CS205 Project2

Leng Ziyang 12011513

September 30, 2021

1 Analysis

1.1 Briefly

To implement a matrix multiplication algorithm, all we need is the number of rows and columns and every elements in the matrix. Using the i^{th} row of and the j^{th} column to calculate the dot product of them and fill it in the C_{ij} of the result matrix C, we then get the result. Therefore theoretically three arrays with two dimensions can perfectly satisfy all we need.

1.2 Think More

Actually being the most fundamental problem, matrix multiplication is famous for its complexity. While the logic behind calculation is quite simple and can be implemented with several *for-loop*, different kinds of implementation may vary a lot in the times they take. How to implement the algorithm in a elegant and high efficient way is the final goal of this project actually.

1.3 Faster?

To implement a program with faster running speed, we first need to consider the component of the running time and then optimize each part to realize the high efficiency. For a program that perform matrix multiplication, the mainly cost of its running time can basically be divided into two parts which are IO costs and calculation cost.

1.3.1 IO cost

To load the two operating matrices into memory from disk, the program need to traverse the files that store the matrices and fetch every element from the .txt file to fill the right position in the two dimension matrices. This cost does not takes much portions of the running time and there is little we can do to optimize this process. But another IO is fetching elements from the two dimension arrays that store the original elements. For a traditional algorithm that multiply two N * N matrices, every element should be visited N times to finish the calculation[1]. Therefore the fetching time of elements in the arrays influence the program a lot. A optional optimization can be implemented in this way.

1.3.2 Calculation Cost

Calculation consists a large part of the time complexity in this algorithm. In the traditional implementation, the time complexity is $O(n^3)$ which is considered to be a awful situation since the execution time grows in a significant speed as the growth of the problem size. When we need to dealing with large matrix multiplication such as the 2048 * 2048 matrix multiplication, the execution time is far beyond our expectations. Hence this is another enter point where we can optimize our program.

[1]C++ 加速矩阵乘法的最简单方法. (2020). 知乎专栏. https://zhuanlan.zhihu.com/p/146250334?from_voters_page=true

1.4 Faster!

After the analysis above, we then have a vague direcations of what we can optimized in order to realize the high efficiency of the program. That is to improve the fetching speed of data stored in the memory, implement faster algorithm with less time complexity and also increase the calculation speed.

2 Codes

The codes behind the matrix multiplier are simply three parts, the matrix class that can store and manipulate matrix, the read and write function that reads matrix from .txt file and load it into memory or writes matrix into .txt file from memory. While the different optimization are written in the different methods in the matrix class.

2.1 Matrix Class(Definition)

```
#include "matrix.h"
#include <iostream>
#include "arm_neon.h"

#define MathType float

Matrix::Matrix(int r, int c) : rowNum(r), colNum(c) {
    matrix=new MathType*[r];
    for (int i = 0; i < r; ++i) {
        matrix[i] = new MathType[c];
        for (int j = 0; j < c; ++j) {
            matrix[i][j]=0;
        }
    }
}

Matrix::~Matrix() {
    if (!matrix) return;
    for (int i = 0; i < rowNum; ++i)
         delete[] matrix[i];
    colNum = rowNum = 0;
    delete[] matrix;
}

MathType &Matrix::operator()(int i, int j) {
    return matrix[i][j];
}

Matrix Matrix::operator+(Matrix &matA){
    Matrix resMat(matA.rowNum,matA.rowNum);
    for (int i = 0; i < matA.rowNum; ++i) {
        for (int j = 0; j < matA.colNum; ++j) {
            resMat(i,j) = (*this)(i,j)+matA(i,j);
        }
    }
    return resMat;
}

Matrix Matrix::operator-(Matrix &matA){
    Matrix resMat(matA.rowNum,matA.rowNum);
    for (int i = 0; i < matA.rowNum; ++i) {
        for (int j = 0; j < matA.colNum; ++j) {
            resMat(i,j) = (*this)(i,j)-matA(i,j);
        }
    }
    return resMat;
}

Matrix Matrix::dot_Product(Matrix &matA, Matrix &matB) {
    if (matA.colNum != matB.rowNum){
        cout << "Matrices size failed to match!"<< endl;
        exit(1);
    }
    Matrix resMat(matA.rowNum, matB.colNum);
    for (int i = 0; i < matA.rowNum; ++i) {
        for (int j = 0; j < matB.colNum; ++j) {
            for (int k = 0; k < matA.colNum; ++k) {
                resMat(i, j) += matA(i, k) * matB(k, j);
            }
        }
    }
    return resMat;
}

Matrix Matrix::dot_Product_ikj(Matrix &matA, Matrix &matB){
    if (matA.colNum != matB.rowNum){
        cout << "Matrices size failed to match!"<< endl;
        exit(1);
    }
    Matrix resMat(matA.rowNum, matB.colNum);
    MathType s;
    for(int i=0;i<matA.rowNum;++i) {
        for (int k = 0; k < matA.colNum; ++k) {
            s = matA(i, k);
            for (int j = 0; j < matB.colNum; ++j)
                resMat(i, j) += s * matB(k, j);
        }
    }
    return resMat;
}

Matrix Matrix::subMatrix(Matrix &matA,int rStart,int rowLen,int cStart,int
   cLen){
    Matrix resMat(rowLen,cLen);
    for (int i = 0; i < resMat.rowNum; ++i) {
        for (int j = 0; j < resMat.colNum; ++j) {
            resMat(i,j) = matA(i+rStart,j+cStart);
        }
    }
    return resMat;
}

Matrix Matrix::merge(Matrix &mat11,Matrix &mat12,Matrix &mat21,Matrix &mat22)
```

```
    {
    Matrix resMat(mat11.rowNum*2,mat11.colNum*2);
    for (int i = 0; i < mat11.rowNum; ++i) {
        for (int j = 0; j < mat11.colNum; ++j) {
            resMat(i,j) = mat11(i,j);
            resMat(i,j+mat11.colNum) = mat12(i,j);
            resMat(i+mat11.rowNum,j) = mat21(i,j);
            resMat(i+mat11.rowNum,j+mat11.colNum)=mat22(i,j);
        }
    }
    return resMat;
}

Matrix Matrix::dot_Product_Strassen(Matrix &matA,Matrix &matB){
    int n = matA.rowNum;

    if (n <= 4){
        Matrix resMat(n,n);
        MathType s=0;
        for(int i=0;i<matA.rowNum;++i) {
            for (int k = 0; k < matA.colNum; ++k) {
                s = matA(i, k);
                for (int j = 0; j < matB.colNum; ++j) {
                    resMat(i, j) += s * matB(k, j);
                }
            }
        }
        return resMat;
    }

    n /=2;
    Matrix A11 = subMatrix(matA, 0, n, 0, n);
    Matrix A12 = subMatrix(matA, 0, n, n, n);
    Matrix A21 = subMatrix(matA, n, n, 0, n);
    Matrix A22 = subMatrix(matA, n, n, n, n);

    Matrix B11 = subMatrix(matB, 0, n, 0, n);
    Matrix B12 = subMatrix(matB, 0, n, n, n);
    Matrix B21 = subMatrix(matB, n, n, 0, n);
    Matrix B22 = subMatrix(matB, n, n, n, n);

    Matrix S1 = B12 - B22;
    Matrix S2 = A11 + A12;
    Matrix S3 = A21 + A22;
    Matrix S4 = B21 - B11;
    Matrix S5 = A11 + A22;
    Matrix S6 = B11 + B22;
    Matrix S7 = A12 - A22;
    Matrix S8 = B21 + B22;
    Matrix S9 = A11 - A21;
    Matrix S10 = B11 + B12;

    Matrix P1 = dot_Product_Strassen(A11, S1);
    Matrix P2 = dot_Product_Strassen(S2, B22);
    Matrix P3 = dot_Product_Strassen(S3, B11);
    Matrix P4 = dot_Product_Strassen(A22, S4);
    Matrix P5 = dot_Product_Strassen(S5, S6);
    Matrix P6 = dot_Product_Strassen(S7, S8);
    Matrix P7 = dot_Product_Strassen(S9, S10);

    Matrix C11 = P5 + P4 - P2 + P6;
    Matrix C12 = P1 + P2;
    Matrix C21 = P3 + P4;
    Matrix C22 = P5 + P1 - P3 - P7;

    return merge(C11, C12, C21, C22);

}

Matrix Matrix::dot_Product_Neon(Matrix &matA,Matrix &matB){
    int n = matA.rowNum;

    if (n<=4){
        Matrix resMat(matA.rowNum,matB.colNum);
        float32x4_t vc0 = vdupq_n_f32(0.0f);
        float32x4_t vc1 = vdupq_n_f32(0.0f);
        float32x4_t vc2 = vdupq_n_f32(0.0f);
        float32x4_t vc3 = vdupq_n_f32(0.0f);

        for (int j=0;j<4;j++)
        {
            float32x4_t vb = vld1q_f32(&matB.matrix[j][0]);

            vc0 = vmlaq_f32(vc0, vdupq_n_f32(matA.matrix[0][j]), vb);
            vc1 = vmlaq_f32(vc1, vdupq_n_f32(matA.matrix[1][j]), vb);
            vc2 = vmlaq_f32(vc2, vdupq_n_f32(matA.matrix[2][j]), vb);
            vc3 = vmlaq_f32(vc3, vdupq_n_f32(matA.matrix[3][j]), vb);
        }

        vst1q_f32(&resMat.matrix[0][0], vc0);
        vst1q_f32(&resMat.matrix[1][0], vc1);
        vst1q_f32(&resMat.matrix[2][0], vc2);
        vst1q_f32(&resMat.matrix[3][0], vc3);

        return resMat;
    }

    n /=2;
    Matrix A11 = subMatrix(matA, 0, n, 0, n);
    Matrix A12 = subMatrix(matA, 0, n, n, n);
    Matrix A21 = subMatrix(matA, n, n, 0, n);
    Matrix A22 = subMatrix(matA, n, n, n, n);

    Matrix B11 = subMatrix(matB, 0, n, 0, n);
    Matrix B12 = subMatrix(matB, 0, n, n, n);
    Matrix B21 = subMatrix(matB, n, n, 0, n);
    Matrix B22 = subMatrix(matB, n, n, n, n);

```

```cpp
    Matrix S1 = dot_Product_Neon(A11,B11);
    Matrix S2 = dot_Product_Neon(A12,B21);
    Matrix S3 = dot_Product_Neon(A11,B12);
    Matrix S4 = dot_Product_Neon(A22,B21);
    Matrix S5 = dot_Product_Neon(A21,B11);
    Matrix S6 = dot_Product_Neon(A22,B21);
    Matrix S7 = dot_Product_Neon(A21,B12);
    Matrix S8 = dot_Product_Neon(A22,B22);

    Matrix C11 = S1 + S2;
    Matrix C12 = S3 + S4;
    Matrix C21 = S5 + S6;
    Matrix C22 = S7 + S8;

    return merge(C11, C12, C21, C22);
}

void Matrix::Neon_Element_Calc(Matrix *tar, const Matrix *m1, const Matrix *
   m2, int row, int col) {
    auto *v2 = new float[m2->rowNum];
    for (size_t k = 0; k < m2->rowNum; ++k)
        v2[k] = m2->matrix[k][col];
    tar->matrix[row][col] = Neon_Vector_Dot(m1->matrix[row], v2, m1->colNum);
    delete[] v2;
}

float Matrix::Neon_Vecor_Dot(const float *v1, const float *v2, const int len)
    {
    float s[4] = {0};
    float32x4_t vc0 = vdupq_n_f32(0.0f);
    float32x4_t vc1 = vdupq_n_f32(0.0f);
    float32x4_t vc2 = vdupq_n_f32(0.0f);

    for (int i = 0; i < len; i += 4) {
        vc0 = vld1q_f32(v1+i);
        vc1 = vld1q_f32(v2+i);
        vc2 = vmlaq_f32(vc2, vc0, vc1);
    }

    vst1q_f32(s, vc2);
    auto sum = s[0] + s[1] + s[2] + s[3];
    for (int i = len - 1; i >= len - len % 4; i--) sum += v1[i] * v2[i];
    return sum;
}
```

2.2 Matrix Class(Declaration)

To further explain some of the function, the codes of header file are listed below which contain the definition of the class Matrix and the declaration of its function.

```cpp
#pragma once
#include <cstdlib>
using namespace std;
```

```

#define MathType float

class Matrix {
public:
    MathType **matrix;
    int rowNum, colNum;

    Matrix(int r, int c);

    ~Matrix();

    MathType &operator()(int i, int j);

    Matrix operator+(Matrix &matA);

    Matrix operator-(Matrix &matA);

    static Matrix dot_Product(Matrix& matA,Matrix& matB);

    static Matrix dot_Product_ikj(Matrix &matA, Matrix &matB);

    static Matrix subMatrix(Matrix &matA,int rStart,int rEnd,int cStart,int
       cEnd);

    static Matrix dot_Product_Strassen(Matrix &matA,Matrix &matB);

    static Matrix merge(Matrix &mat11,Matrix &mat12,Matrix &mat21,Matrix &
       mat22);

    static Matrix dot_Product_Neon(Matrix &matA,Matrix &matB);

    static void Neon_Element_Calc(Matrix *tar, const Matrix *m1, const Matrix
        *m2, int row, int col);

    static float Neon_Vecor_Dot(const float *v1, const float *v2, const int
       len);

};
```

Class Matrix contains a two dimension array that stores the M * N elements that consists the **matrix** and also the number of **rows** and **columns**. Below is the constructor and distrcuctor of the class matrix. Also in order to implement algorithm other than the traditional one, add and subtraction operators are defined. Then is the core function of different implementations of multiplcation process, which are **the traditional $O(n^3)$ implementation**, **the memory acess optimized $O(n^3)$ implementation**, **the Strassen algorithm implementation**, **the Strassen algorith using SIMD and instruction set** and **SIMD and instruction set multiplication with multiple threads**. All of the above would be explained in the 4th *Solutions* part.

2.3 Main Class

```
#include <iostream>
#include <fstream>
#include <thread>
```

```cpp
#include "ThreadPool.h"
#include "matrix.h"

#define TIME_START start=std::chrono::steady_clock::now();
#define TIME_END(NAME) endd=std::chrono::steady_clock::now(); \
             duration=std::chrono::duration_cast<std::chrono::milliseconds>(endd-start).count();\
             cout<<(NAME) \
             <<" duration = "<<duration<<"ms"<<endl;

#define THREADS 8

int get_Row_Num(const char *string);
int get_Col_Num(const char *string);
Matrix read_Matrix(int r,int c,const char* filename);
void write_Matrix(Matrix& res,const char* filename);

using namespace std;

auto start = std::chrono::steady_clock::now();
auto endd = std::chrono::steady_clock::now();
auto duration = 0L;

int main(int argc, char const *argv[]) {

    int rowA,colA,rowB,colB;
    rowA = get_Row_Num(argv[1]);
    colA = get_Col_Num(argv[2]);
    rowB = get_Row_Num(argv[1]);
    colB = get_Col_Num(argv[2]);

    Matrix matA = read_Matrix(rowA,colA,argv[1]);
    Matrix matB = read_Matrix(rowB,colB,argv[2]);

    TIME_START

    Matrix resMatrix = Matrix::dot_Product(matA,matB);

    TIME_END("Matrix multiplication using ijk")

    TIME_START

    Matrix resMatrix2 = Matrix::dot_Product_ikj(matA,matB);

    TIME_END("Matrix multiplication using ikj")

    TIME_START

    Matrix resMatrix3 = Matrix::dot_Product_Strassen(matA,matB);

    TIME_END("Matrix multiplication using Strassen")

    TIME_START

    Matrix resMatrix4 = Matrix::dot_Product_Neon(matA,matB);

    TIME_END("Matrix multiplication using Strassen-Neon")

    TIME_START

    Matrix resMatrix5(matA.rowNum,matB.colNum);
    ThreadPool pool(THREADS);
    for (int i = 0; i < matA.rowNum; ++i) {
        for (int j = 0; j < matB.colNum; ++j) {
            pool.enqueue(Matrix::Neon_Element_Calc, &resMatrix4, &matA, &matB
               , i, j);
//             Matrix::calc_tar_pos(&resMatrix4,&matA,&matB,i,j);
        }
    }

    TIME_END("Matrix multiplication using Pure-Neon with multiple threads")

    write_Matrix(resMatrix,argv[3]);

    write_Matrix(resMatrix2,argv[4]);

    write_Matrix(resMatrix3,argv[5]);

    write_Matrix(resMatrix4,argv[6]);

    write_Matrix(resMatrix5,argv[7]);

}

Matrix read_Matrix(int r,int c,const char* filename){
    TIME_START
    ifstream infile(filename);
    Matrix mat(r,c);
    for (int i = 0; i < r; ++i) {
        for (int j = 0; j < c; ++j) {
            infile >> mat(i,j);
        }
    }
    infile.close();
    TIME_END("Reading matrix" + string(filename))
    return mat;
}

void write_Matrix(Matrix& res,const char* filename){
    TIME_START
    ofstream outfile(filename);
    outfile.setf(ios_base::fixed);
    for (int i = 0; i < res.rowNum; ++i) {
        for (int j = 0; j < res.colNum; ++j) {
            outfile << res(i,j) << " ";
        }
        outfile<<endl;
    }
    outfile.close();
    TIME_END("Writing matrix"+string(filename))
}


int get_Row_Num(const char *filename) {
    ifstream infile(filename);
    if (!infile) {
        cout << filename << " dose not exists!";
        exit(1);
    } else {
        char p;
        int rowCnt=0;
        p = infile.get();
        if (infile.eof()){
            cout<<filename<<" is empty!";
            exit(1);
        }
        else{
            while (!infile.eof()){
                p = infile.get();
                if (p=='\r'||p=='\n'){
                    rowCnt++;
                }
            }
            infile.close();
            return rowCnt;
        }
    }
}

int get_Col_Num(const char *filename) {
    ifstream infile(filename);
    char p;
    int colCnt=0;
    p = infile.get();
    while (!(p=='\r'||p=='\n') && (!infile.eof())){
        if (p==' '){
            colCnt++;
        }
        p = infile.get();
    }
    return colCnt+1;
}
```

The main class contain the reading of matrices, calling of the function and the writing of the result matrix. Also in order to show the time each step takes, a timer is defined to display the corresponding time. To take fully use of my computer performance, threadpool is used to implement multiple threads when using **neon instruction set**.

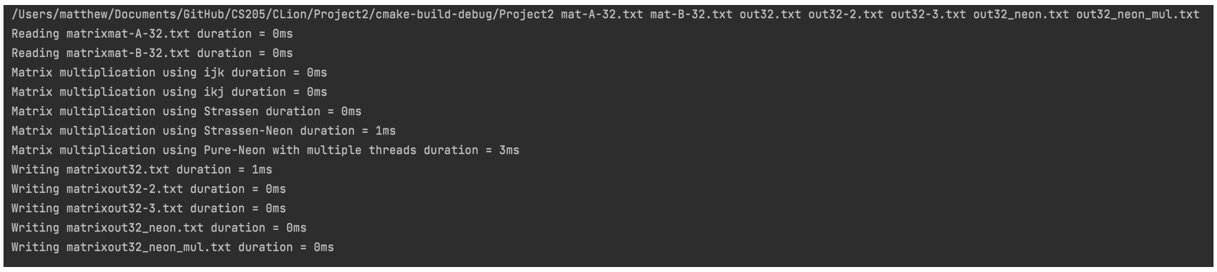
3 Performance & Result

A first declaration is that all of the performances and results below are based on the fact that all the codes are running on the MacBook Air (M1, 2020) equipped with the Apple M1 processor and 8 GB of memory that runs the macOS Big Sur Version 11.5.2.

3.1 Float Precision

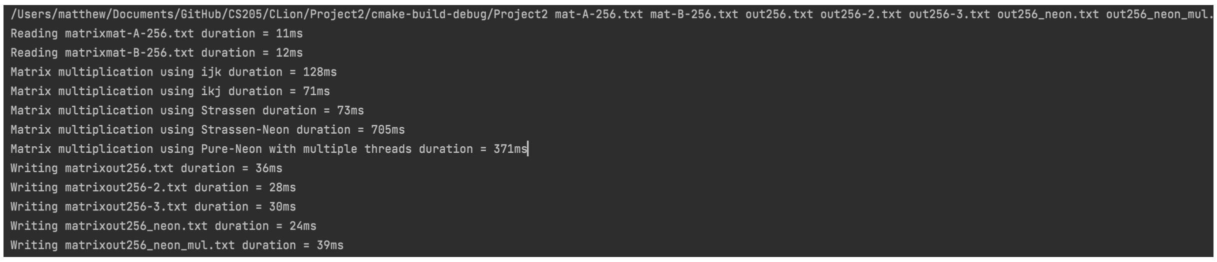
When implementing all of the construction and computation using float precision, all of the function above are called seperatly in order to compare the differences between their performance.

3.1.1 Float 32 * 32



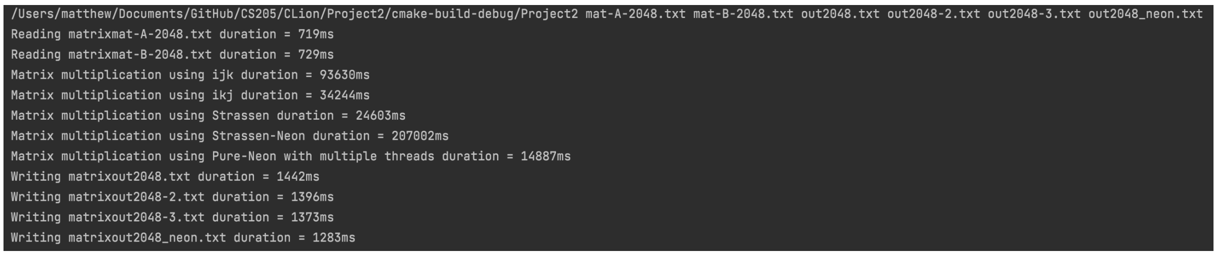
```
/Users/matthew/Documents/GitHub/CS205/CLion/Project2/cmake-build-debug/Project2 mat-A-32.txt mat-B-32.txt out32.txt out32-2.txt out32-3.txt out32_neon.txt out32_neon_mul.txt
Reading matrixmat-A-32.txt duration = 0ms
Reading matrixmat-B-32.txt duration = 0ms
Matrix multiplication using ijk duration = 0ms
Matrix multiplication using ikj duration = 0ms
Matrix multiplication using Strassen duration = 0ms
Matrix multiplication using Strassen-Neon duration = 1ms
Matrix multiplication using Pure-Neon with multiple threads duration = 3ms
Writing matrixout32.txt duration = 1ms
Writing matrixout32-2.txt duration = 0ms
Writing matrixout32-3.txt duration = 0ms
Writing matrixout32_neon.txt duration = 0ms
Writing matrixout32_neon_mul.txt duration = 0ms
```

3.1.2 Float 256 * 256



```
/Users/matthew/Documents/GitHub/CS205/CLion/Project2/cmake-build-debug/Project2 mat-A-256.txt mat-B-256.txt out256.txt out256-2.txt out256-3.txt out256_neon.txt out256_neon_mul.
Reading matrixmat-A-256.txt duration = 11ms
Reading matrixmat-B-256.txt duration = 12ms
Matrix multiplication using ijk duration = 128ms
Matrix multiplication using ikj duration = 71ms
Matrix multiplication using Strassen duration = 73ms
Matrix multiplication using Strassen-Neon duration = 705ms
Matrix multiplication using Pure-Neon with multiple threads duration = 371ms
Writing matrixout256.txt duration = 36ms
Writing matrixout256-2.txt duration = 28ms
Writing matrixout256-3.txt duration = 30ms
Writing matrixout256_neon.txt duration = 24ms
Writing matrixout256_neon_mul.txt duration = 39ms
```

3.1.3 Float 2048 * 2048



```
/Users/matthew/Documents/GitHub/CS205/CLion/Project2/cmake-build-debug/Project2 mat-A-2048.txt mat-B-2048.txt out2048.txt out2048-2.txt out2048-3.txt out2048_neon.txt
Reading matrixmat-A-2048.txt duration = 719ms
Reading matrixmat-B-2048.txt duration = 729ms
Matrix multiplication using ijk duration = 93630ms
Matrix multiplication using ikj duration = 34244ms
Matrix multiplication using Strassen duration = 24603ms
Matrix multiplication using Strassen-Neon duration = 207002ms
Matrix multiplication using Pure-Neon with multiple threads duration = 14887ms
Writing matrixout2048.txt duration = 1442ms
Writing matrixout2048-2.txt duration = 1396ms
Writing matrixout2048-3.txt duration = 1373ms
Writing matrixout2048_neon.txt duration = 1283ms
```

3.1.4 Efficiency Comparison

Apart from the reading and writing of matrix, while the program execute beyond the least precision of the timer that counts the time, the time of the 32 * 32 matrix multiplication is 0 ms, the time can be listed as below,

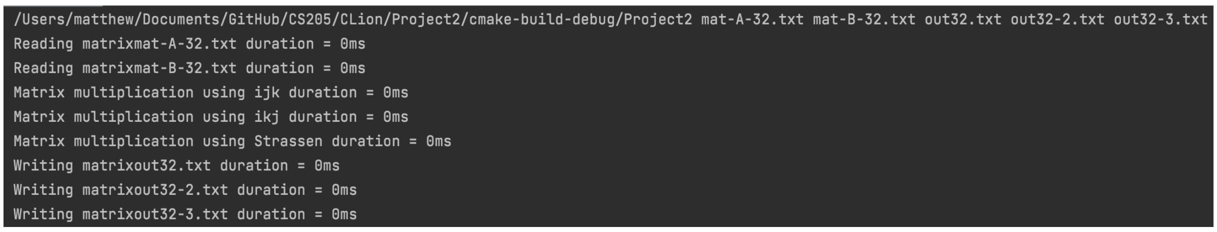
Implementation Size\ms	traditional $O(n^3)$	memory access optimized $O(n^3)$	Strassen algorithm	Strassen algorith using SIMD and instruction set	SIMD and instruction set multiplication with multiple threads
32 * 32	0	0	0	1	3
256 * 256	128	71	73	705	371
2048 * 2048	93,630	34,244	24,603	207,002	14,887

From the table above, it's obvious that the memory access optimized one has improved the speed for about 3 times, while the Strassen algorith improve the speed for about 4 times and the implementation with SIMD and multi-threads improve the speed for almost 7 times. This really is a significantly improvement. The accuracy of the implementation will be discussed later. Now let's turn to the double precision.

3.2 Double Precision

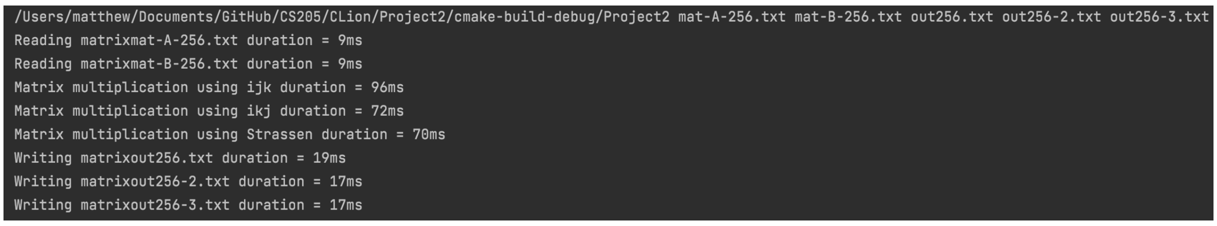
While in the float precision, five different methods are implemented to test their speeds and efficiencies. But noticing that the SIMD that uses the instruction set can only accept 4 float variable, and that isn't compatible with the double precision we needed here, therefore when operating in double precision, only three of the five implementation will be tested, the results are shown as below.

3.2.1 Double 32 * 32



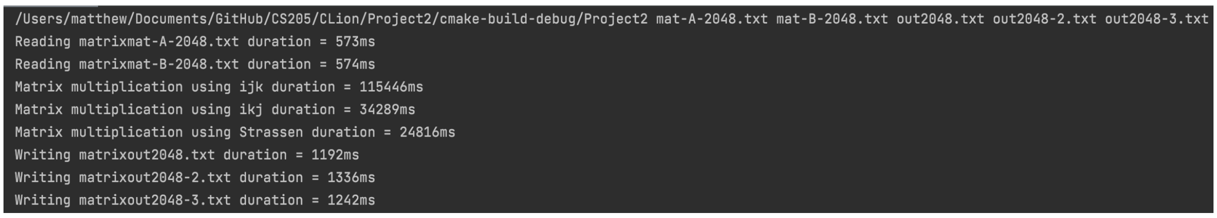
```
/Users/matthew/Documents/GitHub/CS205/CLion/Project2/cmake-build-debug/Project2 mat-A-32.txt mat-B-32.txt out32.txt out32-2.txt out32-3.txt
Reading matrixmat-A-32.txt duration = 0ms
Reading matrixmat-B-32.txt duration = 0ms
Matrix multiplication using ijk duration = 0ms
Matrix multiplication using ikj duration = 0ms
Matrix multiplication using Strassen duration = 0ms
Writing matrixout32.txt duration = 0ms
Writing matrixout32-2.txt duration = 0ms
Writing matrixout32-3.txt duration = 0ms
```

3.2.2 Double 256 * 256



```
/Users/matthew/Documents/GitHub/CS205/CLion/Project2/cmake-build-debug/Project2 mat-A-256.txt mat-B-256.txt out256.txt out256-2.txt out256-3.txt
Reading matrixmat-A-256.txt duration = 9ms
Reading matrixmat-B-256.txt duration = 9ms
Matrix multiplication using ijk duration = 96ms
Matrix multiplication using ikj duration = 72ms
Matrix multiplication using Strassen duration = 70ms
Writing matrixout256.txt duration = 19ms
Writing matrixout256-2.txt duration = 17ms
Writing matrixout256-3.txt duration = 17ms
```

3.2.3 Double 2048 * 2048



```
/Users/matthew/Documents/GitHub/CS205/CLion/Project2/cmake-build-debug/Project2 mat-A-2048.txt mat-B-2048.txt out2048.txt out2048-2.txt out2048-3.txt
Reading matrixmat-A-2048.txt duration = 573ms
Reading matrixmat-B-2048.txt duration = 574ms
Matrix multiplication using ijk duration = 115446ms
Matrix multiplication using ikj duration = 34289ms
Matrix multiplication using Strassen duration = 24816ms
Writing matrixout2048.txt duration = 1192ms
Writing matrixout2048-2.txt duration = 1336ms
Writing matrixout2048-3.txt duration = 1242ms
```

3.2.4 Efficiency Comparison

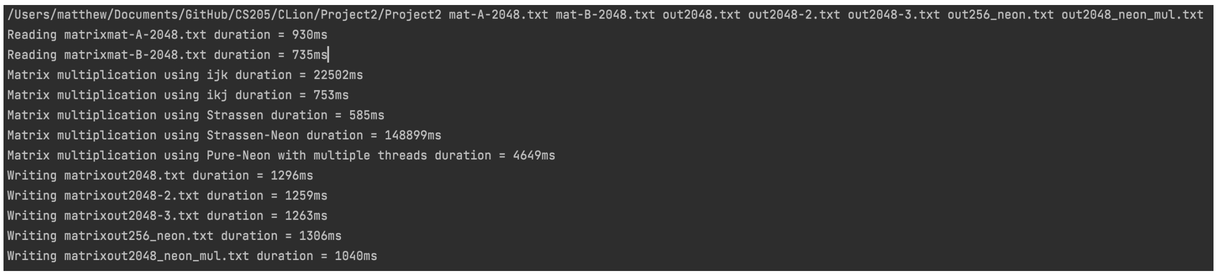
Also here the time consumption of the 32 * 32 matrices multiplication is 0 ms here, but still significant difference of time is shown between different methods, table listed as below,

Implementation Size\ms	traditional $O(n^3)$	memory access optimized $O(n^3)$	Strassen algorithm
32 * 32	0	0	0
256 * 256	96	72	70
2048 * 2048	115,446	34,289	24,816

While the difference between methods when size is 256 * 256 is not quite obvious, the difference when 2048 * 2048 is quite considerable. With the memory access optimized one improving about 4 times, the Strassen algorithm improved 5 times. The improving times is larger than the float precision, since the improving is the optimization of the time complexity, the larger the size of the problem, the better performance it reveals.

3.3 Float Precision with O3 Optimization

While the results above seem have reach the ceiling of our program, but their is another optimization that takes place when the code is compiled and generate the binary file that contains all kinds of optimization even down to the register and instrucation level. After going through some of instructions, I gradually realize that the optimization mainly works on simplifying the instructions, rearranging the sequence of the execution in order to most satisfied with the execution of the processor based on the task of each function perform, taking fully use of the registers, etc. While this takes longer to compile, the improvement of the speed and efficiency probably worth for it. Below reveal the power of O3 optimization.



```
/Users/matthew/Documents/GitHub/CS205/CLion/Project2/Project2 mat-A-2048.txt mat-B-2048.txt out2048.txt out2048-2.txt out2048-3.txt out256_neon.txt out2048_neon_mul.txt
Reading matrixmat-A-2048.txt duration = 930ms
Reading matrixmat-B-2048.txt duration = 735ms
Matrix multiplication using ijk duration = 22502ms
Matrix multiplication using ikj duration = 753ms
Matrix multiplication using Strassen duration = 585ms
Matrix multiplication using Strassen-Neon duration = 148899ms
Matrix multiplication using Pure-Neon with multiple threads duration = 4649ms
Writing matrixout2048.txt duration = 1296ms
Writing matrixout2048-2.txt duration = 1259ms
Writing matrixout2048-3.txt duration = 1263ms
Writing matrixout256_neon.txt duration = 1306ms
Writing matrixout2048_neon_mul.txt duration = 1040ms
```

Implementation Size\ms	traditional $O(n^3)$	memory access optimized $O(n^3)$	Strassen algorithm	Strassen algorithm using SIMD and instruction set	SIMD and instruction set multiplication wit multi-threads
2048 * 2048	22502	753	585	148899	4649

From the table above and compare it with the table before, there is a huge gap between them. For the first time the program is able to perform the 2048 * 2048 matrices multiplication within 1 seconds and all types of implementation have gain a huge improvement. But while here the improvement of the SIMD with multi-threads seems fail to competing with other algorithm, probably it's because of the optimization corresponding to multi-threads and SIMD is not that much improve as traditional code and algorithm. But still this is a significantly improvement!

3.4 Accuracy Comparison

Of course we all know that the accuracy of the *double* is more precise than *float*, but how does it influence the result of matrix multiplication? To verify the result of my program, I use Python's numpy to calculate the input matrices and compare my result with the numpy's result. To calculate the error, I use

$$ERROR = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}|R_{ij} - E_{ij}|}{i * j}$$

and below the table are the results.

3.4.1 Float Error

Implementation Size\ERROR	traditional $O(n^3)$	memory access optimized $O(n^3)$	Strassen algorithm	Strassen algorithm using SIMD and instruction set	SIMD and instruction set multiplication wit multi-threads
32 * 32	0.07316	0.07316	0.07316	0.05494	0.05494
256 * 256	0.34956	0.34956	0.34956	0.192203	0.192203
2048 * 2048	1.67397	1.67397	2.07203	0.843296	0.765575

While the float may cause a considerable precision loss, the error of them all seems very large. Besides since the Strassen only is different from the traditional methods when matrix size is larger than 256 * 256, the error difference only reveals in 2048 * 2048 matrices. Also thanks to the optimization of the SIMD, the float vector multiplication seems more accurate than the traditional way.

3.4.2 Double Error

Implementation Size\ERROR	traditional $O(n^3)$	memory access optimized $O(n^3)$	Strassen algorithm	Strassen algorithm using SIMD and instruction set	SIMD and instruction set multiplication wit multi-threads
32 * 32	3.146087e-06	3.146087e-06	3.146087e-06	\	\
256 * 256	1.507473e-05	1.507473e-05	1.507473e-05	\	\
2048 * 2048	2.676767e-05	2.676767e-05	2.676767e-05	\	\

After implemented the double in the computation process, the time consumptions increase but trade off with a great improvement of the precision, as all errors of the double are around 10^{-5} to 10^{-6}, there we can say that the program has reached almost as accurate as the numpy.

4 Problems & Solutions

This part mainly focus on the optimization each implementation has and analyze its cause and reasons.

4.1 Memory Access Optimization

Different from the traditional sequence that mainly used when we are computing matrices multiplication by hands, the memory access optimization change the sequence of access and computing elements on the matrix to take fully use of the cache. While the time complexity of this algorith is O(n^3), the discontinuity of the memory access can cause lower hit rate of cache. In order to improve the speed, the access of memory should be more continues in order to increase the hit rate of cache. By computing the discontinues access of memory and denote the dimension of the matrix as n, we can reckon that the traditional *ijk* sequence takes

$$n^3 + n^2 + n \text{ (2D)— } n^3 + n^2 - n\text{(1D)}$$

while another sequence *ikj* takes about

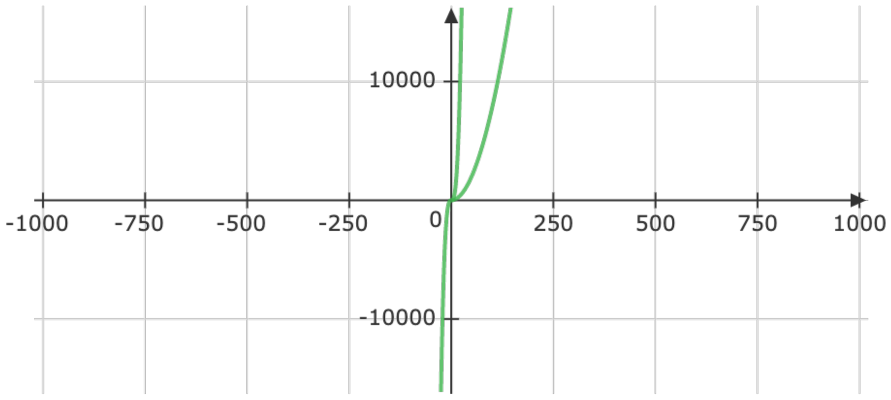
$$2n^2 + n\text{(2D)—}n^2\text{(1D)}$$

From here it's obvious that the *ikj* sequence greatly reduce the time of discontinues accessing the memory and therefore taking fully advantages of the high speed cache besides the processor. From the Part3 display, we can discover that the optimized memory accessing methods always is 3 to 4 times faster than the tradional sequence accessing method. This method can be listed as a optimization around *IO Speed* and achieved a well improvement.

4.2 Strassen Algorithm

While the traditional method's time complexity is $O(n^3)$, the Strassen algorithm[2] is about $O(n^{\log_2 7})$. The decrease of the time complexity has a more powerful effect when the problem size is quite large. Therefore in Part3 comparisons, the Strassen algorithm didn't show its advantages until the size of the matrix reached about 256 * 256.

Since Strassen is done in a recursive way, and the smaller the matrix is divided into, the more level of stack it enters, this will always cause a greate time consumption. Therefore in order to increase the speed, here I only use Strassen decomposition when the matrix is larger than 256 * 256, for the smaller matrices, memory access optimized matrix multiplication is used to compute and then merge the smaller one into the final result.



4.3 SIMD and instruction set

While most of the processors supports the SIMD, which takes only one instruction to operate on 128 bits for a single time, this can reduce the iteration times and increase the speed by reducing the constant of the time complexity. For the Arm processor, the neon instruction offers such way to calculate 4 float(32 bit) variables at the same time and reduce the constant by 4 times.

This method[3] mainly works like transforming the traditional loop

```
for(int i = 0, i < rowNum, ++i){
    C[i][j] += A[i][k] * B[k][j];
}
```

into a more convinient and faster one like this

```
if (n<=4){
        Matrix resMat(matA.rowNum,matB.colNum);
        float32x4_t vc0 = vdupq_n_f32(0.0f);
        float32x4_t vc1 = vdupq_n_f32(0.0f);
        float32x4_t vc2 = vdupq_n_f32(0.0f);
        float32x4_t vc3 = vdupq_n_f32(0.0f);

```

[2]详解矩阵乘法中的 Strassen 算法. (2021). 知乎专栏. https://zhuanlan.zhihu.com/p/78657463

[3]B. (2020). how-to-optimize-gemm/armv7a/src at master · BBuf/how-to-optimize-gemm. GitHub. https://github.com/BBuf/how-to-optimize-gemm/tree/master/armv7a/src

```
        for (int j=0;j<4;j++)
        {
            float32x4_t vb = vld1q_f32(&matB.matrix[j][0]);

            vc0 = vmlaq_f32(vc0, vdupq_n_f32(matA.matrix[0][j]), vb);
            vc1 = vmlaq_f32(vc1, vdupq_n_f32(matA.matrix[1][j]), vb);
            vc2 = vmlaq_f32(vc2, vdupq_n_f32(matA.matrix[2][j]), vb);
            vc3 = vmlaq_f32(vc3, vdupq_n_f32(matA.matrix[3][j]), vb);
        }

        vst1q_f32(&resMat.matrix[0][0], vc0);
        vst1q_f32(&resMat.matrix[1][0], vc1);
        vst1q_f32(&resMat.matrix[2][0], vc2);
        vst1q_f32(&resMat.matrix[3][0], vc3);

        return resMat;
    }
```

The program first apply for four float storage sets[4] and set them to zero, then we copy the elements of the operating matrix four at a time and use the instruction *vmlaq_f32(sum, a, b)* to compute the product of a and b then add it on to the sum. This is exactly what we want to do in the *for-loop*. After computing all of the elements on the result matrix, the result is than copy from the f32 sets into the corresponding memory address in the result matrix we define.

But as the operation can only manipulate 128 bits at a time, we can only use the float precision and get the prduct of 4 elements at the same time. Therefore for the larger matrices we have to decomposite them into smaller sub-matrix that are size 4 * 4 for computing.

During the test, when computing 4 * 4 size matrices, the SIMD is about 3 - 4 times faster than the traditional methods, which can be explained that the SIMD had the four operations done in a single one and this has reduce the time for about 4 times. The time complexity of this is still $O(n^3)$, but with a smaller constant that is about $\frac{1}{4}$ of the previous one.

But when computing the larger matrices such as 2048 * 2048, since we have to divide the matrix size 2 at a time, and that produce 4 sub-matrices, therefore there will be 4^9 which will be 262,144 matrices to compute. Although each 4 * 4 matrix only takes 4 * 4 operation to done, the total operation is still about $4.19 * 10^6$ operation. To include the cost to divide the matrix and the matrix merge cost, also the cost of enter and exit the stack, the final cost of the method is significantly high that the running time is over 200s. Then to fully use the of the SIMD, another implementation is needed.

4.4 SIMD and instruction set using Multi-threads

The traditional method takes one row from the first operating matrix and multiply it with one colomn from the second operating matrix, then fill the value into the corresponding position of the result matrix. Therefore we can divide a row and a column into several length 4 segments and multiply them seperately, then we add their values together and produce the final result.

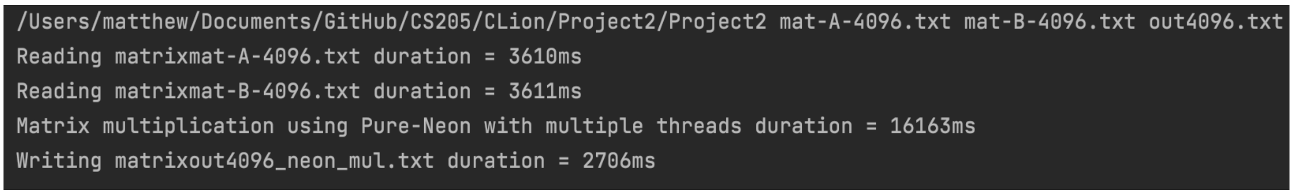
To fully implement the SIMD, the program needs to fetch one column from the second matrix and calculate dot product with each row of the first matrix. While the first process involves in discontinuesly accessing the memory, the second step can be done with the SIMD. The first step corresponding to the function *Neon_Element_Calc()* and the second step involves the function *Neon_Vecor_Dot*, which is basically the same as the 4 * 4 multiplication above.

[4]Arm Ltd. (2020). SIMD ISAs | Neon –. Arm Developer. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon

Until now we can fully use the SIMD without gain redundant costs, but the speed is still not very ideal, therefore multi-threads is used here to increase the calculation efficiency of the SIMD. Here I use a github project[5] which offers a *ThreadPool.h* file to directly use the multi-threads to calculate the result elements of matrix.

```
Matrix resMatrix5(matA.rowNum,matB.colNum);
    ThreadPool pool(THREADS);
    for (int i = 0; i < matA.rowNum; ++i) {
        for (int j = 0; j < matB.colNum; ++j) {
            pool.enqueue(Matrix::Neon_Element_Calc, &resMatrix5, &matA, &matB
                , i, j);
//              Matrix::calc_tar_pos(&resMatrix4,&matA,&matB,i,j);
        }
    }
```

In fact, multi-threads significantly increase the programs efficiency since the use rate of the CPU reached about 90%, this is a greate improve because when running the previous methods, the use rate of CPU rarely gain over 30%.



```
/Users/matthew/Documents/GitHub/CS205/CLion/Project2/Project2 mat-A-4096.txt mat-B-4096.txt out4096.txt
Reading matrixmat-A-4096.txt duration = 3610ms
Reading matrixmat-B-4096.txt duration = 3611ms
Matrix multiplication using Pure-Neon with multiple threads duration = 16163ms
Writing matrixout4096_neon_mul.txt duration = 2706ms
```

But using the multi-threads also needs the compatible threads number to achieve the maximum use rate of CPU. It's not that the higher the threads number is, the better efficiency it has. For my situation, 8 threads is a compatible condition.

4.5 Other Details

Apart from the efficiency improvement above, other details are also impelmented into the program which enable the program to be more reliable and compatible. Including reading the rows and columns before store the matrix in order to decide the size of the matrices, matching the size of the matrices before multiplicate them in order to avoid the breakdown caused by the wrong size multiplication, the distructor that distruct the useless matrices to avoid memory leak, etc. All of the features ensure the program to run more reliably and efficiently.

5 Summary

This project seems like a traditional project, but actually it's a perfect chance for me to learn how to improve and amend a program from the algorithm level and also the hardware level. A deeper understanding of the memory access and instruction set has achieved when I did research on the different optimization. Also the Strassen algorithm offers a new way of optimizing using recursive. There is nothing more satisfying than watching the program optimizated both algorithm way and hardware level reached much faster speed and a higher accuracy!

[5]P. (2021). GitHub - progschj/ThreadPool: A simple C++11 Thread Pool implementation. GitHub. https://github.com/progschj/ThreadPool