C/C++ Pro 2 – 计算矩阵乘法

姓名：林洁芳

学号：12011543

第一部分 问题思路和分析

1 问题分析

① 矩阵乘法实现

依据题目要求，需要实现三种不同规模的浮点数矩阵乘法，首先考虑将矩阵读入后利用矩阵乘法的特性，采用n^3的复杂度，将矩阵乘法实现后输出。在小规模的矩阵中，普通的n^3（指IJK）还没有表现出劣势，但在大规模的矩阵中，就会表现出时间耗费长的劣势，于是考虑从降低算法复杂度（strassen 算法）和采用硬件优化（IKJ、多并行、循环展开、SIMD）这两个大方面进行分析；

② 对比float和double

单精度浮点数占32字节和双精度浮点数占64字节，在计算精度上也有差距，且不同的计算顺序也会造成不同精度的丢失（尤其开启多并行之后可能会导致结果相差很大）；且计算速度也不完全是float比double快。

2 问题思路

先完成IJK矩阵乘法计算，并能正确输出结果，而后再去尝试strassen 算法、硬件优化（KIJ、IKJ等）、并行化、CUDA优化（展开循环）等，在此过程中遇到了很多问题，并查找了很多资料，**具体会呈现在"难点和方案"中**；

此外，本次project使用到的libraries有：

```
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include <sstream>
```

第二部分 源码

此处展示一维数组版本代码，另有二维数组版本在zip中另附；

1 主方法体

```
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include <sstream>

int row1 = 0;
int row2 = 0;
int column1 = 0;
int column2 = 0;
```

```cpp
using namespace std;

#pragma GCC optimize(3)

int main(int argc, char **argv) {
    string inp01, inp02, outfileF_IKJ, outfileD_IKJ, outfileF_N3, outfileD_N3,
outfileF_N281, outfileD_N281;
    if (argc > 7) {
        inp01 = argv[1];
        inp02 = argv[2];
        outfileF_IKJ = argv[3];
        outfileD_IKJ = argv[4];
        outfileF_N3 = argv[5];
        outfileD_N3 = argv[6];
        outfileF_N281 = argv[7];
        outfileD_N281 = argv[8];
    } else {
        inp01 = argv[1];
        inp02 = argv[2];
        outfileF_IKJ = argv[3];
        outfileD_IKJ = argv[4];
        outfileF_N3 = argv[5];
        outfileD_N3 = argv[6];
    }
    row1 = getRow(inp01);
    column1 = getColumn(inp01);
    row2 = column1;
    column2 = getColumn(inp02);


    double *aD = new double[row1 * column1];
    float *aF = new float[row1 * column1];
    double *bD = new double[row2 * column2];
    float *bF = new float[row2 * column2];

    // 对第一个矩阵的处理
    ifstream infile1, infile2;
    infile1.open(inp01);
    string temp;
    for (int i = 0; i < row1; ++i) {
        for (int j = 0; j < column1; ++j) {
            infile1 >> temp;
            aF[i * column1 + j] = stof(temp);
            aD[i * column1 + j] = stod(temp);
        }
    }
    // 对第二个矩阵的处理
    infile2.open(inp02);
    for (int i = 0; i < row2; ++i) {
        for (int j = 0; j < column2; ++j) {
            infile2 >> temp;
            bF[i * column1 + j] = stof(temp);
            bD[i * column1 + j] = stod(temp);
        }
    }
    float *resultF_IKJ = new float[row1 * column2];
    double *resultD_IKJ = new double[row1 * column2];
    float *resultF_N3 = new float[row1 * column2];
    double *resultD_N3 = new double[row1 * column2];
    float *resultF_N281 = new float[row1 * column2];
    double *resultD_N281 = new double[row1 * column2];

    for (int i = 0; i < row1; ++i) {
        for (int j = 0; j < column2; ++j) {
            resultF_IKJ[i * column2 + j] = 0.0f;
            resultF_N281[i * column2 + j] = 0.0f;
            resultF_N3[i * column2 + j] = 0.0f;
            resultD_IKJ[i * column2 + j] = 0.0;
            resultD_N3[i * column2 + j] = 0.0;
            resultD_N281[i * column2 + j] = 0.0;
        }
    }

    auto start = chrono::steady_clock::now();
    IKJF((float *) aF, (float *) bF, (float *) resultF_IKJ);
    auto end1 = std::chrono::steady_clock::now();
    chrono::duration<double, micro> elapsed = end1 - start;
    cout << "Use float to calculate and the complexity is IKJ: " <<
elapsed.count() / 1000 << " ms"
         << endl;

    IKJD((double *) aD, (double *) bD, (double *) resultD_IKJ);
    auto end2 = std::chrono::steady_clock::now();
    elapsed = end2 - end1;
    cout << "Use double to calculate and the complexity is IKJ: " <<
elapsed.count() / 1000 << " ms"
         << endl;

    simpleMatrixMulF((float *) aF, (float *) bF, (float *) resultF_N3);
    auto end3 = std::chrono::steady_clock::now();
    elapsed = end3 - end2;
    cout << "Use float to calculate and the complexity is O(n^3): " <<
elapsed.count() / 1000 << " ms"
         << endl;

    simpleMatrixMulD((double *) aD, (double *) bD, (double *) resultD_N3);
    auto end4 = std::chrono::steady_clock::now();
    elapsed = end4 - end3;
    cout << "Use double to calculate and the complexity is O(n^3): " <<
elapsed.count() / 1000 << " ms"
         << endl;


    if (row1 < 1024) {
        strassenF((float *) aF, (float *) bF, (float *) resultF_N281, row1,
column1, column2);
        auto end5 = std::chrono::steady_clock::now();
        elapsed = end5 - end4;
        cout << "Use float to calculate and the complexity is O(n^2.81): " <<
elapsed.count() / 1000 << " ms"
             << endl;

        strassenD((double *) aD, (double *) bD, (double *) resultD_N281, row1,
column1, column2);
        auto end6 = std::chrono::steady_clock::now();
        elapsed = end6 - end5;
        cout << "Use double to calculate and the complexity is O(n^2.81): " <<
elapsed.count() / 1000 << " ms"
             << endl;
    }

    if (row1 < 1024) {
        fileOutPutF(outfileF_IKJ, resultF_IKJ);
        fileOutPutF(outfileF_N3, resultF_N3);
        fileOutPutF(outfileF_N281, resultF_N281);
        fileOutPutD(outfileD_IKJ, resultD_IKJ);
        fileOutPutD(outfileD_N3, resultD_N3);
        fileOutPutD(outfileD_N281, resultD_N281);
    } else {
        fileOutPutF(outfileF_IKJ, resultF_IKJ);
        fileOutPutF(outfileF_N3, resultF_N3);
        fileOutPutD(outfileD_IKJ, resultD_IKJ);
        fileOutPutD(outfileD_N3, resultD_N3);
    }

    delete[] aD;
    delete[] aF;
    delete[] bD;
    delete[] bF;
    delete[] resultD_IKJ;
    delete[] resultF_IKJ;
    delete[] resultD_N3;
    delete[] resultF_N3;
    delete[] resultD_N281;
    delete[] resultF_N281;

    return 0;
}
```

2 其他方法体

```
int getColumn(string file) {
    ifstream goalFile(file);
    string unit, line;
    getline(goalFile, line);
    stringstream input(line);
    int edgeNum = 0;
    while (input >> unit) {
        edgeNum++;
    }
    return edgeNum;
}

int getRow(string file) {
    ifstream goalFile(file);
    string line;
    int nums = 0;
    while (getline(goalFile, line)) {
        nums++;
    }
    return nums;
}
```

```c
void simpleMatrixMulF(float *arr1, float *arr2, float *res) {
    float c;
    for (int i = 0; i < row1; ++i) {
        for (int j = 0; j < column2; ++j) {
            c = 0.0f;
            for (int k = 0; k < column1; ++k) {
                c += (float) arr1[i * column1 + k] * (float) arr2[k * column2 +
j];
            }
            res[i * column2 + j] = c;
        }
    }
}

void simpleMatrixMulD(double *arr1, double *arr2, double *res) {
    double c;
    for (int i = 0; i < row1; ++i) {
        for (int j = 0; j < column2; ++j) {
            c = 0.0;
            for (int k = 0; k < column1; ++k) {
                c += arr1[i * column1 + k] * arr2[k * column2 + j];
            }
            res[i * column2 + j] = c;
        }
    }
}

void smallMatrixMulF(float *arr1, float *arr2, float *res, int ro1, int col1,
int col2) {
    float c;
    for (int i = 0; i < ro1; ++i) {
        for (int j = 0; j < col2; ++j) {
            c = 0.0f;
            for (int k = 0; k < col1; ++k) {
                c += arr1[i * col1 + k] * arr2[k * col2 + j];
            }
            res[i * col2 + j] = c;
        }
    }
}

void smallMatrixMulD(double *arr1, double *arr2, double *res, int ro1, int col1,
int col2) {
    double c;
    for (int i = 0; i < ro1; ++i) {
        for (int j = 0; j < col2; ++j) {
            c = 0.0;
            for (int k = 0; k < col1; ++k) {
                c += arr1[i * col1 + k] * arr2[k * col2 + j];
            }
            res[i * col2 + j] = c;
        }
    }
}

static void strassenF(float *arr1, float *arr2, float *res, int ro1, int col1,
int col2) {
    if (col1 % 2 != 0 || col2 % 2 != 0 || ro1 % 2 != 0)
```

```
        return smallMatrixMulF(arr1, arr2, res, ro1, col1, col2);

    float *M1 = new float[ro1 / 2 * col2 / 2];
    float *M2 = new float[ro1 / 2 * col2 / 2];
    float *M3 = new float[ro1 / 2 * col2 / 2];
    float *M4 = new float[ro1 / 2 * col2 / 2];
    float *M5 = new float[ro1 / 2 * col2 / 2];
    float *M6 = new float[ro1 / 2 * col2 / 2];
    float *M7 = new float[ro1 / 2 * col2 / 2];
    float *A11 = new float[ro1 / 2 * col1 / 2];
    float *A12 = new float[ro1 / 2 * col1 / 2];
    float *A22 = new float[ro1 / 2 * col1 / 2];
    float *A21 = new float[ro1 / 2 * col1 / 2];
    float *A11022 = new float[ro1 / 2 * col1 / 2];
    float *A21022 = new float[ro1 / 2 * col1 / 2];
    float *A11012 = new float[ro1 / 2 * col1 / 2];
    float *A21111 = new float[ro1 / 2 * col1 / 2];
    float *A12122 = new float[ro1 / 2 * col1 / 2];
    float *B22 = new float[col1 / 2 * col2 / 2];
    float *B11 = new float[col1 / 2 * col2 / 2];
    float *B21 = new float[col1 / 2 * col2 / 2];
    float *B12 = new float[col1 / 2 * col2 / 2];
    float *B11022 = new float[col1 / 2 * col2 / 2];
    float *B12122 = new float[col1 / 2 * col2 / 2];
    float *B21111 = new float[col1 / 2 * col2 / 2];
    float *B11012 = new float[col1 / 2 * col2 / 2];
    float *B21022 = new float[col1 / 2 * col2 / 2];

    for (int i = 0; i < ro1 / 2; ++i) {
        for (int j = 0; j < col1 / 2; ++j) {
            A11[i * col1 / 2 + j] = arr1[i * col1 + j];
            A12[i * col1 / 2 + j] = arr1[i * col1 + j + col1 / 2];
            A22[i * col1 / 2 + j] = arr1[(i + ro1 / 2) * col1 + j + col1 / 2];
            A21[i * col1 / 2 + j] = arr1[(i + ro1 / 2) * col1 + j];
            A11022[i * col1 / 2 + j] = A11[i * col1 / 2 + j] + A22[i * col1 / 2
+ j];
            A21022[i * col1 / 2 + j] = A21[i * col1 / 2 + j] + A22[i * col1 / 2
+ j];
            A11012[i * col1 / 2 + j] = A11[i * col1 / 2 + j] + A12[i * col1 / 2
+ j];
            A21111[i * col1 / 2 + j] = A21[i * col1 / 2 + j] - A11[i * col1 / 2
+ j];
            A12122[i * col1 / 2 + j] = A12[i * col1 / 2 + j] - A22[i * col1 / 2
+ j];
        }
    }
    for (int i = 0; i < col1 / 2; ++i) {
        for (int j = 0; j < col2 / 2; ++j) {
            B11[i * col2 / 2 + j] = arr2[i * col2 + j];
            B21[i * col2 / 2 + j] = arr2[(i + col1 / 2) * col2 + j];
            B22[i * col2 / 2 + j] = arr2[(i + col1 / 2) * col2 + j + col2 / 2];
            B12[i * col2 / 2 + j] = arr2[i * col2 + j + col2 / 2];
            B11022[i * col2 / 2 + j] = B11[i * col2 / 2 + j] + B22[i * col2 / 2
+ j];
            B12122[i * col2 / 2 + j] = B12[i * col2 / 2 + j] - B22[i * col2 / 2
+ j];
            B21111[i * col2 / 2 + j] = B21[i * col2 / 2 + j] - B11[i * col2 / 2
+ j];
```

```
            B11012[i * col2 / 2 + j] = B11[i * col2 / 2 + j] + B12[i * col2 / 2
+ j];
            B21022[i * col2 / 2 + j] = B21[i * col2 / 2 + j] + B22[i * col2 / 2
+ j];
        }
    }

    for (int i = 0; i < ro1 / 2; ++i) {
        for (int j = 0; j < col2 / 2; ++j) {
            M1[i * col2 / 2 + j] = 0;
            M2[i * col2 / 2 + j] = 0;
            M3[i * col2 / 2 + j] = 0;
            M4[i * col2 / 2 + j] = 0;
            M5[i * col2 / 2 + j] = 0;
            M6[i * col2 / 2 + j] = 0;
            M7[i * col2 / 2 + j] = 0;
        }
    }
    strassenF((float *) A11022, (float *) B11022, (float *) M1, ro1 / 2, col1 /
2, col2 / 2);
    strassenF((float *) A21022, (float *) B11, (float *) M2, ro1 / 2, col1 / 2,
col2 / 2);
    strassenF((float *) A11, (float *) B12122, (float *) M3, ro1 / 2, col1 / 2,
col2 / 2);
    strassenF((float *) A22, (float *) B21111, (float *) M4, ro1 / 2, col1 / 2,
col2 / 2);
    strassenF((float *) A11012, (float *) B22, (float *) M5, ro1 / 2, col1 / 2,
col2 / 2);
    strassenF((float *) A21111, (float *) B11012, (float *) M6, ro1 / 2, col1 /
2, col2 / 2);
    strassenF((float *) A12122, (float *) B21022, (float *) M7, ro1 / 2, col1 /
2, col2 / 2);

    for (int i = 0; i < ro1 / 2; i++) {
        for (int j = 0; j < col2 / 2; j++) {
            res[i * col2 + j] =
                    M1[i * col2 / 2 + j] + M4[i * col2 / 2 + j] - M5[i * col2 /
2 + j] + M7[i * col2 / 2 + j];
            res[i * col2 + j + col2 / 2] = M3[i * col2 / 2 + j] + M5[i * col2 /
2 + j];
            res[(i + ro1 / 2) * col2 + j + col2 / 2] =
                    M1[i * col2 / 2 + j] + M3[i * col2 / 2 + j] - M2[i * col2 /
2 + j] + M6[i * col2 / 2 + j];
            res[(i + ro1 / 2) * col2 + j] = M2[i * col2 / 2 + j] + M4[i * col2 /
2 + j];
        }
    }
}

static void strassenD(double *arr1, double *arr2, double *res, int ro1, int
col1, int col2) {
    if (col1 % 2 != 0 || col2 % 2 != 0 || ro1 % 2 != 0)
        return smallMatrixMulD(arr1, arr2, res, ro1, col1, col2);
    double *M1 = new double[ro1 / 2 * col2 / 2];
    double *M2 = new double[ro1 / 2 * col2 / 2];
    double *M3 = new double[ro1 / 2 * col2 / 2];
    double *M4 = new double[ro1 / 2 * col2 / 2];
    double *M5 = new double[ro1 / 2 * col2 / 2];
    double *M6 = new double[ro1 / 2 * col2 / 2];
    double *M7 = new double[ro1 / 2 * col2 / 2];
    double *A11 = new double[ro1 / 2 * col1 / 2];
    double *A12 = new double[ro1 / 2 * col1 / 2];
    double *A22 = new double[ro1 / 2 * col1 / 2];
    double *A21 = new double[ro1 / 2 * col1 / 2];
    double *A11022 = new double[ro1 / 2 * col1 / 2];
    double *A21022 = new double[ro1 / 2 * col1 / 2];
    double *A11012 = new double[ro1 / 2 * col1 / 2];
    double *A21111 = new double[ro1 / 2 * col1 / 2];
    double *A12122 = new double[ro1 / 2 * col1 / 2];
    double *B22 = new double[col1 / 2 * col2 / 2];
    double *B11 = new double[col1 / 2 * col2 / 2];
    double *B21 = new double[col1 / 2 * col2 / 2];
    double *B12 = new double[col1 / 2 * col2 / 2];
    double *B11022 = new double[col1 / 2 * col2 / 2];
    double *B12122 = new double[col1 / 2 * col2 / 2];
    double *B21111 = new double[col1 / 2 * col2 / 2];
    double *B11012 = new double[col1 / 2 * col2 / 2];
    double *B21022 = new double[col1 / 2 * col2 / 2];

    for (int i = 0; i < ro1 / 2; ++i) {
        for (int j = 0; j < col1 / 2; ++j) {
            A11[i * col1 / 2 + j] = arr1[i * col1 + j];
            A12[i * col1 / 2 + j] = arr1[i * col1  + j + col1 / 2];
            A22[i * col1 / 2 + j] = arr1[(i + ro1 / 2) * col1  + j + col1 / 2];
            A21[i * col1 / 2 + j] = arr1[(i + ro1 / 2) * col1  + j];
            A11022[i * col1 / 2 + j] = A11[i * col1 / 2 + j] + A22[i * col1 / 2
+ j];
            A21022[i * col1 / 2 + j] = A21[i * col1 / 2 + j] + A22[i * col1 / 2
+ j];
            A11012[i * col1 / 2 + j] = A11[i * col1 / 2 + j] + A12[i * col1 / 2
+ j];
            A21111[i * col1 / 2 + j] = A21[i * col1 / 2 + j] - A11[i * col1 / 2
+ j];
            A12122[i * col1 / 2 + j] = A12[i * col1 / 2 + j] - A22[i * col1 / 2
+ j];
        }
    }
    for (int i = 0; i < col1 / 2; ++i) {
        for (int j = 0; j < col2 / 2; ++j) {
            B11[i * col2 / 2 + j] = arr2[i * col2 + j];
            B21[i * col2 / 2 + j] = arr2[(i + col1 / 2) * col2  + j];
            B22[i * col2 / 2 + j] = arr2[(i + col1 / 2) * col2  + j + col2 / 2];
            B12[i * col2 / 2 + j] = arr2[i * col2 + j + col2 / 2];
            B11022[i * col2 / 2 + j] = B11[i * col2 / 2 + j] + B22[i * col2 / 2
+ j];
            B12122[i * col2 / 2 + j] = B12[i * col2 / 2 + j] - B22[i * col2 / 2
+ j];
            B21111[i * col2 / 2 + j] = B21[i * col2 / 2 + j] - B11[i * col2 / 2
+ j];
            B11012[i * col2 / 2 + j] = B11[i * col2 / 2 + j] + B12[i * col2 / 2
+ j];
            B21022[i * col2 / 2 + j] = B21[i * col2 / 2 + j] + B22[i * col2 / 2
+ j];
        }
    }
    for (int i = 0; i < ro1 / 2; ++i) {
        for (int j = 0; j < col2 / 2; ++j) {
            M1[i * col2 / 2 + j] = 0;
            M2[i * col2 / 2 + j] = 0;
            M3[i * col2 / 2 + j] = 0;
            M4[i * col2 / 2 + j] = 0;
            M5[i * col2 / 2 + j] = 0;
            M6[i * col2 / 2 + j] = 0;
            M7[i * col2 / 2 + j] = 0;
        }
    }

    strassenD((double *) A11022, (double *) B11022, (double *) M1, ro1 / 2, col1
/ 2, col2 / 2);
    strassenD((double *) A21022, (double *) B11, (double *) M2, ro1 / 2, col1 /
2, col2 / 2);
    strassenD((double *) A11, (double *) B12122, (double *) M3, ro1 / 2, col1 /
2, col2 / 2);
    strassenD((double *) A22, (double *) B21111, (double *) M4, ro1 / 2, col1 /
2, col2 / 2);
    strassenD((double *) A11012, (double *) B22, (double *) M5, ro1 / 2, col1 /
2, col2 / 2);
    strassenD((double *) A21111, (double *) B11012, (double *) M6, ro1 / 2, col1
/ 2, col2 / 2);
    strassenD((double *) A12122, (double *) B21022, (double *) M7, ro1 / 2, col1
/ 2, col2 / 2);

    for (int i = 0; i < ro1 / 2; i++) {
        for (int j = 0; j < col2 / 2; j++) {
            res[i * col2 + j] =
                    M1[i * col2 / 2 + j] + M4[i * col2 / 2 + j] - M5[i * col2 /
2 + j] + M7[i * col2 / 2 + j];
            res[i * col2 + j + col2 / 2] = M3[i * col2 / 2 + j] + M5[i * col2 /
2 + j];
            res[(i + ro1 / 2) * col2 + j + col2 / 2] =
                    M1[i * col2 / 2 + j] + M3[i * col2 / 2 + j] - M2[i * col2 /
2 + j] + M6[i * col2 / 2 + j];
            res[(i + ro1 / 2) * col2 + j] = M2[i * col2 / 2 + j] + M4[i * col2 /
2 + j];
        }
    }

}


static void IKJF(float *arr1, float *arr2, float *res) {
    float d = 0.0F;
    {
#pragma omp parallel for
        for (int i = 0; i < row1; i++) {
            for (int k = 0; k < column1; k++) {
                d = arr1[i * column1 + k];
                for (int j = 0; j < column2; j++) {
                    res[i * column2 + j] += d * arr2[k * column2 + j];
                }
            }
        }
    }
```

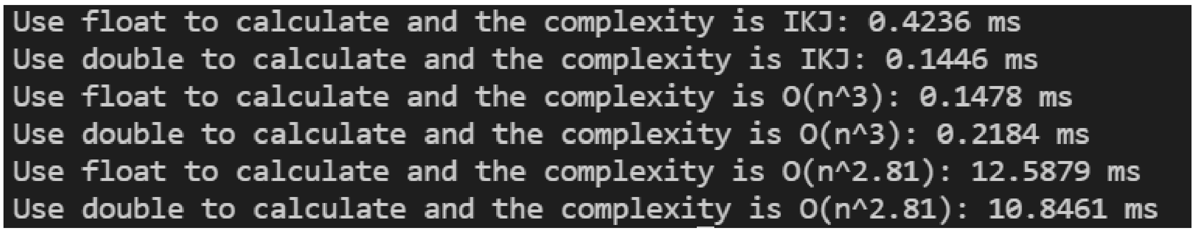
```
}

static void IKJD(double *arr1, double *arr2, double *res) {
    double d = 0.0;
    {
#pragma omp parallel for
        for (int i = 0; i < row1; i++) {
            for (int k = 0; k < column1; k++) {
                d = arr1[i * column1 + k];
                for (int j = 0; j < column2; j++) {
                    res[i * column2 + j] += d * arr2[k * column2 + j];
                }
            }
        }
    }
}

static void fileOutPutF(const string &out, float *res) {
    ofstream outfile;
    outfile.open(out);
    for (int i = 0; i < row1; ++i) {
        for (int j = 0; j < column2; ++j) {
            outfile << to_string(res[i * column2 + j]) << " ";
        }
        outfile << endl;
    }
    outfile.close();
}

static void fileOutPutD(const string &out, double *res) {
    ofstream outfile;
    outfile.open(out);
    for (int i = 0; i < row1; ++i) {
        for (int j = 0; j < column2; ++j) {
            outfile << to_string(res[i * column2 + j]) << " ";
        }
        outfile << endl;
    }
    outfile.close();
}
```

第三部分 结果展示

结果展示的是32和256矩阵乘法的IKJ+并行+编译优化（O3级别）、IJK、strassen 算法；2048矩阵乘法的IKJ+并行+ 编译优化（O3级别）、IJK；

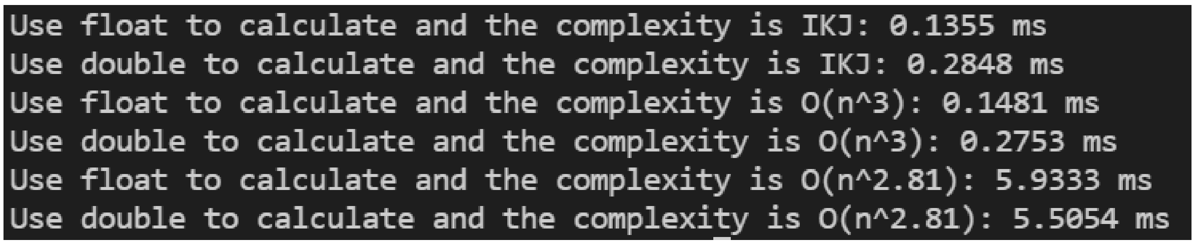
1 处理 32 * 32 的矩阵

利用二维数组：



```
Use float to calculate and the complexity is IKJ: 0.4236 ms
Use double to calculate and the complexity is IKJ: 0.1446 ms
Use float to calculate and the complexity is O(n^3): 0.1478 ms
Use double to calculate and the complexity is O(n^3): 0.2184 ms
Use float to calculate and the complexity is O(n^2.81): 12.5879 ms
Use double to calculate and the complexity is O(n^2.81): 10.8461 ms
```

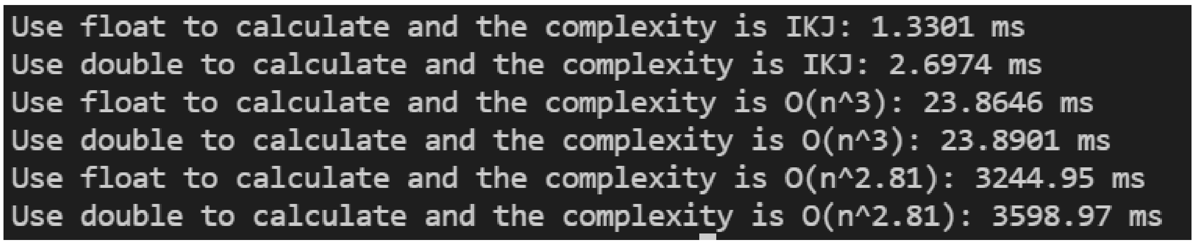
利用一维数组：



```
Use float to calculate and the complexity is IKJ: 0.1355 ms
Use double to calculate and the complexity is IKJ: 0.2848 ms
Use float to calculate and the complexity is O(n^3): 0.1481 ms
Use double to calculate and the complexity is O(n^3): 0.2753 ms
Use float to calculate and the complexity is O(n^2.81): 5.9333 ms
Use double to calculate and the complexity is O(n^2.81): 5.5054 ms
```

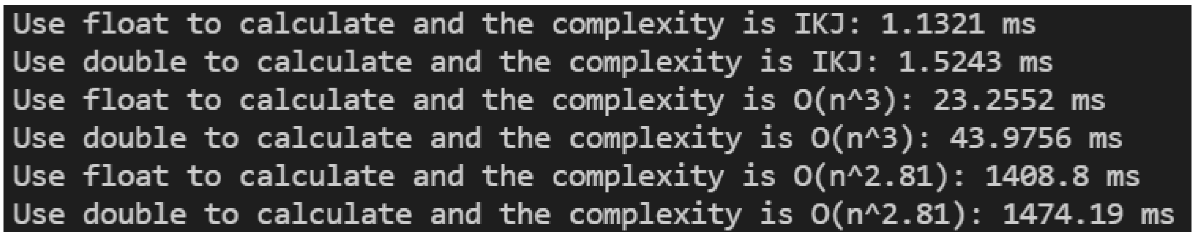
2 处理 256 * 256 的矩阵

利用二维数组：



```
Use float to calculate and the complexity is IKJ: 1.3301 ms
Use double to calculate and the complexity is IKJ: 2.6974 ms
Use float to calculate and the complexity is O(n^3): 23.8646 ms
Use double to calculate and the complexity is O(n^3): 23.8901 ms
Use float to calculate and the complexity is O(n^2.81): 3244.95 ms
Use double to calculate and the complexity is O(n^2.81): 3598.97 ms
```

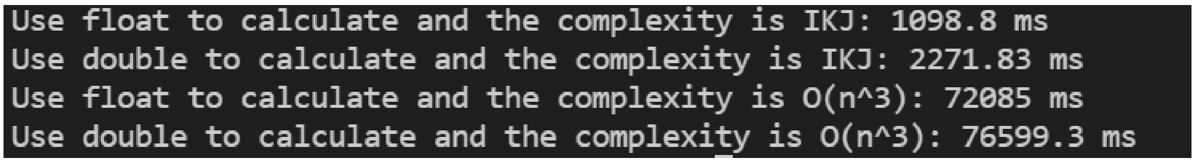
利用一维数组：



```
Use float to calculate and the complexity is IKJ: 1.1321 ms
Use double to calculate and the complexity is IKJ: 1.5243 ms
Use float to calculate and the complexity is O(n^3): 23.2552 ms
Use double to calculate and the complexity is O(n^3): 43.9756 ms
Use float to calculate and the complexity is O(n^2.81): 1408.8 ms
Use double to calculate and the complexity is O(n^2.81): 1474.19 ms
```

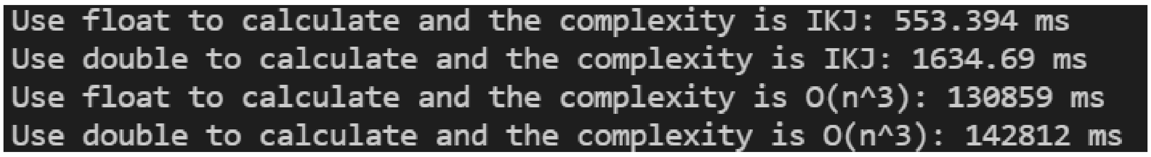
3 处理 2048 * 2048 的矩阵

利用二维数组：



```
Use float to calculate and the complexity is IKJ: 1098.8 ms
Use double to calculate and the complexity is IKJ: 2271.83 ms
Use float to calculate and the complexity is O(n^3): 72085 ms
Use double to calculate and the complexity is O(n^3): 76599.3 ms
```

利用一维数组：



```
Use float to calculate and the complexity is IKJ: 553.394 ms
Use double to calculate and the complexity is IKJ: 1634.69 ms
Use float to calculate and the complexity is O(n^3): 130859 ms
Use double to calculate and the complexity is O(n^3): 142812 ms
```

第四部分 难点和方案

1 如何获取矩阵规模

假设预先不知道矩阵规模，就需要读取矩阵规模，采用以下方法，分别可以获得矩阵的行数和列数：**即可以处理各种规模的矩阵乘法问题；**

```
int getColumn(string file) {
    ifstream goalFile(file);
    string unit, line;
    getline(goalFile, line); //获取一行数据
    stringstream input(line); //通过“ ”将字符串断开
    int edgeNum = 0;
    while (input >> unit) { //并将每个值赋值给unit,当读取完毕，循环跳出
        edgeNum++;
    }
    return edgeNum; //返回列数
}

int getRow(string file) {
    ifstream goalFile(file);
    string line;
    int nums = 0;
    while (getline(goalFile, line)) { //获取一行数据，读到没有下一行退出循环
        nums++;
    }
    return nums; //返回行数
}
```

2 如何创建数组

将数组创建在堆上**可以规避栈溢出**，利用new关键词（以double的一维和二维数组为例）：

```
double *  array2 = new double [length];//一维数组创建

double ** array1 = new double *[row];//二维数组创建
for (int i = 0; i < row; ++i) {
    array1[i] = new double[column];
}
```

3 double和float的差异

①精度差异

float为32位字节，是单精度浮点数，double为64位字节，是双精度浮点数。

float和double的**整数部分始终是“1”，由于它是不变的，故不能对精度造成影响**，因此其精度是由尾数的位数来决定的，浮点数在内存中是按照科学计数法来储存的，由于这一特性，**float的精度为6~7位有效数字，所以它并不准确，如果该数无法正确表示，它将选择离该数最近的数加以取代，在计算中就会造成精度丢失，带来结果与double存在差异**；由于double的精度为15~16位，因此，在本次数据中几乎不会出现数据精度丢失，可以认为double的计算结果是正确的（在多并行的条件下不成立）。

②速度差异

从直观上看，由于float只有32位，相比double的64位，在字节上就存在空间优势，于是认为**float的运算速度高于double**，这个想法是有道理的。浮点数计算本身并不是影响速度的最大原因，基于现代CPU的微观并行化，真正的问题在于内存吞吐量。**float数据的读写比double数据的读写省去了一半的数据量，这个差距在数据量大时非常明显，而在数据量小的时候可能由于缓存或者回收机制等影响而产生double速度优于float的情况；**

网上还有一个说法：**关于double速度优于float**

“浮点运算在底层实现上都是以双精度进行的，即使只有float的运算，也要先转换成double再进行计算“，**所以float比double速度慢是因为float还要进行若干次数据类型转换**。以上对于比较旧的架构FPU是这样的，**FPU指令集内部使用80位精度计算浮点数，意思就是无论你的浮点数数据类型是什么，始终是利用80位精度计算。**

总结如下：在现在CPU处理器的基础下，float的计算速度理论上应当快于double，但也无法排除内存状态、程序运行加载、并行处理以及回收机制等因素影响，因此也会出现double快于float的情况。

4、运算速度优化

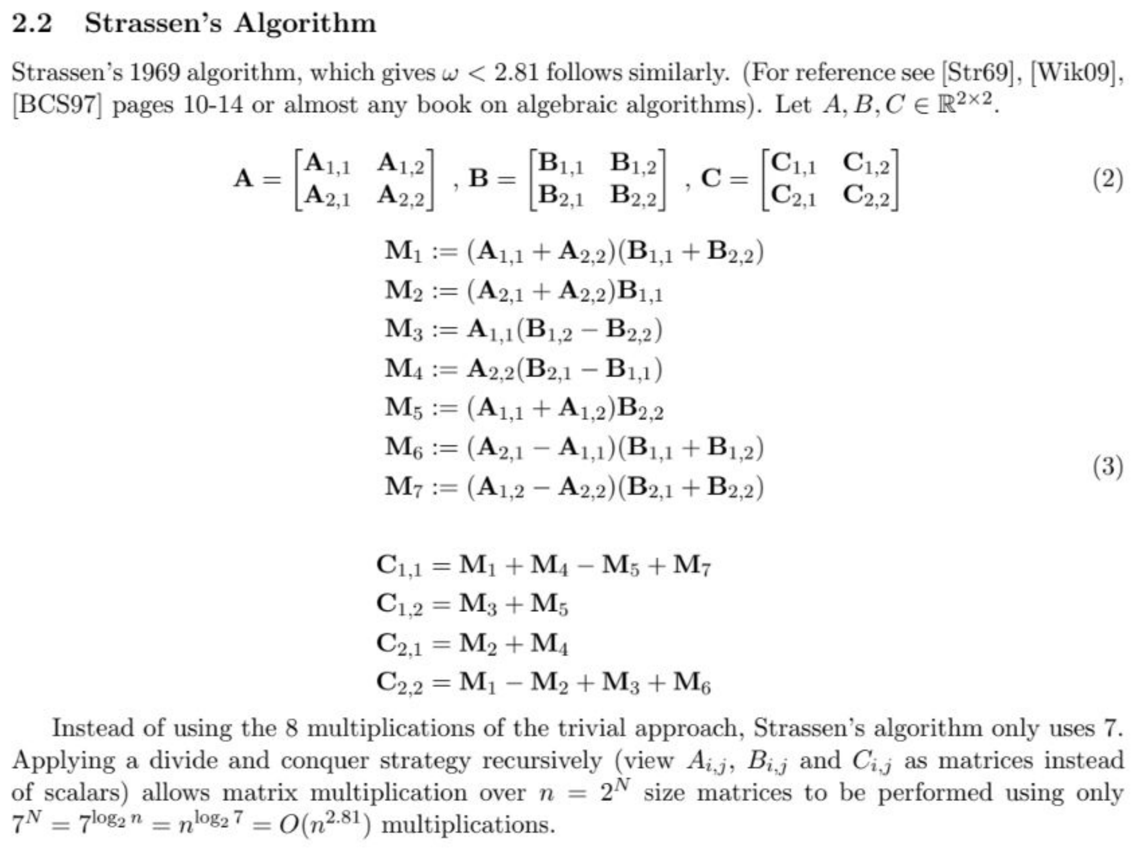
①基于时间复杂度

通过查找资料，迄今为止矩阵乘法的复杂度已经优化到O(n^2.37)左右，算法时间复杂度优化历程如下：

时间复杂度	算法名称
n^3	
n^2.81	Strassen (1969)
n^2.376	Coppersmith-Winograd (1990)
n^2.374	Stothers (2010)
n^2.3729	Williams (2011)
n^2.37287	Le Gall (2014)

基于上述算法，后三种算法没有得到充分证明，因此目前 Coppersmith-Winograd (1990) 算法为矩阵乘法时间复杂度的最优算法，但更为普遍的还是Strassen (1969) 算法，于是本次project采用Strassen (1969) 算法；

Strassen (1969) 算法参考下图，



2.2 Strassen's Algorithm

Strassen's 1969 algorithm, which gives $\omega < 2.81$ follows similarly. (For reference see [Str69], [Wik09], [BCS97] pages 10-14 or almost any book on algebraic algorithms). Let $A, B, C \in \mathbb{R}^{2\times 2}$.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix} \tag{2}$$

$$\begin{aligned} \mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\ \mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\ \mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\ \mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ \mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\ \mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\ \mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}) \end{aligned} \tag{3}$$

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\ \mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\ \mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6 \end{aligned}$$

Instead of using the 8 multiplications of the trivial approach, Strassen's algorithm only uses 7. Applying a divide and conquer strategy recursively (view $A_{i,j}$, $B_{i,j}$ and $C_{i,j}$ as matrices instead of scalars) allows matrix multiplication over $n = 2^N$ size matrices to be performed using only $7^N = 7^{\log_2 n} = n^{\log_2 7} = O(n^{2.81})$ multiplications.

利用将矩阵乘法转换成多个矩阵加法以及递归的思想来对该算法进行描述，但是正因为该算法充分利用分治思想，**在面对小矩阵（<300 * 300），这个算法毫无优势，但是面对2048 * 2048的大型矩阵，由于递归对栈内存的消耗，导致栈溢出**，因此在老师所给出的数据中，没有能够体现该算法优势的地方，算法方面的优化较为失败。

②基于硬件优化

造成矩阵乘法慢的原因，除了算法上的**O(n^3)**以外，还有**内存访问不连续**。这会导致cache命中率不高，即缓存回收命中率不高。所以为了加速，就要尽可能使内存访问连续，因此可以通过改变IJK的计算顺序，来实现速度优化。通过查找资料，如下图所示：

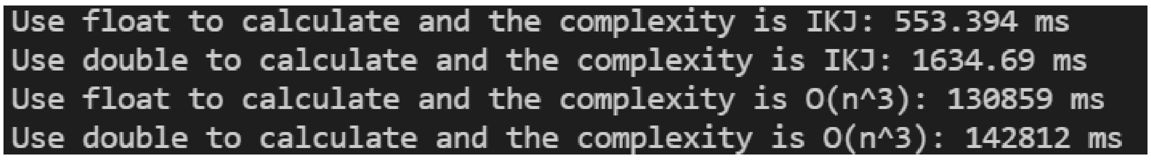
- 顺序 ikj —— $2n^2+n$ （二维数组）—— n^2 （一维数组）
- 顺序 kij —— $3n^2$ （二维数组）—— $2n^2$ （一维数组）
- 顺序 jik —— n^3+2n^2 （二维数组）—— n^3+n^2+n （一维数组）
- 顺序 ijk —— n^3+n^2+n （二维数组）—— n^3+n^2-n （一维数组）
- 顺序 kji —— $2n^3+n$ （二维数组）—— $2n^3$ （一维数组）
- 顺序 jki —— $2n^3+n^2$ （二维数组）—— $2n^3+n^2$ （一维数组）

因此从速度来说：

$$ikj > kij > jik > ijk > kji > jki$$

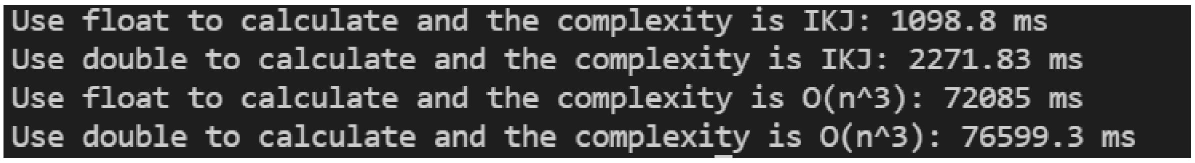
本次project采用IKJ来实现速度优化，不难发现（未使用并行）针对一维数组速度可以提升80-300倍，二维数组速度可以提升30-70倍，且结果几乎一致（由于改变计算顺序造成的精度丢失不一致的影响很小）：以2048 * 2048 为例，

一维数组对比：



```
Use float to calculate and the complexity is IKJ: 553.394 ms
Use double to calculate and the complexity is IKJ: 1634.69 ms
Use float to calculate and the complexity is O(n^3): 130859 ms
Use double to calculate and the complexity is O(n^3): 142812 ms
```

二维数组对比：



```
Use float to calculate and the complexity is IKJ: 1098.8 ms
Use double to calculate and the complexity is IKJ: 2271.83 ms
Use float to calculate and the complexity is O(n^3): 72085 ms
Use double to calculate and the complexity is O(n^3): 76599.3 ms
```

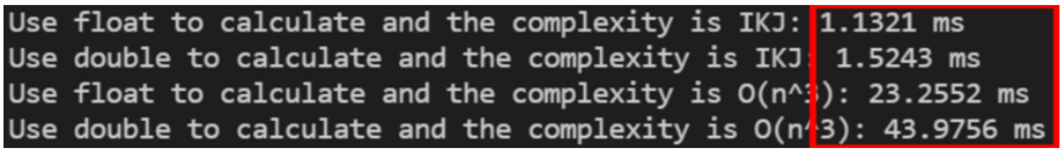
③并行化处理

（参考https://blog.csdn.net/weixin_39568744/article/details/88576576?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522163250087316780357275156%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request_id=163250087316780357275156&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-2-88576576.pc_search_ecpm_flag&utm_term=%23program+omp+parallel+for&spm=1018.2226.3001.4187)

OpenMp是由OpenMP Architecture Review Board牵头提出的，并已被广泛接受的，用于共享内存并行系统的多处理器程序设计的一套指导性的编译处理方案(Compiler Directive)。

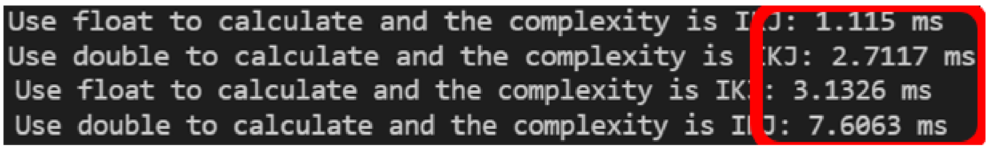
OpenMP中的一个最强大的一个功能是：在之前串行程序的源码基础上（即IJK—N^3），只要进行少量的改动，就可以并行化许多串行的for循环，达到明显提高性能的效果。

因此可以利用并行处理的特性，将多个内核同时利用，从而提高计算效率（可以提高20倍左右的计算速度，与IJK相比），以256 * 256为例：



```
Use float to calculate and the complexity is IKJ: 1.1321 ms
Use double to calculate and the complexity is IKJ: 1.5243 ms
Use float to calculate and the complexity is O(n^3): 23.2552 ms
Use double to calculate and the complexity is O(n^3): 43.9756 ms
```

并行化处理和关闭并行化处理程序执行效率对比如下：



```
Use float to calculate and the complexity is IKJ: 1.115 ms
Use double to calculate and the complexity is IKJ: 2.7117 ms
 Use float to calculate and the complexity is IKJ: 3.1326 ms
 Use double to calculate and the complexity is IKJ: 7.6063 ms
```

但是由于多并行处理，因此很容易挤占CPU内存，从而导致**数据错乱**，**计算结果准确性下降**（与准确值相差较大），因此在底层优化（硬件层面）**要同时考虑效率和准确性**，**不能为了提升效率而降低计算的准确性**；

④循环展开 — loop unrolling（CUDA优化）

通过增加步长，及将内层循环步长设为4（32、256、2048都可以被4整除，其中空间换时间的折中最优位置需要针对具体问题来做具体的分析，因此选择4作为步长），原理上可以提升计算速度，但是由于编译优化（**系统自带的开发人员已经提供的底层优化**）为**O3**，可能内部已经实现了循环展开，从而导致手动展开时效果不好，**可能还会延长程序运行时间，因此此优化方案失败**；

⑤SIMD

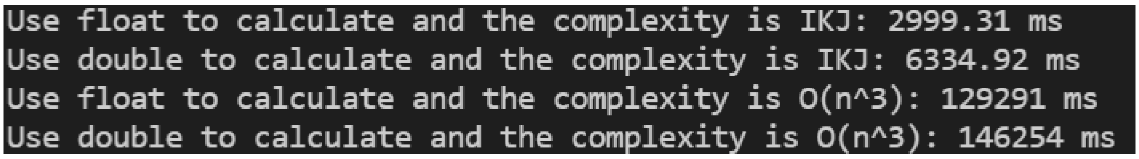
尝试SIMD的优化，**但是由于SIMD是较为底层的优化，所以在编译优化（O3）中已经进行自动优化，且其执行会与并行化执行冲突，所以效果不明显。**尝试实现SIMD代码如下：

```
{
        __m256d num;
#pragma omp parallel for
        for (int k = 0; k < column1; k++) {
            for (int i = 0; i < row1; i++) {
                __m256d d = _mm256_broadcast_sd(arr1 + i * column1 + k);
                for (int j = 0; j < column2; j += 4) {
                    int n = i * column2;
                    num = _mm256_load_pd(res + n + j);
                    num = _mm256_add_pd(num,
                                        _mm256_mul_pd(d,
                                                      _mm256_load_pd(arr2 +
                                                                     k * column2
+ j)));
                    _mm256_store_pd(res + i * column2 + j, num);
                }
            }
        }
    }
}
```

⑥一维与二维数组的选择

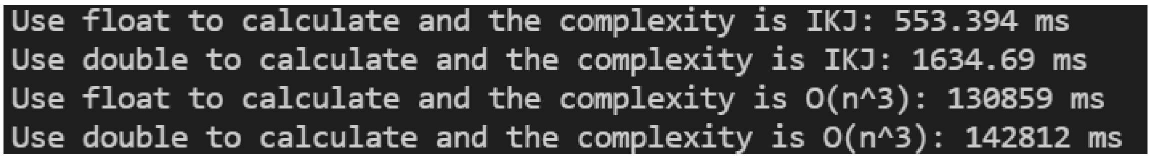
经过程序测试，在数据量较小的时候（这里仅知 < 256 * 256），表现出明显优势，速度比二维数组缩小一倍（在结果展示中体现），但在2048 * 2048的矩阵计算时，结果反倒表现出明显的效率下降，结果对比（2048 * 2048）如下：（**注：多并行只用于IKJ式中**）

一维数组关闭多并行：



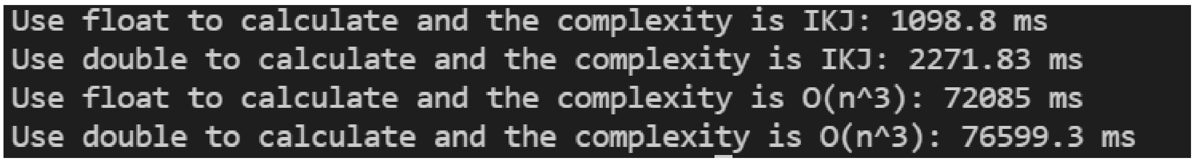
```
Use float to calculate and the complexity is IKJ: 2999.31 ms
Use double to calculate and the complexity is IKJ: 6334.92 ms
Use float to calculate and the complexity is O(n^3): 129291 ms
Use double to calculate and the complexity is O(n^3): 146254 ms
```

一维数组多并行：



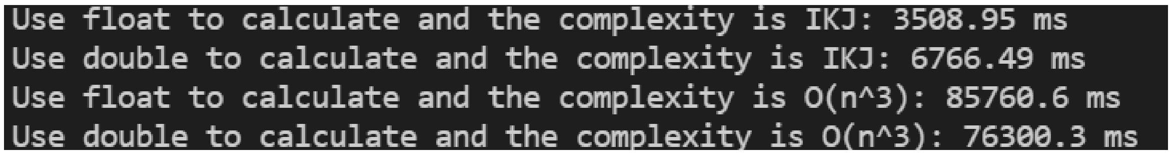
```
Use float to calculate and the complexity is IKJ: 553.394 ms
Use double to calculate and the complexity is IKJ: 1634.69 ms
Use float to calculate and the complexity is O(n^3): 130859 ms
Use double to calculate and the complexity is O(n^3): 142812 ms
```

二维数组多并行：



```
Use float to calculate and the complexity is IKJ: 1098.8 ms
Use double to calculate and the complexity is IKJ: 2271.83 ms
Use float to calculate and the complexity is O(n^3): 72085 ms
Use double to calculate and the complexity is O(n^3): 76599.3 ms
```

二维数组关闭多并行：

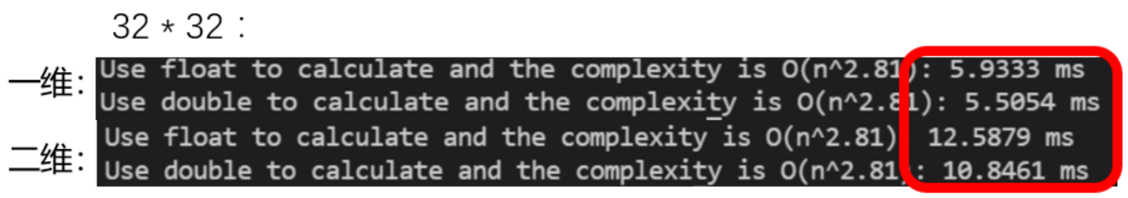


```
Use float to calculate and the complexity is IKJ: 3508.95 ms
Use double to calculate and the complexity is IKJ: 6766.49 ms
Use float to calculate and the complexity is O(n^3): 85760.6 ms
Use double to calculate and the complexity is O(n^3): 76300.3 ms
```

由以上数据可以作出以下总结：

· 在一维数组和二维数组同时开启多并行时，一维数组的速度较二维数组快一点，**在一维数组有数据存储优势的基础上，说明多并行对一维数组运行效率的优化更为显著**；

· 对IJK而言，二维数组普遍比一维数组快，说明在大数据的计算，**一维数组的运算效率较二维数组低下**，但在较小规模矩阵乘法计算中，二维数组的速度较一维数组慢，但在大频率的矩阵计算中，一维数组表现极大的优势（下图以N^2.81为例，计算过程中运用大量矩阵计算），体现了连续空间的优势；



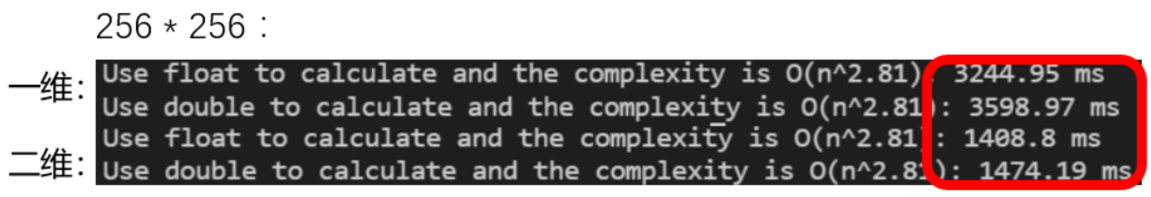
32 * 32：

一维：

```
Use float to calculate and the complexity is O(n^2.81): 5.9333 ms
Use double to calculate and the complexity is O(n^2.81): 5.5054 ms
```

二维：

```
Use float to calculate and the complexity is O(n^2.81): 12.5879 ms
Use double to calculate and the complexity is O(n^2.81): 10.8461 ms
```



256 * 256：

一维：

```
Use float to calculate and the complexity is O(n^2.81): 3244.95 ms
Use double to calculate and the complexity is O(n^2.81): 3598.97 ms
```

二维：

```
Use float to calculate and the complexity is O(n^2.81): 1408.8 ms
Use double to calculate and the complexity is O(n^2.81): 1474.19 ms
```

· 当一维和二维数组同时关闭多并行时，针对IKJ，二维数组与一维数组之间计算速度差异较大，可以说明二维数组比一维数组在大规模的数组计算中体现高效的计算效率，在IKJ的硬件优化中表现出一维数组较N^3速度提升幅度比二维数组大，**一维数组在硬件优化中表现出更明显的优化效果**；

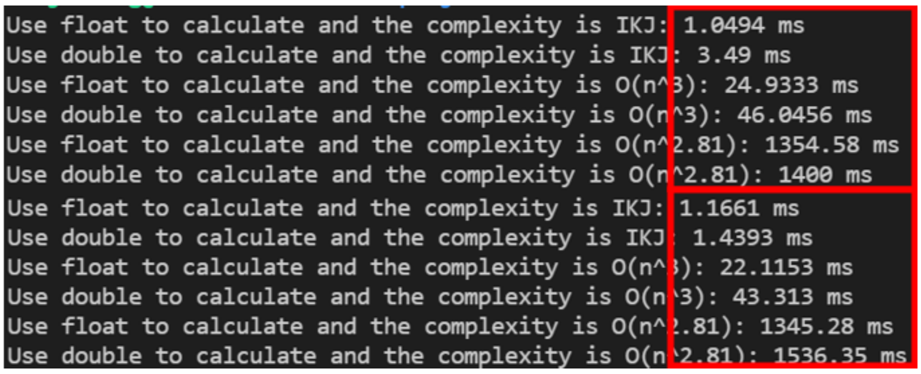
5 结果正确性校验

通过三种计算方式以及MATLAB辅助检验，结果在整数部分几乎一致，小数部分因浮点数计算有精度丢失，无法确保准确；且由于一些优化对计算顺序、计算存储过程做出一定的变动，导致有些数据偏差较大；

MATLAB辅助计算结果在zip文件data目录下另附（以截图的形式呈现）。

6 程序执行用时差异

以256 * 256矩阵乘法两次计算时间对比为例：



```
Use float to calculate and the complexity is IKJ: 1.0494 ms
Use double to calculate and the complexity is IKJ: 3.49 ms
Use float to calculate and the complexity is O(n^3): 24.9333 ms
Use double to calculate and the complexity is O(n^3): 46.0456 ms
Use float to calculate and the complexity is O(n^2.81): 1354.58 ms
Use double to calculate and the complexity is O(n^2.81): 1400 ms
Use float to calculate and the complexity is IKJ: 1.1661 ms
Use double to calculate and the complexity is IKJ: 1.4393 ms
Use float to calculate and the complexity is O(n^3): 22.1153 ms
Use double to calculate and the complexity is O(n^3): 43.313 ms
Use float to calculate and the complexity is O(n^2.81): 1345.28 ms
Use double to calculate and the complexity is O(n^2.81): 1536.35 ms
```

结果虽然相近，但不完全相同，相同的程序每次都能跑出不一样的用时，依据查找资料，给出可能原因如下：其一，**与操作系统的调度有关**，每次计算电脑都会处于不同的状态，例如同时开启的程序有了不同的进程，这都会使操作系统的调度出现不同；其二，**CPU支持动态调频**，依据电脑开启的进程数，可以动态调整频率，也会导致每次程序运行结果不同；

7 不同规模矩阵计算时差分析

不同规模的矩阵计算速度差异很大，小规模的矩阵即使不进行任何优化，也能在很短的时间内运行出来，但随着矩阵规模增大，如果不进行优化，在很长时间内都没法运行出来（2048 * 2048 最长达3分钟左右），因此优化显得势在必行；

在优化中，在小规模的矩阵计算中，表现出与IJK无异的运行速度，但在大规模的数组中，运行速度成倍提升，能够充分体现优越性能；

8 I/O读取与内存读取

由于将数组创建在堆上（及内存中），在读取上较I/O（磁盘读取）快，举例如下（例子来源：https://blog.csdn.net/truelove12358/article/details/105692909/?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_utm_term~default-1.no_search_link&spm=1001.2101.3001.4242）：

以红色威龙增强版DDR4电脑性能为例，

连续读取：
DDR4内存大概是60GB/s的水平；
机械硬盘大概是100～150MB/s的水平；

由此可见，内存读取较磁盘快近4个数量级，原因在于存储介质的特性，磁盘本身存取比主存慢，以及机械运动的耗费造成效率的进一步下降，因此一般的，数据存储在内存中，以便于提高读取速度，从而提升程序性能。