Contents

1 Requirement Analysis

The concise purpose of the project is to implement GEMM for double matrices multiplication referring to cblas_dgemm() in openblas library and focuses on improving the efficiency of function.

1.1 Understanding cblas_dgemm() in openblas

According to the requirement, the interface of the function should be consistent with the openblas library. (cblas_dgemm() as an example)

```
void cblas_dgemm(const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE TransA,
                 const CBLAS_TRANSPOSE TransB, const int M, const int N,
                 const int K, const double alpha, const double  *A,
                 const int lda, const double  *B, const int ldb,
                 const double beta, double  *C, const int ldc);
```

The meanings of its parameters are explained as follows.

Order: Specifies the storage order of the matrices. It can be either CblasRowMajor or CblasColMajor.

TransA: Specifies whether matrix A should be transposed before the multiplication. It can be CblasNoTrans for no transpose, CblasTrans for transpose, or CblasConjTrans for conjugate transpose.

TransB: Specifies whether matrix B should be transposed before the multiplication. It follows the same options as TransA.

By choosing TransA and TransB, we can compute the matrix in these four ways:

$$C = \alpha AB + \beta C;\ C = \alpha A^T B + \beta C;\ C = \alpha AB^T + \beta B;\ C = \alpha A^T B^T + \beta B.$$

M: Specifies the number of rows of matrix A and matrix C.

N: Specifies the number of columns of matrix B and matrix C.

K: Specifies the number of columns of matrix A and the number of rows of matrix B.

alpha: Specifies the scalar value to scale the matrix product.

A: Pointer to the first element of matrix A.

lda: Specifies the leading dimension (stride) of matrix A. If Order is CblasRowMajor, lda should be at least K. If Order is CblasColMajor, lda should be at least M.

B: Pointer to the first element of matrix B.

ldb: Specifies the leading dimension (stride) of matrix B. If Order is CblasRowMajor, ldb should be at least N. If Order is CblasColMajor, ldb should be at least K.

beta: Specifies the scalar value to scale matrix C before adding the matrix product.

C: Pointer to the first element of matrix C.

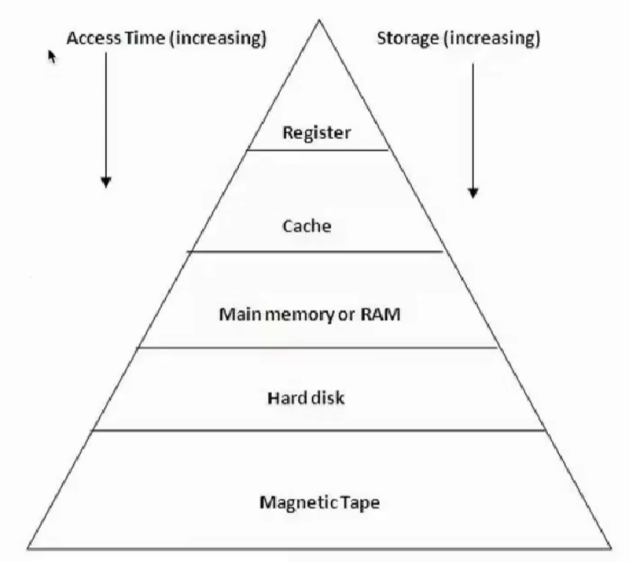
ldc: Specifies the leading dimension (stride) of matrix C. If Order is CblasRowMajor, ldc should be at least N. If Order is CblasColMajor, ldc should be at least M.

1.2 Improve the Efficiency

This project focuses on improving the efficiency of our matrix multiplication function.

According to what we have learned in this course, this project focuses on improving the speed without changing the algorithm. There are 2 major aspects that need to be improved in order to boost the efficiency of matrix multiplication.

Firstly, optimize the access to inventory. As we all know, the access architecture of the computer is shown in the following figure. From the figure, we can easily find out that the capacity increases the further down the pyramid you go but the speed of computation becomes slower. Thus, in order to decrease the time accessing the memory, we can use some ways to make full use of the register and cache. We can change the loop sequence, implement matrix blocking, implement matrix packing, use pointers, use register variables, and align memory to realize the goal.



Secondly, improve the speed of computation. Doing multiple computations in one step can improve the overall speed of computation. We can make use of SIMD vectorization, loop unrolling, and OpenMP Multithreading to realize our goal.

Moreover, making use of the automatic optimization of the compiler is a very important aspect of improving the efficiency of our functions. In C++, it provides O0, O1, O2, and O3 with these different levels of compiler optimization. According to the information online, we find that -O3 will Optimize your program to the fullest extent by using loop unrolling, inline function, SIMD vectorization, and Automatic Parallelization.

2 Code Implementation

2.1 Get Random Double Matrix

The Random() function obtains random numbers by calculating formulas and obtaining the seeds of random numbers through the system clock. (In this project the range of the random double is from 0 to 255)

```
#define randomInt(a,b)       (rand()% (b - a) + (a) ) //Get int from a to b
#define randomDouble(RAND_MAX) (rand() / double(RAND_MAX)) //Get double from 0 to 1

double* Random(size_t row, size_t col)
{
    if(row == 0 || col == 0){
        cerr << "File " << __FILE__ << "Line " << __LINE__ << "Func " << __FUNCTION__ << "()" << endl;
        cerr << "The input (" << row << ", " << col << ") is invalid" << endl;
        exit(EXIT_FAILURE);
    }
    double* data = new double[row*col];
    if(data==NULL)
    {
        cerr << "File " << __FILE__ << "Line " << __LINE__ << "Func " << __FUNCTION__ << "()" << endl;
        cerr << "Fail to allocate memory for data" << endl;
        exit(EXIT_FAILURE);
    }
    srand((unsigned)time(NULL));
    #pragma omp parallel for schedule(dynamic)
    for(size_t i=0; i < row*col; i++)
        *(data + i) = randomDouble(RAND_MAX) + randomInt(0, 10);
    return data;
}
```

2.2 Final DGEMM

The final DGEMM is consistent with cblas_dgemm() in the openblas library.

```
void cblas_dgemm(const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE TransA,
                 const CBLAS_TRANSPOSE TransB, const int M, const int N,
                 const int K, const double alpha, const double  *A,
                 const int lda, const double  *B, const int ldb,
                 const double beta, double  *C, const int ldc);
```

Parameter consistency:

The first parameter is a const enum type, which is intended to indicate whether the incoming matrix stores data in rows or columns. It is worth noting that the input and output matrices have the same type of data stored in them, so it is not necessary to distinguish whether it is stored in rows or columns when multiplying matrices inside the function.

The second and third parameters are const enum type, which is intended to tell us whether the incoming matrix needs to be transposed when multiplying it.

The latter parameters are intended to tell us information about the matrix as well as to tell us the values of the alpha and beta factors.

The specific code implementations are as follows. (The more specific design ideas are mentioned in Part 3.10.1)

```
typedef enum {RowMajor, ColMajor} Order;
typedef enum {NoTrans, Trans, ConjTrans} TransType;

#define left(i, j) A[i * K + j]
#define right(i, j) B[i * N + j]
#define res(i, j) C[i * N + j]

bool dgemm_final(Order order, TransType TransA, TransType TransB, const size_t M, const size_t N,
```

```
                const size_t K, const double alpha, const double* A, const size_t lda, const double* B,
                const size_t ldb, const double beta, double* C, const size_t ldc)
{
    if(M == 0 || N == 0 || K == 0 || lda == 0 || ldb == 0 || (M != N) || (M != K))
    {
        cerr << "File " << __FILE__ << " Line " << __LINE__ << " Func " << __FUNCTION__ << "()" << endl;
        cerr << "The sizes of the matrix are wrong ( " << M << ", " << N << ", " << K << ", " << lda << ", "
        << ldb << ")" << endl;
        exit(EXIT_FAILURE);
    }
    if(A == NULL || B == NULL || C == NULL)
    {
        cerr << "File " << __FILE__ << " Line " << __LINE__ << " Func " << __FUNCTION__ << "()" << endl;
        cerr << "NULL pointer!" << endl;
        exit(EXIT_FAILURE);
    }
    if(M < 512)
    {
        if(TransA == NoTrans && TransB == NoTrans)
        {
            #pragma omp parallel for schedule(dynamic)
            for(size_t r = 0; r < M; r++)
                for(size_t k =0; k < K; k++)
                    for(size_t c = 0; c < N; c++)
                        res(r, c) = beta * res(r, c) + alpha * left(r, k) * right(k, c);
        }
        else if((TransA == Trans || TransB ==  ConjTrans) && TransB == NoTrans)
        {
            #pragma omp parallel for schedule(dynamic)
            for(size_t k =0; k < K; k++)
                for(size_t r = 0; r < M; r++)
                    for(size_t c = 0; c < N; c++)
                        res(r, c) = beta * res(r, c) + alpha * left(r, k) * right(k, c);
        }
        else if(TransA == NoTrans && (TransB == Trans || TransB == ConjTrans))
        {
            #pragma omp parallel for schedule(dynamic)
            for(size_t r = 0; r < M; r++)
                for(size_t c = 0; c < N; c++)
                    for(size_t k =0; k < K; k++)
                        res(r, c) = beta * res(r, c) + alpha * left(r, k) * right(k, c);
        }
        else if((TransA == Trans || TransB ==  ConjTrans) && (TransB == Trans || TransB == ConjTrans))
        {
            #pragma omp parallel for schedule(dynamic)
            for(size_t r = 0; r < M; r++)
                for(size_t k =0; k < K; k++)
                    for(size_t c = 0; c < N; c++)
                        res(r, c) = beta * res(r, c) + alpha * left(r, k) * right(k, c);
        }
        else
        {
            cerr << "File " << __FILE__ << " Line " << __LINE__ << " Func " << __FUNCTION__ << "()" << endl;
            cerr << "Wrong Matrix TransType input" << endl;
            exit(EXIT_FAILURE);
        }
    }
    else
    {
        if(TransA == NoTrans && TransB == NoTrans)
        {
            double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            #pragma omp parallel for schedule(dynamic)
            for(size_t r = 0; r < M; r+=32)
                for(size_t k =0; k < K; k+=32)
                {
                    for(size_t c = 0; c < N; c+=32)
                    {
                        for(size_t k0 = 0; k0 < 32; k0++)
                            for(size_t c0 = 0; c0 < 32; c0++)
                                right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                            for(size_t k0 = 0; k0 < 32; k0++)
                                left_packed[r0 * 32 + k0] = left(r + r0, k + k0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                        {
                            for(size_t k0 = 0; k0 < 32; k0++)
                            {
                                register double temp1 = left_packed[r0 * 32 + k0];
                                for(size_t c0 = 0; c0 < 32; c0++)
                                    res(r + r0, c + c0) = alpha * (temp1 * right_packed[k0 * 32 + c0])
                                    + beta * res(r + r0, c + c0);
                            }
                        }
                    }
                }
```

```
            delete[] right_packed;
            delete[] left_packed;
        }
        else if((TransA == Trans || TransB ==  ConjTrans) && TransB == NoTrans)
        {
            double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            #pragma omp parallel for schedule(dynamic)
            for(size_t r = 0; r < M; r+=32)
                for(size_t k =0; k < K; k+=32)
                {
                    for(size_t c = 0; c < N; c+=32)
                    {
                        for(size_t k0 = 0; k0 < 32; k0++)
                            for(size_t c0 = 0; c0 < 32; c0++)
                                right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                            for(size_t k0 = 0; k0 < 32; k0++)
                                left_packed[r0 * 32 + k0] = left(k + k0, r + r0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                        {
                            for(size_t k0 = 0; k0 < 32; k0++)
                            {
                                register double temp1 = left_packed[r0 * 32 + k0];
                                for(size_t c0 = 0; c0 < 32; c0++)
                                    res(r + r0, c + c0) = alpha * (temp1 * right_packed[k0 * 32 + c0])
                                    + beta * res(r + r0, c + c0);
                            }
                        }
                    }
                }
            delete[] right_packed;
            delete[] left_packed;
        }
        else if(TransA == NoTrans && (TransB == Trans || TransB == ConjTrans))
        {
            double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            #pragma omp parallel for schedule(dynamic)
            for(size_t r = 0; r < M; r+=32)
                for(size_t k =0; k < K; k+=32)
                {
                    for(size_t c = 0; c < N; c+=32)
                    {
                        for(size_t k0 = 0; k0 < 32; k0++)
                            for(size_t c0 = 0; c0 < 32; c0++)
                                right_packed[k0 * 32 + c0] = right(c + c0, k + k0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                            for(size_t k0 = 0; k0 < 32; k0++)
                                left_packed[r0 * 32 + k0] = left(r + r0, k + k0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                        {
                            for(size_t k0 = 0; k0 < 32; k0++)
                            {
                                register double temp1 = left_packed[r0 * 32 + k0];
                                for(size_t c0 = 0; c0 < 32; c0++)
                                    res(r + r0, c + c0) = alpha * (temp1 * right_packed[k0 * 32 + c0])
                                    + beta * res(r + r0, c + c0);
                            }
                        }
                    }
                }
            delete[] right_packed;
            delete[] left_packed;
        }
        else if((TransA == Trans || TransB ==  ConjTrans) && (TransB == Trans || TransB == ConjTrans))
        {
            double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            #pragma omp parallel for schedule(dynamic)
            for(size_t r = 0; r < M; r+=32)
                for(size_t k =0; k < K; k+=32)
                {
                    for(size_t c = 0; c < N; c+=32)
                    {
                        for(size_t k0 = 0; k0 < 32; k0++)
                            for(size_t c0 = 0; c0 < 32; c0++)
                                right_packed[k0 * 32 + c0] = right(c + c0, k + k0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                            for(size_t k0 = 0; k0 < 32; k0++)
                                left_packed[r0 * 32 + k0] = left(k + k0, r + r0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                        {
                            for(size_t k0 = 0; k0 < 32; k0++)
                            {
```

```
                                register double temp1 = left_packed[r0 * 32 + k0];
                                for(size_t c0 = 0; c0 < 32; c0++)
                                    res(r + r0, c + c0) = alpha * (temp1 * right_packed[k0 * 32 + c0])
                                    + beta * res(r + r0, c + c0);
                            }
                        }
                    }
                }
            delete[] right_packed;
            delete[] left_packed;
        }
        else
        {
            cerr << "File " << __FILE__ << " Line " << __LINE__ << " Func " << __FUNCTION__ << "()" << endl;
            cerr << "Wrong Matrix TransType input" << endl;
            exit(EXIT_FAILURE);
        }
    }
    return true;
}
```

2.3 Write data into a file

When conducting a large number of experiments that collect a large amount of data, we can write our data to a CSV file, which is more convenient for us to process the data.

If we do not have a file, we can create a CSV named 'TimeCost.csv' through ofstream object. Then we write our data into the file by ¡¡ operator, using ',' to separate the data in a line, and using 'endl' to write the data in the next line.

```
ofstream file("TimeCost.csv");
if(!file.is_open())
    ......
file << "data1" << "," << "data2" << endl;
file.close();
```

2.4 Compile the file & CMakeLists.txt

This project needs to link openblas library, implement SIMD instructions and implement OpenMP instructions. Thus, there are servals things needed to link when compiling the code.

```
//On x86
g++ mian.cpp mul.cpp -DWITH_AVX2 -mavx -fopenmp -lopenblas -O3
//On arm
g++ mian.cpp mul.cpp -DWITH_NEON -fopenmp -lopenblas -O3
```

As it is shown above, it is tedious to write all the commands in the command window by ourselves. Writing a CMakeLists.txt is a good way to solve this problem.

```
cmake_minimum_required(VERSION 3.12)

add_definitions(-DWITH_AVX2)
add_definitions(-mavx)
add_definitions(-O3)

set(CMAKE_CXX_STANDARD 11)

project(pro)

ADD_EXECUTABLE(pro main.cpp mul.cpp)

set(BLA_VENDER OpenBLAS)
find_package(BLAS REQUIRED)
if (BLAS_FOUND)
    #message(STATUS "BLAS library found: ${BLAS_LIBRARIES}")
    target_link_libraries(pro /usr/lib/x86_64-linux-gnu/libopenblas.so)
    message("openblas Found")
else()
    message("Can't find Openblas library")
endif()

find_package(OpenMP)
if(OpenMP_CXX_FOUND)
    message("OpenMP found.")
    target_link_libraries(pro OpenMP::OpenMP_CXX)
endif()
```

3 Experiments & Analysis

In this project, we analyzed the principle of various methods of accelerated matrix multiplication, the code implementation and its correctness check, the speed optimization experiment, and its result analysis. (All the functions are checked to be correct, only the Final DGEMM Function correctness is mentioned in the report)

We check the correctness of the code by comparing the results calculated by our function with the results calculated by the function in openblas. When the size of the matrix is relatively small, we use the function printInfo() to output the data of the matrix to find the differences by ourselves.

```
bool printInfo(const size_t row, const size_t col, const double* data);
```

When the size of the matrix is relatively large, we output the result into a TXT file using ofstream object, and we can use the built-in Git tool and Diff comparison feature in Visual Studio Code (VS Code) to compare two documents and show the differences.

To explore the efficiency of a function doing matrix multiplication, we first generate a random array of the specified size. Then calculate the elapsed time doing the matrix multiplication. Finally, use the number of floating-point operations per unit of time to evaluate its efficiency.

We obtain random doubles by the Random() function. (In this project the range of the random double is from 0 to 255)

```
double* Random(size_t row, size_t col);
```

Since this program is running under a Linux environment, we use the Linux environment timing function gettimeofday() to time the program. The timing function has a high accuracy of microseconds. This function gets the elapsed time and time zone (UTC time) from January 1, 1970 to the present, but according to the official Linux documentation, the time zone is no longer used, so just pass NULL when using it. The difference between the current time before the convolution operation and the current time after the convolution operation is completed is the difference between them, which is the program's running time.

```
#include <sys/time.h>
struct timeval start, end;
gettimeofday(&start, NULL);
//Convolution operation
gettimeofday(&end, NULL);
printf("Time Cost : %lf s\n",(end.tv_sec - start.tv_sec) + (double)(end.tv_usec - start.tv_usec) / 1000000.0);
```

We use Giga Floating-point Operations Per Second(GFLOPS) to measure the efficiency of the function doing matrix multiplication. For A_{mxk} and matrix B_{kxn} doing multiplication, there are $2 * m * n * k$ floating-point operations. If the computation time is s, then GFLOPS is the total number of floating-point operations. then GFLOPS is

$$GFLOPS = \frac{2 * n * m * k}{10^9 * s}$$

Simultaneously, we compare our best function with the function in openblas library.

Ultimately, we compare the differences between x86 and arm.

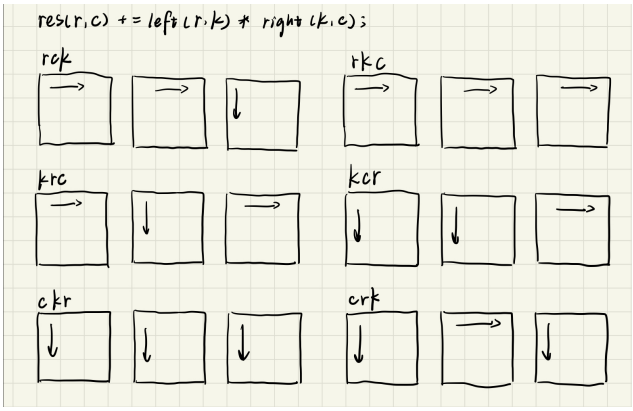
3.1 Change loop sequence

In this experiment, we compare the speed of matrix multiplication among different loop sequences (rkc, rck, krc, kcr, ckr, crk).

3.1.1 Theorem Analysis

The matrices in this experiment are stored by rows, and both matrices A and B are untransposed. Therefore, when we access the elements in the matrix in the order of rows, the memory accessed is continuous, which

will reduce the number of cache misses and theoretically bring speedup, so it can be inferred that rkc is the fastest and ckr is the slowest. At the same time, from the calculation formula can be obtained, for the result matrix, and the right matrix are using the variable c to bring to access the elements in the order of rows, and the left matrix is using the variable k to access the elements in the order of rows, and when the number of cycles in each layer is the same, the closer the inner layer will have more cycles, so we can infer that rck is slower than krc, and crk is slower than kcr.



3.1.2 Code Implementation

The Code implementation is below.

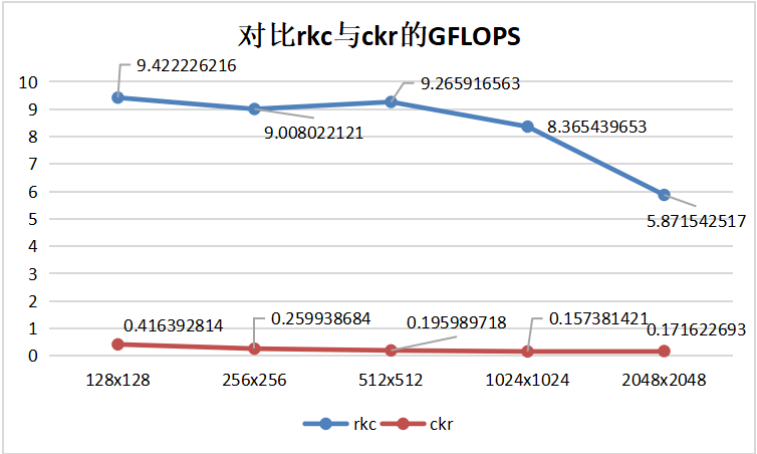
```
//rck
for(size_t r = 0; r < M; r++)
            for(size_t c = 0; c < N; c++)
                for(size_t k =0; k < K; k++)
                    res(r, c) += left(r, k) * right(k, c);

//rkc
for(size_t r = 0; r < M; r++)
            for(size_t k =0; k < K; k++)
                for(size_t c = 0; c < N; c++)
                    res(r, c) += left(r, k) * right(k, c);

//krc
for(size_t k =0; k < K; k++)
            for(size_t r = 0; r < M; r++)
                for(size_t c = 0; c < N; c++)
                    res(r, c) += left(r, k) * right(k, c);

//kcr
for(size_t k =0; k < K; k++)
            for(size_t c = 0; c < N; c++)
                for(size_t r = 0; r < M; r++)
                    res(r, c) += left(r, k) * right(k, c);

//ckr
for(size_t c = 0; c < N; c++)
            for(size_t k =0; k < K; k++)
                for(size_t r = 0; r < M; r++)
                    res(r, c) += left(r, k) * right(k, c);

//crk
for(size_t c = 0; c < N; c++)
            for(size_t r = 0; r < M; r++)
                for(size_t k =0; k < K; k++)
                    res(r, c) += left(r, k) * right(k, c);
```

3.1.3 Results Analysis

Functions dgemm_rck(), dgemm_rkc(), dgemm_krc(), dgemm_kcr(), dgemm_ckr() and dgemm_crk() are compared in this experiment.

The results are as follows.

不同循环次序下不同规模的矩阵进行乘法的平均耗时（s）					
矩阵大小	128x128	256x256	512x512	1024x1024	2048x2048
rck	0.002759	0.03588585	0.406535	4.544402	51.71665
rkc	0.00044515	0.00372495	0.0289702	0.256709	2.925955
krc	0.00043095	0.0034447	0.0287623	0.2767148	6.517475
kcr	0.0098668	0.1262078	1.364236	13.815	104.2089
ckr	0.01007295	0.12908595	1.3696405	13.64509	100.10255
crk	0.0028347	0.0383602	0.44038385	4.990566	55.34555



The results are generally correct and correspond to our theorem analysis.

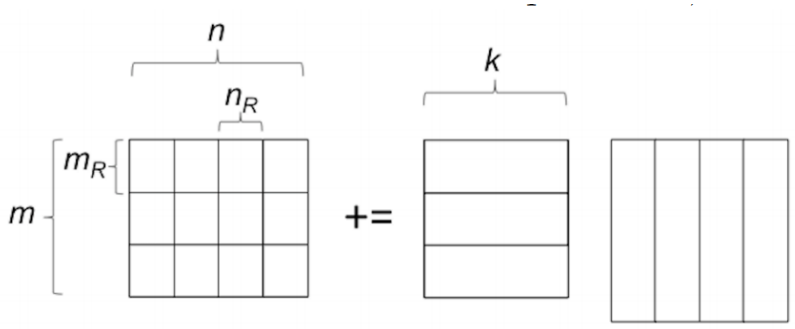
3.2 Matrix Blocking

In this experiment, we compare the speed of matrix multiplication among different blocking sizes: 4x4, 16x16, 32x32, 64x64, and 128x128.

3.2.1 Theorem Analysis

Our computers generally have three caches, L1, L2, and L3, and it is much faster to perform data computation when the data is in the cache. Based on the above information, the idea of matrix chunking to speed up matrix multiplication is to divide the matrix into small blocks so that when computing a small block of the matrix, the matrix can all be stored in the cache, thus speeding up matrix multiplication. Moreover, it can also reduce the time of cache misses and speed up matrix multiplication.

The principle is as follows.



$$\text{A} = \begin{pmatrix} a_{0,0} & \cdots & a_{0,K-1} \\ \vdots & \ddots & \vdots \\ a_{M-1,0} & \cdots & a_{M-1,K-1} \end{pmatrix}, \text{B} = \begin{pmatrix} b_{0,0} & \cdots & b_{0,N-1} \\ \vdots & \ddots & \vdots \\ b_{K-1,0} & \cdots & b_{K-1,N-1} \end{pmatrix}, \text{C} = \begin{pmatrix} c_{0,0} & \cdots & c_{0,N-1} \\ \vdots & \ddots & \vdots \\ c_{M-1,0} & \cdots & c_{M-1,N-1} \end{pmatrix}$$

where $C_{i,j}$ is m_i x n_j, $A_{i,p}$ is m_i x k_p, and $B_{p,j}$ is k_p x n_j. Then,

$$C_{i,j} = \sum_{p=0}^{K-1} A_{i,p} B_{p.j} + C_{i,j}$$

3.2.2 Code Implementation

The Code implementation is below.

```
//T is the size of the blocking
//TxT
for(size_t r = 0; r < M; r+=T)
    for(size_t k =0; k < K; k+=T)
        for(size_t c = 0; c < N; c+=T)
            for(size_t r0 = 0; r0 < T; r0++)
                for(size_t k0 = 0; k0 < T; k0++)
                    for(size_t c0 = 0; c0 < T; c0++)
                        res(r + r0, c + c0) += left(r + r0, k + k0) * right(k + k0, c + c0);
```

Correctness:

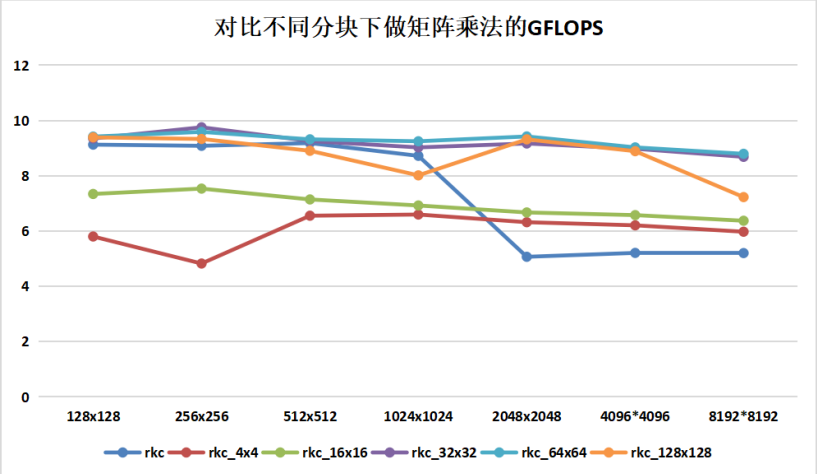
3.2.3 Results Analysis

Functions dgemm_rkc_4x4(), dgemm_rkc_16x16(), dgemm_rkc_32x32(), dgemm_rkc_64x64(), and dgemm_rkc_128x128() are compared in this experiment.

The results are as follows.

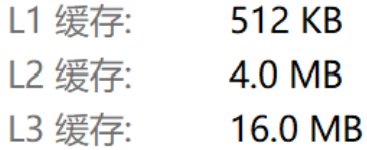
不同矩阵分块下不同规模的矩阵进行乘法的平均耗时（s）

	128x128	256x256	512x512	1024x1024	2048x2048	4096*4096	8192*8192
rkc	0.0004596	0.003694	0.0292324	0.2462516	3.390574	26.393175	211.403
rkc_4x4	0.000723	0.006957	0.040959	0.325554	2.7191	22.1473	184.055
rkc_16x16	0.0005714	0.0044535	0.0375832	0.3101456	2.575044	20.9029	172.571
rkc_32x32	0.0004487	0.00344	0.0289944	0.2379222	1.87405	15.309575	126.608
rkc_64x64	0.0004454	0.0034999	0.0288086	0.2322122	1.823338	15.22665	125.031
rkc_128x128	0.0004467	0.003596	0.030149	0.2679758	1.843848	15.46145	152.136



From the graph, we can see that matrix chunking does not give a speedup to matrix multiplication when the size of the matrix is less than 1024. When the size of the matrix exceeds 1024, the GFLOPS of the algorithm without chunking decreases rapidly, probably because when it is less than 1024, the data of the matrix can all be loaded into the cache, so there is not much difference between chunking and not chunking, but chunking will bring a little more time-consuming. However, when the size of the matrix is larger than 1024, chunking will significantly reduce cache misses, thereby accelerating the multiplication of the matrix.

By looking at the configuration of the computer where the experiment was conducted, I found that the size of its L1 cache is 512 KB and the size of the L2 cache is 4M.



Since 1KB = 1024 bytes, 1M = 1024 bytes, the two caches are 4,718,592 bytes in total, which can store 589,824 doubles, and when the size of the matrix is 512x512 When the matrix size is 512x512, there are 262,144 doubles, which is less than the size of the two caches, while at 1024x1024, there are 1,048,576 doubles, which is more than the size of the two caches. Our analysis fits with the results.

Also through the chart, we can find that when the chunk size varies, it brings different efficiency gains. When the matrix chunks are less than 100, the larger the chunks are, the more efficient the matrix multiplication is, because more data is put into the cache at one time, reducing the time spent on transferring data to the cache. However, when the chunks of the matrix are larger than 100, the larger the chunks, the lower the efficiency when computing a matrix with a larger amount of data, probably because the larger the chunks, the lower the cache hit rate.

3.3 Loop unrolling

In this experiment, we compare the speed of matrix multiplication among different loop unrolling sizes: 4, 8, and 16.

3.3.1 Theorem Analysis

Loop costs a lot. Each loop requires repeated calculations and comparisons, operations that take up processor time and slow down program execution. In addition, loops need to store multiple variables and intermediate results in memory, which can increase memory usage and reduce program efficiency. Thus, lowering the time of the looping will improve the efficiency of the program.

3.3.2 Code Implementation

The Code Implementation is below.

```
//matrix blocking 16x16 and loop unrolling for 4
for(size_t r = 0; r < M; r+=16)
    for(size_t k =0; k < K; k+=16)
        for(size_t c = 0; c < N; c+=16)
        for(size_t r0 = 0; r0 < 16; r0++)
            for(size_t k0 = 0; k0 < 16; k0++)
                for(size_t c0 = 0; c0 < 16; c0+=4)
                {
                    res(r + r0, c + c0) += left(r + r0, k + k0) * right(k + k0, c + c0);
                    res(r + r0, c + c0 + 1) += left(r + r0, k + k0) * right(k + k0, c + c0 + 1);
                    res(r + r0, c + c0 + 2) += left(r + r0, k + k0) * right(k + k0, c + c0 + 2);
                    res(r + r0, c + c0 + 3) += left(r + r0, k + k0) * right(k + k0, c + c0 + 3);
                }
```

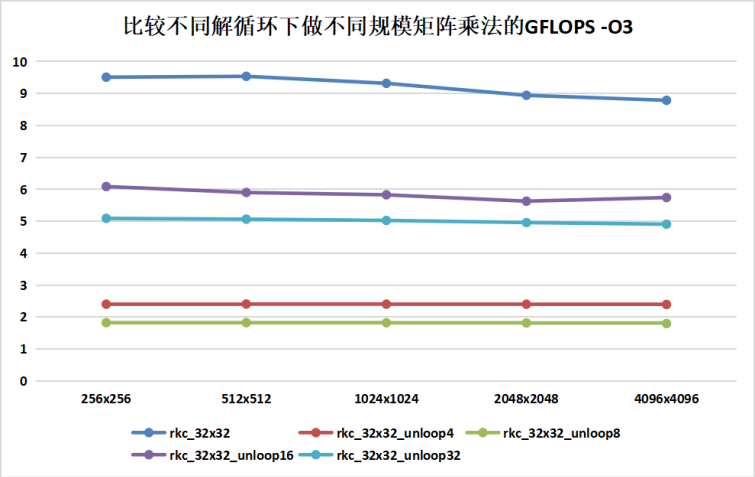
3.3.3 Results Analysis

Functions dgemm_rkc_32x32(), dgemm_rkc_32x32_unloop4(), dgemm_rkc_32x32_unloop8(), dgemm_rkc_32x32_unloop16() and dgemm_rkc_32x32_unloop32() are compared in this experiment.

The results are as follows. (with -O3)

不同解循环规模下不同规模矩阵乘法的平均耗时（s）-O3

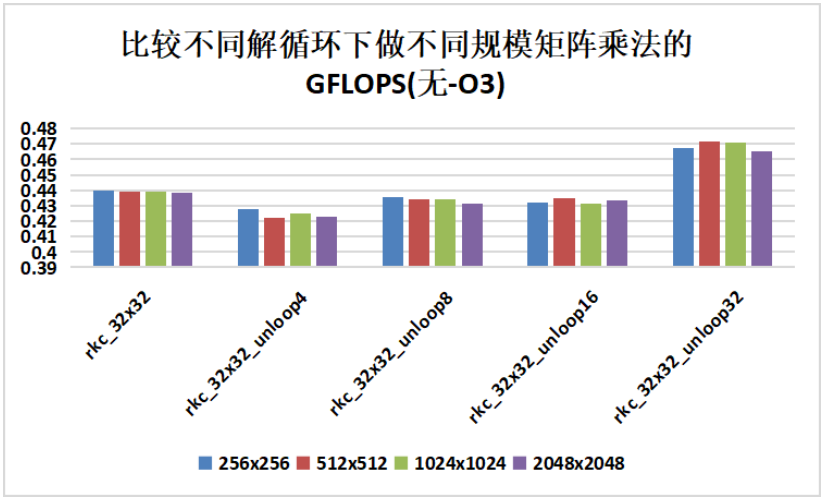
	256x256	512x512	1024x1024	2048x2048	4096x4096
rkc_32x32	0.0035282	0.0281444	0.2304436	1.92097	15.6421
rkc_32x32_unloop4	0.0139785	0.1116127	0.894435	7.16283	57.466
rkc_32x32_unloop8	0.0184123	0.1473682	1.181184	9.47826	76.6355
rkc_32x32_unloop16	0.0055132	0.0455046	0.3685896	3.05286	23.9415
rkc_32x32_unloop32	0.0065938	0.0530109	0.4274454	3.4652	28.018



From the chart and the graph, we will find that the function without loop unrolling will still be faster than the one calculated by unrolling the loop. This may be because when we do the experiment, we turn on the -O3 instruction, which may help us optimize the most primitive function, while it may not be convenient to optimize the function with loop unrolling.

Then we turn off the -O3 instruction and check the results.

不同解循环规模下不同规模矩阵乘法的平均耗时（s）没-O3				
	256x256	512x512	1024x1024	2048x2048
rkc_32x32	0.076288	0.611265	4.89347	39.1769
rkc_32x32_unloop4	0.078415	0.635913	5.05607	40.6378
rkc_32x32_unloop8	0.07698	0.618461	4.94841	39.8392
rkc_32x32_unloop16	0.0777	0.61688	4.98153	39.6546
rkc_32x32_unloop32	0.071816	0.569289	4.55727	36.9516



From the results, we can find that unrolling the loop only speeds up the computation when the size of unrolling is 32. However, the speed is much slower than the speed when we turn on the -O3 instruction.

3.4 Register variables

In this experiment, we evaluate the speed of matrix multiplication under different scales of loop unrolling using register variables.

3.4.1 Theorem Analysis

Computation can only happen if data is stored in registers. A compiler will automatically transform code so that the intermediate steps that place certain data in registers are inserted. One can give a hint to the compiler

that it would be good to keep certain data in registers, which may improve the speed of computation.

3.4.2 Code Implementation

The Code Implementation is below.

```
//dgemm_rkc_32x32_register
for(size_t r = 0; r < M; r+=32)
    for(size_t k =0; k < K; k+=32)
        for(size_t c = 0; c < N; c+=32)
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t k0 = 0; k0 < 32; k0++)
                    for(size_t c0 = 0; c0 < 32; c0++)
                    {
                        register double temp = res(r + r0, c + c0);
                        temp += left(r + r0, k + k0) * right(k + k0, c + c0);
                        res(r + r0, c + c0) = temp;
                    }
```

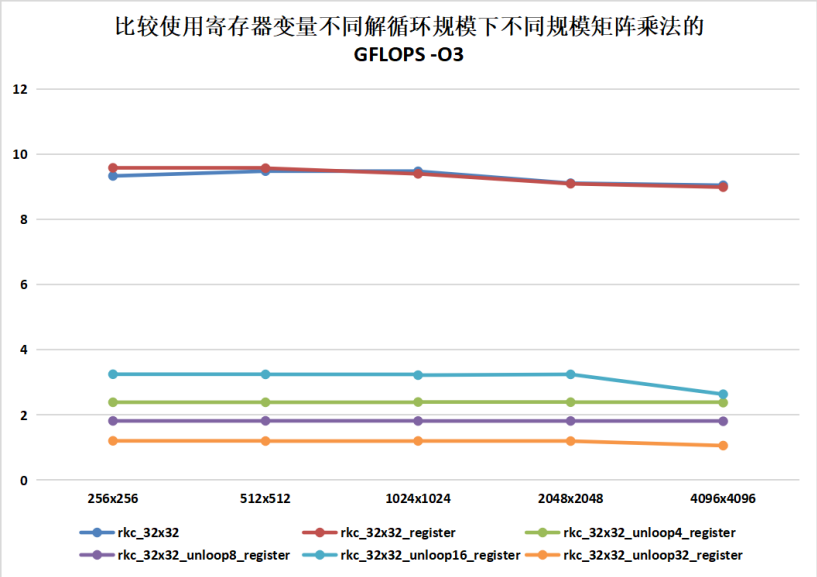
3.4.3 Results Analysis

Functions dgemm_rkc_32x32(), dgemm_rkc_32x32_unloop4_register(), dgemm_rkc_32x32_unloop8_register(), dgemm_rkc_32x32_unloop16_register() and dgemm_rkc_32x32_unloop32_register() are compared in this experiment.

The results are as follows. (with -O3)

使用寄存器变量不同解循环规模下不同规模矩阵乘法的平均耗时（s）-O3

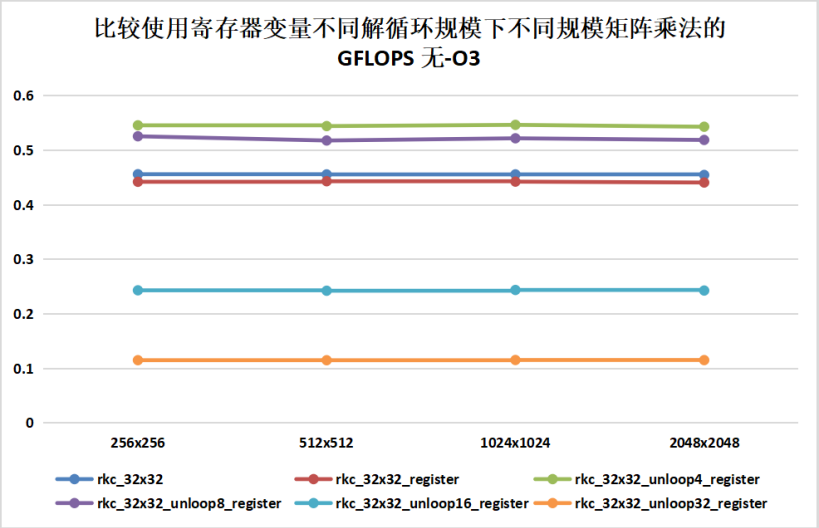
	256x256	512x512	1024x1024	2048x2048	4096x4096
rkc_32x32	0.0035934	0.0282988	0.226515333	1.884575	15.1844
rkc_32x32_register	0.0035001	0.0280304	0.228383667	1.8881	15.2813
rkc_32x32_unloop4_register	0.0140456	0.112389	0.896173	7.183115	57.8152
rkc_32x32_unloop8_register	0.0184686	0.1475694	1.183563333	9.4663	76.1051
rkc_32x32_unloop16_register	0.0103259	0.0827244	0.666676333	5.293	52.1431
rkc_32x32_unloop32_register	0.027864	0.2240868	1.78857	14.3529	129.84



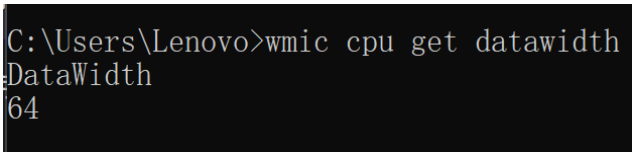
From the graph we can see that, as discussed in 3.3, the addition of loop unrolling slows down the computation when -O3 is on. Simultaneously, without loop unrolling, the addition of register variables does not improve the speed of the computation.

Then, we do the same experiment without -O3 instruction. The results are as follows.

使用寄存器变量不同规模矩阵乘法的平均耗时（s）无-O3				
	256x256	512x512	1024x1024	2048x2048
rkc_32x32	0.0735954	0.5891996	4.71299	37.7928
rkc_32x32_register	0.075907	0.6058926	4.85484	38.9825
rkc_32x32_unloop4_register	0.061503	0.4933026	3.92996	31.6402
rkc_32x32_unloop8_register	0.0638544	0.518499	4.11547	33.1175
rkc_32x32_unloop16_register	0.1380172	1.107698	8.81107	70.7875
rkc_32x32_unloop32_register	0.2917702	2.335686	18.6167	149.21



As we can see from the graph, adding register variables does speed up the computation of the matrix when the loop is expanded to 0, 4, or 8 and the speeds of factors 4 and 8 are fast. However, when the loop is 16 or 32, the computation time increases exponentially, probably because there is not enough memory in the registers, and what can be done in one step will be done in multiple steps, which leads to an exponential increase in time. As has been said above, it is likely that our register can store 8 doubles at a time.



To verify our analysis, we check the size of the register in our computer and find it is 64 bytes, which means it can store 8 double at a time. The information is corresponding to our results and analysis.

3.5 Matrix Packing

In this experiment, we compare the speed of matrix multiplication among different levels of packing data on the base of 32x32 matrix blocking.

3.5.1 Theorem Analysis

When multiplying matrices in chunks of size 32x32 to speed up the multiplication of matrices, the inner layers compute the 32x32 chunks with discontinuous access to the right matrix, the left matrix, and the result matrix. Therefore, we put the data to be computed into the aligned contiguous memory before computing the 32x32 blocks, so that all the data to be computed can be put into the cache continuously during the computation, reducing cache misses and increasing the cache hit rate, thus accelerating matrix multiplication.

3.5.2 Code Implementation

The Code implementation is below. (Memory Alignment and register variables are used to improve the speed)

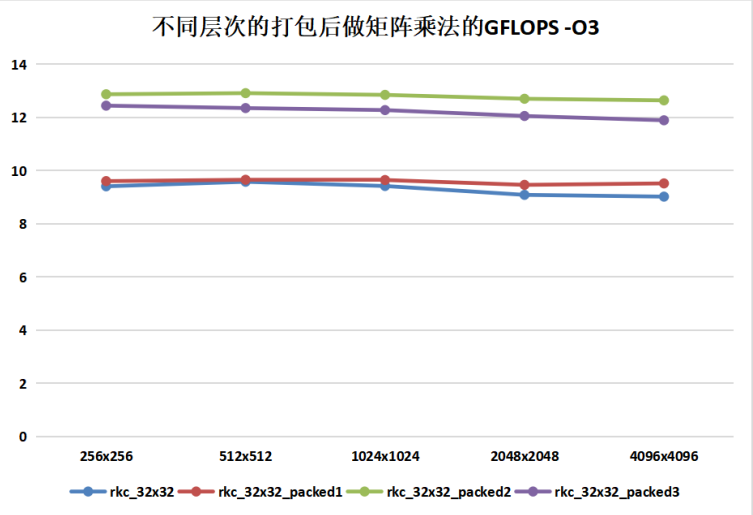
```
//dgemm_rkc_32x32_packed1
//pack right matrix
double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
for(size_t r = 0; r < M; r+=32)
    for(size_t k =0; k < K; k+=32)
        for(size_t c = 0; c < N; c+=32)
        {
            for(size_t k0 = 0; k0 < 32; k0++)
                for(size_t c0 = 0; c0 < 32; c0++)
                    right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t k0 = 0; k0 < 32; k0++)
                {
                    register double temp = left(r + r0, k + k0);
                    for(size_t c0 = 0; c0 < 32; c0++)
                        res(r + r0, c + c0) += temp * right_packed[k0 * 32 + c0];
                }
        }
delete[] right_packed;

//dgemm_rkc_32x32_packed2
//pack the left matrix
double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
for(size_t r = 0; r < M; r+=32)
    for(size_t k =0; k < K; k+=32)
        for(size_t c = 0; c < N; c+=32)
        {
            for(size_t k0 = 0; k0 < 32; k0++)
                for(size_t c0 = 0; c0 < 32; c0++)
                    right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t k0 = 0; k0 < 32; k0++)
                    left_packed[r0 * 32 + k0] = left(r + r0, k + k0);
            for(size_t r0 = 0; r0 < 32; r0++)
            {
                for(size_t k0 = 0; k0 < 32; k0++)
                {
                    register double temp1 = left_packed[r0 * 32 + k0];
                    for(size_t c0 = 0; c0 < 32; c0++)
                        res(r + r0, c + c0) += temp1 * right_packed[k0 * 32 + c0];
                }
            }
        }
delete[] right_packed;
delete[] left_packed;

//dgemm_rkc_32x32_packed3
//pack the result matrix
double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
double* res_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
memset(res_packed, 0, 32 * 32 * sizeof(double));
for(size_t r = 0; r < M; r+=32)
    for(size_t k =0; k < K; k+=32)
        for(size_t c = 0; c < N; c+=32)
        {
            for(size_t k0 = 0; k0 < 32; k0++)
                for(size_t c0 = 0; c0 < 32; c0++)
                    right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t k0 = 0; k0 < 32; k0++)
                    left_packed[r0 * 32 + k0] = left(r + r0, k + k0);
            for(size_t r0 = 0; r0 < 32; r0++)
            {
                for(size_t k0 = 0; k0 < 32; k0++)
                {
                    register double temp1 = left_packed[r0 * 32 + k0];
                    for(size_t c0 = 0; c0 < 32; c0++)
                        res_packed[r0 * 32 + c0] += temp1 * right_packed[k0 * 32 + c0];
                }
            }
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t c0 = 0; c0 < 32; c0++)
                    res(r + r0, c + c0) += res_packed[r0 * 32 + c0];
        }
delete[] right_packed;
delete[] left_packed;
delete[] res_packed;
```

3.5.3 Results Analysis

Functions dgemm_rkc_32x32_packed1(), dgemm_rkc_32x32_packed2() and dgemm_rkc_32x32_packed3() are compared in this experiment.

The results with -O3 are as follow.

不同层次的打包后矩阵乘法的平均耗时（s）-O3					
	256x256	512x512	1024x1024	2048x2048	4096x4096
rkc_32x32	0.0035648	0.0280099	0.227873	1.889783333	15.2295
rkc_32x32_packed1	0.0034923	0.0277921	0.2225482	1.814716667	14.4339
rkc_32x32_packed2	0.0026062	0.0207824	0.1670978	1.3522	10.8689
rkc_32x32_packed3	0.002696	0.021727	0.1748906	1.424866667	11.5536

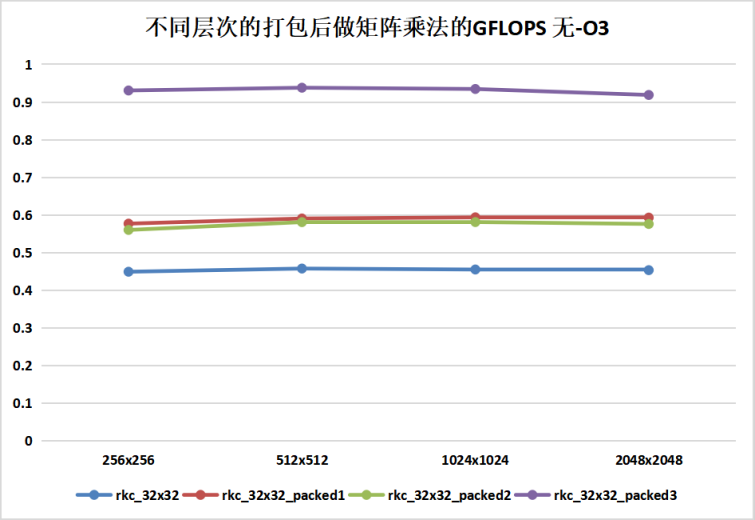


From the results of the graph, we can see that when -O3 compilation is turned on during the compilation process, the compiler will automatically help us to optimize our program, probably with SIMD, unloop, etc. The optimization is best when we only pack the left matrix and the right matrix (rkc_32x32_packed2), probably in this case, not only the way of accessing the data memory optimized is by manual operations but also the program is written in a way that is easier to be understood by the compiler so that the compiler can recognize our program and optimize it through -O3.

To better explore whether our packing of the left matrix, the right matrix, and the resultant matrix have a speedup effect on the multiplication of the matrices, we turned off -O3 and repeated the experiment again.

The results without -O3 are as follows.

不同层次的打包后矩阵乘法的平均耗时（s）无-O3				
	256x256	512x512	1024x1024	2048x2048
rkc_32x32	0.0747222	0.586725	4.719786667	37.9042
rkc_32x32_packed1	0.0581706	0.4545732	3.61793	28.9774
rkc_32x32_packed2	0.059902	0.4621392	3.696063333	29.8273
rkc_32x32_packed3	0.0360552	0.286183	2.29769	18.6992



From the graph, we found that these optimizations were all effective in speeding up the matrix multiplication.

3.6 Pointer accessing

In this experiment, we use a pointer to access the memory with the ++ operator and aim to find out if this way is effective in speeding up the matrix multiplication.

3.6.1 Theorem Analysis

3.6.2 Code Implementation

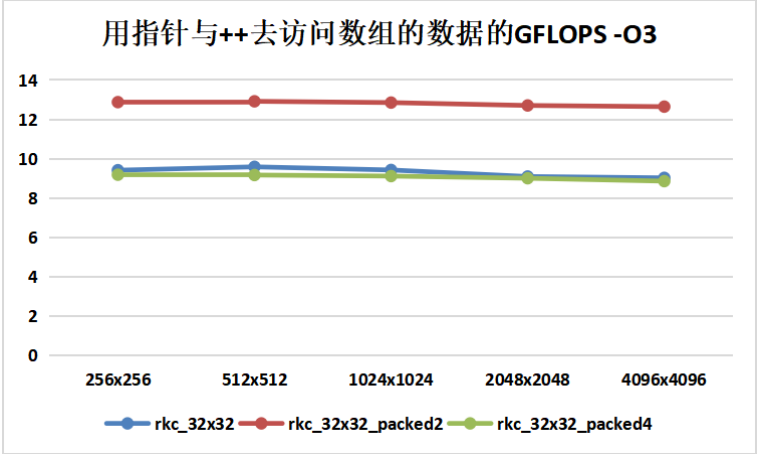
The Code implementation is as follows. (Based on the dgemm_rkc_32x32_packed3 function)

```
//dgemm_rkc_32x32_packed4
double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
double* res_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
memset(res_packed, 0, 32 * 32 * sizeof(double));
for(size_t r = 0; r < M; r+=32)
    for(size_t k =0; k < K; k+=32)
        for(size_t c = 0; c < N; c+=32)
        {
            for(size_t k0 = 0; k0 < 32; k0++)
                for(size_t c0 = 0; c0 < 32; c0++)
                    right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t k0 = 0; k0 < 32; k0++)
                    left_packed[r0 * 32 + k0] = left(r + r0, k + k0);
            for(size_t r0 = 0; r0 < 32; r0++)
            {
                for(size_t k0 = 0; k0 < 32; k0++)
                {
                    double* pright = &right_packed[k0 * 32];
                    double* pres = &res_packed[r0 * 32];
                    register double temp1 = left_packed[r0 * 32 + k0];
                    for(size_t c0 = 0; c0 < 32; c0++)
                        *(pres++) += temp1 * *(pright++);
                }
            }
            double* pres = res_packed;
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t c0 = 0; c0 < 32; c0++)
                    res(r + r0, c + c0) += *(pres++);
        }
delete[] right_packed;
delete[] left_packed;
delete[] res_packed;
```

3.6.3 Results Analysis

We compare the speed among dgemm_rkc_32x32, dgemm_rkc_32x32_packed2(performs best in 3.6 -O3 experiment) and dgemm_rkc_32x32_packed4. The results are as follows. (with -O3)

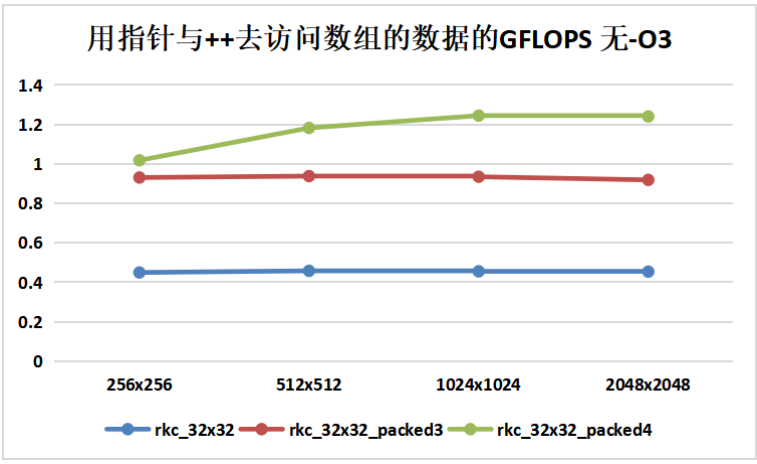
用指针与++去访问数组的数据 -O3					
	256x256	512x512	1024x1024	2048x2048	4096x4096
rkc_32x32	0.0035648	0.0280099	0.227873	1.889783333	15.2295
rkc_32x32_packed2	0.0026062	0.0207824	0.1670978	1.3522	10.8689
rkc_32x32_packed4	0.0036539	0.0292806	0.2356212	1.90804	15.5171



From the graph, we can see that the speed of matrix multiplication decreases when we use pointer access. The reason for this is that using pointers makes the program more complex and the compiler cannot recognize our program well enough to give us a better optimization through -O3.

In order to find out if changing the way to access the data through pointer and ++ will improve the speed of matrix multiplication. We turn off the -O3 and compare the speed among dgemm_rkc_32x32, dgemm_rkc_32x32_packed3(performs best in 3.6 without-O3 experiment) and dgemm_rkc_32x32_packed4. The results are as follows. (without -O3)

用指针与++去访问数组的数据 无-O3				
	256x256	512x512	1024x1024	2048x2048
rkc_32x32	0.0747222	0.586725	4.719786667	37.9042
rkc_32x32_packed3	0.0360552	0.286183	2.29769	18.6992
rkc_32x32_packed4	0.0329662	0.2270502	1.725236667	13.8495



From the graph, we can find out that changing the way to access the data through pointer and ++ does improve the speed of the matrix multiplication. Because the speed of computing by the ++ operator is much faster than computing the index.

3.7 Memory Alignment

In this experiment, we compare the speed of matrix multiplication under the matrix packing without Memory Alignment, with Memory Alignment, and with explicit Memory Alignment.

3.7.1 Theorem Analysis

Theoretically, aligned memory allows the cache to read and write data faster. Explicit aligned memory can booster the efficiency of -O3.

3.7.2 Code Implementation

The Code Implementation is as follows.

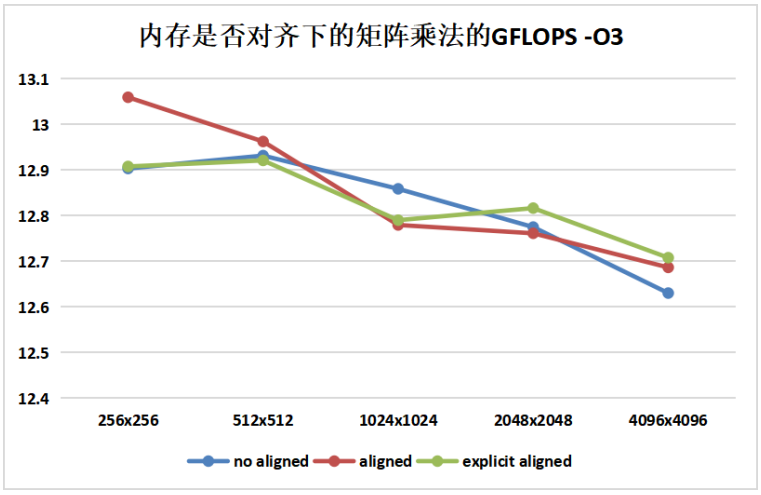
```
//Without Memory Alignment
double* data = new double[size];
//Memory Alignment
double* data = (double*)aligned_alloc(1024, size * sizeof(double));
//Explicit Memory Alignment
__attribute__((aligned(1024))) double* data = (double*)aligned_alloc(1024, size * sizeof(double));
```

3.7.3 Results Analysis

Functions dgemm_rkc_32x32_packed5() (use new to allocate), dgemm_rkc_32x32_packed2() (allocate Alignment Memory) and dgemm_rkc_32x32_packed6() (Explicit allocate Alignment Memory)are compared in this experiment.

The results are as follows. (with -O3)

内存是否对齐下的矩阵乘法平均耗时（s）-O3					
	256x256	512x512	1024x1024	2048x2048	4096x4096
no aligned	0.0026004	0.0207577	0.1670027	1.344794	10.88163333
aligned	0.0025693	0.0207082	0.1680382	1.346232	10.83333333
explicit aligned	0.0025995	0.0207747	0.1679004	1.34043	10.81516667



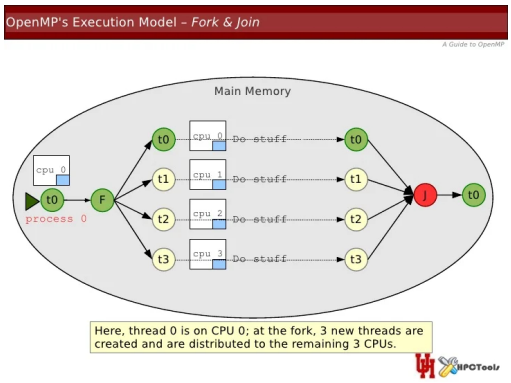
From the graph, we find that the speed of the computation does not change too much under 3 circumstances. Maybe the -O3 instruction has optimized the code to the same degree.

3.8 OpenMP

In this experiment, we compare the speed of matrix multiplication under single-thread and multi-threaded. (using OpenMP)

3.8.1 Theorem Analysis

On a multi-core computer, OpenMP can map multiple threads to different CPU cores so that different threads can execute different tasks at the same time. This parallel execution can significantly improve the running efficiency of programs because it can fully utilize the multi-core processing power of the computer.



3.8.2 Code Implementation

By including 'omp.h' head file, adding '-fopenmp' when compiling the file, and adding '#pragma omp parallel for schedule(dynamic)' before the loop to let the computer automatically allocates computational resources, thus maximizing process efficiency.

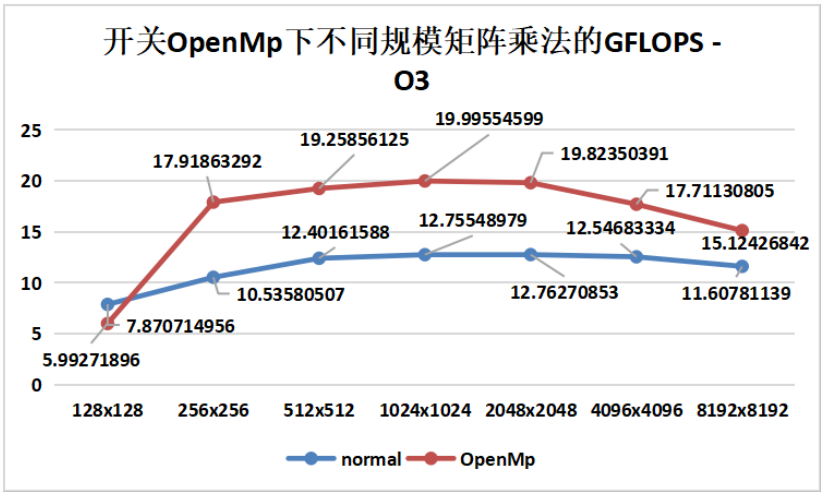
```
#ifdef _OPENMP
#include <omp.h>
#endif

//dgemm_rkc_32x32_packed2_OpenMP
double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
#pragma omp parallel for schedule(dynamic)
for(size_t r = 0; r < M; r+=32)
    for(size_t k =0; k < K; k+=32)
    {
        for(size_t c = 0; c < N; c+=32)
        {
            for(size_t k0 = 0; k0 < 32; k0++)
                for(size_t c0 = 0; c0 < 32; c0++)
                    right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t k0 = 0; k0 < 32; k0++)
                    left_packed[r0 * 32 + k0] = left(r + r0, k + k0);
            for(size_t r0 = 0; r0 < 32; r0++)
            {
                for(size_t k0 = 0; k0 < 32; k0++)
                {
                    register double temp1 = left_packed[r0 * 32 + k0];
                    for(size_t c0 = 0; c0 < 32; c0++)
                        res(r + r0, c + c0) += temp1 * right_packed[k0 * 32 + c0];
                }
            }
        }
    }
delete[] right_packed;
delete[] left_packed;
```

3.8.3 Results Analysis

Functions dgemm_rkc_32x32_packed2() and dgemm_rkc_32x32_packed2_OpenMP() are compared.

The results are as follows.

开关OpenMp下不同规模矩阵乘法的平均耗时（s）-O3							
	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096	8192x8192
normal	0.0005329	0.0031848	0.0216452	0.1683576	1.346099	10.954075	94.7217
OpenMp	0.0006999	0.0018726	0.0139385	0.1073981	0.8666414	7.759955	72.6985



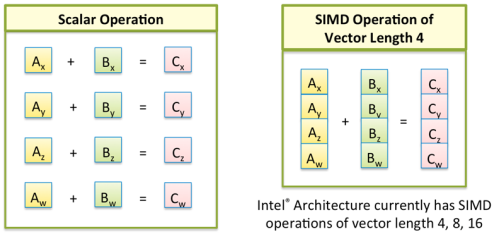
As we can see from the graph, when the size of the matrix is relatively small, the improvement of the multi-threaded operation is hard to see. However, as the size of the matrix gets bigger. The multi-threaded operation does improve the speed of matrix multiplication.

3.9 SIMD

In this experiment, we aim to figure out whether the simd will improve the speed of matrix multiplication or not.

3.9.1 Theorem Analysis

In SIMD architecture, a single instruction can operate multiple data elements at the same time, making full use of register space. It can greatly reduce the number of instruction executions and improves the running efficiency of the program.



3.9.2 Code Implementation

The Code Implementations are as follows. (avx2 version 2 is like a loop unrolling with a factor of 4)

```
//dgemm_rkc_32x32_packed_OpenMP_avx2_v1
double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
#pragma omp parallel for schedule(dynamic)
for(size_t r = 0; r < M; r+=32)
    for(size_t k =0; k < K; k+=32)
    {
        for(size_t c = 0; c < N; c+=32)
        {
            for(size_t k0 = 0; k0 < 32; k0++)
                for(size_t c0 = 0; c0 < 32; c0++)
                    right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t k0 = 0; k0 < 32; k0++)
                    left_packed[r0 * 32 + k0] = left(r + r0, k + k0);
            __m256d right_data, left_data, res_data;
            for(size_t r0 = 0; r0 < 32; r0++)
            {
                for(size_t k0 = 0; k0 < 32; k0++)
                {
```

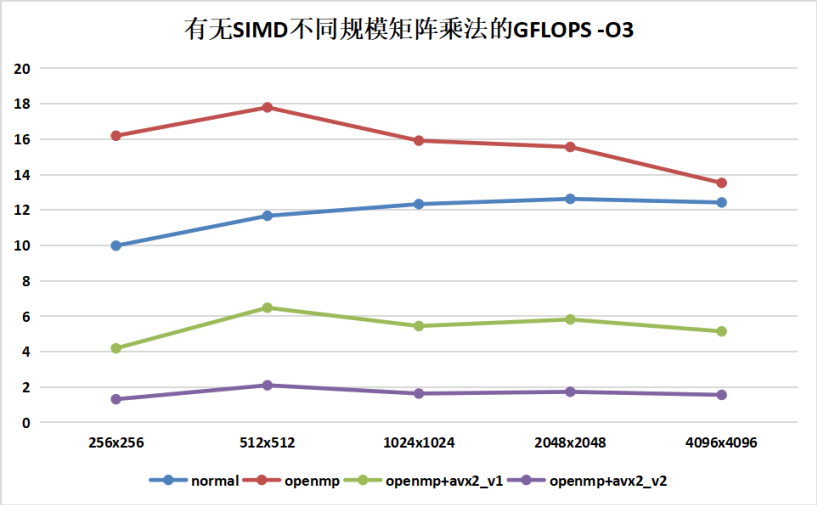
```
                    left_data = _mm256_broadcast_sd(&left_packed[r0 * 32 + k0]);
                    double* temp = (double*)aligned_alloc(32, 4 * sizeof(double));
                    for(size_t c0 = 0; c0 < 32; c0+=4)
                    {
                        res_data = _mm256_setzero_pd();
                        right_data = _mm256_loadu_pd(&right_packed[k0 * 32 + c0]);
                        res_data = _mm256_add_pd(res_data, _mm256_mul_pd(left_data, right_data));
                        _mm256_storeu_pd(temp, res_data);
                        for(int i = 0; i < 4; i++)
                            res(r + r0, c + c0 + i) += temp[i];
                    }
                    delete[] temp;
                }
            }
        }
    }
delete[] right_packed;
delete[] left_packed;

//dgemm_rkc_32x32_packed_OpenMP_avx2_v2
double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
#pragma omp parallel for schedule(dynamic)
for(size_t r = 0; r < M; r+=32)
    for(size_t k =0; k < K; k+=32)
    {
        for(size_t c = 0; c < N; c+=32)
        {
            for(size_t k0 = 0; k0 < 32; k0++)
                for(size_t c0 = 0; c0 < 32; c0++)
                    right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
            for(size_t r0 = 0; r0 < 32; r0++)
                for(size_t k0 = 0; k0 < 32; k0++)
                    left_packed[r0 * 32 + k0] = left(r + r0, k + k0);
            __m256d right_data1, left_data, res_data1, right_data2, res_data2, right_data3, res_data3, right_data4
            for(size_t r0 = 0; r0 < 32; r0++)
            {
                for(size_t k0 = 0; k0 < 32; k0++)
                {
                    left_data = _mm256_broadcast_sd(&left_packed[r0 * 32 + k0]);
                    double* temp1 = (double*)aligned_alloc(32, 4 * sizeof(double));
                    double* temp2 = (double*)aligned_alloc(32, 4 * sizeof(double));
                    double* temp3 = (double*)aligned_alloc(32, 4 * sizeof(double));
                    double* temp4 = (double*)aligned_alloc(32, 4 * sizeof(double));
                    for(size_t c0 = 0; c0 < 32; c0+=16)
                    {
                        res_data1 = _mm256_setzero_pd();
                        res_data2 = _mm256_setzero_pd();
                        res_data3 = _mm256_setzero_pd();
                        res_data4 = _mm256_setzero_pd();
                        right_data1 = _mm256_loadu_pd(&right_packed[k0 * 32 + c0]);
                        right_data2 = _mm256_loadu_pd(&right_packed[k0 * 32 + c0 + 4]);
                        right_data3 = _mm256_loadu_pd(&right_packed[k0 * 32 + c0 + 8]);
                        right_data4 = _mm256_loadu_pd(&right_packed[k0 * 32 + c0 + 12]);
                        res_data1 = _mm256_add_pd(res_data1, _mm256_mul_pd(left_data, right_data1));
                        res_data2 = _mm256_add_pd(res_data2, _mm256_mul_pd(left_data, right_data2));
                        res_data3 = _mm256_add_pd(res_data3, _mm256_mul_pd(left_data, right_data3));
                        res_data4 = _mm256_add_pd(res_data4, _mm256_mul_pd(left_data, right_data4));
                        _mm256_storeu_pd(temp1, res_data1);
                        _mm256_storeu_pd(temp2, res_data2);
                        _mm256_storeu_pd(temp3, res_data3);
                        _mm256_storeu_pd(temp4, res_data4);
                        for(int i = 0; i < 4; i++)
                        {
                            res(r + r0, c + c0 + i) += temp1[i];
                            res(r + r0, c + c0 + i + 4) += temp2[i];
                            res(r + r0, c + c0 + i + 8) += temp3[i];
                            res(r + r0, c + c0 + i + 12) += temp4[i];
                        }
                    }
                    delete[] temp1;
                    delete[] temp2;
                    delete[] temp3;
                    delete[] temp4;
                }
            }
        }
    }
delete[] right_packed;
delete[] left_packed;
```

3.9.3 Results Analysis

We compare the speed of matrix multiplication between dgemm_rkc_32x32_packed2(), dgemm_rkc_32x32_packed2_OpenMP(), dgemm_rkc_32x32_packed2_OpenMP_avx2_v1() and dgemm_rkc_32x32_packed2_OpenMP_avx2_v2() functions.

The results are as follows.(with -O3)

有无SIMD不同规模矩阵乘法的平均耗时（s）-O3					
	256x256	512x512	1024x1024	2048x2048	4096x4096
normal	0.0033608	0.0229987	0.1741173	1.36057	11.06176667
openmp	0.002072	0.015084	0.1349214	1.104156667	10.15926
openmp+avx2_v1	0.0080076	0.0414199	0.3942065	2.952523333	26.70493333
openmp+avx2_v2	0.0255331	0.1277175	1.315042	9.919266667	88.1253

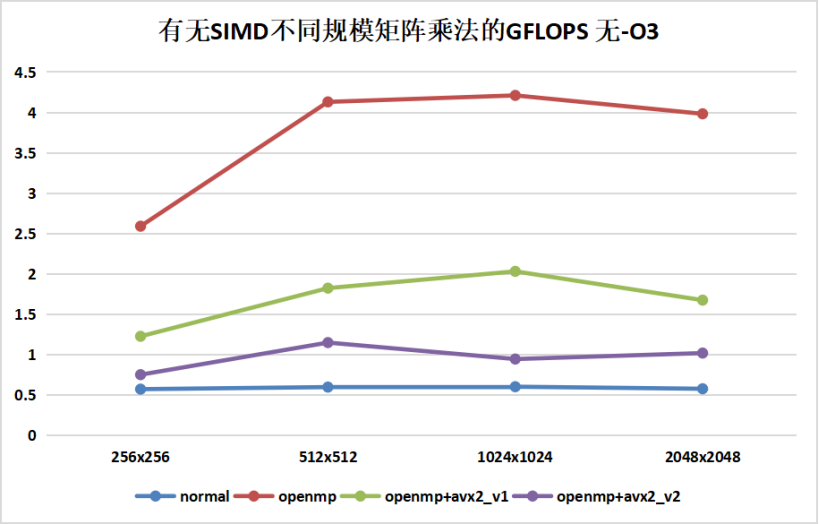


From the graph, we can find out that the SIMD operations will slow down the speed of the matrix multiplication when compared with the normal function. This may be because -O3 instruction has already help us to do the SIMD instruction and some other optimizations in an excellent way. When we add the SIMD by ourselves, the compiler cannot understand our code and thus cannot give us the best optimizations.

Moreover, the function with SIMD and OpenMP is also slower than the function with only OpenMP. We use the command '#pragma omp parallel for schedule(dynamic)' to let the compiler to choose the best way to do multiple-tread operations. When we add SIMD operations in the function, the code will become difficult for the compiler to understand and thus cannot generate the best way of doing the multiple-tread operations.

However, we want to find out the effect of SIMD instruction on matrix multiplication. Thus, we repeat the experiment again without the -O3 instruction.

有无SIMD不同规模矩阵乘法的平均耗时（s）无-O3				
	256x256	512x512	1024x1024	2048x2048
normal	0.0585199	0.4482505	3.55902	29.718
openmp	0.0129484	0.0649846	0.509955	4.31306
openmp+avx2_v1	0.0273325	0.1471367	1.05682	10.253
openmp+avx2_v2	0.044588	0.2334272	2.26999	16.843



Comparing the normal function with the function with SIMD, we can find out that the SIMD operations improve the speed of the function. However, the computation that includes only openmp will be faster than one that includes both openmp and SIMD. This is probably because the compiler can give us a better multi-threaded optimization solution without SIMD, which is higher than the optimization we get with the SIMD operation.

3.10 Final DGEMM Evaluation

In this subsection, I will describe the specific implementation of my Final DGEMM, and the rationale for its design. Then the efficiency of our equation is evaluated by comparing our design of the final function with that of openblas and the most parsimonious matrix multiplication (rkc).

3.10.1 Design Analysis

From our previous experimental analysis, some manual optimization of matrix multiplication, such as SIMD, manual loop expansion, and use of pointer for sequential access will conflict with the effect of the compiler's automatic optimization of -O3, making matrix multiplication slower, and using register variables with memory alignment has little noticeable optimization effect.

Moreover, it is obvious that the compiler's -O3 efficiency is much higher than our own manual optimizations mentioned above, so our main idea when designing the final DGEMM function is to write the function that the compiler can read better so that the compiler can maximize the effect of -O3 optimization to improve the efficiency of our matrix multiplication.

Based on the previous experimental analysis, we found that matrix access optimization, such as swapping loop order, matrix chunking, and matrix packing will make the matrix multiplication rate faster. Also, using multi-threads will make the matrix multiplication faster.

Last but not least, when the size of the matrix is relatively small, it is better to swap the loop order only as the cost of the operation of matrix chunking or matrix packing will affect the speed of the matrix multiplication. When the size of the matrix is relatively large, it is time to use all of our techniques to improve the speed of our matrix multiplication, like matrix chunking, matrix packing, and multi-threads. The dividing line that separates the two situations is approximately 512x512. (Experiment result is as follows)

不同加速方法下的平均耗时（s）		
	交换顺序加速	多种加速
256x256	0.0065766	0.0190778
512x512	0.0508606	0.022282
1024x1024	0.3363041	0.1490238

3.10.2 Code Implementation & Correctness

The streamlined version of the code is below.

```
bool dgemm_final(Order order, TransType TransA, TransType TransB, const size_t M, const size_t N, const size_t K,
const double alpha, const double* A, const size_t lda, const double* B, const size_t ldb,
const double beta, double* C, const size_t ldc)
{
    //Safety Check
    if(M < 512)
    {
        if(TransA == NoTrans && TransB == NoTrans)
        {
            for(size_t r = 0; r < M; r++)
                for(size_t k =0; k < K; k++)
                    for(size_t c = 0; c < N; c++)
                        res(r, c) = beta * res(r, c) + alpha * left(r, k) * right(k, c);
        }
        else if ......
    }
    else
    {
        if(TransA == NoTrans && TransB == NoTrans)
        {
            double* right_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            double* left_packed = (double*)aligned_alloc(1024, 32 * 32 * sizeof(double));
            #pragma omp parallel for schedule(dynamic)
            for(size_t r = 0; r < M; r+=32)
                for(size_t k =0; k < K; k+=32)
                {
                    for(size_t c = 0; c < N; c+=32)
                    {
                        for(size_t k0 = 0; k0 < 32; k0++)
                            for(size_t c0 = 0; c0 < 32; c0++)
                                right_packed[k0 * 32 + c0] = right(k + k0, c + c0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                            for(size_t k0 = 0; k0 < 32; k0++)
                                left_packed[r0 * 32 + k0] = left(r + r0, k + k0);
                        for(size_t r0 = 0; r0 < 32; r0++)
                        {
                            for(size_t k0 = 0; k0 < 32; k0++)
                            {
                                register double temp1 = left_packed[r0 * 32 + k0];
                                for(size_t c0 = 0; c0 < 32; c0++)
                                    res(r + r0, c + c0) = alpha * (temp1 * right_packed[k0 * 32 + c0]) + beta * re
                            }
                        }
                    }
                }
            delete[] right_packed;
            delete[] left_packed;
        }
        else if ......
    return true;
}
```

Correctness:

we output the result into a TXT file using ofstream object, and we can use the built-in Git tool and Diff comparison feature in Visual Studio Code (VS Code) to compare two documents and show the differences.

```
double* A = Random(Col, Row);
double* B = Random(Col, Row);
double* C = new double[Col*Row]();

//NoTrans NoTrans
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, Col, Row, Row, 1, A, Col, B, Col, 1, C, Col);
ofstream file1("openblas_output1.txt");
for(size_t i = 0; i < Col*Row; i++)
    file1 << C[i] << endl;
file1.close();
memset(C, 0, sizeof(double) * Col * Row);
dgemm_final(RowMajor, NoTrans, NoTrans, Col, Row, Row, 1, A, Col, B, Col, 1, C, Col);
ofstream file2("final_output1.txt");
for(size_t i = 0; i < Col*Row; i++)
    file2 << C[i] << endl;
file2.close();

//Trans NoTrans
cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, Col, Row, Row, 1, A, Col, B, Col, 1, C, Col);
ofstream file1("openblas_output2.txt");
for(size_t i = 0; i < Col*Row; i++)
```

```
    file1 << C[i] << endl;
file1.close();
memset(C, 0, sizeof(double) * Col * Row);
dgemm_final(RowMajor, Trans, NoTrans, Col, Row, Row, 1, A, Col, B, Col, 1, C, Col);
ofstream file2("final_output2.txt");
for(size_t i = 0; i < Col*Row; i++)
    file2 << C[i] << endl;
file2.close();

//NoTrans Trans
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, Col, Row, Row, 1, A, Col, B, Col, 1, C, Col);
ofstream file1("openblas_output3.txt");
for(size_t i = 0; i < Col*Row; i++)
    file1 << C[i] << endl;
file1.close();
memset(C, 0, sizeof(double) * Col * Row);
dgemm_final(RowMajor, NoTrans, Trans, Col, Row, Row, 1, A, Col, B, Col, 1, C, Col);
ofstream file2("final_output3.txt");
for(size_t i = 0; i < Col*Row; i++)
    file2 << C[i] << endl;
file2.close();

//Trans Trans
cblas_dgemm(CblasRowMajor, CblasTrans, CblasTrans, Col, Row, Row, 1, A, Col, B, Col, 1, C, Col);
ofstream file1("openblas_output3.txt");
for(size_t i = 0; i < Col*Row; i++)
    file1 << C[i] << endl;
file1.close();
memset(C, 0, sizeof(double) * Col * Row);
dgemm_final(RowMajor, Trans, Trans, Col, Row, Row, 1, A, Col, B, Col, 1, C, Col);
ofstream file2("final_output3.txt");
for(size_t i = 0; i < Col*Row; i++)
    file2 << C[i] << endl;
file2.close();
```

NoTrans NoTrans:

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$ git init
Initialized empty Git repository in /mnt/e/桌面/Computer learning/cpp/Project5/Code/build/.git/
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$ git add --  openblas_output1.txt final_output1.txt
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$ git diff -- openblas_output1.txt final_output1.txt
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$
```

Trans NoTrans:

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$ git add --  openblas_output2.txt final_output2.txt
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$ git diff -- openblas_output2.txt final_output2.txt
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$
```

NoTrans NoTrans:

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$ git add --  openblas_output3.txt final_output3.txt
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$ git diff -- openblas_output3.txt final_output3.txt
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$
```

Trans Trans:

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$ git add --  openblas_output4.txt final_output4.txt
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$ git diff -- openblas_output4.txt final_output4.txt
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project5/Code/build$
```

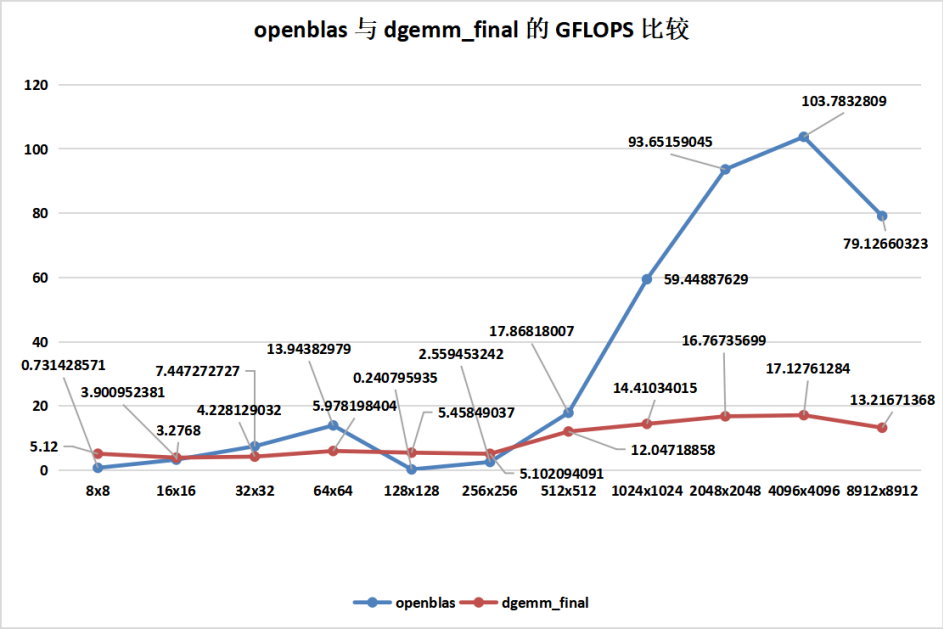
Since there is no difference in output in the terminal, these two files are the same. Thus, the result of the calculation of our function is correct.

3.10.3 Comparative experiment

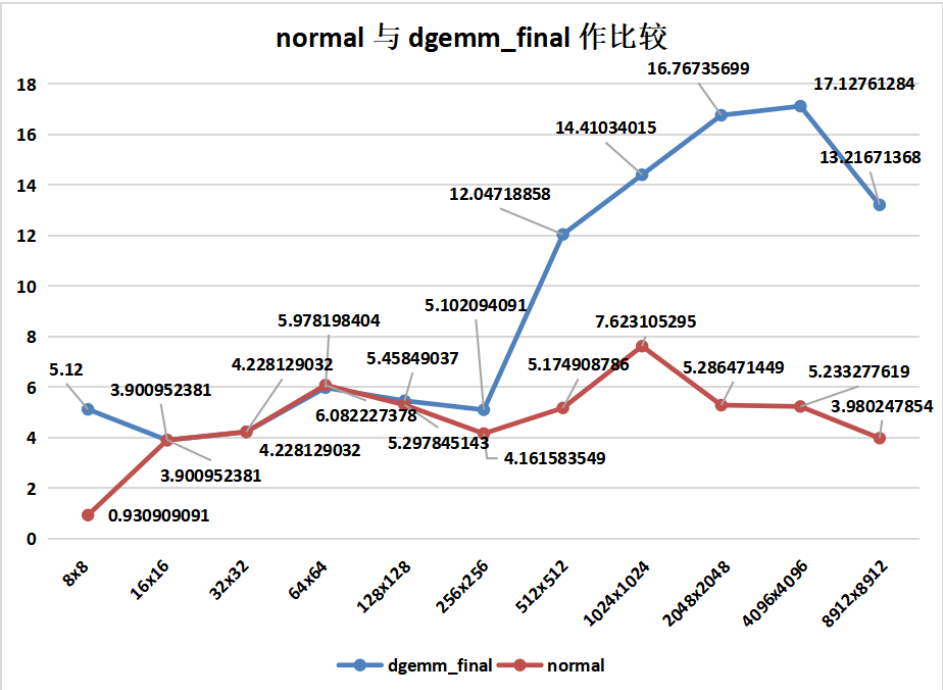
In this experiment, we compare the speed among functions cblas_dgemm() (The openblas function), dgemm_final(), and dgemm_rkc() (The normal function).

The results are as follows.

不同规模下矩阵乘法耗费的平均时间（s）-O3						
矩阵规模	8x8	16x16	32x32	64x64	128x128	256x256
openblas	0.0000014	0.0000025	0.0000088	0.0000376	0.0174185	0.01311
dgemm_final	0.0000002	0.0000021	0.0000155	0.0000877	0.0007684	0.0065766
normal	0.0000011	0.0000021	0.0000155	0.0000862	0.0007917	0.0080629
矩阵规模	512x512	1024x1024	2048x2048	4096x4096	8912x8912	
openblas	0.0150231	0.0361232	0.1834445	1.324288	13.8956	
dgemm_final	0.022282	0.1490238	1.0246021	8.024408	83.191	
normal	0.0518725	0.2817072	3.24978	26.2625	276.242	



Openblas and dgemm_final: From the graph, we find that our function has a better optimization effect until the matrix size is smaller than 512x512, which can be basically kept at the same level as openblas. However, when the size of the matrix gradually increases, the matrix multiplication of openblas will have a very large speedup, while our function will have a very slow improvement, and the final effect will be about 16% of openblas.



dgemm_final and normal: From the graph, we can see that when the size of the matrix is relatively small, there is basically no difference in the efficiency of the two, while when the size of the matrix becomes larger, beyond 256x256, the effect of our optimization is shown, and GFLOPS can be roughly 4 times higher.

3.11 Comparison under x86 and ARM

In this experiment, we run cblas_dgemm() (The openblas function) on both x86(PC), x86(Ser), and arm(PC) architectures.

The information of the computers are as follows.

An x86_64 personal computer:

Terms	information
Architecture	x86_64
CPU	An eight-core CPU, AMD Ryzen 7 5800H with Radeon Graphics @ 3.20GHz
Cache	512KB (L1) 4.0MB (L2) 16.0MB (L3)

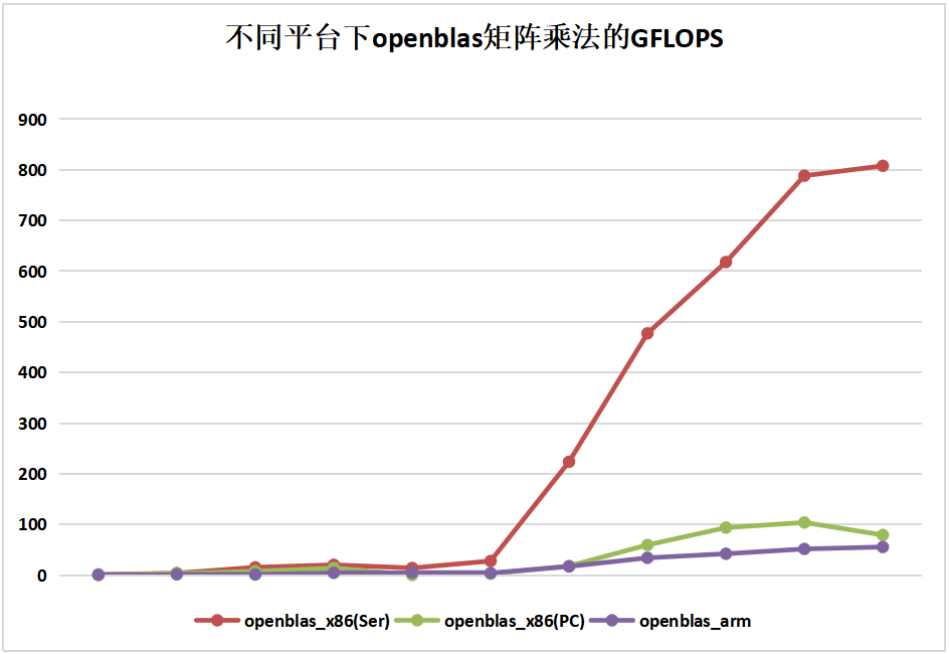
An x86_64 service computer:

Terms	information
Architecture	x86_64
CPU	An 28-core CPU, Intel Xeon W processor with Turbo @ 4.4GHz
Cache	57MB

An ARM personal computer:

Terms	information
Architecture	arm_64
CPU	An eight-core CPU, Apple M1 Pro @ 3.35GHz
Cache	192KB (L1) 4.0MB (L2)

Matrix multiplication operations of different sizes of the matrix are done on each of the three CPU architectures. The results are as follows.



We can see from the graph that when the matrix size is small, there is not much difference between matrix multiplication on an arm personal computer, an x86 service computer, and an x86 personal computer. But when the matrix size is larger, the matrix multiplication through the openblas function on the x86 service computer is much faster than on the x86 personal computer and the arm personal computer. And the x86 personal computer is a little bit faster than the arm personal computer.

The following are some possible reasons for the appearance of this phenomenon:

First of all, the size of the cache. The cache is a high-speed computing area in a computer organization. The bigger size of the cache, the faster the computational speed will be. Known from the information of the three computers, the x86_64 service computer, the x86_64 personal computer, and the arm personal computer have increasing sizes of cache, which corresponds to the results.

Secondly, the number of the core. We can take advantage of the core to do multiple operations at a single time. The more cores, the faster the computation is. Known from the information of the three computers, the x86_64 service computer, the x86_64 personal computer, and the arm personal computer have increasing numbers of cores, which corresponds to the results.

Finally, the -O3 operation on different CPU architectures is different. ARM and x86 have different instruction set architectures and cache sizes, so the compiler will use different instruction sequences and optimization strategies when generating optimized code. In terms of SIMD, ARM typically has smaller instructions, which may make ARM a little slower than x86. In terms of access optimization, ARM architectures typically have smaller instruction and data caches and therefore may focus more on reducing the number of memory accesses when optimizing. The x86 architecture, on the other hand, typically has a larger cache and therefore may focus more on leveraging cache coherency and prefetching to improve performance.

4 Difficulties & Solutions

4.1 Difficulties in invoking the OpenBlas library

Invoking the OpenBlas library has some difficulties in downloading the library, linking the library, and using the functions of the library correctly.

We use the input of the instruction in the terminal to download the Openblas library in Ubuntu.

```
sudo apt update
sudo apt install libopenblas-dev
```

To import the Openblas library in C++, we include 'cblas.h' head file. Then when we compile the file, we should add '-lopenblas' to link the library.

4.2 Difficulties in checking the correctness of the function

When the size of the matrix is large, it is impossible to check the correctness of the result by printing out the result on the screen and finding the differences through our own observation. In this project, we use the built-in Git tool and Diff comparison feature in Visual Studio Code (VS Code) to compare two documents and show the differences.

The concrete steps are as follows.

Firstly, we initialize a Git repository in the current folder.

```
git init
```

Secondly, we add documents to the Git Repository.

```
git add -- document1.txt document2.txt
```

Finally, we run Diff to make a comparison between the two files. The differences will output in the terminal.

```
git diff -- document1.txt document2.txt
```

4.3 Difficulties in writing CMakeLists.txt

Initially, I failed to link the openblas library using CMakeLists.txt.

```
set(BLA_VENDER OpenBLAS)
find_package(BLAS REQUIRED)
if (BLAS_FOUND)
    target_link_libraries(pro BLAS::BLAS)
    message("openblas Found")
else()
    message("Can't find Openblas library")
endif()
```

I solved this problem by finding the exact address of the openblas library and changing BLAS::BLAS to the exact address of the openblas library.

```
set(BLA_VENDER OpenBLAS)
find_package(BLAS REQUIRED)
if (BLAS_FOUND)
    #message(STATUS "BLAS library found: ${BLAS_LIBRARIES}")//Find the address
    target_link_libraries(pro /usr/lib/x86_64-linux-gnu/libopenblas.so)
    message("openblas Found")
else()
    message("Can't find Openblas library")
endif()
```

5 Brief Summary

I gained a lot from this project.

First of all, it exercised my ability to call the library to use it and deepened my understanding of the library openblas. I read the code implementation of some functions of this library in this project, and mainly ran the function cblas_dgemm, and found that the openblas library has strong functions for linear algebra such as matrix multiplication.

Secondly, by continuously studying how to improve the efficiency of matrix multiplication, I have gained a deeper understanding of computer architecture as well as compilers. I found that the automatic optimizations such as -O3 of the compiler are really strong and will be much better than the optimizations we can do manually by ourselves. However, I still do experiments without -O3 and compare the manually optimized matrix multiplication with the plainest matrix multiplication, and actually found that the optimization we added manually did make the matrix multiplication rate increase, and this process deepened my understanding of the implementation of matrix multiplication optimization. Although the compiler's -O3 can optimize matrix multiplication by using loop unrolling, inline function, SIMD vectorization, and Automatic Parallelization, it does not do a good job of the optimization from the access to the matrix. In order to optimize the access to inventory. We use matrix packing, swapping the order of loops, and matrix chunking. Of course, implementing multi-threads is also a direction of our optimization.

Finally, I compared the speed of implementing cblas_dgemm() in x86 and arm architectures. The process of the research deepens my understanding of both x86 and arm CPU architectures. The ARM (Advanced RISC Machines) architecture uses a RISC (Reduced Instruction Set Computing) design philosophy, while the x86 architecture uses a CISC (Complex Instruction Set Computing) design philosophy. The difference in their designs may result in different optimizations in the cblas_dgemm() which contributed to different performances under different sizes of matrices.