

CS205 C/C++ Programming - Project Report 2

Name:张闻城

SID:12010324

Part 1 - Analysis

1. Time consumption: **float** vs. **double**

After some tests, it seems that multiplication of **float** is more faster than multiplication of **double**. I think there are two main reasons that cause this result:

- **Memory consumed:**

double takes more memory to store data. Take multiplication between two 2048x2048 matrices as example. A 2048x2048 matrix has 4194304 elements. If stored in **float**, it takes 16Mb of bits, while it takes 32Mb when stored in **double**. The huge difference between memory consumed to store data will cause huge difference between the times of data manipulation and data access. Thus **double** takes more time to get the result compared with **float**, especially when the size of matrix is large.

- **Hardware:**

The performance of multiplication maybe differ in **float** and **double**. Refer to a statement in *Optimizing software in C++*,

Single precision division, square root and mathematical functions are calculated faster than double precision when the XMM registers are used, while the speed of addition, subtraction, multiplication, etc. is still the same regardless of precision on most processors (when vector operations are not used).

So maybe underlying structure of hardware will also cause the difference of time consumption.

- **Something interesting:**

- The explanation above seems to be right. But after I compare the time consumed during calculating some data (the number of data is not large, just 100) of **float** with that of **double**, I find that **double** is faster than **float**, which is weird.

Test program

```
1  #include <iostream>
2  #include <ctime>
3  using namespace std;
4
5  int main()
6  {
7      float fArr[] = {88.3f};
8      double dArr[] = {88.3};
9      struct timespec start = {0, 0};
```

```

10     struct timespec end = {0, 0};
11
12     float fRes = 0.0f;
13     clock_gettime(CLOCK_REALTIME, &start);
14     for (int i = 0; i < 100; i++)
15     {
16         fRes += fArr[0] * fArr[0];
17     }
18     clock_gettime(CLOCK_REALTIME, &end);
19     printf("float计算运行时间:%lds %ldns\n", end.tv_sec -
start.tv_sec, end.tv_nsec - start.tv_nsec);
20
21     double dRes = 0.0;
22     clock_gettime(CLOCK_REALTIME, &start);
23     for (int i = 0; i < 100; i++)
24     {
25         dRes += dArr[0] * dArr[0];
26     }
27     clock_gettime(CLOCK_REALTIME, &end);
28     printf("double计算运行时间:%lds %ldns\n", end.tv_sec -
start.tv_sec, end.tv_nsec - start.tv_nsec);
29
30     return 0;
31 }

```

Output:

```

float计算运行时间:0s 441ns
double计算运行时间:0s 421ns

```

After searching on the Internet, I find out the reason. The calculations between floating-point number in computer are all performed as double-precision, no matter it is **double** or not. Thus, the first step of calculation between **float** is to convert **float** to **double**. So, without the time consumption of converting progress, **double** will be faster (if the amount of data is not large).

Such being the case, why **float** is still faster than **double** in the progress of matrix multiplication? I believe it is still due to the memory consumption. Since the difference of memory consumption is extremely large (that of the [example](#) at the top of article is **16Mb!**), the effect of manipulating such huge data dominates the reason of time consumption difference.

- The explanation seems to be right again. **However**, after I search on the Internet over again and again, I find something new, again. It is said that there is no such a rule that says **float** will be converted into **double** in C++. It also has something to do with the instruction set (**FPU** and **SSE/AVX**). Different instruction set has different performance with floating-point number calculation (**FPU** does all floating-point number's calculation as 80-bit number, while **SSE/AVX** do **float** and **double** calculation separately).

To conclude, the difference of data length is the main reason that cause the difference of matrix multiplication's time consumption between float and double.

2. Accuracy: float vs. double

On a 64-bit operating system with a x64 CPU, the size of **float** in C++ is 4 bytes while that of **double** is 8 bytes. Thus **double** can store more bits of mantissa and exponent (11-bit of exponent and 52-bit of mantissa). Obviously, **double** has more accuracy than **float**.

Part 2 - Improve the program

1. Improve the speed:

When the size of is huge, it will take much time to finish the multiplication (eg.

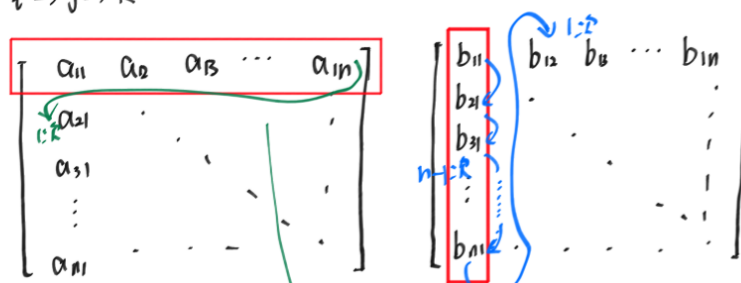
$A_{2048 \times 2048} \times B_{2048 \times 2048}$ takes about 50s). After searching on the Internet, I found a way to improve the speed.

- Reference: [C++加速矩阵乘法的最简单方法](#)

One way to improve the speed is to change the order of loop. It is said that the data stored in a row of an array is continuous while in column is not. Thus it takes few time to access the data that stored in a row (because you don't have to jump to other address to access discontinuous data). The simple order of loop is $i \rightarrow j \rightarrow k$. Assume the size of two matrices are all $n \times n$. The number leaps to access discontinuous data is $n^3 + n^2 + n$. If change the order to $i \rightarrow k \rightarrow j$, it needs $2n^2 + n$ leaps. Thus the speed will be improve.

Details of how to get the number of leaps in a matrix multiplication

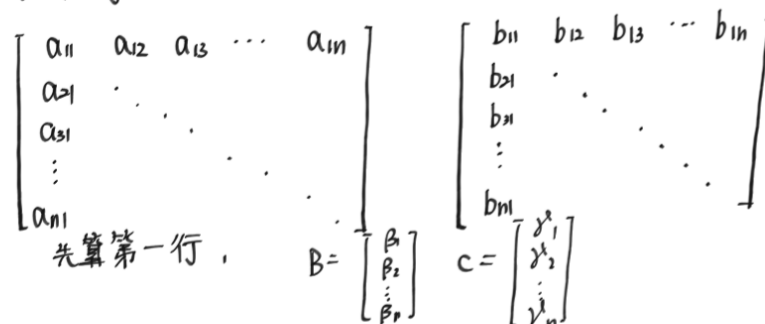
① $i \rightarrow j \rightarrow k$



算一个元素共 n 次

\therefore 共 $n^2 \cdot (n+1) + n$ 次
矩阵 C 跳转次数

② $i \rightarrow k \rightarrow j$



先算第一行

$$B = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

$$C = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_n \end{bmatrix}$$

$$\gamma_1 = a_{11}\beta_1 + a_{12}\beta_2 + a_{13}\beta_3 + \dots + a_{1n}\beta_n$$

共 n 次 (包括跳回 b_{11} 的那次)

$$\gamma_2 = a_{21}\beta_1 + a_{22}\beta_2 + a_{23}\beta_3 + \dots + a_{2n}\beta_n$$

计算一行结果需 $n+1$ 次, n 行结果需 n^2+n 次

C 共跳转 n^2 次

\therefore 共跳转 $2n^2+n$ 次

- Comparison of time consumption:

```
zephyrus@LAPTOP-IIC396SP: /mnt/d/OneDrive - Office/File/Project/VS Code/C++/CS205-Project 2$ ./main mat-A-2048.txt mat-B-2048.txt out.txt
2048x2048的矩阵与2048x2048的矩阵以float,按ijk的顺序相乘得出结果共耗时: 43.827401s
2048x2048的矩阵与2048x2048的矩阵以float,按ikj的顺序相乘得出结果共耗时: 28.053239s
2048x2048的矩阵与2048x2048的矩阵以double,按ijk的顺序相乘得出结果共耗时: 49.933257s
2048x2048的矩阵与2048x2048的矩阵以double,按ikj的顺序相乘得出结果共耗时: 28.836138s
```

Note: The time consumed only contains the progress of matrix multiplication, not including file reading.

2. Get the size of matrix:

I get the size of matrix by count how many lines (to get number of rows) and how many spaces within a single line (to get number of columns) in **mat.txt**, so user don't need to add another argument in the file name to indicate the size of matrix.

3. Multiplication between random size of matrices:

Since we can get size of matrix, it is easy to implement. Besides, my program will check if it is possible to do multiplication of two matrices (eg. $A_{m \times n} \times B_{s \times t}$ requires **n = s**).

Part 3 - Code

Things to be known before testing the program:

The format of command line parameter is not strictly restricted. The size of matrix is not required to be mentioned in the file's name. And the name of the file that stored the result is also not necessary.

```
1 | ./matmul mat-A.txt mat-B.txt
```

```
1  #include <iostream>
2  #include <string>
3  #include <cstring>
4  #include <fstream>
5  #include <time.h>
6  using namespace std;
7
8  class Matrix
9  {
10 private:
11     //attributes
12     float **fmat;
13     double **dmat;
14     unsigned int row;
15     unsigned int col;
16     string filename;
17
18     /**
19      * @brief 获取矩阵的行数与列数
20      */
21     void setSize()
22     {
23         ifstream ifs;
24         ifs.open(filename, ios::in);
25         if (!ifs.is_open())
26         {
27             cout << "读取矩阵 " << filename << " 失败，程序退出" << endl;
28             exit(100);
29         }
30         string buffer;
31         while (getline(ifs, buffer))
32         {
33             row++;
34         }
```

```

35     ifs.close();
36     ifs.open(filename, ios::in);
37     if (getline(ifs, buffer))
38     {
39         for (int i = 0; i < buffer.length(); i++)
40         {
41             if (buffer[i] == 32)
42             {
43                 col++;
44             }
45         }
46         col++;
47     }
48     ifs.close();
49 }

```

```

51 public:
52     /**
53      * @brief 构造矩阵
54      * @param filename 要读取的txt文件的名称
55      */
56     Matrix(string filename)
57     {
58         row = 0;
59         col = 0;
60         this->filename = filename;
61         setSize();
62     }
63
64     unsigned getRow()
65     {
66         return row;
67     }
68
69     unsigned getCol()
70     {
71         return col;
72     }
73
74     /**
75      * @brief 将txt文件中的矩阵读取到二维数组中去
76      */
77     void prepareMat()
78     {
79         fmat = new float *[row];
80         dmat = new double *[row];
81         for (int i = 0; i < row; i++)
82         {
83             fmat[i] = new float[col];
84             dmat[i] = new double[col];
85         }
86
87         //Initialization
88         for (int i = 0; i < row; i++)
89         {
90             for (int j = 0; j < col; j++)
91             {
92                 fmat[i][j] = 0.0f;

```

```

93         dmat[i][j] = 0.0;
94     }
95 }
96
97 //Read file
98 ifstream ifs;
99 ifs.open(filename, ios::in);
100 if (!ifs.is_open())
101 {
102     cout << "读取矩阵 " << filename << " 失败，程序退出" << endl;
103     exit(100);
104 }
105 char buffer[32];
106 unsigned int i = 0;
107 unsigned int j = 0;
108 while (ifs >> buffer)
109 {
110     if (i == row && j == col)
111     {
112         break;
113     }
114     fmat[i][j] = stof(buffer);
115     dmat[i][j] = stod(buffer);
116     if (++j >= col)
117     {
118         i++;
119         j = 0;
120     }
121 }
122 ifs.close();
123 }
124
125 /**
126  * @brief float矩阵的乘法，乘法顺序为当前对象的矩阵乘以参数的矩阵
127  * @param mat 另一个矩阵对象
128  */
129 void fmatMul(Matrix mat)
130 {
131     float **res = new float *[this->row];
132     for (int j = 0; j < this->row; j++)
133     {
134         res[j] = new float[mat.col];
135     }
136     //Initialization
137     for (int i = 0; i < this->row; i++)
138     {
139         for (int j = 0; j < mat.col; j++)
140         {
141             res[i][j] = 0.0f;
142         }
143     }
144
145     time_t start, end;
146     start = clock();
147
148     //将矩阵按照 i->j->k 的顺序进行乘法运算
149     for (int i = 0; i < this->row; i++)
150     {

```

```

151         for (int j = 0; j < mat.col; j++)
152         {
153             float c_i_j = 0.0f;
154             for (int k = 0; k < this->col; k++)
155             {
156                 c_i_j += this->fmat[i][k] * mat.fmat[k][j];
157             }
158             res[i][j] = c_i_j;
159         }
160     }
161
162     end = clock();
163     printf("%dx%d的矩阵与%dxd%d的矩阵以float,按ijk的顺序相乘得出结果共耗时:
164     %fs\n", this->row, this->col, mat.row, mat.col, (double(end - start) /
165     CLOCKS_PER_SEC));
166
167     //将矩阵中的每个元素都重置,为下次运算做准备
168     for (int i = 0; i < this->row; i++)
169     {
170         for (int j = 0; j < mat.col; j++)
171         {
172             res[i][j] = 0.0f;
173         }
174     }
175
176     start = clock();
177
178     //将矩阵按照 i->k->j 的顺序进行乘法运算
179     for (int i = 0; i < this->row; i++)
180     {
181         for (int k = 0; k < this->col; k++)
182         {
183             float temp = this->fmat[i][k];
184             for (int j = 0; j < mat.col; j++)
185             {
186                 res[i][j] += temp * mat.fmat[k][j];
187             }
188         }
189     }
190
191     end = clock();
192     printf("%dx%d的矩阵与%dxd%d的矩阵以float,按ikj的顺序相乘得出结果共耗时:
193     %fs\n", this->row, this->col, mat.row, mat.col, (double(end - start) /
194     CLOCKS_PER_SEC));
195
196     //将结果写入txt文件中
197     string ofile_name = "out-float-" + to_string(this->row) + "x" +
198     to_string(mat.col) + ".txt";
199     ofstream ofs;
200     ofs.open(ofile_name, ios::out);
201     for (int i = 0; i < this->row; i++)
202     {
203         for (int j = 0; j < mat.col; j++)
204         {
205             ofs << res[i][j] << " ";
206         }
207         ofs << endl;
208     }

```

```

204
205 //释放内存
206 for (int i = 0; i < this->row; i++)
207 {
208     delete[] res[i];
209 }
210 delete[] res;
211 }
212
213 /**
214  * @brief double矩阵的乘法，乘法顺序为当前对象的矩阵乘以参数的矩阵
215  * @param mat 另一个矩阵对象
216  */
217 void dmatMul(Matrix mat)
218 {
219     double **res = new double *[this->row];
220     for (int j = 0; j < this->row; j++)
221     {
222         res[j] = new double[mat.col];
223     }
224     //Initialization
225     for (int i = 0; i < this->row; i++)
226     {
227         for (int j = 0; j < mat.col; j++)
228         {
229             res[i][j] = 0.0;
230         }
231     }
232
233     time_t start, end;
234     start = clock();
235     for (int i = 0; i < this->row; i++)
236     {
237         for (int j = 0; j < mat.col; j++)
238         {
239             double c_i_j = 0.0;
240             for (int k = 0; k < this->col; k++)
241             {
242                 c_i_j += this->dmat[i][k] * mat.dmat[k][j];
243             }
244             res[i][j] = c_i_j;
245         }
246     }
247     end = clock();
248     printf("%dx%d的矩阵与%dx%d的矩阵以double,按ijk的顺序相乘得出结果共耗时:
249 %fs\n", this->row, this->col, mat.row, mat.col, (double)(end - start) /
250     CLOCKS_PER_SEC));
251
252     for (int i = 0; i < this->row; i++)
253     {
254         for (int j = 0; j < mat.col; j++)
255         {
256             res[i][j] = 0.0;
257         }
258     }
259
260     start = clock();
261     for (int i = 0; i < this->row; i++)

```



```

260     {
261         for (int k = 0; k < this->col; k++)
262         {
263             double temp = this->dmat[i][k];
264             for (int j = 0; j < mat.col; j++)
265             {
266                 res[i][j] += temp * mat.dmat[k][j];
267             }
268         }
269     }
270     end = clock();
271     printf("%dx%d的矩阵与%dx%d的矩阵以double,按ikj的顺序相乘得出结果共耗时:
%s\n", this->row, this->col, mat.row, mat.col, (double(end - start) /
CLOCKS_PER_SEC));
272
273     string ofile_name = "out-double-" + to_string(this->row) + "x" +
to_string(mat.col) + ".txt";
274     ofstream ofs;
275     ofs.open(ofile_name, ios::out);
276     for (int i = 0; i < this->row; i++)
277     {
278         for (int j = 0; j < mat.col; j++)
279         {
280             ofs << res[i][j] << " ";
281         }
282         ofs << endl;
283     }
284
285     for (int i = 0; i < this->row; i++)
286     {
287         delete[] res[i];
288     }
289     delete[] res;
290 }
291
292 /**
293  * @brief 将矩阵打印到命令行上
294  */
295 void printMat()
296 {
297     for (int i = 0; i < row; i++)
298     {
299         for (int j = 0; j < col; j++)
300         {
301             printf("%5.1f ", dmat[i][j]);
302         }
303         cout << endl;
304     }
305 }
306 };
307
308 int main(int argc, char **argv)
309 {
310     if (argc < 3)
311     {
312         cout << "文件的数量不够。退出!" << endl;
313         exit(100);
314     }

```

```
315
316     string ifile1_name = argv[1];
317     string ifile2_name = argv[2];
318
319     Matrix mat_A(ifile1_name);
320     Matrix mat_B(ifile2_name);
321     if (mat_A.getCol() != mat_B.getRow())
322     {
323         cout << "矩阵A的列数不等于矩阵B的行数，无法做矩阵乘法。退出！" << endl;
324         exit(100);
325     }
326
327     mat_A.prepareMat();
328     mat_B.prepareMat();
329     mat_A.fmatMul(mat_B);
330     mat_A.dmatMul(mat_B);
331     return 0;
332 }
```