similar to the function of getting the determinant (just plus one dimension).

   I use 1 simple example to present the 2 functions above. It is closely related to matrix multiplication.

$$m4 = \begin{pmatrix} 1 & 1 \\ 0 & 2 \end{pmatrix}, m6 = \begin{pmatrix} 1 & 1 \\ 0 & 2 \end{pmatrix}.$$

   We know that $m4 * m5 = m6$ and $m4$ is invertible, then $m5 = (m4)^{-1} * m6$. Clearly m5 should be an 2*2 identity matrix, the program can check it.

```
transpose of m6:
1 0
1 2
inverse of m4:
1 -0.5
-0 0.5
m4^{-1} * m6 =
1 0
0 1
```

Part 3. Efficiency analysis

1. After version 1-3, the efficiency of computing matrix multiplication is improved every time. I make a chart to list the improvement.

Version	0	1	2	3	OpenBLAS
Average Time / ms	8090	6087	820(1892)	776	260
Increased ratio		32%	642%(322%)	6%	

   As shown in the chart, transpose greatly contributes to the improvement of efficiency.

2. The scale of column and row can also influence the cost of time when the total elements of the matrices are the same. So I make another chart to show the difference.

Row*Column	10*20000000	100*2000000	1000*200000	10000*20000
matrixCompute_t / ms	776	4424	29835	199698
OpenBLAS / ms	260	245	1409	12176

   As the scale of row and column approaches 1, the cost of time grows geometrically. I think it is may because reading the data from the cache to cpu occupy the most time of cost. As the number of rows increases, the time(次数) needed to exchange the data increases geometrically. Thus the total time also increases geometrically.

Part 4. Source Code

All the source codes are included in a GitHub repository: https://github.com/Beauchamp-West/2020Fall_CS205/tree/master/Project