中山大学计算机学院本科生实验报告

（**2021**学年秋季学期）

课程名称：高性能计算程序设计基础　　批改人：

实验	**LAB1**	专业	计算机科学与技术（超算）
学号	19335162	姓名	潘思晗
Email	pansh25@mail2.sysu.edu.cn	完成日期	2021.09.16

一、实验目的

> *1.*用 *C* 语言实现通用矩阵乘法，并记录计算的时间。
>
> *2.*实现基于算法分析的方法对矩阵乘法的优化，可选择 *Strassen* 算法或*Coppersmith–Winograd* 算法，并与 *GEMM* 进行对比分析。
>
> *3.*实现基于软件优化的方法对矩阵优化，如循环拆分向量化或内存重排，并与 *GEMM* 进行对比分析。
>
> *4.*优化后的矩阵乘法与*Intel MKL*函数库库的矩阵乘法函数，进行性能对比（相同规模矩阵乘法完成时间），并试着解释原因
>
> *5.*考虑大规模矩阵计算优化，从性能方面和可靠性方面进行思考与分析。

二、实验过程和核心代码

1.通用矩阵乘法

（1）算法描述

通用矩阵乘法通常定义为: $C = AB$， $C_{m,n} = \sum_{n=1}^{N} A_{m,n}B_{n,k}$

（2）核心代码

主要使用两个函数来完成，分别是随生成矩阵的函数`generate_matrix()`和计算矩阵乘法的函数`GEMM()`。

`generate_matrix()`函数根据传入的参数r=行数rows，c=列数columns来分配矩阵空间并用rand函数随机填充。可以设置随机数的范围（如整数0~10）。

```
int** generate_matrix(int r,int c)
{
    int** mat = (int**)malloc(sizeof(int*)*r);
    for(int i=0;i<r;i++){
        mat[i]=(int*)malloc(sizeof(int)*c);
    }

    srand((unsigned)time(0));
    for(int i=0;i<r;i++){
        for(int j=0;j<c;j++){
            mat[i][j] = rand()%10;
            //printf("%d",mat[i][j]);
        }
    }

    return mat;
}
```

`GEMM()`函数传入矩阵A、B并为矩阵C申请空间，通过循环嵌套实现矩阵的相乘，最后返回计算结果的指针。

```
int** GEMM(int** matA,int** matB)
{
    int** ans = (int**)malloc(sizeof(int*)*m);
    for(int i=0;i<m;i++){
        ans[i]=(int*)malloc(sizeof(int)*k);
    }
    for(int i=0;i<m;i++){
        for(int j=0;j<k;j++){
```

```
                ans[i][j] = 0;
                for(int t=0;t<n;t++){
                    ans[i][j] += matA[i][t]*matB[t][j];
                }
            }
        }
        return ans;
    }
```

在main函数中，调用两次`generate_matrix()`函数，然后标记时间，调用函数`GMEE()`，标记时间并计算得到运行总时长，最后打印出三个矩阵A、B、C以及完成矩阵乘法所用时间。

2.基于算法分析的矩阵乘法优化

（1）算法描述

选择Strassen算法对矩阵乘法进行优化。

Strassen基于分治的思想，Strassen算法将矩阵$A, B, C \in R^{n^2 \times n^2}$分别拆分为更小的矩阵

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

根据矩阵基本的运算法则，矩阵C可以由下列公式求出：

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

共需要八次小矩阵乘法和四次小矩阵加法计算。

Strassen通过减少矩阵相乘的次数，实现了降低算法复杂度

将A，B，C分解后，如下创建10个$\frac{n}{2} \times \frac{n}{2}$的矩阵$S_1, S_2, \ldots, S_{10}$

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

递归地计算7个矩阵积$P_1, P_2, \ldots, P_7$，每个矩阵P_i都是$\frac{n}{2} \times \frac{n}{2}$的

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A12 \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}
$$

最后通过P_i计算$C_{11}, C_{12}, C_{21}, C_{22}$

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

将4个矩阵合并，即得到最终的结果。

（2）核心代码

1.将矩阵扩展成可分块方阵

在矩阵分解的过程中，遇到的一个问题是：如何判断一个$n \times n$的矩阵是否能恰好分解为4个大小相同的矩阵。即判断n是否为2的幂。

一个基本的方法是将n不断除以2，直到余数不为0，判断当前的被除数是否为1，是则为2的幂，否则不是2的幂。这相当于通过右移检查n的二进制形式是否1000...0。

但这种方式比较繁琐，需要循环判断。为了提高效率，上网搜集资料发现可以用下面这行代码解决这一问题：

```
n & (n - 1) == 0
```

在使用Strassen计算前，先把矩阵通过补 0，扩展成为边长为2的整数次幂的方阵，才能够方便使用分治方法。

因此扩展矩阵的函数如下：

```
int extend_matrix(int m, int n, int k)
{
    int ext = m;
    if(ext<n)   ext = n;
    if(ext<k)   ext = k;

    while(ext & (ext-1) != 0)
        ext++;

    return ext;
}
```

2.矩阵分块

将矩阵A，B均分成4块，矩阵分块的函数如下：

```
double** divide_matrix(double** mat, int size, int div_pos)
//矩阵分块
{
    double **matrix = (double**)malloc(sizeof(double*)*(size/2));
    if(div_pos == 1){
        for (int i=0;i<size/2;i++){
            matrix[i] = (double *)malloc(sizeof(double)*(size/2));
            for (int j=0;j<size/2;j++){
                matrix[i][j] = mat[i][j];
            }
        }
    }
    else if(div_pos == 2){
        for (int i=0;i<size/2;i++){
            matrix[i]=(double *)malloc(sizeof(double)*(size/2));
```

```
            for (int j=0;j<size/2;j++){
                matrix[i][j] = mat[i][size/2+j];
            }
        }
    }
    else if(div_pos == 3){
        for (int i=0;i<size/2;i++) {
            matrix[i]=(double *)malloc(sizeof(double)*(size/2));
            for (int j=0;j<size/2;j++) {
                matrix[i][j] = mat[i+size/2][j];
            }
        }
    }
    else if(div_pos == 4){
        for (int i=0;i<size/2;i++){
            matrix[i]=(double *)malloc(sizeof(double)*(size/2));
            for (int j=0;j<size/2;j++){
                matrix[i][j] = mat[i+size/2][j+size/2];
            }
        }
    }
    return matrix;
}
```

3.分治和递归

Strassen算法核心就是分治思想。首先是分治，需要注意当分块小到了某一临界值（维度64）时，如果继续拆分会导致效率低下，因此当拆分到边长为64时，直接调用GEMM方法计算结果。

之后按照算法的表达式计算，Strassen算法的核心代码如下：

```
double** Strassen(double** matA, double** matB, int size)
{
    if(size <= 64){
        return GEMM(matA, matB, size);
    }
    else{
        double** a11 = divide_matrix(matA,size,1);
        double** a12 = divide_matrix(matA,size,2);
        double** a21 = divide_matrix(matA,size,3);
        double** a22 = divide_matrix(matA,size,4);
```

```
        double** b11 = divide_matrix(matB,size,1);
        double** b12 = divide_matrix(matB,size,2);
        double** b21 = divide_matrix(matB,size,3);
        double** b22 = divide_matrix(matB,size,4);

        double** p1 = Strassen(a11, sub_matrix(b12, b22, size/2),
size/2);
        double** p2 = Strassen(add_matrix(a11, a12, size/2), b22,
size/2);
        double** p3 = Strassen(add_matrix(a21, a22, size/2), b11,
size/2);
        double** p4 = Strassen(a22, sub_matrix(b21, b11, size/2),
size/2);
        double** p5 = Strassen(add_matrix(a11, a22, size/2),
add_matrix(b11, b22, size/2), size/2);
        double** p6 = Strassen(sub_matrix(a12, a22, size/2),
add_matrix(b21, b22, size/2), size/2);
        double** p7 = Strassen(sub_matrix(a11, a21, size/2),
add_matrix(b11, b12, size/2), size/2);
        double** C11 = add_matrix(add_matrix(p4, p5, size/2),
sub_matrix(p6, p2, size/2), size/2);

        double** C12 = add_matrix(p1, p2, size/2);
        double** C21 = add_matrix(p3, p4, size/2);
        double** C22 = add_matrix(sub_matrix(p1, p3, size/2),
sub_matrix(p5, p7, size/2), size/2);

        return merge_matrix(C11,C12,C21,C22,size);
    }
}
```

4.整合矩阵

将C11，C12，C21，C22四个矩阵整合得到最终答案，整合矩阵的函数`merge_matrix()`如下所示：

```
double** merge_matrix(double** mat1, double** mat2, double** mat3,
double** mat4, int size)     //矩阵合并
{
    double **matrix=(double**)malloc(sizeof(double*)*size);
    for (int i=0;i<size;i++)
            matrix[i]=(double *)malloc(sizeof(double)*size);
```

```
    for(int i=0;i<size/2;i++){
        for(int j=0;j<size/2;j++){
            matrix[i][j] = mat1[i][j];
        }
        for(int j=size/2;j<size;j++){
            matrix[i][j] = mat2[i][j-size/2];
        }
    }

    for(int i=size/2;i<size;i++){
        for(int j=0;j<size/2;j++){
            matrix[i][j] = mat3[i-size/2][j];
        }
        for(int j=size/2;j<size;j++){
            matrix[i][j] = mat2[i-size/2][j-size/2];
        }
    }
    return matrix;
}
```

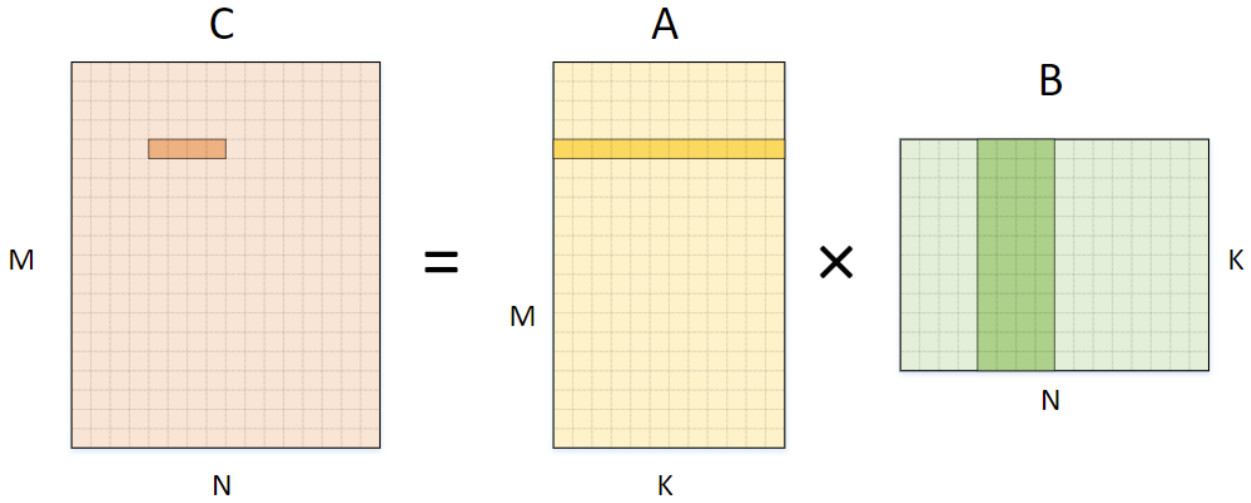
最后打印输出矩阵和运行时间，也可以计算GEMM算法和Strassen算法的用时对比，得出优化结论。

3.基于软件优化的矩阵乘法优化

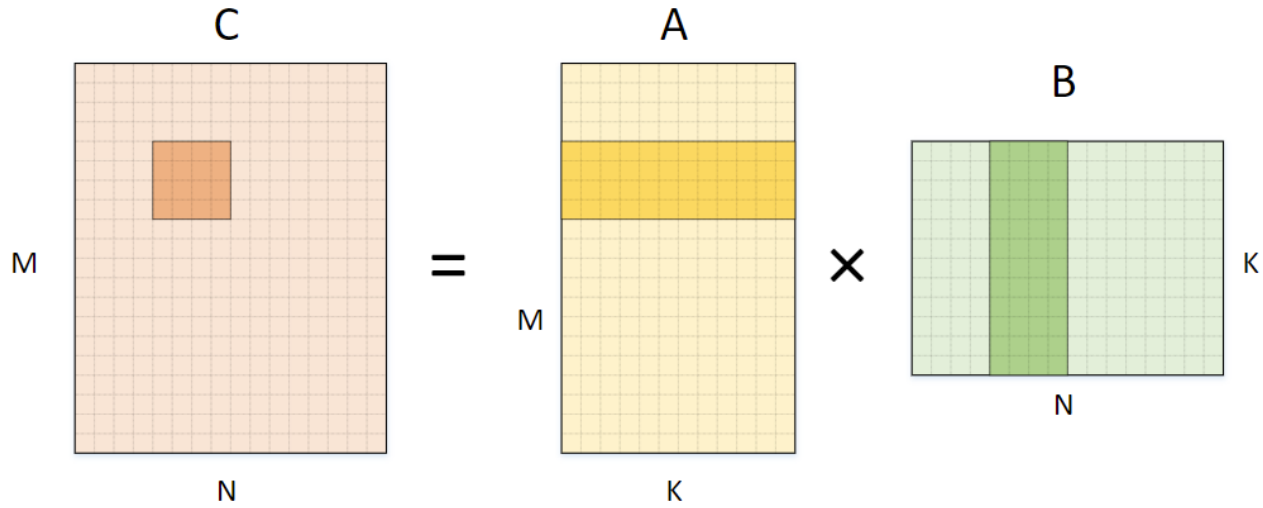
（1）方法描述

选择计算拆分方法进行矩阵的乘法优化。

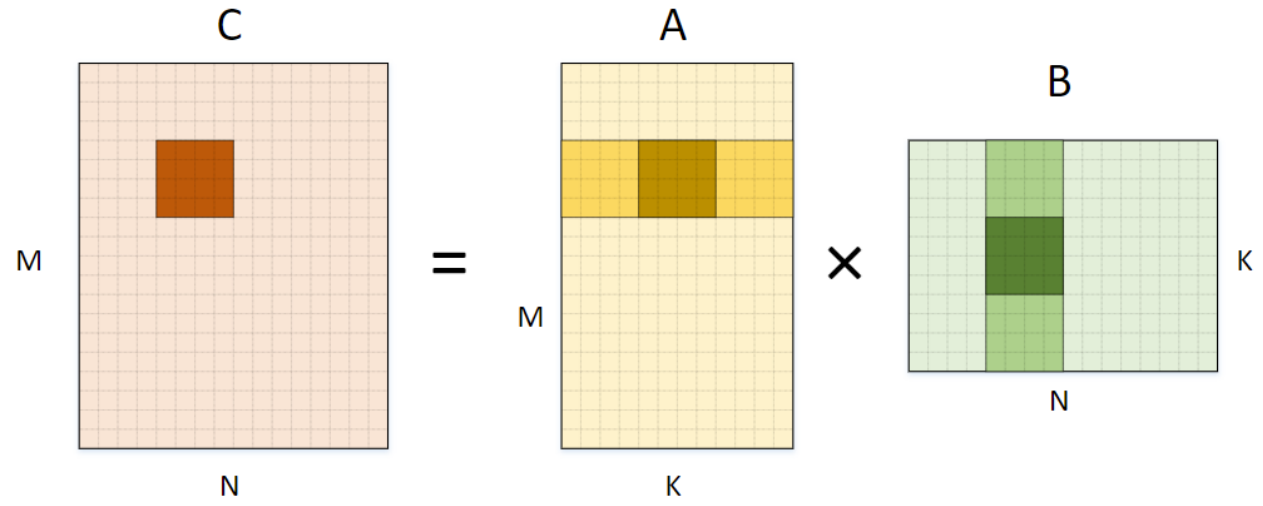
首先，将输出的计算拆分为 1×4 的小块，即将N维度拆分为两部分。计算该块输出时，需要使用A矩阵的1行，和B矩阵的4列。



类似地，我们可以继续拆分输出的M维度，从而在内侧循环中计算 4×4 输出，



最后，在计算 4×4 输出时，将维度K拆分，从而每次最内侧循环计算出输出矩阵C的4/K部分和。



在对M和N展开时，我们可以分别复用B和A的数据；在对K展开时，我们可以将部分和累加在寄存器中，最内层循环整个迭代结束时一次写到C的内存中。那么内存访问次数为：

$$
\begin{aligned}
MemAccess &= (4*4)*(\frac{M}{4}*\frac{N}{4})+\frac{1}{4}MNK+\frac{1}{4}MNK \\
&= MN+\frac{1}{2}MNK \\
&\approx \frac{1}{2}MNK
\end{aligned}
$$

（2）核心代码

首先，与上述图形描述不同的是，在设计中矩阵A是 m×n 矩阵，B是 n×k 矩阵。因此实际上我们是先拆分k维度，然后拆分m和n。

首先拆分k维度

```
double** divide_dimensionK(double** matA, double** matB)
{
    double** ans = (double**)malloc(sizeof(double*)*m);
    for(int i=0;i<m;i++){
        ans[i]=(double*)malloc(sizeof(double)*k);
    }
    for(int i=0;i<m;i++){
        for(int j=0;j<k;j+=4){
            ans[i][j+0] = 0;
            ans[i][j+1] = 0;
            ans[i][j+2] = 0;
            ans[i][j+3] = 0;
            for(int t=0;t<n;t++){
                ans[i][j+0] += matA[i][t]*matB[t][j+0];
                ans[i][j+1] += matA[i][t]*matB[t][j+1];
                ans[i][j+2] += matA[i][t]*matB[t][j+2];
                ans[i][j+3] += matA[i][t]*matB[t][j+3];
            }
        }
    }
    return ans;
}
```

接着在此基础上拆分M维度

```
double** divide_dimensionM(double** matA, double** matB)
{
    double** ans = (double**)malloc(sizeof(double*)*m);
```

```
    for(int i=0;i<m;i++){
        ans[i]=(double*)malloc(sizeof(double)*k);
    }
    for(int i=0;i<m;i+=4){
        for(int j=0;j<k;j+=4){
            ans[i+0][j+0] = 0;
            ans[i+0][j+1] = 0;
            ans[i+0][j+2] = 0;
            ans[i+0][j+3] = 0;
            ans[i+1][j+0] = 0;
            ans[i+1][j+1] = 0;
            ans[i+1][j+2] = 0;
            ans[i+1][j+3] = 0;
            ans[i+2][j+0] = 0;
            ans[i+2][j+1] = 0;
            ans[i+2][j+2] = 0;
            ans[i+2][j+3] = 0;
            ans[i+3][j+0] = 0;
            ans[i+3][j+1] = 0;
            ans[i+3][j+2] = 0;
            ans[i+3][j+3] = 0;
            for(int t=0;t<n;t++){
                ans[i+0][j+0] += matA[i+0][t]*matB[t][j+0];
                ans[i+0][j+1] += matA[i+0][t]*matB[t][j+1];
                ans[i+0][j+2] += matA[i+0][t]*matB[t][j+2];
                ans[i+0][j+3] += matA[i+0][t]*matB[t][j+3];
                ans[i+1][j+0] += matA[i+1][t]*matB[t][j+0];
                ans[i+1][j+1] += matA[i+1][t]*matB[t][j+1];
                ans[i+1][j+2] += matA[i+1][t]*matB[t][j+2];
                ans[i+1][j+3] += matA[i+1][t]*matB[t][j+3];
                ans[i+2][j+0] += matA[i+2][t]*matB[t][j+0];
                ans[i+2][j+1] += matA[i+2][t]*matB[t][j+1];
                ans[i+2][j+2] += matA[i+2][t]*matB[t][j+2];
                ans[i+2][j+3] += matA[i+2][t]*matB[t][j+3];
                ans[i+3][j+0] += matA[i+3][t]*matB[t][j+0];
                ans[i+3][j+1] += matA[i+3][t]*matB[t][j+1];
                ans[i+3][j+2] += matA[i+3][t]*matB[t][j+2];
                ans[i+3][j+3] += matA[i+3][t]*matB[t][j+3];
            }
        }
    }
    return ans;
}
```

最后拆分reduction维度N，方法与前面类似，篇幅原因不详细展开。

一个存在的问题是，使用循环拆分方法优化矩阵乘法，适用的矩阵其m,n,k需要是4的倍数。

4.使用Intel MKL的矩阵乘法函数

使用mkl的内置函数库直接进行矩阵乘法的运算。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mkl.h"

MKL_INT m, n, k;

int main()
{

    MKL_INT         lda, ldb, ldc;
    MKL_Complex8    alpha, beta;
    MKL_Complex8   *a, *b, *c;
    CBLAS_LAYOUT    layout = CblasRowMajor;
    CBLAS_TRANSPOSE transA = CblasNoTrans;
    CBLAS_TRANSPOSE transB = CblasNoTrans;
    MKL_INT         ma, na, mb, nb;

    /***************** 参数初始化 *****************/
    printf("ENTER 3 INTERGERS (512~2048) :");
    scanf("%d%d%d",&m,&n,&k);

    alpha.real = 1;
    alpha.imag = 0;
    beta.real = beta.imag = 0;

    if (transA == CblasNoTrans) {
        ma = m;
        na = k;
    } else {
        ma = k;
        na = m;
    }
```

```
    if (transB == CblasNoTrans) {
        mb = k;
        nb = n;
    } else {
        mb = n;
        nb = k;
    }

    a = (MKL_Complex8 *)mkl_calloc(ma*na, sizeof(MKL_Complex8),
64);
    b = (MKL_Complex8 *)mkl_calloc(mb*nb, sizeof(MKL_Complex8),
64);
    c = (MKL_Complex8 *)mkl_calloc(ma*nb, sizeof(MKL_Complex8),
64);

    /************** 矩阵与向量赋值 ******************/
    for (int i = 0; i < ma*na; i++) {
        a[i].real = (float)(i + 1);
        a[i].imag = (float)i;
    }
    for (int i = 0; i < mb*nb; i++) {
        b[i].real = (float)i;
        b[i].imag = (float)(i + 1);
    }

    if (layout == CblasRowMajor) {
        lda = na;
        ldb = nb;
        ldc = nb;
    } else {
        lda = ma;
        ldb = mb;
        ldc = ma;
    }

    clock_t begin,end;
    begin = clock();
    //      Call CGEMM subroutine ( C Interface )
    cblas_cgemm(layout, transA, transB, m, n, k, &alpha, a, lda, b,
ldb,
        &beta, c, ldc);
    end = clock();
    double time=(double)(end-begin)/CLOCKS_PER_SEC;
```

```
    //print time
    printf("THE TIME OF mkl: %f s\n",time);

    mkl_free(a);
    mkl_free(b);
    mkl_free(c);

    return 0;
}
```

5.大规模矩阵计算优化方向

（1）性能

> 提高大规模稀疏矩阵乘法性能。

在稀疏矩阵中的非零元的数量非常少（一般非零元数量少于同规模稠密矩阵元素的5%，甚至低于1%），为了避免大量零元造成的冗余计算和存储，不能使用和普通稠密矩阵一样的二维数组来保存，而使用其它数据结构来存储稀疏矩阵。

可以用三元组`(i,j,val)`来存储稀疏矩阵中的非零元素，其中`(i,j)`分别是非零元素所在的矩阵的行和列，`val`则是该非零元素的值，则稀疏矩阵可以用三元组序列来表示，由于矩阵的非零元素很少，该序列的规模远小于矩阵规模。对于零元素，则不做处理。

进行矩阵乘法运算时，利用三元组的性质和rpos数组的信息。为了得到非零元的乘积，只要对M.data[1,2...]中的每个元素，找到N.data中所有相应的元素相乘即可。

```
#define MAX_SIZE 1500
#define MAX_ROW 1500
class Triple
{
    public:
    int i,j;
    int val;
};
class RLSMatrix
{
    public:
    Triple data[MAX_SIZE];
    int rpos[MAX_ROW];
```

```
    int row_num,col_num,cnt;
};

void MultRLSMatrix(RLSMatrix M,RLSMatrix N,RLSMatrix &rs){
    int arow,brow,p,q,ccol,ctemp[MAX_ROW + 1],t,tp;
    if(M.col_num != N.row_num){
        return;
    }
    if(0 == M.cnt * N.cnt ){
        return;
    }

    rs.row_num = M.row_num;
    rs.col_num = N.col_num;
    rs.cnt = 0;

    for(arow = 1;arow <= M.row_num;arow++){
        for(ccol=1;ccol <= rs.col_num;ccol++){
            ctemp[ccol] = 0;
        }
        rs.rpos[arow] = rs.cnt + 1;
        if(arow < M.row_num){
            tp = M.rpos[arow+1];
        }else{
            tp = M.cnt + 1;
        }
        for(p = M.rpos[arow];p < tp;p++){
            brow = M.data[p].j;
            if(brow < N.row_num){
                t = N.rpos[brow + 1];
            }else{
                t = N.cnt + 1;
            }
            for(q = N.rpos[brow];q < t;q++){
                ccol = N.data[q].j;
                ctemp[ccol] += M.data[p].val * N.data[q].val;
            }
        }
        for(ccol = 1;ccol <= rs.col_num;ccol++){
            if(0 != ctemp[ccol]){
                if(++rs.cnt > MAX_SIZE){
                    return;
                }
```

```
                rs.data[rs.cnt].i = arow;
                rs.data[rs.cnt].j = ccol;
                rs.data[rs.cnt].val = ctemp[ccol];
            }
        }
    }
}
```

（**2**）可靠性

> 在内存有限的情况下，如何保证大规模矩阵乘法计算完成（*M, N, K >> 100000*），不触发内存溢出异常。

考虑使用矩阵拆分或者通过并行架构进行矩阵计算。

三、实验结果

1.通用矩阵乘法结果

首先用较小规模的矩阵验证一下随机生成矩阵的功能及计算功能

4×4矩阵计算：

```
ENTER 3 INTERGERS (512~2048) :4 4 4
matrixA(4*4):
8.343150 8.580889 5.336772 3.028657
9.974059 2.370983 0.593280 9.348735
8.826868 0.128788 1.108432 6.223029
3.589282 7.749870 8.320872 6.957610
matrixB(4*4):
8.343150 8.580889 5.336772 3.028657
9.974059 2.370983 0.593280 9.348735
8.826868 0.128788 1.108432 6.223029
3.589282 7.749870 8.320872 6.957610
matrixC(4*4)=matrixA*matrixB:
213.172138 116.095801 80.732871 159.772099
145.655447 163.735750 133.083168 121.110543
107.048621 124.418158 100.193039 78.132778
205.663646 104.166244 90.869520 183.511549
```

10×10矩阵计算：

```
ENTER 3 INTERGERS (512~2048) :10 10 10
matrixA(10*10):
8.263497 6.168401 9.199805 5.545518 8.153325 6.503494 4.605853 0.815149 5.711844 0.227973
4.719382 5.147557 3.089389 3.786737 6.519974 3.664052 1.743217 7.343974 0.905789 4.493240
9.812006 3.910337 1.564684 3.415937 4.430982 3.646962 8.311411 5.091403 4.302499 7.405622
2.717063 7.429121 2.831202 8.570513 8.831446 0.667745 7.139500 7.265236 6.817225 7.722404
3.272195 5.982543 2.307199 7.363506 2.909635 0.703146 9.295633 6.793725 1.891842 6.568499
9.377728 5.176855 5.875423 2.160405 5.149083 4.690695 3.429365 1.464278 0.578936 0.659810
1.886654 7.563707 6.167791 7.332682 6.261788 7.139805 1.699576 0.865200 6.545610 9.833064
4.411756 6.078066 3.694266 1.529893 5.378582 8.539384 9.620655 5.639515 1.937315 8.547319
1.499680 1.419111 8.958098 8.568072 5.710013 3.413800 6.773888 0.691549 1.706595 1.073031
8.585772 5.673086 1.157262 2.603229 5.149693 0.114139 2.638630 2.122562 9.565416 3.356731
matrixB(10*10):
8.263497 6.168401 9.199805 5.545518 8.153325 6.503494 4.605853 0.815149 5.711844 0.227973
4.719382 5.147557 3.089389 3.786737 6.519974 3.664052 1.743217 7.343974 0.905789 4.493240
9.812006 3.910337 1.564684 3.415937 4.430982 3.646962 8.311411 5.091403 4.302499 7.405622
2.717063 7.429121 2.831202 8.570513 8.831446 0.667745 7.139500 7.265236 6.817225 7.722404
3.272195 5.982543 2.307199 7.363506 2.909635 0.703146 9.295633 6.793725 1.891842 6.568499
9.377728 5.176855 5.875423 2.160405 5.149083 4.690695 3.429365 1.464278 0.578936 0.659810
1.886654 7.563707 6.167791 7.332682 6.261788 7.139805 1.699576 0.865200 6.545610 9.833064
4.411756 6.078066 3.694266 1.529893 5.378582 8.539384 9.620655 5.639515 1.937315 8.547319
1.499680 1.419111 8.958098 8.568072 5.710013 3.413800 6.773888 0.691549 1.706595 1.073031
8.585772 5.673086 1.157262 2.603229 5.149693 0.114139 2.638630 2.122562 9.565416 3.356731
matrixC(10*10)=matrixA*matrixB:
313.208904 291.533596 265.046949 306.777921 321.556064 209.207055 317.925275 217.096293 193.019977 257.552366
235.213900 238.393655 162.642268 188.072757 235.735940 163.884586 248.203625 187.639760 155.345663 217.537671
281.047244 299.466667 263.303630 269.224634 316.400394 224.702183 254.891441 159.541090 242.547312 244.004732
265.788506 337.679348 241.822738 334.777926 350.437520 207.450640 343.155391 262.687098 256.260985 345.140517
220.776543 287.303647 190.868082 246.599137 292.401485 193.467743 238.011886 194.250574 230.212443 294.704438
245.744148 218.009620 189.526860 192.307343 230.513998 167.490202 206.002606 146.148632 143.300521 171.582957
320.437410 286.775522 211.226155 280.018542 315.076921 148.257984 289.679242 226.505165 228.161646 246.697143
322.547704 318.981110 239.476752 249.289346 310.752674 233.730964 258.107742 186.811961 231.111545 282.025437
208.567865 231.019382 150.551457 235.288613 232.440239 133.590154 242.350168 176.512281 178.438282 256.935431
191.578920 202.903459 231.935180 246.816606 250.524193 156.741408 224.456771 136.612569 156.529954 155.642021
```

可见该方法实现了随机生成矩阵和矩阵乘法计算功能，下面查看使用通用矩阵乘法GEMM的时间性能。

矩阵维度是100×512和512×600时：

```
ENTER 3 INTERGERS (512~2048) :100 512 600
THE TIME OF GEMM: 0.142000 s
```

GEMM的运行时间大约是0.142秒。

矩阵维度是512×600和600×1000时：

```
ENTER 3 INTERGERS (512~2048) :512 600 1000
THE TIME OF GEMM: 2.035000 s
```

花费时间2.035秒。

矩阵维度提高到512×1000和1000×1024时：

```
ENTER 3 INTERGERS (512~2048) :512 1000 1024
THE TIME OF GEMM: 4.940000 s
```

大约花费4.94秒。

随着维度的增加，GEMM的劣势越来越明显，所花时间增长越来越快。

2.基于算法分析的优化结果

使用Strassen方法的运行结果如下。

对于不同的矩阵，Strassen算法的优化效果不同，下面比较对于不同矩阵用GEMM求解和Strassen求解的时间。

矩阵维度是100×100和100×100时：

```
ENTER 3 INTERGERS (512~2048) :100 100 100
THE TIME OF GEMM:     0.007000 s
THE TIME OF Strassen: 0.009000 s
```

通用矩阵乘法GEMM性能优于Strassen。

矩阵维度是512×512和512×512时：

```
ENTER 3 INTERGERS (512~2048) :512 512 512
THE TIME OF GEMM:     1.134000 s
THE TIME OF Strassen: 0.677000 s
```

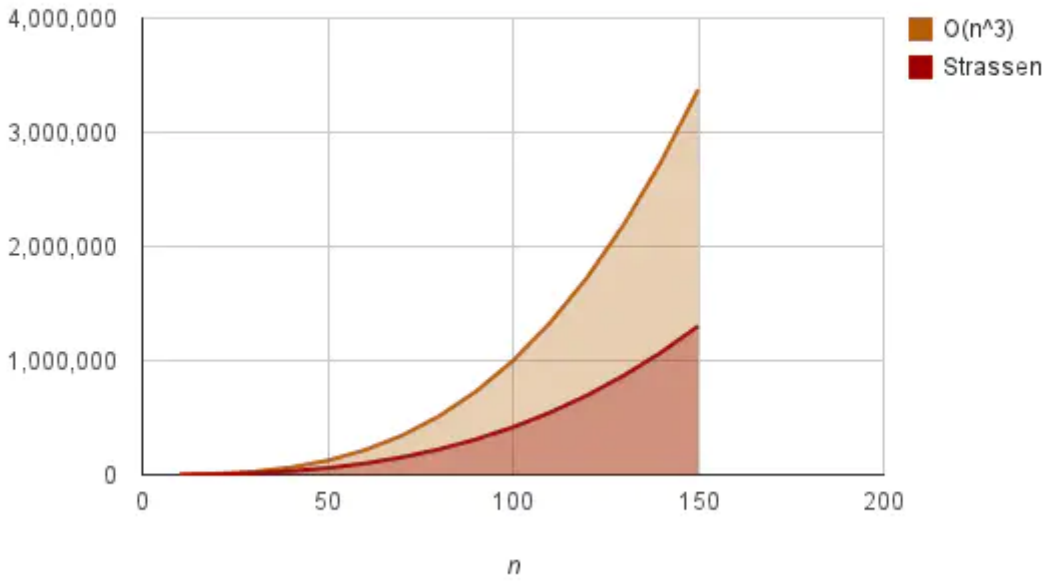
Strassen明显快于GEMM，加速比约为1.675

矩阵维度提高到1024×1024和1024×1024时：

```
ENTER 3 INTERGERS (512~2048) :1024 1024 1024
THE TIME OF GEMM:     19.256000 s
THE TIME OF Strassen: 3.837000 s
```

Strassen明显快于GEMM，加速比约为5.019

分析原因，此处实现的Strassen算法对大矩阵进行了补零，使其成为规模为2^n的矩阵，这会影响性能，此外，运行的硬件平台也可能影响算法性能。



这是网上搜索到的平凡算法和Strassen算法的性能差异，可以看到理性情况下n越大，Strassen算法的性能优势越明显。

3.基于软件优化的优化结果

使用循环拆分的方法实现的矩阵乘法优化结果如下。

矩阵维度是100×100和100×100时：

```
ENTER 3 INTERGERS (512~2048) :
100 100 100
THE TIME OF GEMM:     0.006000 s
THE TIME OF divide_dimension: 0.012000 s
```

优化结果不明显，使用GEMM反而更迅速。

矩阵维度是512×512和512×512时：

```
ENTER 3 INTERGERS (512~2048) :512 512 512
THE TIME OF GEMM:     1.085000 s
THE TIME OF divide_dimension: 0.722000 s
```

优化结果开始显现，加速比约为1.5。

矩阵维度提高到1024×1024和1024×1024时：

```
ENTER 3 INTERGERS (512~2048) :1024 1024 1024
THE TIME OF GEMM:      17.503000 s
THE TIME OF divide_dimension: 8.728000 s
```

加速比提升至2.0。

矩阵维度分别为512×1000和1000×1024时：

```
ENTER 3 INTERGERS (512~2048) :512 1000 1024
THE TIME OF GEMM:      8.202000 s
THE TIME OF divide_dimension: 3.331000 s
```

加速比2.46，优化效果更为显著。

4.使用Intel MKL的矩阵乘法函数

使用MKL无疑是所有方法中速度最快性能最好的，下面直接以前面几种方法速度较慢的大规模矩阵对比。

矩阵维度是512×512和512×512时：

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab$ ./mkl
ENTER 3 INTERGERS (512~2048) :512 512 512
THE TIME OF mkl: 0.020625 s
```

仅用时0.02秒。

矩阵维度提高到1024×1024和1024×1024时：

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab$ ./mkl
ENTER 3 INTERGERS (512~2048) :1024 1024 1024
THE TIME OF mkl: 0.140546 s
```

维数扩大到1024，也仅仅用时0.14秒，前面的优化方法都至少用时3s以上。

矩阵维度分别为2048×2048和2048×2048时：

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab$ ./mkl
ENTER 3 INTERGERS (512~2048) :2048 2048 2048
THE TIME OF mkl: 1.161494 s
```

仅仅用时1.161秒，跟GEMM算法计算512维度的矩阵速度差不多。

四、实验感想

这次实验主要针对矩阵乘法做了不同的优化，我了解了不同的算法以及关于计算时内存访问的知识。在实验过程中也遇到了许多值得思考的问题，总结如下。

（1）一开始矩阵中使用int类型的数据，但是发现从性能上和mkl库函数的调用上都遇到了问题，之后还是改用double类型，走了一些弯路。

（2）有关如何进行Strassen算法的编写，如何寻找拆分矩阵的“终点”，搜集了很多资料，总结之前的成果决定了拆分矩阵的分界线。

（3）关于Coppersmith–Winograd 算法的资料还是不多，没有更加深入地思考和研究。

（4）循环拆分法和向量法都是比较新的知识，还需要再学习。

总体而言，通过这次实验，我对于矩阵乘法的优化有了更加深入的了解。

五、参考资料

1.https://zhuanlan.zhihu.com/p/78657463

2.https://en.wikipedia.org/wiki/Strassen_algorithm

3.https://blog.csdn.net/qq_20880415/article/details/104332743

4.https://jackwish.net/2019/gemm-optimization.html

5.https://zh.wikipedia.org/wiki/%E7%9F%A9%E9%98%B5