CS205 C/C++ Program Design Project 2

Name: 眭和(SUI He)

SID: 12012929

CS205 C/C++ Program Design Project 2

- Part 1 - Analysis
  - Requirements
  - Load and save the matrixes
  - Matrix Multiplication
    - Start with the conventional approach(version 1)
    - Speed up the Calculating
  - Timing the program
  - The comparison between `float` and `double`
    - Time Analysis
    - Precision Analysis
- Part 2 - Code
- Part 3 - Result & Verification
- Part 4 - Difficulties & Solutions
  - Use transpose matrix (Version 2)
  - Change Loop Order (Version 3)
  - T-n Analysis

Part 1 - Analysis

Requirements

1. When you run the program as follows, it will output the result into an output file such as `out32.txt`.

   `$./matmul mat-A-32.txt mat-B-32.txt out32.txt`

   `$./matmul mat-A-256.txt mat-B-256.txt out256.txt`

   `$./matmul mat-A-2048.txt mat-B-2048.txt out2048.txt`

2. Please implement the matrix multiplication in `float` and `double` separately, and compare the time consumed and the results. Give detailed analysis on the speed and accuracy.
3. You can try to improve the speed, and introduce the methods in the report.

Load and save the matrixes

In order to load the matrixes from `.txt` file, I need to `#include <fstream>` and use `ifstream` to read the data from file and use `ofstream` to save data into file.

```
string inFileName1, inFileName2, outFileName;

inFileName1 = argv[1];
inFileName2 = argv[2];
outFileName = argv[3];
ifstream inFile1(inFileName1);
ifstream inFile2(inFileName2);
ofstream outFile(outFileName);
```

The program read the name from command line and create corresponding ifstream or ofstream. And then get the size of the matrix from the file name and check whether it is same as the real size of the data matrix.

In order to have enough space to save the `2048*2048` matrix, I use `new` statement to create arrays so that it is stored in heap. And then we can load the data into the array.

```
double **matrixA = new double *[size];
    double **matrixB = new double *[size];
    double **result = new double *[size];

    for (int i = 0; i < size; ++i)
    {
        matrixA[i] = new double[size];
        matrixB[i] = new double[size];
        result[i] = new double[size];
    }

    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
        {
            inFile1 >> matrixA[i][j];
            inFile2 >> matrixB[i][j];
        }
```

Matrix Multiplication

For the multiplication of a N rows and M columns matrix A and a M rows and P columns matrix B, every element of the $N \times P$ result matrix C as follows:

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

Start with the conventional approach(version 1)

According to the definition of the matrix multiplication, we can easily finish the task by the conventional solution. This method imitate human to calculate matrix multiplication. It use a double loop to iterate every position of the result matrix and use another loop to calculate value of each position by the formula above.

```
void matrixMultiplication(double **matrixA, double **matrixB, double
**result, int size)
{
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
            for (int k = 0; k < size; ++k)
                result[i][j] += matrixA[i][k] * matrixB[k][j];
}
```

Speed up the Calculating

This part can be seen in Part 4.

Timing the program

In this project, I think I should mainly focus on the efficiency on the matrix multiplication. The time spend on input and output are not the main part I should consider about. Therefore, I use `ctime` to record the time before multiplication and after multiplication to calculate the time cost in this process and compare the time consumed between using `float` and `double`.

```
startTime = clock();
matrixMultiplication(matrixA, matrixB, result, size);
endTime = clock();
cout << "The calculate time is: " << (double)(endTime - startTime) /
CLOCKS_PER_SEC << "s" << endl;
```

The comparison between `float` and `double`

The program runs under WSL, and my CPU is `Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz`. All testing are be done under this environment.

Firstly I use `float` and `double` separately to save and calculate the matrix. To insure fair in timing, the output significant bits are the same. Each version will run three times and record the calculating time.

`float`	32×32	256×256	2048×2048
1	0.000123s	0.075903s	98.152s
2	0.000125s	0.073795s	95.3507s
3	0.000122s	0.074098s	96.5678s
average	0.000123s	0.074599s	96.6902s

`double`	32×32	256×256	2048×2048
1	0.000123s	0.083332s	127.222s
2	0.000122s	0.082607s	126.823s
3	0.000123s	0.084471s	121.182s
average	0.000123s	0.08347s	125.076s

Time Analysis

From the result above, we can easily find that when the matrix size is small, the speed of `float` and `double` are nearly the same. But when the matrix size become larger and larger, we can obviously find the difference between `float` and `double`. And when the size become 2048×2048 , the difference of the time is nearly 30s. Therefore, as the size grow bigger, using `float` is quicker than `double`.

From my point of view, this is because `float` is float-point number with single precision, and `double` is double precision floating point number. Which means `float` use half of the space of `double`. Therefore, when calculating matrix with `float`, `float` use less space to save data, which means it has less digit to calculate, obviously it will quicker than `double`.

However, I also find on the internet that whether `float` or `double` is quicker is depends on hardware. If `CPU` use `FPU` to calculate float-point number. When calculating `float`, it need to convert into `double` first and calculate and then convert back to `float`. In this case, without doubt `float` will be slower than `double`. But if `CPU` use `SSE/AVX` to calculate float-point number, it needn't converting. And It can handle four `float` or two `double` at a time. Therefore, in this case, `float` is quicker than `double`. All in all, different device has different situations, and it mainly depends on the hardware.

Precision Analysis

As we all know, `float` use less space than `double`. Mostly, `float` use 4 `byte` and `double` use 8 `byte`. This means `float` has less precision than `double`. And `float` has less significant digit than `double`. `float` can promise 6 digits and `double` can promise 15 digits. Therefore, I use `setprecision(20)` to compare their precision. I use size 2048 matrix to compare the difference in this part. For lack of space, I only show the first 10 numbers in line 1 for comparison:

5252342.2900000093505 5053432.2499999841675 5087801.5800000010058
5006596.2999999998137 5185736.2400000002235 5085108.7700000060722
5122403.839999997057 5107456.2399999909103 5100696.6700000036508
4986783.2099999869242

Result of using double

5252342 5053434 5087805 5006593.5 5185730.5 5085111 5122402.5 5107453.5 5100698
4986780

Result of using float

Comparing the result of size 2048 matrix, we can easily find the difference between them. We can find that `float` can only promise the value of first 6 digits. And after that it cannot promise its correctness. And `double` can only promise the value of first 15 digits. And after that, it is uncertain. Therefore, we can easily know that `double` has more precision than `double`. And

when the size become larger, the difference is more obvious to observe. Because it adds up the difference, and farther away from the correct value. `double` has higher precision than `float`.

Part 2 - Code

> The code below only shows the version of `double`. The version of `float` only need to change all the `double` into `float`

```
#include <iostream>
#include <cmath>
#include <string>
#include <fstream>
#include <ctime>
#include <iomanip>

using namespace std;
int matrixSize(string);
void matrixMultiplication(float **, float **, float **, int);

int main(int argc, char **argv)
{
    string inFileName1, inFileName2, outFileName;
    int size;
    clock_t startTime, endTime;
    if (argc == 4)
    {
        inFileName1 = argv[1];
        inFileName2 = argv[2];
        outFileName = argv[3];
    }
    else
    {
        cerr << "Missing file!" << endl;
        return -1;
    }
    ifstream inFile1(inFileName1);
    ifstream inFile2(inFileName2);
    if (!inFile1.is_open() || !inFile2.is_open())
    {
        cerr << "Can not open the file!" << endl;
        return -1;
    }
    int size1 = matrixSize(inFileName1);
    int size2 = matrixSize(inFileName2);
    if (size1 != size2)
    {
        cerr << "Two matrix are not the same size!" << endl;
        return -1;
    }

    size = size1;
    float **matrixA = new float *[size];
    float **matrixB = new float *[size];
    float **result = new float *[size];

```

```
    for (int i = 0; i < size; ++i)
    {
        matrixA[i] = new float[size];
        matrixB[i] = new float[size];
        result[i] = new float[size];
    }

    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
        {
            inFile1 >> matrixA[i][j];
            inFile2 >> matrixB[i][j];
        }
    cout << "Load successfully!" << endl;
    inFile1.close();
    inFile2.close();

    startTime = clock();
    matrixMultiplication(matrixA, matrixB, result, size);
    endTime = clock();
    cout << "The calculate time is: " << (double)(endTime - startTime) /
CLOCKS_PER_SEC << "s" << endl;

    ofstream outFile(outFileName);
    // outFile << setprecision(20);

    for (int i = 0; i < size; ++i)
    {
        for (int j = 0; j < size; ++j)
        {
            outFile << result[i][j] << " ";
            if (result[i][j] <= 0)
                cout << i << " " << j << endl;
        }
        outFile << endl;
    }
    outFile.close();
    cout << "Write successfully!" << endl;
    return 0;
}

int matrixSize(string name)
{
    int startIndex = -1;
    int endIndex = -1;
    for (int counter = 0; counter < name.size(); ++counter)
    {
        if (name[counter] >= 48 && name[counter] <= 57 && startIndex == -1)
            startIndex = counter;
        else if ((name[counter] < 48 || name[counter] > 57) && startIndex
!= -1)
        {
            endIndex = counter - 1;
            break;
        }
    }
    int size = 0;
    for (int counter = startIndex; counter <= endIndex; counter++)
```

```
        size += (name[counter] - 48) * (int)pow(10, endIndex - counter);
    return size;
}

// Version 1
// void matrixMultiplication(double **matrixA, double **matrixB, double
**result, int size)
// {
//     for (int i = 0; i < size; ++i)
//         for (int j = 0; j < size; ++j)
//             for (int k = 0; k < size; ++k)
//                 result[i][j] += matrixA[i][k] * matrixB[k][j];
// }

// Version 2
// void matrixMultiplication(double **matrixA, double **matrixBT, double
**result, int size)
// {
//     for (int i = 0; i < size; i++)
//         for (int j = 0; j < size; j++)
//             for (int k = 0; k < size; k++)
//                 result[i][j] += matrixA[i][k] * matrixBT[j][k];
// }

// Version 3
void matrixMultiplication(float **matrixA, float **matrixB, float **result,
int size)
{
    for (int i = 0; i < size; ++i)
        for (int k = 0; k < size; ++k)
        {
            float value = matrixA[i][k];
            for (int j = 0; j < size; ++j)
                result[i][j] += value * matrixB[k][j];
        }
}
```

Part 3 - Result & Verification

In order to verify the result, we only need to ensure the method of calculating the matrix multiplication is correct. Therefore, I use `matlab` to calculate the 32×32 matrix to check whether the result are nearly the same (the less significant digits may be different due to precision problem).

> Due to layout limitations, the data in the first line may be squeezed into the second line

> result =
>
> **1.0e+05** *
>
> 列 1 至 12
>
> 0.6028 0.6733 0.5663 0.6081 0.7337 0.6670 0.7198 0.7143 0.5737 0.7711
> 0.7895 0.7674
> 0.6279 0.7780 0.7643 0.7372 0.9096 0.7413 0.9606 0.7347 0.6672 0.9285
> 0.8563 0.8448

0.5834 0.6766 0.5733 0.5746 0.7832 0.5800 0.7633 0.6231 0.6335 0.7910
0.7550 0.7124
0.6512 0.7344 0.7033 0.6117 0.8107 0.7230 0.7714 0.6736 0.6949 0.9190
0.8123 0.7558
0.7330 0.8833 0.7680 0.8112 0.9537 0.8573 0.9695 0.8325 0.7636 0.9625
0.9657 0.8787
0.7063 0.7484 0.6783 0.7183 0.8942 0.7755 0.8657 0.7799 0.6460 0.8396
0.8024 0.8466
0.7004 0.8650 0.7413 0.7203 1.0179 0.8103 0.9079 0.8254 0.7725 0.9995
0.9274 0.8322
0.7192 0.8066 0.7584 0.6432 0.9050 0.8024 0.8582 0.7194 0.7494 1.0093
0.8248 0.7361
0.5635 0.5871 0.6012 0.5830 0.7425 0.6083 0.7275 0.6670 0.5926 0.6966
0.7070 0.6867
0.7665 0.8539 0.7411 0.7030 0.8473 0.7796 0.8450 0.8048 0.7037 0.8970
0.9229 0.8499
0.6266 0.6815 0.7291 0.5832 0.7831 0.7680 0.8108 0.7867 0.6012 0.8132
0.7515 0.8004
0.5746 0.6645 0.5835 0.5358 0.7032 0.6714 0.6379 0.7318 0.5704 0.7650
0.7404 0.7112
0.6385 0.7768 0.6977 0.5683 0.8222 0.8182 0.7656 0.7716 0.6706 0.8415
0.8189 0.8279
0.5270 0.6367 0.6350 0.4678 0.7328 0.6972 0.6537 0.5394 0.6099 0.8226
0.6537 0.7638
0.7419 0.9575 0.7093 0.8056 0.9899 0.8914 0.9489 0.8464 0.7300 0.9420
0.9187 0.9295
0.7218 0.8133 0.8157 0.6449 0.9264 0.8779 0.9262 0.8230 0.7383 0.9579
0.8098 0.9064
0.7382 0.7552 0.7649 0.6236 0.8569 0.7802 0.8441 0.7360 0.7543 0.8885
0.8662 0.8223
0.8323 0.7538 0.8917 0.6484 0.9378 0.8891 0.9482 0.8436 0.7952 1.0416
0.8692 0.8367
0.6834 0.8113 0.7175 0.5758 0.8052 0.7844 0.7439 0.7211 0.7010 0.8921
0.8734 0.7959
0.6303 0.7516 0.7070 0.5676 0.8302 0.8143 0.6854 0.7920 0.6404 0.8198
0.7951 0.7025
0.7824 0.9584 0.8661 0.8158 0.9896 0.9391 0.9459 0.8843 0.8738 1.1023
1.0689 1.0163
0.7904 0.9081 0.7927 0.7944 0.9536 0.8315 0.8919 0.8813 0.8150 1.0062
1.0764 0.8695
0.7338 0.8394 0.8296 0.7082 0.9365 0.8741 0.8713 0.8105 0.7705 1.1040
0.9308 0.8802
0.6856 0.8632 0.6605 0.6546 0.8148 0.7881 0.8149 0.7610 0.7032 0.8922
0.8924 0.8897
0.8275 0.9422 0.9017 0.7904 0.9148 0.9343 0.9954 0.9200 0.7179 1.0455
0.9675 0.9274
0.7149 0.8354 0.6686 0.7958 0.8476 0.7090 0.9298 0.9269 0.6875 0.8824
0.9047 0.8308
0.7108 0.7858 0.7076 0.6405 0.8481 0.6755 0.8192 0.6678 0.6938 0.8374
0.8928 0.7590
0.6238 0.7000 0.6806 0.5593 0.7903 0.7330 0.8674 0.7684 0.6564 0.8563
0.8389 0.7346

0.7533 0.8181 0.8389 0.6120 0.8710 0.8297 0.8700 0.7436 0.6915 1.0003
0.9032 0.8336
0.7530 1.0285 0.8062 0.7775 1.0174 1.0022 0.9231 0.9518 0.8663 1.1016
1.0183 0.9824
0.8398 1.0643 0.8463 0.9497 1.1335 0.9802 1.0728 1.0237 0.9246 1.0791
1.1061 1.0178
0.7224 0.7296 0.6779 0.6458 0.8008 0.7429 0.7781 0.8093 0.6831 0.8365
0.8418 0.7588

列 13 至 24

0.8260 0.8128 0.6969 0.6321 0.6408 0.6722 0.8167 0.5944 0.6611 0.6652
0.7653 0.7608
0.9364 0.9294 0.8014 0.6839 0.6994 0.7873 0.8562 0.7171 0.7273 0.7965
0.9132 0.8325
0.7795 0.7463 0.5910 0.6581 0.5760 0.6302 0.7840 0.5905 0.6239 0.5615
0.6888 0.6367
0.9220 0.8439 0.6660 0.7676 0.6069 0.7472 0.8470 0.6293 0.7190 0.7134
0.8024 0.7853
0.9489 0.9684 0.8268 0.8119 0.7500 0.9753 0.9745 0.8098 0.7235 0.7313
0.8602 0.8547
0.8567 0.8613 0.8491 0.8030 0.7157 0.7963 0.8566 0.7627 0.6996 0.6904
0.7690 0.7365
0.9603 0.8770 0.8160 0.7354 0.8255 0.7903 0.9167 0.7833 0.7564 0.7995
0.9087 0.7826
0.9446 0.8963 0.7554 0.7430 0.6726 0.6786 0.8820 0.7080 0.6825 0.7537
0.8394 0.7779
0.7045 0.7166 0.6679 0.6324 0.5793 0.6774 0.7261 0.6638 0.6214 0.6140
0.7534 0.6728
0.9156 0.9305 0.8112 0.8606 0.6924 0.8947 0.9369 0.7197 0.7463 0.6274
0.7689 0.8357
0.7523 0.8289 0.8079 0.6700 0.6927 0.6949 0.7600 0.6213 0.7169 0.6510
0.7871 0.7327
0.7841 0.7850 0.7326 0.5661 0.6195 0.5714 0.7492 0.5664 0.6920 0.6541
0.7312 0.6812
0.8630 0.8962 0.8020 0.6736 0.8155 0.7376 0.8677 0.6981 0.7055 0.6561
0.7792 0.7703
0.7574 0.7212 0.6139 0.6238 0.5697 0.6503 0.7429 0.5052 0.5906 0.5480
0.6992 0.7098
1.0293 0.9675 0.8809 0.9299 0.8423 0.8825 0.9394 0.8094 0.7740 0.7966
0.8826 0.9228
0.9088 0.9009 0.8251 0.7601 0.7535 0.8874 0.8793 0.7091 0.8103 0.6738
0.8537 0.8259
0.8883 0.8644 0.8050 0.7821 0.6551 0.7435 0.8747 0.6721 0.7150 0.7276
0.9279 0.8738
0.9347 0.9564 0.8484 0.7885 0.7611 0.9164 0.9205 0.7382 0.8255 0.7056
0.8586 0.8326
0.8914 0.9210 0.7835 0.8053 0.7219 0.7710 0.9255 0.7412 0.7001 0.7536
0.7827 0.8514
0.8445 0.7859 0.8031 0.6926 0.7085 0.6495 0.7315 0.6241 0.6823 0.6727
0.8028 0.7948
1.0031 0.9898 0.8706 0.8754 0.7411 0.8936 0.9801 0.7294 0.8596 0.8064
0.9985 0.8463

1.0319 1.0121 0.9009 0.8188 0.8168 0.8151 1.0095 0.7894 0.7735 0.8469
0.9267 0.8194
0.9756 0.9378 0.8434 0.7277 0.7452 0.7533 0.9344 0.6995 0.7813 0.8078
0.8851 0.8227
0.8858 0.9292 0.7510 0.7296 0.6970 0.8080 0.9418 0.7138 0.7676 0.7628
0.8734 0.8493
1.0212 1.0828 0.9656 0.8176 0.7691 0.9138 0.9934 0.7825 0.7709 0.9084
0.9725 0.9343
0.8999 0.9851 0.9243 0.7285 0.7275 0.7541 0.9268 0.7959 0.7466 0.7659
0.7986 0.7949
0.8891 0.8583 0.7717 0.8031 0.6226 0.6987 0.8623 0.6751 0.6787 0.7102
0.8171 0.8020
0.7424 0.8718 0.7513 0.7317 0.6613 0.7000 0.8423 0.6415 0.7105 0.6822
0.7950 0.7751
0.9778 1.0377 0.8046 0.7216 0.6766 0.8138 0.9396 0.7031 0.7307 0.8599
0.9380 0.8178
1.0130 1.0452 0.8603 0.8889 0.8786 0.8648 1.0289 0.8584 0.8824 0.8341
0.9611 0.9538
1.1232 1.0930 0.9834 0.9672 0.9423 0.9142 1.0635 0.9226 0.9441 0.8424
1.0941 1.0583
0.8128 0.8749 0.8358 0.6912 0.6994 0.7161 0.8813 0.6763 0.6474 0.6162
0.7358 0.7116

列 25 至 32

0.8606 0.6470 0.6822 0.8274 0.7586 0.5178 0.5203 0.7543
0.9664 0.9273 0.8027 0.9424 0.7500 0.6252 0.6616 0.7680
0.7748 0.7269 0.5740 0.8327 0.7009 0.6251 0.4495 0.6192
0.8722 0.8000 0.6887 0.9560 0.6822 0.6424 0.6031 0.7550
1.0884 0.8839 0.8450 1.0450 0.9708 0.8126 0.6738 0.8651
0.9914 0.8139 0.6786 0.9110 0.7607 0.6567 0.6015 0.8118
0.9446 0.8881 0.7881 0.9960 0.8274 0.6522 0.6132 0.8142
0.9214 0.7958 0.7196 0.8717 0.6785 0.6499 0.6786 0.7252
0.7932 0.6322 0.6237 0.8060 0.6704 0.5316 0.4663 0.6087
1.0532 0.7579 0.7728 1.0177 0.9228 0.7819 0.6462 0.7748
0.9564 0.7403 0.7513 0.7737 0.7447 0.5871 0.6477 0.7464
0.7704 0.5805 0.5438 0.8005 0.6896 0.4670 0.5148 0.6523
0.9357 0.6752 0.7628 0.8465 0.8307 0.5921 0.6590 0.7972
0.7682 0.6639 0.6520 0.8198 0.7093 0.5124 0.4869 0.6231
1.1434 0.9481 0.8862 1.0355 0.8752 0.7293 0.7184 0.9851
0.9683 0.8680 0.7702 0.8778 0.8567 0.7517 0.6483 0.8822
0.9860 0.7144 0.7334 0.9824 0.8003 0.6433 0.6246 0.7289
1.0853 0.7747 0.8021 0.9371 0.8489 0.7711 0.6941 0.8083
0.9383 0.7669 0.8243 0.9923 0.8249 0.6570 0.5908 0.7186
0.8891 0.6626 0.7083 0.7463 0.7566 0.6089 0.5924 0.7223
1.0976 0.8977 0.8708 1.1920 0.9624 0.7828 0.7347 0.8292
1.0352 0.8761 0.8907 1.0319 0.8609 0.7464 0.6531 0.8514
0.9701 0.9190 0.8508 1.0277 0.8249 0.6493 0.6086 0.6792
0.9371 0.8118 0.7438 1.0535 0.8888 0.6446 0.5495 0.7705
1.1495 0.9183 0.9590 1.0624 0.9096 0.7135 0.7697 0.8944
1.0479 0.8149 0.7977 0.8846 0.8412 0.6724 0.5784 0.7335
0.9342 0.7783 0.7235 0.9070 0.7559 0.7215 0.6063 0.7601
0.9812 0.7255 0.6915 0.7799 0.7860 0.6565 0.6306 0.7173

0.9862	0.7388	0.7805	1.0205	0.7859	0.6460	0.7580	0.7581
1.1263	1.0175	0.9633	1.1599	0.9976	0.7849	0.7208	0.8847
1.2808	1.0535	0.9490	1.1388	1.0325	0.8711	0.7521	1.0299
0.9384	0.6620	0.7742	0.8504	0.8185	0.6412	0.5143	0.6521

Compared with the result from my program (Attached in the compressed file), they are the same result. It proves that there is no problem in the implementation of my matrix multiplication. Therefore, the other two result are also correct under my program.

> **All the result `.txt` file are attached in the compressed file. I will only show program performance below.**

Test case 1: `float` 32×32 matrix

```
sui@LAPTOP-2J3CU669:/mnt/d/SUSTech/课程/CS205 CC++程序设计/Project2$ ./matmul mat-A-32.txt mat-B-32.txt out32.txt
Load successfully!
The calculate time is: 9.6e-05s
Write successfully!
```

Test case 2: `float` 256×256 matrix

```
sui@LAPTOP-2J3CU669:/mnt/d/SUSTech/课程/CS205 CC++程序设计/Project2$ ./matmul mat-A-256.txt mat-B-256.txt out256.txt
Load successfully!
The calculate time is: 0.052883s
Write successfully!
```

Test case 3: `float` 2048×2048 matrix

```
sui@LAPTOP-2J3CU669:/mnt/d/SUSTech/课程/CS205 CC++程序设计/Project2$ ./matmul mat-A-2048.txt mat-B-2048.txt out2048.txt
Load successfully!
The calculate time is: 26.1729s
Write successfully!
```

Test case 4: `double` 32×32 matrix

```
sui@LAPTOP-2J3CU669:/mnt/d/SUSTech/课程/CS205 CC++程序设计/Project2$ ./matmul mat-A-32.txt mat-B-32.txt out32.txt
Load successfully!
The calculate time is: 9.8e-05s
Write successfully!
```

Test case 5: `double` 256×256 matrix

```
sui@LAPTOP-2J3CU669:/mnt/d/SUSTech/课程/CS205 CC++程序设计/Project2$ ./matmul mat-A-256.txt mat-B-256.txt out256.txt
Load successfully!
The calculate time is: 0.054651s
Write successfully!
```

Test case 6: `double` 2048×2048 matrix

```
sui@LAPTOP-2J3CU669:/mnt/d/SUSTech/课程/CS205 CC++程序设计/Project2$ ./matmul mat-A-2048.txt mat-B-2048.txt out2048.txt
Load successfully!
The calculate time is: 28.4901s
Write successfully!
```

Part 4 - Difficulties & Solutions

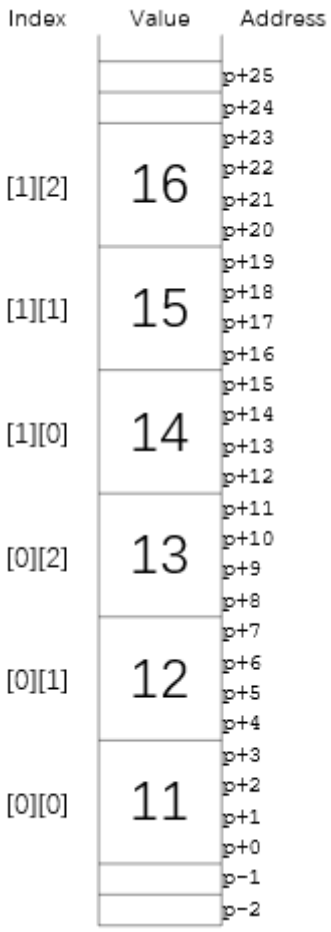
> We notice that the speed of calculating the matrix is slow, especially 2048×2048 matrix. Therefore, in this part, I will improve the program to speed up calculating. For conveniently, I will only use `double` to calculate in this part.

Use transpose matrix (Version 2)

The memory storage of a multidimensional array is still linear. For a two-dimensional array, the memory storage of its second dimension is continuous. The first dimension is not continuous. According to the multiplication of matrix, each element in the result matrix is the dot product of the first matrix's row vector and the second matrix's column vector. In my version 1, I access the

second matrix through its column. However, the adjacent element in columns is not continuous in the array. This may result in extra time in memory addressing.

> The following figure is from power point of CS205, which can clearly show the memory storage of the multidimensional array.



In order to make the second matrix can continuously access the value from array, we can transpose the second matrix. Then each element in the result matrix is the dot product of the two row matrix. And then we can improve our program as below:

> The second matrix is already been transposed

```
void matrixMultiplication(double **matrixA, double **matrixBT, double
**result, int size)
{
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
            for (int k = 0; k < size; k++)
                result[i][j] += matrixA[i][k] * matrixBT[j][k];
}
```

Average Time	32×32	256×256	2048×2048
Version 1	0.000123s	0.08347s	125.076s
Version 2	0.000123s	0.06612s	32.9104s

Above is the time cost comparation, we can see that the program speed up a lot, especially the 2048×2048. This shows that continuously access the array may save a lot of time.

Change Loop Order (Version 3)

There are also some ways to reduce the time spent on memory access. Let's look back on the version 1.

```
void matrixMultiplication(double **matrixA, double **matrixB, double
**result, int size)
{
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
            for (int k = 0; k < size; ++k)
                result[i][j] += matrixA[i][k] * matrixB[k][j];
}
```

We already known that repeated access to the same element in an array leads to many unnecessary times. And we can also find that `matrixA[i][k]` has been repeatedly access in second loop and the change of `j` won't change its value. Therefore, we can consider to swap the second loop and the third loop to minimize memory access. We can improve the program as below:

```
void matrixMultiplication(double **matrixA, double **matrixB, double
**result, int size)
{
    for (int i = 0; i < size; ++i)
        for (int k = 0; k < size; ++k)
        {
            double value = matrixA[i][k];
            for (int j = 0; j < size; ++j)
                result[i][j] += value * matrixB[k][j];
        }
}
```

Average Time	32×32	256×256	2048×2048
Version 1	0.000123s	0.08347s	125.076s
Version 2	0.000123s	0.06612s	32.9104s
Version 3	9.8e-05s	0.05445s	28.5711s

From the above result, we can see the speed of the version 3 is much quicker, especially the large size matrix.

T-n Analysis

It is obvious the time complexity is $O(n) = n^3$ because of the three layers loop. And then we can verify it is the trend of n^3.

We use the data of version 3 to verify it:

$$\frac{256^3}{32^3} = 512 \approx 556 \approx \frac{0.05445}{9.8 \times 10^{-5}}$$

$$\frac{2048^3}{256^3} = 512 \approx 524 \approx \frac{28.5711}{0.05445}$$

Therefore, we can know the relation between T and n is approximately conform $T(n) \propto n^3$. There may be some difference caused by the memory access when the `n` is small. But when `n` become large enough, it will more conform to $T(n) \propto n^3$.