

高性能计算实验报告 lab1

1. 通用矩阵乘法

GEMM

数学上，一个 $m \times n$ 的矩阵是一个由 m 行 n 列元素排列成的矩形阵列。矩阵是高等代数中常见的数学工具，也常见于统计分析等应用数学学科中。矩阵运算是数值分析领域中的重要问题。

通用矩阵乘法 (GEMM) 通常定义为：

$$C = AB$$

$$C_{m,n} = \sum_{n=1}^N A_{m,n} B_{n,k}$$

请根据定义用C/C++语言实现一个矩阵乘法：

题目：用C/C++语言实现通用矩阵乘法

输入：M, N, K三个整数 (512 ~ 2048)

问题描述：随机生成MN和NK的两个矩阵A,B,对这两个矩阵做乘法得到矩阵C.

输出：A,B,C三个矩阵以及矩阵计算的时间

代码

完整代码见文件gemm.cpp

变量说明：

A, B是两个矩阵，用来做GEMM的矩阵乘法。矩阵C=A*B，是运算结果

A是 $m \times n$ 的矩阵

B是n*k的矩阵

C是m*k的矩阵

需要用到的头文件：

```
1 #include<iostream>
2 #include<vector>
3 #include<time.h>
4 using namespace std;
```

首先随机生成两个m*n和n*k大小的矩阵，大小在50以内，数据类型为浮点型：

```
1 int main() {
2     ...
3     cin>>m>>n>>k;
4     srand(time(NULL));
5     vector<vector<double>> A(m, vector<double>(n, 0)),
6                                B(n, vector<double>(k, 0)),
7                                C(m, vector<double>(k, 0));
8     for(int i=0; i<m; ++i)
9         for(int j=0; j<n; ++j) A[i][j] = rand() % 50;
10    for(int i=0; i<n; ++i)
11        for(int j=0; j<k; ++j) B[i][j] = rand() % 50;
12    ...
13 }
```

利用GEMM的乘法公式，对两个随机矩阵进行乘法运算，并返回结果“矩阵C”：

```

1 //常规GEMM进行矩阵乘法
2 vector<vector<double>> GEMM(vector<vector<double>> A,
3   vector<vector<double>> B, int m, int n, int k) {
4     vector<vector<double>> C(m, vector<double>(k, 0));
5     int sumt;
6     for(int i=0;i<k;++i) {
7       for(int j=0;j<m;++j) {
8         sumt=0;
9         for(int x=0;x<n;++x) sumt+=A[j][x]*B[x][i];
10        C[j][i]=sumt;
11      }
12    }
13  }

```

对整个计算过程进行计时，用于与优化之后的算法进行对比：

```

1 int main() {
2   ...
3   srand(time(NULL));
4   time_t st,fin;
5   st=clock(); //开始计时
6   C=GEMM(A,B,m,n,k); //调用计算矩阵乘法的函数
7   fin=clock(); //结束计时
8   cout<<"time cost = "<<double(fin -st)/CLOCKS_PER_SEC<<" s"
9   <<endl;
10  //计算并输出运行时间
11 }

```

最后还需要输出生成的随即矩阵A、B以及结果C：

```

1 //输出矩阵
2 void PRINT(vector<vector<double>> x,int m,int n) {
3   for(int i=0;i<m;++i) {
4     for(int j=0;j<n;++j) cout<<x[i][j]<<" ";
5     cout<<endl;
6   }
7   cout<<endl;
8 }

```

```
1 int main() {
2     ...
3     //当矩阵过大时，如A, B都是1024*1024*1024的矩阵，将这些矩阵都输出过于冗
4     //因此只在计算小型矩阵乘法时输出，用于查矩阵乘法是否正确
5     //print
6     cout<<"A:"<<endl;
7     PRINT(A, m, n);
8     cout<<"B:"<<endl;
9     PRINT(B, n, k);
10    cout<<"C=A*B:"<<endl;
11    PRINT(C, m, k);
12    ...
13 }
```

运行结果以及时间

输出运行结果：

```
wwwj@ubuntu:~/lab1$ ./gemm
3 4 5
time cost = 1.4e-05 s
A:
31 38 36 24
4 27 48 16
6 16 46 1

B:
49 19 37 47 19
30 25 1 18 34
28 9 19 10 28
29 24 6 37 5

C=A*B:
4363 2439 2013 3389 3009
2814 1567 1183 1746 2418
2091 952 1118 1067 1951
```

在matlab中进行验算：

结果正确，算法无误。

```
A=[31,38,36,24;  
 4,27,48,16;  
 6,16,46,1];  
B=[49,19,37,47,19;  
 30,25,1,18,34;  
 28,9,19,10,28;  
 29,24,6,37,5];  
C=A*B;
```

命令窗口

C =

4363	2439	2013	3389	3009
2814	1567	1183	1746	2418
2091	952	1118	1067	1951

在终端编译并运行后，输入不同数值，可看到对于不同大小的矩阵运算结果也不同：

```
wwwj@ubuntu:~/lab1$ g++ gemm.cpp -o gemm  
wwwj@ubuntu:~/lab1$ ./gemm  
100 200 300  
time cost = 0.069532 s  
wwwj@ubuntu:~/lab1$ ./gemm  
128 128 128  
time cost = 0.034667 s  
wwwj@ubuntu:~/lab1$ ./gemm  
256 256 256  
time cost = 0.158958 s  
wwwj@ubuntu:~/lab1$ ./gemm  
512 512 512  
time cost = 1.40912 s  
wwwj@ubuntu:~/lab1$ ./gemm  
1024 1024 1024  
time cost = 12.6819 s
```

2. 通用矩阵乘法优化

1) 基于算法分析的方法进行优化

典型的算法包括 Strassen 算法和 Coppersmith–Winograd 算法，本实验中我完成了 Strassen 算法。

Strassen 算法

完整代码见文件strassen.cpp

代码

由于strassen是要将矩阵不断二分并递归，因此矩阵A，B必须是两个方阵并且矩阵大小 $n = 2^x \quad x = 0, 1, 2\dots$

因此在输入时只输入一个矩阵的大小n，A，B，C三个矩阵都是n*n的。

并且判断是否 $n = 2^x \quad x = 0, 1, 2\dots$ ，如果不是就直接返回：

```
1 int main() {
2     ...
3     if( (n & (n-1)) != 0 ){//judge n if 2^x
4         cout<<n<<" isn't power of 2, we can't us strassen to
5             computer."<<endl;
6         return 0;
7     }
8 }
```

然后生成两个随机矩阵A、B（与之前GEMM中说到的一样），并调用strassen函数进行矩阵乘法的运算：

```
1 int main() {
2     ...
3     //computer
4     st=clock();
5     C=strassen(A,B,n); //调用函数，输入的三个参数分别是矩阵A、B，以及矩阵大
6     小n
7     fin=clock();
8     cout<<"time cost = "<<double(fin - st)/CLOCKS_PER_SEC<<" s"
9     <<endl;
10    ...
11 }
```

由于在strassen算法中会遇到大量的矩阵加减法运算，因此编写两个函数进行这两个操作：

```
1 //将两个矩阵相加
2 vector<vector<double>> ADDmatrix(vector<vector<double>> A,
3                                         vector<vector<double>> B, int n) {
4     for(int i=0;i<n;++i) {
5         for(int j=0;j<n;++j) A[i][j] += B[i][j];
6     }
7 }
```

```

6     return A;
7 }
8
9 //将两个矩阵相减
10 vector<vector<double>> SUBmatrix(vector<vector<double>> A,
11    vector<vector<double>> B, int n) {
12     for(int i=0;i<n;++i) {
13         for(int j=0;j<n;++j) A[i][j]=A[i][j]-B[i][j];
14     }
15     return A;
16 }
```

函数strassen()实现步骤

在函数中首先判断矩阵的大小是否为1，即判断矩阵是否还能继续划分并递归下去，如果不能则直接对这两个矩阵进行乘法运算并返回，如果可以继续划分就让 $n = \frac{n}{2}$ ，并进行下一步计算。

```

1 vector<vector<double>> strassen(vector<vector<double>> A,
2    vector<vector<double>> B, int n) {
3     ...
4     if(n==1) {
5         C[0][0]=A[0][0]*B[0][0];
6         return C;
7     }
8     n=n/2;
9 }
```

然后，将矩阵A,B,C分解：

```

1 vector<vector<double>> strassen(vector<vector<double>> A,
2                                     vector<vector<double>> B, int n) {
3     ...
4     //break down matrix A B
5     for(int i=0;i<n;i++) {
6         for(int j=0;j<n;j++) {
7             A11[i][j]=A[i][j],   A12[i][j]=A[i][j+n];
8             A21[i][j]=A[i+n][j], A22[i][j]=A[i+n][j+n];
9             B11[i][j]=B[i][j],   B12[i][j]=B[i][j+n];
10            B21[i][j]=B[i+n][j], B22[i][j]=B[i+n][j+n];
11        }
12    ...
13 }

```

再如下创建10个 $\frac{n}{2} \times \frac{n}{2}$ 的矩阵 S_1, S_2, \dots, S_{10} (花费时间 $\Theta(n^2)$)

$$\begin{aligned}
S1 &= B12 - B22 \\
S2 &= A11 + A12 \\
S3 &= A21 + A22 \\
S4 &= B21 - B11 \\
S5 &= A11 + A22 \\
S6 &= B11 + B22 \\
S7 &= A12 - A22 \\
S8 &= B21 + B22 \\
S9 &= A11 - A21 \\
S10 &= B11 + B12
\end{aligned}$$

```

1 vector<vector<double>> strassen(vector<vector<double>> A,
2                                     vector<vector<double>> B, int n) {
3     ...
4     //computer S1,S2,...,S10
5     //S1=B12-B22, S2=A11+A12
6     s1=SUBmatrix(B12,B22,n);
7     s2=ADDmatrix(A11,A12,n);
8     //S3=A21+A22, S4=B21-B11
9     s3=ADDmatrix(A21,A22,n);

```

```

9   s4=SUBmatrix(B21,B11,n);
10  //S5=A11+A22, S6=B11+B22
11  s5=ADDmatrix(A11,A22,n);
12  s6=ADDmatrix(B11,B22,n);
13  //S7=A12-A22, S8=B21+B22
14  s7=SUBmatrix(A12,A22,n);
15  s8=ADDmatrix(B21,B22,n);
16  //S9=A11-A21, S10=B11+B12
17  s9=SUBmatrix(A11,A21,n);
18  s10=ADDmatrix(B11,B12,n);
19  ...
20 }
```

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

$$\begin{aligned}
P1 &= A11 \cdot S1 = A11 \cdot B12 - A11 \cdot B22 \\
P2 &= S2 \cdot B22 = A11 \cdot B22 + A12 \cdot B22 \\
P3 &= S3 \cdot B11 = A21 \cdot B11 + A22 \cdot B11 \\
P4 &= A22 \cdot S4 = A22 \cdot B21 - A22 \cdot B11 \\
P5 &= S5 \cdot S6 = A11 \cdot B11 + A11 \cdot B22 + A22 \cdot B11 + A22 \cdot B22 \\
P6 &= S7 \cdot S8 = A12 \cdot B21 + A12 \cdot B22 - A22 \cdot B21 - A22 \cdot B22 \\
P7 &= S9 \cdot S10 = A11 \cdot B11 + A11 \cdot B12 - A21 \cdot B11 - A21 \cdot B12
\end{aligned}$$

```

1 vector<vector<double>> strassen(vector<vector<double>> A,
2   vector<vector<double>> B, int n) {
3     ...
4     //Start recursion
5     //p1=A11*s1, p2=s2*B22, p3=s3*B11, p4=A22*s4
6     //p5=s5*s6, p6=s7*s8, p7=s9*s10
7     p1=strassen(A11,s1,n);
8     p2=strassen(s2,B22,n);
9     p3=strassen(s3,B11,n);
10    p4=strassen(A22,s4,n);
11    p5=strassen(s5,s6,n);
12    p6=strassen(s7,s8,n);
13    p7=strassen(s9,s10,n);
14 }
```

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

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

```
1 vector<vector<double>> strassen(vector<vector<double>> A,
2                                     vector<vector<double>> B, int n) {
3     ...
4     //computer four parts of matrix C
5     //C11=p5+p4-p2+p6
6     C11=ADDmatrix(p5,p4,n);
7     C11=SUBmatrix(C11,p2,n);
8     C11=ADDmatrix(C11,p6,n);
9     //C12=p1+p2, C21=p3+p4
10    C12=ADDmatrix(p1,p2,n);
11    C21=ADDmatrix(p3,p4,n);
12    //C22=p5+p1-p3-p7
13    C22=ADDmatrix(p5,p1,n);
14    C22=SUBmatrix(C22,p3,n);
15    C22=SUBmatrix(C22,p7,n);
16 }
```

将他们合成一个矩阵C并返回:

```

1 vector<vector<double>> strassen(vector<vector<double>> A,
2                                     vector<vector<double>> B, int n) {
3     ...
4     //Integration matrix C
5     for(int i=0;i<n;++i) {
6         for(int j=0;j<n;++j) {
7             C[i][j]=C11[i][j],C[i][j+n]=C12[i][j];
8             C[i+n][j]=C21[i][j],C[i+n][j+n]=C22[i][j];
9         }
10    }
11    return C;
12 }
```

和前面的GEMM一样也会有一个函数PRINT()用来输出矩阵，此处不再赘述。

运行结果以及时间

输出运行结果：

```

wwwj@ubuntu:~/lab1$ g++ strassen.cpp -o strassen
wwwj@ubuntu:~/lab1$ ./strassen
4
time cost = 0.001321 s
A:
7 7 2 8
29 10 1 13
22 3 42 29
18 46 24 3

B:
13 34 44 41
3 31 21 38
5 42 17 26
23 46 28 30

C=A*B:
306 907 713 845
711 1936 1867 1985
1172 3939 2557 2978
561 3184 2250 3200
```

在matlab中进行验算：

结果正确，算法无误。

```

A=[7, 7, 2, 8;
29, 10, 1, 13;
22, 3, 42, 29;
18, 46, 24, 3];
B=[13, 34, 44, 41;
3, 31, 21, 38;
5, 42, 17, 26;
23, 46, 28, 30];
C=A*B;

```

命令窗口

```

C =

```

306	907	713	845
711	1936	1867	1985
1172	3939	2557	2978
561	3184	2250	3200

运行时间：

不设置阈值，即一直迭代到矩阵大小为1：

```

wwwj@ubuntu:~/lab1$ g++ strassen.cpp -o strassen
wwwj@ubuntu:~/lab1$ ./strassen
128
time cost = 4.87875 s
wwwj@ubuntu:~/lab1$ ./strassen
256
time cost = 33.6877 s
wwwj@ubuntu:~/lab1$ ./strassen
512
time cost = 242.435 s

```

我们发现运行时间过长，可以在利用strassen算法进行矩阵乘法时设置一个阈值，比如在矩阵划分到大小 $n = 1, 2, 4, 8, \dots$ 时不再进行划分，而是直接利用GEMM算法进行计算：

```

1 vector<vector<double>> strassen(vector<vector<double>> A,
2   vector<vector<double>> B, int n) {
3     ...
4     if (n==1) {
5       C[0][0]=A[0][0]*B[0][0];
6       return C;
7     }
8     else if (n==16) return C=GEMM(A,B,n,n,n); //不再继续使用strassen，开始调用GEMM
9   ...
}

```

设置一定阈值， $n=16$ 时停止迭代，直接用GEMM得到此时的小矩阵乘法结果：

比完全用strassen要快很多，但是仍比只使用GEMM要慢。

```
wwwj@ubuntu:~/lab1$ g++ strassen.cpp -o strassen
wwwj@ubuntu:~/lab1$ ./strassen
128
time cost = 0.047312 s
wwwj@ubuntu:~/lab1$ ./strassen
256
time cost = 0.321031 s
wwwj@ubuntu:~/lab1$ ./strassen
512
time cost = 2.26019 s
wwwj@ubuntu:~/lab1$ ./strassen
1024
time cost = 15.9974 s
```

阈值n=32，此时的运行时间和只用GEMM的运行时间相差不多，甚至更快：

```
wwwj@ubuntu:~/lab1$ g++ strassen.cpp -o strassen
wwwj@ubuntu:~/lab1$ ./strassen
128
time cost = 0.029044 s
wwwj@ubuntu:~/lab1$ ./strassen
256
time cost = 0.185917 s
wwwj@ubuntu:~/lab1$ ./strassen
512
time cost = 1.33171 s
wwwj@ubuntu:~/lab1$ ./strassen
1024
time cost = 9.42962 s
```

阈值n=64：

```
wwwj@ubuntu:~/lab1$ g++ strassen.cpp -o strassen
wwwj@ubuntu:~/lab1$ ./strassen
128
time cost = 0.031501 s
wwwj@ubuntu:~/lab1$ ./strassen
256
time cost = 0.155505 s
wwwj@ubuntu:~/lab1$ ./strassen
512
time cost = 1.08812 s
wwwj@ubuntu:~/lab1$ ./strassen
1024
time cost = 7.70933 s
```

可以看到，此时对两个方阵进行矩阵乘法时都要比只用strassen更快。

优化GEMM

完整代码见文件*gemm_strassen.cpp*

代码

由于计算的矩阵不都是方阵，且即使是方阵也不可能大小都是 $n = 2^x \quad x = 0, 1, 2\dots$ （即有可能在进行strassen算法运算时，进行到一定地步矩阵就无法再二分递归）。

因此我们可以利用strassen算法能优化部分矩阵的特点对GEMM算法进行优化。

首先我们这里输入的三个数m、n、k分别表示矩阵A、B、C的大小。

在输入这三个数据后就进行判断这两个矩阵能否使用strassen算法进行矩阵乘法：如果矩阵A、B是方阵且大小能被2整除，说明这两个矩阵是可以用strassen算法进行一定程度的递归运算，就使用strassen算法；如果不满足这些条件，说明不能使用strassen算法，因而直接使用GEMM：

```
1 int main() {
2     ...
3     //computer
4     st=clock();
5     if(m==n && n==k && (n%2==0) ) {
6         //如果矩阵A、B都是方阵，并且矩阵边长是2的倍数，就能够使用strassen
7         //算法
8         cout<<"firstly, use strassen."<<endl;
9         C=strassen(A,B,n);
10        //由于用strassen的一定是方阵，因此只需要传递一个矩阵大小的参数n
11        //等到在strassen中递归时无法将一个方阵划分为四个相等的小方阵，就再
12        //调用GEMM
13    }
14    else{
15        //如果第一次就不能使用strassen，就直接使用GEMM
16        cout<<"firstly, use GEMM."<<endl;
17        C=GEMM(A,B,m,n,k);
18    }
19    fin=clock();
20    cout<<"time cost = "<<double(fin - st)/CLOCKS_PER_SEC<<" s"
<<endl;
21    ...
22 }
```

函数strassen()和上面介绍的一样，此处不再赘述，只是有可能矩阵划分到一定地步就不能完全二分了，此时就转而调用函数GEMM()，因此在函数strassen()的开始还有个判断条件：

```
1 vector<vector<double>> strassen(vector<vector<double>> A,
2     vector<vector<double>> B,  int n) {
3
4     if(n==1) {
5         C[0][0]=A[0][0]*B[0][0];
6         return C;
7     }
8     //判断是否能继续二分下去
```

```

9     else if(n%2!=0) {
10        //递归下来调用这个函数的矩阵很多，这个if条件是为了不输出过多信息
11        if(!marksmall) {
12            marksmall=true;
13            cout<<"Cannot be divided into smaller matrices, use
14            GEMM."<<endl;
15            //当有一个小矩阵不能再划分时，输出一条信息
16        }
17        return C=GEMM(A,B,n,n,n);
18        //此时的n是个奇数
19    }
20    //设置一定阈值，递归到划分的小矩阵小于这个值，就不再使用strassen，而使
21    //用GEMM这样更节省时间
22    else if(n < 200) {
23        if(!marklimlit) {
24            marklimlit=true;
25            cout<<"Reached the limit n="<<n<<" ,use GEMM."<<endl;
26            //当达到一定阈值时，输出一条信息
27        }
28        return C=GEMM(A,B,n,n,n);
29    }
30    ...
31 }
```

运行结果以及时间

输出运行结果：

```

wwwj@ubuntu:~/lab1$ ./gemm_strassen
3 4 5
firstly, use GEMM.
time cost = 4e-05 s
A:
33 3 36 32
34 29 8 29
34 43 23 30

B:
17 13 15 2 9
1 26 35 38 8
13 15 9 43 4
12 6 48 12 41

C=A*B:
1416 1239 2460 2112 1777
1059 1490 2989 1862 1759
1280 2085 3662 3051 1972

```

在matlab中进行验算：

结果正确，算法无误。

```

- A=[33, 3, 36, 32;
  34, 29, 8, 29;
  34, 43, 23, 30];
- B=[17, 13, 15, 2, 9;
  1, 26, 35, 38, 8;
  13, 15, 9, 43, 4;
  12, 6, 48, 12, 41];
- C=A*B;

```

命令窗口

```

>> C
C =

```

1416	1239	2460	2112	1777
1059	1490	2989	1862	1759
1280	2085	3662	3051	1972

运行时间：

当矩阵A，B不是方阵时，会直接用GEMM：

```

wwwj@ubuntu:~/lab1$ g++ gemm_strassen.cpp -o gemm_strassen
wwwj@ubuntu:~/lab1$ ./gemm_strassen
200 400 500
firstly, use GEMM.
time cost = 0.437557 s

```

当矩阵A，B时方阵，但不是2的倍数，也只能直接使用GEMM：

```

wwwj@ubuntu:~/lab1$ g++ gemm_strassen.cpp -o gemm_strassen
wwwj@ubuntu:~/lab1$ ./gemm_strassen
555 555 555
firstly, use GEMM.
time cost = 2.00973 s

```

当是矩阵时，优化结果要比之前只是用GEMM要快，达到了优化的目的：

```

wwwj@ubuntu:~/lab1$ g++ gemm.cpp -o gemm
wwwj@ubuntu:~/lab1$ ./gemm
704 704 704
time cost = 3.68114 s → GEMM的运行时间
wwwj@ubuntu:~/lab1$ g++ gemm_strassen.cpp -o gemm_strassen
wwwj@ubuntu:~/lab1$ ./gemm_strassen
704 704 704
firstly, use strassen. → 可看到在迭代到n=88时，开始使用GEMM
Reached the limit n=88 ,use GEMM.
time cost = 2.58627 s → 优化后阈值设为100的运行时间
wwwj@ubuntu:~/lab1$ g++ gemm_strassen.cpp -o gemm_strassen
wwwj@ubuntu:~/lab1$ ./gemm_strassen
704 704 704
firstly, use strassen.
Reached the limit n=176 ,use GEMM. → 阈值为200的运行时间
time cost = 2.68484 s → n=176时，开始使用GEMM

```

```

wwwj@ubuntu:~/lab1$ g++ gemm.cpp -o gemm
wwwj@ubuntu:~/lab1$ ./gemm
1408 1408 1408
time cost = 42.7226 s → 只用GEMM的运行时间
wwwj@ubuntu:~/lab1$ g++ gemm_strassen.cpp -o gemm_strassen
wwwj@ubuntu:~/lab1$ ./gemm_strassen
1408 1408 1408
firstly, use strassen. → 阈值为100, n=88使用GEMM
Reached the limit n=88 ,use GEMM. → 运行时间甚至不足原来只用GEMM的一半
time cost = 18.8012 s → 优化后时间减少到原来的一半
wwwj@ubuntu:~/lab1$ g++ gemm_strassen.cpp -o gemm_strassen
wwwj@ubuntu:~/lab1$ ./gemm_strassen
1408 1408 1408
firstly, use strassen. → 阈值为200, n=176使用GEMM
Reached the limit n=176 ,use GEMM. → 优化后时间减少到原来的一半
time cost = 19.2039 s →

```

2) 基于软件优化的方法进行优化

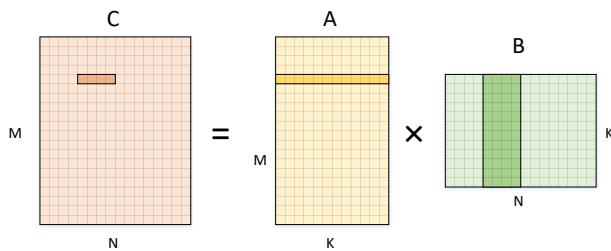
如循环拆分向量化和内存重排，下面主要通过访存局部性，来逐步进行优化。

完整代码见文件gemm2.cpp

a. 矩阵乘计算 1×4 输出

代码

由于 $A * B$ 要A的每一行去乘B的每一列然后进行求和，对B来说访问列上的每一个数时由于局部性原理缓存的是该数所处行上周围的数，因此每次进行乘法运算时连续对B的4列进行乘法求和运算，这样就可以每次访问：



相较于之前的GEMM。只改变函数GEMM()：

```
1 vector<vector<double>> GEMM(vector<vector<double>> A,
2     vector<vector<double>> B, int M, int N, int K) {
3     vector<vector<double>> C(M, vector<double>(K, 0));
4     for(int m=0; m<M; m++) {
5         for(int k=0; k<K; k+=4) {
6             for(int n=0; n<N; n++) {
7                 //first
8                 C[m][k+0] += A[m][n] * B[n][k+0], C[m][k+1] += A[m]
9 [n] * B[n][k+1];
10                C[m][k+2] += A[m][n] * B[n][k+2], C[m][k+3] += A[m]
11 [n] * B[n][k+3];
12            }
13        }
14    }
15    return C;
16 }
```

运行结果以及时间

(只是改变了一次运算的元素个数，算法实际并未改变，因此对于运算结果的正确与否这里不再验证)

运行时间：

优化后：

```
wwwj@ubuntu:~/lab1$ g++ gemm2.cpp -o gemm2
wwwj@ubuntu:~/lab1$ ./gemm2
100 200 300
time cost = 0.033473 s
wwwj@ubuntu:~/lab1$ ./gemm2
128 128 128
time cost = 0.008509 s
wwwj@ubuntu:~/lab1$ ./gemm2
256 256 256
time cost = 0.067685 s
wwwj@ubuntu:~/lab1$ ./gemm2
512 512 512
time cost = 0.463145 s
wwwj@ubuntu:~/lab1$ ./gemm2
1024 1024 1024
time cost = 4.35199 s
```

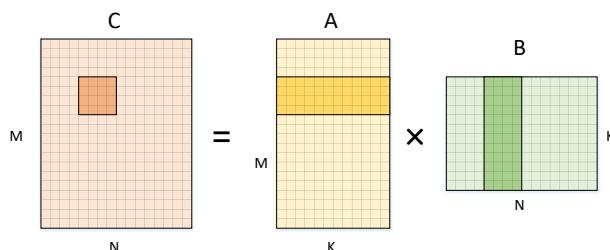
之前只用GEMM的运行时间：

```
wwwj@ubuntu:~/lab1$ g++ gemm.cpp -o gemm
wwwj@ubuntu:~/lab1$ ./gemm
100 200 300
time cost = 0.069532 s
wwwj@ubuntu:~/lab1$ ./gemm
128 128 128
time cost = 0.034667 s
wwwj@ubuntu:~/lab1$ ./gemm
256 256 256
time cost = 0.158958 s
wwwj@ubuntu:~/lab1$ ./gemm
512 512 512
time cost = 1.40912 s
wwwj@ubuntu:~/lab1$ ./gemm
1024 1024 1024
time cost = 12.6819 s
```

可以看出时间几乎是原来的 $\frac{1}{3}$ ，优化效果很明显。

b. 矩阵乘计算 4×4 输出

代码



依然是利用局部性原理，对数据进行复用，减少内存开销：

```

1  vector<vector<double>> GEMM(vector<vector<double>> A,
2  vector<vector<double>> B, int M, int N, int K) {
3      vector<vector<double>> C(M, vector<double>(K, 0));
4      for(int m=0;m<M;m+=4) {
5          for(int k=0;k<K;k+=4) {
6              for(int n=0;n<N;n++) {
7                  //second
8                  C[m+0][k+0] += A[m+0][n]*B[n][k+0], C[m+0]
9 [k+1] += A[m+0][n]*B[n][k+1];
10                 C[m+0][k+2] += A[m+0][n]*B[n][k+2], C[m+0]
11 [k+3] += A[m+0][n]*B[n][k+3];
12                 C[m+1][k+0] += A[m+1][n]*B[n][k+0], C[m+1]
13 [k+1] += A[m+1][n]*B[n][k+1];
14                 C[m+1][k+2] += A[m+1][n]*B[n][k+2], C[m+1]
15 [k+3] += A[m+1][n]*B[n][k+3];
16                 C[m+2][k+0] += A[m+2][n]*B[n][k+0], C[m+2]
17 [k+1] += A[m+2][n]*B[n][k+1];
18                 C[m+2][k+2] += A[m+2][n]*B[n][k+2], C[m+2]
19 [k+3] += A[m+2][n]*B[n][k+3];
20             }
21         }
22     }
23     return C;
24 }
```

运行结果以及时间

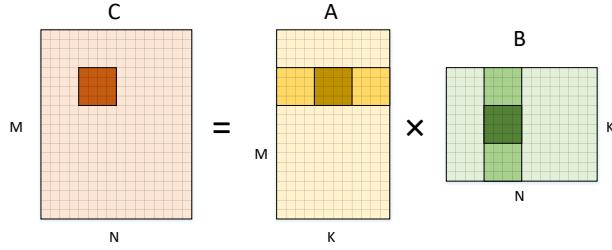
```

wwwj@ubuntu:~/lab1$ g++ gemm2.cpp -o gemm2
wwwj@ubuntu:~/lab1$ ./gemm2
100 200 300
time cost = 0.088161 s
wwwj@ubuntu:~/lab1$ ./gemm2
128 128 128
time cost = 0.032207 s
wwwj@ubuntu:~/lab1$ ./gemm2
256 256 256
time cost = 0.220393 s
wwwj@ubuntu:~/lab1$ ./gemm2
512 512 512
time cost = 1.70023 s
wwwj@ubuntu:~/lab1$ ./gemm2
1024 1024 1024
time cost = 14.2593 s
```

可以看到此次改进后运行时间和之前只用GEMM的时间相差不多，并没有达到优化的目的。

c. 矩阵乘计算4×4输出对K维度的拆分

代码



```
1 | vector<vector<double>> GEMM(vector<vector<double>> A,
2 |   vector<vector<double>> B, int M, int N, int K) {
3 |     vector<vector<double>> C(M, vector<double>(K, 0));
4 |     for (int m=0;m<M;m+=4) {
5 |       for (int k=0;k<K;k+=4) {
6 |         for (int n=0;n<N;n++) {
7 |           //end
8 |           C[m+0][k+0] += A[m+0][n+0]*B[n+0][k+0],
9 |           C[m+0][k+1] += A[m+0][n+0]*B[n+0][k+1],
10 |          C[m+0][k+2] += A[m+0][n+0]*B[n+0][k+2],
11 |          C[m+0][k+3] += A[m+0][n+0]*B[n+0][k+3],
12 |          C[m+1][k+0] += A[m+1][n+0]*B[n+0][k+0],
13 |          C[m+1][k+1] += A[m+1][n+0]*B[n+0][k+1],
14 |          C[m+1][k+2] += A[m+1][n+0]*B[n+0][k+2],
15 |          C[m+1][k+3] += A[m+1][n+0]*B[n+0][k+3],
16 |          C[m+2][k+0] += A[m+2][n+0]*B[n+0][k+0],
17 |          C[m+2][k+1] += A[m+2][n+0]*B[n+0][k+1],
18 |          C[m+2][k+2] += A[m+2][n+0]*B[n+0][k+2],
19 |          C[m+2][k+3] += A[m+2][n+0]*B[n+0][k+3],
20 |          C[m+3][k+0] += A[m+3][n+0]*B[n+0][k+0],
21 |          C[m+3][k+1] += A[m+3][n+0]*B[n+0][k+1],
22 |          C[m+3][k+2] += A[m+3][n+0]*B[n+0][k+2],
23 |          C[m+3][k+3] += A[m+3][n+0]*B[n+0][k+3],
24 |
25 |          C[m+0][k+0] += A[m+0][n+1]*B[n+1][k+0],
26 |          C[m+0][k+1] += A[m+0][n+1]*B[n+1][k+1],
27 |          C[m+0][k+2] += A[m+0][n+1]*B[n+1][k+2],
28 |          C[m+0][k+3] += A[m+0][n+1]*B[n+1][k+3],
29 |          C[m+1][k+0] += A[m+1][n+1]*B[n+1][k+0],
30 |          C[m+1][k+1] += A[m+1][n+1]*B[n+1][k+1],
```

```

31      C[m+1][k+3] += A[m+1][n+1]*B[n+1][k+3],
32      C[m+2][k+0] += A[m+2][n+1]*B[n+1][k+0],
33      C[m+2][k+1] += A[m+2][n+1]*B[n+1][k+1],
34      C[m+2][k+2] += A[m+2][n+1]*B[n+1][k+2],
35      C[m+2][k+3] += A[m+2][n+1]*B[n+1][k+3],
36      C[m+3][k+0] += A[m+3][n+1]*B[n+1][k+0],
37      C[m+3][k+1] += A[m+3][n+1]*B[n+1][k+1],
38      C[m+3][k+2] += A[m+3][n+1]*B[n+1][k+2],
39      C[m+3][k+3] += A[m+3][n+1]*B[n+1][k+3],
40
41      C[m+0][k+0] += A[m+0][n+2]*B[n+2][k+0],
42      C[m+0][k+1] += A[m+0][n+2]*B[n+2][k+1],
43      C[m+0][k+2] += A[m+0][n+2]*B[n+2][k+2],
44      C[m+0][k+3] += A[m+0][n+2]*B[n+2][k+3],
45      C[m+1][k+0] += A[m+1][n+2]*B[n+2][k+0],
46      C[m+1][k+1] += A[m+1][n+2]*B[n+2][k+1],
47      C[m+1][k+2] += A[m+1][n+2]*B[n+2][k+2],
48      C[m+1][k+3] += A[m+1][n+2]*B[n+2][k+3],
49      C[m+2][k+0] += A[m+2][n+2]*B[n+2][k+0],
50      C[m+2][k+1] += A[m+2][n+2]*B[n+2][k+1],
51      C[m+2][k+2] += A[m+2][n+2]*B[n+2][k+2],
52      C[m+2][k+3] += A[m+2][n+2]*B[n+2][k+3],
53      C[m+3][k+0] += A[m+3][n+2]*B[n+2][k+0],
54      C[m+3][k+1] += A[m+3][n+2]*B[n+2][k+1],
55      C[m+3][k+2] += A[m+3][n+2]*B[n+2][k+2],
56      C[m+3][k+3] += A[m+3][n+2]*B[n+2][k+3],
57
58      C[m+0][k+0] += A[m+0][n+3]*B[n+3][k+0],
59      C[m+0][k+1] += A[m+0][n+3]*B[n+3][k+1],
60      C[m+0][k+2] += A[m+0][n+3]*B[n+3][k+2],
61      C[m+0][k+3] += A[m+0][n+3]*B[n+3][k+3],
62      C[m+1][k+0] += A[m+1][n+3]*B[n+3][k+0],
63      C[m+1][k+1] += A[m+1][n+3]*B[n+3][k+1],
64      C[m+1][k+2] += A[m+1][n+3]*B[n+3][k+2],
65      C[m+1][k+3] += A[m+1][n+3]*B[n+3][k+3],
66      C[m+2][k+0] += A[m+2][n+3]*B[n+3][k+0],
67      C[m+2][k+1] += A[m+2][n+3]*B[n+3][k+1],
68      C[m+2][k+2] += A[m+2][n+3]*B[n+3][k+2],
69      C[m+2][k+3] += A[m+2][n+3]*B[n+3][k+3],
70      C[m+3][k+0] += A[m+3][n+3]*B[n+3][k+0],
71      C[m+3][k+1] += A[m+3][n+3]*B[n+3][k+1],
72      C[m+3][k+2] += A[m+3][n+3]*B[n+3][k+2],

```

```
73     C[m+3][k+3] += A[m+3][n+3] * B[n+3][k+3];  
74 }  
75 }  
76 }  
77 return C;  
78 }
```

运行结果以及时间

```
问题 输出 调试控制台 终端  
> Executing task: c:\test\computer\bin\gemm2.exe <  
704 704 704  
time cost = 7.196  
按任意键关闭终端。  
□
```

只用GEMM时：

```
问题 输出 调试控制台 终端  
> Executing task: c:\test\computer\bin\gemm.exe <  
704 704 704  
time cost = 6.036  
按任意键关闭终端。  
□
```

可以看到此次改进也并没有优化效果。

d. 向量化

完整代码见文件gemm2_3.cpp

代码

老师给的文章后还提到将矩阵相乘用向量化进行优化，虽然没能实现像文章中写的4个数值进行向量化，但是可以利用python中的dot()函数进行一整行A乘一整列B并求和的向量化，整个代码如下：

```
1 import time  
2 import numpy as np  
3
```

```

4 A = np.random.rand(1024,1024)
5 B = np.random.rand(1024,1024)
6
7 C = np.zeros((1024,1024))
8
9 # print(A)
10 # print(B)
11
12 B1=np.transpose(B)
13 tic = time.time()
14 for m in range(1024):
15     for n in range(1024):
16         C[m][n]=np.dot(A[m],B1[n])
17
18 toc = time.time()
19 print ("Vectorized: time=" + str((toc-tic)) +"s")
20
21 tic = time.time()
22 C=np.dot(A,B)
23 toc = time.time()
24 print ("Vectorized: time=" + str((toc-tic)) +"s")
25 # print(C)

```

而且函数dot()也可以直接计算矩阵乘法，运算速度要更快：

```

1 ...
2 tic = time.time()
3 C=np.dot(A,B)
4 toc = time.time()
5 print ("Vectorized version:" + str((toc-tic)) +"s")
6 ...

```

运行结果以及时间

向量化：

从上到下依次是矩阵大小为128*128, 256*256, 512*512, 1024*1024时的运行时间，这也是由于这个函数会进行指令级并行：

```

wwwj@ubuntu:~/lab1$ python vectorized.py
Vectorized : time = 0.0154418945312 s
wwwj@ubuntu:~/lab1$ python vectorized.py
Vectorized : time = 0.0806221961975 s
wwwj@ubuntu:~/lab1$ python vectorized.py
Vectorized : time = 0.591318130493 s
wwwj@ubuntu:~/lab1$ python vectorized.py
Vectorized : time = 5.18112206459 s

```

直接使用python的dot()函数进行矩阵乘法：

从上到下依次是矩阵大小为 128×128 , 256×256 , 512×512 , 1024×1024 时的运行时间，函数dot()实际是使用了并行指令集，因此要快很多。

```
wwwj@ubuntu:~/lab1$ python vectorized.py
Vectorized : time = 0.00102686882019s
wwwj@ubuntu:~/lab1$ python vectorized.py
Vectorized : time = 0.00193905830383s
wwwj@ubuntu:~/lab1$ python vectorized.py
Vectorized : time = 0.00564813613892s
wwwj@ubuntu:~/lab1$ python vectorized.py
Vectorized : time = 0.0686061382294s
wwwj@ubuntu:~/lab1$
```

经过算法分析优化以及软件优化后的GEMM

从上面几种优化结果来看，只有“矩阵乘计算 1×4 输出”有一定的优化效果（不包括利用python中的numpy库的向量化），将这部分优化加入到之前GEMM已经经过strassen优化的代码中：

完整代码见gemm3.cpp

运行结果以及时间对比

对于一些同样规模的矩阵进行乘法运算，我们来观察经过两次优化后的GEMM的运算时间的变化

GEMM (gemm.cpp) :

```
wwwj@ubuntu:~/lab1$ ./gemm
100 200 300
time cost = 0.061483 s
wwwj@ubuntu:~/lab1$ ./gemm
128 128 128
time cost = 0.018911 s
wwwj@ubuntu:~/lab1$ ./gemm
256 256 256
time cost = 0.175992 s
wwwj@ubuntu:~/lab1$ ./gemm
512 512 512
time cost = 1.40259 s
wwwj@ubuntu:~/lab1$ ./gemm
1024 1024 1024
time cost = 11.6774 s
wwwj@ubuntu:~/lab1$
```

进行了算法优化以后 (gemm_strassen.cpp) :

可以看到有一定的优化效果：

```
wwwj@ubuntu:~/lab1$ g++ gemm_strassen.cpp -o gemm_strassen
wwwj@ubuntu:~/lab1$ ./gemm_strassen
100 200 300
firstly, use GEMM.
time cost = 0.062845 s
wwwj@ubuntu:~/lab1$ ./gemm_strassen
128 128 128
firstly, use strassen.
Reached the limit n=64 ,use GEMM.
time cost = 0.024708 s
wwwj@ubuntu:~/lab1$ ./gemm_strassen
256 256 256
firstly, use strassen.
Reached the limit n=64 ,use GEMM.
time cost = 0.172589 s
wwwj@ubuntu:~/lab1$ ./gemm_strassen
512 512 512
firstly, use strassen.
Reached the limit n=64 ,use GEMM.
time cost = 1.10094 s
wwwj@ubuntu:~/lab1$ ./gemm_strassen
1024 1024 1024
firstly, use strassen.
Reached the limit n=64 ,use GEMM.
time cost = 8.00081 s
```

进行了算法优化以及软件优化以后（gemm3.cpp）：

可以看出对比之前的gemm，效率达到了原来的3倍：

```
wwwj@ubuntu:~/lab1$ ./gemm3
100 200 300
firstly, use GEMM.
time cost = 0.021355 s
wwwj@ubuntu:~/lab1$ ./gemm3
128 128 128
firstly, use strassen.
Reached the limit n=128 ,use GEMM.
time cost = 0.009341 s
wwwj@ubuntu:~/lab1$ ./gemm3
256 256 256
firstly, use strassen.
Reached the limit n=128 ,use GEMM.
time cost = 0.069126 s
wwwj@ubuntu:~/lab1$ ./gemm3
512 512 512
firstly, use strassen.
Reached the limit n=128 ,use GEMM.
time cost = 0.531985 s
wwwj@ubuntu:~/lab1$ ./gemm3
1024 1024 1024
firstly, use strassen.
Reached the limit n=128 ,use GEMM.
time cost = 3.53017 s
```

3) 利用INTEL MKL函数库进行矩阵乘法

优化后的矩阵乘法与Intel MKL函数库的矩阵乘法函数，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。

完整代码见mkl.c

代码

首先要在ubuntu下安装MKL库，这个过程一直遇到报错，参考一些文章博客可以成功解决问题并安装。

利用INTEL MKL函数库进行矩阵乘法，最常用的函数主要就是degmm。

其中最主要的函数是：

```
1 | cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
2 |           m, n, k, alpha, A, k, B, n, beta, C, n);
```

- 第一个参数表示输入的矩阵是行主序（CblasRowMajor）还是列主序（CblasColMajor）：行主序（CblasRowMajor）表示矩阵是按照按一行接一行的顺序存储，列主序（CblasColMajor）是按照按一列接一列的顺序存储，在计算时也就按照他们的存储方式确定行列并进行计算。
- 第二个参数表示是否对矩阵A进行转置（CblasTrans or CblasNoTrans）：与前面的参数（行或列主序）配合使用。
- 第三个参数表示是否对矩阵B进行转置（CblasTrans or CblasNoTrans）
- m, n, k 分别表示A是 $m * k$ 的矩阵，B是 $k * n$ 的矩阵，C是 $m * n$ 的矩阵
- alpha和beta是两个参数，degmm的公式是 $C <= \alpha A * B + \beta C$ ，由于我们要计算 $C = A * B$ ，因此将 $alpha = 1, beta = 0$

按照我们的矩阵的要求，这个函数的各个参数被写为：

```
1 | cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
2 |           m, k, n, alpha, A, n, B, k, beta, C, k);
```

代码的主要过程：输入有关矩阵大小变量 m, n, k ，然后给矩阵分配空间，空间不够时输出信息并直接退出程序，然后将随机数赋给矩阵A，B并进行计算，如果有需要还可以输出矩阵A，B，C来验证结果。

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include "mkl.h"
4 |
5 | void PRINT(double *A, int m, int n) {
```

```

6   for(int i=0;i<m;++i) {
7     for(int j=0;j<n;++j) printf("%lf ",A[i*n+j]);
8     printf("\n");
9   }
10 }
11
12 int main()
13 {
14   double *A, *B, *C; //matrix
15   int m, k, n;
16   double alpha=1.0, beta=0.0;
17
18   scanf("m=%d n=%d k=%d", &m, &n, &k);
19
20   //allocate memory
21   A = (double *)mkl_malloc( m*n*sizeof( double ), 64 );
22   B = (double *)mkl_malloc( n*k*sizeof( double ), 64 );
23   C = (double *)mkl_malloc( m*k*sizeof( double ), 64 );
24
25   //if we dont't have memory, return
26   if (A == NULL || B == NULL || C == NULL) {
27     printf( "ERROR: Can't allocate memory for matrices. \n");
28     mkl_free(A);
29     mkl_free(B);
30     mkl_free(C);
31     return 1;
32   }
33
34   //Intialize random matrix
35   for(int i=0;i<m*n;++i)      A[i]=(double)(rand()%50);
36   for(int i=0;i<n*k;++i)      B[i]=(double)(rand()%50);
37   for(int i=0;i<m*k;++i)      C[i]=0;
38
39   //computer
40   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, k, n,
alpha, A, n, B, k, beta, C, k);
41   printf ("Computations completed.\n\n");
42
43   //print
44   // printf("A:\n");
45   // PRINT(A,m,n);
46   // printf("B:\n");
47   // PRINT(B,n,k);

```

```

48     // printf("C=A*B:\n");
49     // PRINT(C,m,k);
50
51     //Deallocating memory
52     mkl_free(A);
53     mkl_free(B);
54     mkl_free(C);
55
56     return 0;
57 }
```

运行结果以及时间

输出运行结果：

```

wwwj@ubuntu:~/lab1$ mpicc mkl.c -o mkl -lmkl_rt
wwwj@ubuntu:~/lab1$ ./mkl
3 4 5
time cost = 0.009655 s
Computations completed.

A:
12.000000 45.000000 25.000000 17.000000
8.000000 46.000000 41.000000 10.000000
10.000000 7.000000 25.000000 20.000000

B:
37.000000 45.000000 3.000000 26.000000 17.000000
23.000000 4.000000 7.000000 32.000000 2.000000
25.000000 22.000000 40.000000 24.000000 31.000000
5.000000 32.000000 5.000000 43.000000 44.000000

C=A*B:
2189.000000 1814.000000 1436.000000 3083.000000 1817.000000
2429.000000 1766.000000 2036.000000 3094.000000 1939.000000
1256.000000 1668.000000 1179.000000 1944.000000 1839.000000
```

在matlab中进行验算：

结果正确，算法无误。

```

uptest.m × [ + ]
-
A=[12.000000,45.000000,25.000000,17.000000;
8.000000,46.000000,41.000000,10.000000;
10.000000,7.000000,25.000000,20.000000];
-
B=[37.000000,45.000000,3.000000,26.000000,17.000000;
23.000000,4.000000,7.000000,32.000000,2.000000;
25.000000,22.000000,40.000000,24.000000,31.000000;
5.000000,32.000000,5.000000,43.000000,44.000000];
-
C=A*B;
```

命令窗口

```

>> C
```

```

C =
2189      1814      1436      3083      1817
2429      1766      2036      3094      1939
1256      1668      1179      1944      1839
```

```

>>
```

mkl:

```
wwwj@ubuntu:~/lab1$ mpicc mkl.c -o mkl -lmkl_rt
wwwj@ubuntu:~/lab1$ ./mkl
100 200 300
time cost = 0.003789 s
Computations completed.

wwwj@ubuntu:~/lab1$ ./mkl
128 128 128
time cost = 0.003934 s
Computations completed.

wwwj@ubuntu:~/lab1$ ./mkl
256 256 256
time cost = 0.003565 s
Computations completed.

wwwj@ubuntu:~/lab1$ ./mkl
512 512 512
time cost = 0.008378 s
Computations completed.

wwwj@ubuntu:~/lab1$ ./mkl
1024 1024 1024
time cost = 0.058234 s
Computations completed.
```

之前只用GEMM的运行时间：

```
wwwj@ubuntu:~/lab1$ g++ gemm.cpp -o gemm
wwwj@ubuntu:~/lab1$ ./gemm
100 200 300
time cost = 0.069532 s
wwwj@ubuntu:~/lab1$ ./gemm
128 128 128
time cost = 0.034667 s
wwwj@ubuntu:~/lab1$ ./gemm
256 256 256
time cost = 0.158958 s
wwwj@ubuntu:~/lab1$ ./gemm
512 512 512
time cost = 1.40912 s
wwwj@ubuntu:~/lab1$ ./gemm
1024 1024 1024
time cost = 12.6819 s
```

比较二者， mkl计算要快很多（是gemm的200倍左右），性能提高了。

分析原因

Intel® MKL 性能库提供了高度优化的函数，这些函数可充分利用Intel® 多核处理器，从而能够最大限度地获得应用程序的性能并减少开发时间。MKL在底层架构上使用了指令级并行，实现了矢量化和良好的数据缓存。

利用MKL库可以在由TLB(TransferringLookAbout缓冲区)定义的矩阵块上执行乘法，从而将处理的数据量准确地流到处理器。另一方面是矢量化，它们使用处理器的向量化指令来优化指令吞吐量，这在跨平台C++代码中是无法做到的。

3. 进阶：大规模矩阵计算优化

进阶问题描述：如何让程序支持大规模矩阵乘法？

考虑两个优化方向

1) 性能，提高大规模稀疏矩阵乘法性能；

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。

对优化方法及思想进行详细描述，提供大规模矩阵计算优化代码可加分。

对于大规模矩阵进行乘法运算，尝试了之前优化后的代码，都会产生内存溢出的错误。

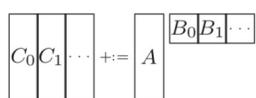
首先能想到的是将矩阵分块和并行化处理。

分块与并行化处理

比如矩阵A为 $m \times k$ ，矩阵B是 $k \times n$ 。进行拆分将矩阵A拆分成多列，B拆成多行。然后再对B的行进行拆分，如B的一行被拆分为三个block。对A的列也进行拆分，A的一列被拆分为多个slice。A的一列乘以B的每个block，得到矩阵C的一列的部分值。

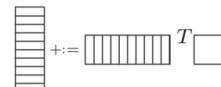
```
1 for p = 1 : k
2   for j = 1 : n
3     for i = 1 : m
4       Cij += Aip * Bpj
5     endfor
6   endfor
7 endfor
```

Algorithm: $C := \text{GEPP_BLK_VAR2}(A, B, C)$



Pack A into \tilde{A}
for $j = 0, \dots, N - 1$
 $C_j := \tilde{A}^T B_j$ (GEPB_OPT1)
endfor

Algorithm: $C := \text{GEPB_OPT1}(A, B, C)$



Assumption: A “packed” in memory.

Pack B into \tilde{B}
for $i = 0, \dots, M - 1$
 $\square := \square^T \square$ ($C_{\text{aux}} := A_i^T \tilde{B}$)
Unpack $\tilde{C}_i = \tilde{C}_i + C_{\text{aux}}$
endfor

还有并行：

尝试使用MPI进行矩阵乘法并行化

(注：下面的代码根据文章做了些更改，并不是都自己实现的)

这个代码虽然进行了并行化，但是仍然无法计算两个 100000×100000 的矩阵相乘，仍需要进一步的优化。

```
1 #include<stdio.h>
2 #include <iostream>
3 #include<math.h>
4 #include<mpi.h>
5 #include<time.h>
6 using namespace std;
7
8 #define random(x) (rand()%50)
9
10 int main(int argc, char *argv[]){
11     int size;
12     if (argc == 2){
13         //判断有无矩阵大小size接收，无则初始化为5
14         size = atoi(argv[1]);
15     }
16     else { size = 5; }
17     srand((int)time(0));
18     int *a, *b, *c, *pce, *ans;
19     int rank, numprocess, line;
20     double starttime, endtime;
21     MPI_Init(&argc, &argv); //MPI Initialize
22     MPI_Comm_rank(MPI_COMM_WORLD, &rank); //获得当前进程号
23     MPI_Comm_size(MPI_COMM_WORLD, &numprocess); //获得进程个数
24     line = size / numprocess; //将数据分为(进程数)个块，主进程也要处理数
据
25     a = (int*)malloc(sizeof(int)*size*size);
26     b = (int*)malloc(sizeof(int)*size*size);
27     c = (int*)malloc(sizeof(int)*size*size);
28     pce = (int*)malloc(sizeof(int)*size*line);
29     ans = (int*)malloc(sizeof(int)*size*line);
30
31     starttime = MPI_Wtime();
32     if (rank==0){
33         for (int i = 0; i<size; i++) {
```

```

34         for (int j = 0; j<size; j++) {
35             //随机两个相乘的矩阵a, b
36             a[i*size + j] = random(10);
37             b[i*size + j] = random(10);
38         }
39     }
40     for (int i = 1; i<numprocess; i++)
41     {
42         //将b矩阵传递给各进程
43         MPI_Send(b, size*size, MPI_INT, i, 0, MPI_COMM_WORLD);
44     }
45     for (int i = 1; i<numprocess; i++){
46         //将a矩阵分成块矩阵, 传递给各进程
47         MPI_Send(a + (i - 1)*line*size, size*line, MPI_INT, i,
48 1, MPI_COMM_WORLD);
49     }
50     for (int k = 1; k<numprocess; k++)
51     {
52         MPI_Recv(ans, line*size, MPI_INT, k, 2,
53 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
54         //接收传递结果到数组c
55         for (int i = 0; i<line; i++)
56         {
57             for (int j = 0; j<size; j++)
58             {
59                 c[((k - 1)*line + i)*size + j] = ans[i*size +
60 j];
61             }
62         }
63         for (int i = (numprocess - 1)*line; i<size; i++)
64         {
65             //计算分配给主进程的块矩阵
66             for (int j = 0; j<size; j++)
67             {
68                 int temp = 0;
69                 for (int k = 0; k<size; k++){
70                     temp += a[i*size + k] * b[k*size + j];}
71                 c[i*size + j] = temp;
72             }
73         }
74         // for (int i = 0; i < size; i++) {
75         // 打印显示

```

```

74         //for (int j = 0; j<size; j++){ printf("%d ", a[size*i +
j]); }printf("  ");
75         //for (int j = 0; j<size; j++){ printf("%d ", b[size*i +
j]); }printf("  ");
76         //for (int j = 0; j<size; j++){ printf("%d ", c[size*i +
j]); }printf("\n");
77     }
78     endtime = MPI_Wtime();
79     printf("Took %f secodes.\n", endtime - starttime);
80 }
81
82 else{
83     MPI_Recv(b, size*size, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
84     //接收b矩阵
85     MPI_Recv(pce, size*line, MPI_INT, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
86     //接收块矩阵
87     for (int i = 0; i<line; i++){
88         //计算块矩阵与b矩阵相乘的结果
89         for (int j = 0; j<size; j++){
90             int temp = 0;
91             for (int k = 0; k<size; k++)
92                 temp += pce[i*size + k] * b[k*size + j];
93             ans[i*size + j] = temp;
94         }
95     }
96     MPI_Send(ans, line*size, MPI_INT, 0, 2, MPI_COMM_WORLD);
97     //将结果传递到主进程
98 }
99 MPI_Finalize();
100 return 0;
101 }
```

运行结果：

```

wwwj@ubuntu:~/hw$ mpixexec -n 10 a 1000
Took 3.554084 secodes.
wwwj@ubuntu:~/hw$ mpixexec -n 100 a 1000
Took 106.857063 secodes.
wwwj@ubuntu:~/hw$ mpixexec -n 5 a 1000
Took 4.169534 secodes.
wwwj@ubuntu:~/hw$
```

会发现有时候线程越多反而速度越慢，可能是因为它们之间的相互通信造成时间的消耗。

利用hadoop实现超大规模矩阵相乘

在网络上还找到一些优化方法，比如“利用hadoop实现超大规模矩阵相乘”

这个方法主要是针对“计算过程中文件占用存储空间大”这个缺陷提出的，也是使用了Map-Reduce模型。

文章链接:[利用Hadoop实现超大矩阵相乘之我见 (二) - 上品物语 - 博客园 (cnblogs.com)]

<https://www.cnblogs.com/eczhou/p/3600783.html>

算法的基本思路如下：

传统矩阵相乘是左矩阵的行乘以右矩阵的列来进行计算，但对于稀疏矩阵来说会造成过多无效计算。在这个算法中采用列、行相乘计算方式，即利用左矩阵的一列中的元素与右矩阵对应行中的所有元素依次相乘，该方法有效避免了稀疏矩阵相乘过程中产生的无效计算。

1) 首先对矩阵进行预处理：

将所有的矩阵元素都存储在文本文件中，一行记录代表一个矩阵元素，针对稀疏矩阵，0元素不在输入文本中。每一行包含的信息就是非0元素的行号，列号以及他的值。可以把左矩阵的列号和右矩阵的行号作为key值放在每行的第一列，方便在下面Reduce过程中能够按照Key值统计左矩阵第Key列及其对应右矩阵第Key行中的元素。

2) Reduce阶段进行统计与分段：

当矩阵规模大到一定程度时，内存可能会碰到加载不了左矩阵的一列或右矩阵的一行元素的问题。该算法对左矩阵元素按列进行分段，对右矩阵元素按行进行分段的方法，这样，单个计算节点就可以加载左矩阵的一段与右矩阵的一段至内存进行相乘运算，突破了内存的限制。

首先将所有Key相同的Value集合在一起，形成一个Value-List。若Key为k，那么Value-List则代表了左矩阵第k列与右矩阵第k行的所有元素，这些元素混合在一起。在Reduce阶段，我们第一轮遍历Value-List，获得左矩阵第k列的元素个数为M_k，右矩阵第k行的元素个数为N_k。接下来我们通过第二轮遍历对左矩阵第k列、右矩阵第k行的元素进行分段操作，假设每个分段包含w个元素，则左矩阵第k列被分为段，右矩阵被分为段。

然后将L矩阵中第k列中第i个分段表示成如下格式：

$$k \ L - i - [N_k/w] - element_list \quad i \in (1, 2, \dots, [M_k/w])$$

R矩阵中第k行中第j个分段表示成如下格式:

$$k \quad R - [M_k/w] - j - element_list \quad i \in (1, 2, \dots, [N_k/w])$$

注: $[N_k/w]$ 代表该分段在接下来的过程中总共需要 $[N_k/w]$ 个拷贝/该列或者该行的分段数目, N_k 代表第k列或者行的元素个数, w是每个分段的元素数目, *element_list*表示该分段中的元素集合, $[M_k/w]$ 同理。

为了便于后续Map-Reduce过程的处理, 将每一个分段都存储在磁盘文件中, 文件中的一样代表一个分段。同时, 将两个矩阵中的具体分段信息存储在分布式缓存中, 有利于解决后续步骤中不同节点间的通信与数据查询问题。

3) 拷贝任务分发——Map迭代算法

将两个矩阵中的分段一一对应相乘。但是在相乘的过程中可能会出现左矩阵的一段要与右矩阵的每一段相乘, 就要对左矩阵的这一段进行大量拷贝, 可能会造成时间空间的大量浪费。本算法采用了“Map迭代拷贝任务分发法”来对每条记录(每个分段)的拷贝任务进行分发, 这样有效的控制每个节点对每个分段的拷贝数量, 同时更有效的使得更多的节点参与拷贝运算工作。

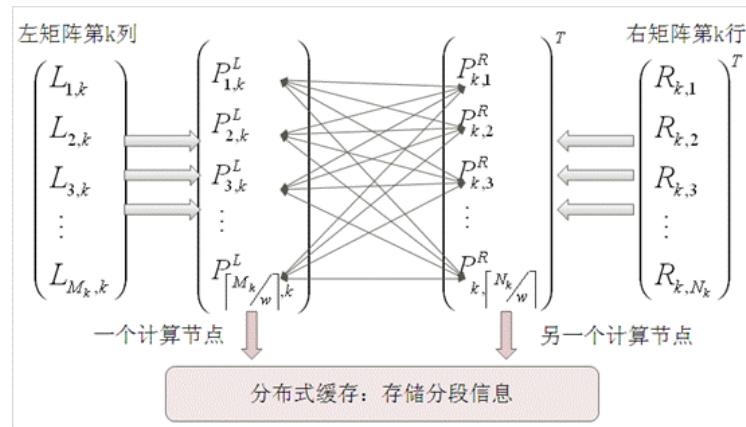
注: 对迭代次数的控制。在现实大矩阵相乘中, 由于大部分情况下矩阵都为稀疏矩阵, 那么每行每列包含的元素个数就不一样, 所以每个分段需拷贝的份数都不确定。这样我们就需要计算Map迭代过程的迭代次数, 依次来控制Map迭代的过程。

4) 计算模块

完成记录的拷贝工作后, 还需要两轮Map-Reduce过程完成矩阵的运算。

第一轮Map-Reduce: 进行分段拷贝与对应

第二轮Map-Reduce: 进行如下图中两个段的相乘工作并汇总



Map阶段对每条记录进行相乘运算，即将L（左矩阵）中每个元素依次与R（右矩阵）中每个元素相乘，若element_list_(L,k,i)中某个元素 $L_{i,k}$ 与element_list_(R,k,j)中某个元素 $R_{k,j}$ 相乘，将结果记录成“i-j value”格式。然后每个Map结束后执行combine操作，combine操作与该轮Reduce操作一样，执行相同key的value相加，便得到了最终的矩阵运算结果。

注：

这里只是对于这个算法的简单概述和我自己的一些简单的理解。这个算法我看过几遍之后能理解一部分，但是对于后半部分我还是有很多地方无法理解，对比如这个迭代的过程，以及reduce操作等等，也对如何实现这个算法并无想法。要想实现这个大规模矩阵计算还是与很多路要走。

实验思考

本次实验中对gemm进行一步步优化后发现都是合理且有效的，并且在本次实验中运用了一些比如numpy以及mkl这些能够告诉运行的库。但是在本次实验中关于并行的内容我做的不是很多，以后还需要继续学习。

主要文章参考：

通用矩阵乘（GEMM）优化算法 <https://jackwish.net/2019/gemm-optimization.html>

5分钟掌握矩阵乘法的Strassen算法 | Long Luo's Life Notes <http://www.longluo.me/blog/2019/06/21/Strassens-Matrix-Multiplication-Algorithm/>

Ubuntu下MKL（Intel Math Kernel Library）安装教程 https://blog.csdn.net/qq_36080114/article/details/108424401

Using Intel® Math Kernel Library for Matrix Multiplication - C <https://software.intel.com/content/www/us/en/develop/documentation/mkl-tutorial-c/top/multiplying-matrices-using-dgemm.html>

OpenBlas库----cblas_dgemm()函数_weixin_34292402的博客-CSDN博客 https://blog.csdn.net/weixin_34292402/article/details/92178636

矩阵相乘优化（Gemm） - Cheney_1016 - 博客园 <https://www.cnblogs.com/jzcbest1016/p/12238185.html>

使用 MapReduce 实现大规模稀疏矩阵相乘 <http://vividfree.github.io/大规模数据处理/2015/11/14/large-scale-matrix-multiplication-using-mapreduce>

基于MPI的大规模矩阵乘法问题 - 陈府 - 博客园 <https://www.cnblogs.com/chenzhihong294/p/10633342.html>