CS205 C/ C++ Program Design

Project 1

Name: 王康

SID: 11510815

Part 1. Source Code

https://github.com/apostlewang/CS205_CPP/blob/master/Proj1/matrixmulti.cpp

Part 2. Result & Verification

1.Struct for matrices

```
struct Matrix{
    int num_rows;
    int num_colums;
    float *data;
    Matrix(int m, int n){
        num_rows = m;
        num_colums = n;
        data = new float[m*n];
    }
};
```

矩阵数据采用一维数组按行储存

2.Function to compute matrix product

```
Matrix multiMatrix1(Matrix &a, Matrix &b){
    assert (a.num_colums == b.num_rows);
    Matrix result = Matrix(a.num_rows, b.num_colums);
    for(int i=0 ; i<a.num_rows; i++){
        for(int j=0 ; j<b.num_colums; j++){
            float sum = 0;
            for(int k=0 ; k<a.num_colums; k++){
                sum += a.data[i*a.num_colums+k]*b.data[k*b.num_colums+j];
            }
            result.data[i*b.num_colums+j] = sum;
        }
    }
```

```
    return result;
}
```

这个是最简单的矩阵乘法实现，效率很低（内存访问不连续且本身算法复杂度为O（n^3）），但是基本实现思路是这样，计算得到的矩阵中的每一个元素都需要k次乘法和加法，而共有i*j个元素。

3.Time for mutiply two matrices containing about 200M elements

200M个元素是亿级别的数量级，两个10000*10000的矩阵相乘，如果用以上的算法，复杂度为O（n^3），那么需要10000^3次运算，一般计算机一亿次运算大概需要1s，所以大概需要10000s时间，时间过长所以无法直接测量，采用估算的方法。

```
Test for two 2000*2000 matrix:
Cblas: duration = 79ms, result:
        48642.2 49058 48261.5 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418.1 47277.4

V1(naive)1: duration = 29649ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4

V1(naive)2: duration = 29621ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4
```

2000*2000的矩阵需要约30s，那么10000*10000的矩阵估算需要30*25^3s，约为47万秒。

而使用改进后的算法可以在可以容忍的时间内测量出结果：

```
Test for two 10000*10000 matrix:
Cblas: duration = 8385ms, result:
        242026 245430 247233 ... 241292 245801 245275
                                 ...
        243122 247172 247922 ... 241703 244822 245272

V4(ikj+omp): duration = 90447ms, result:
        242026 245430 247233 ... 241292 245800 245276
                                 ...
        243122 247172 247922 ... 241703 244822 245272

V5(tanspose+SIMD): duration = 105368ms, result:
        242026 245430 247233 ... 241292 245801 245275
                                 ...
        243122 247172 247922 ... 241703 244822 245272
```

自己实现最优的算法所需时间约为90s。

4.Improvement methods and the improvements

V1即为最原始的版本，在提升之前先看一下这个算法为什么慢：

```
for(int i=0 ; i<a.num_rows; i++){
    for(int j=0 ; j<b.num_colums; j++){
        float sum = 0;
        for(int k=0 ; k<a.num_colums; k++){
            sum += a.data[i*a.num_colums+k]*b.data[k*b.num_colums+j];
        }
        result.data[i*b.num_colums+j] = sum;
    }
}
```

最内层的循环中，对a中一维数组数据的访问是相对连续的，有利于计算机的数据cache，即减少了从内存中读入数据的次数，而对于b来说，内存访问一直是跳跃的（每次间隔b.num_colums），即每次都要访问内存来提取数据，共有i*j*k次这样的操作所以开销是巨大的。

一种避免内存访问不连续的方法是调换ijk的顺序：

```
Matrix multiMatrix2(Matrix &a, Matrix &b){
    assert (a.num_colums == b.num_rows);
    Matrix result = Matrix(a.num_rows, b.num_colums);
    for(int i=0 ; i<a.num_rows; i++){
        for(int k=0 ; k<a.num_colums; k++){
            float s = a.data[i*a.num_colums+k];
            for(int j=0 ; j<b.num_colums; j++){
                result.data[i*b.num_colums+j] += s*b.data[k*b.num_colums+j];
            }
        }
    }

    return result;
}
```

即V2中使用的方法，变ijk为ikj，这样使得b中数据的访问相对连续（跳跃次数从ijk次减小到ik次）。在测试中这种方法比第一种提升了约10倍。

因为两种方法都有for循环，因此考虑分别使用openmp来加速，V1+openmp得到V3，V2+openmp得到V4：

```
Test for two 2000*2000 matrix:
Cblas: duration = 76ms, result:
        50055.9 49734.2 49005 ... 49249.7 49290.8 49490.9
                                  ...
        49307.3 49155.3 48590.8 ... 48355.2 49059 48790.6

V1(naive)1: duration = 28827ms, result:
        50055.9 49734.3 49004.9 ... 49249.7 49290.8 49491
                                  ...
        49307.3 49155.3 48590.8 ... 48355.1 49059 48790.6

V1(naive)2: duration = 28764ms, result:
        50055.9 49734.3 49004.9 ... 49249.7 49290.8 49491
                                  ...
        49307.3 49155.3 48590.8 ... 48355.1 49059 48790.6

V2(change_to_ikj): duration = 1941ms, result:
        50055.9 49734.3 49004.9 ... 49249.7 49290.8 49491
                                  ...
        49307.3 49155.3 48590.8 ... 48355.1 49059 48790.6

V3(naive+openmp): duration = 8749ms, result:
        50055.9 49734.3 49004.9 ... 49249.7 49290.8 49491
                                  ...
        49307.3 49155.3 48590.8 ... 48355.1 49059 48790.6

V4(ikj+omp): duration = 386ms, result:
        50055.9 49734.3 49004.9 ... 49249.7 49290.8 49491
                                  ...
        49307.3 49155.3 48590.8 ... 48355.1 49059 48790.6
```

对于两个2000*2000的矩阵相乘，V1提升了约3倍，V2提升了约6倍。

所以在后续的版本中均会使用openmp加速的方法。

接着想继续优化的话还有一种常用策略SIMD没有使用，但是在V1和V2的实现中均没有连续多条数据与连续多条数据相乘的运算，因此考虑将矩阵b转置后使用，这样可以连续读入bt的行数据即b的列数据。后面的V5-7均使用了矩阵b转置后的数据作为输入。

```
Matrix transpose(Matrix b){
    int kk = b.num_rows;
    int jj = b.num_colums;
    float* colums = new float[kk*jj];
    Matrix bt = Matrix(jj,kk);
    for(int i=0; i<kk*jj; i++){
        int row = i/jj;
        int colum = i%jj;
        colums[colum*kk+row] = b.data[i];
    }
    bt.data = colums;å
    return bt;
}
```

其中V5和V6均为每次计算出结果中第ij个元素的值，不同之处在于5中采用了SIMD使得对于k这个维度的数据每次可以完成8个乘法操作，而6中为了对比SIMD的作用而采用k个元素逐个相乘的方法。还是测试2000*2000的数据：

```
V5(tanspose+SIMD): duration = 363ms, result:
        48642.3 49058 48261.5 ... 49102.2 47922 47992.1
                                ...
        49944.9 49760.8 48948.7 ... 49850 49418.1 47277.4

V6(transpose): duration = 1523ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4
```

可以看到采用了SIMD的版本比没有采用的版本快了将近5倍。V5与之前采用了ikj和openmp的速度相当，而只做转置操作的版本比起原来的ijk版本也有相当大的提升，原因还是在于数据的访问在内存中变得相对连续了。

最后一个版本V7中用到了16个寄存器（因为累加操作比较多这样可以省点时间），每次计算结果中的16个元素，思路是取出a中4行和b中4列（bt中的4行）的起始地址，每次访问a中4行的第k组元素和bt中4行的第k组元素，交叉相乘分别加到对应的寄存器上，循环结束后提取出16个寄存器中的值作为对应位置的计算结果。这样做的好处是降低了同一行/列数据的访问次数（原来的4次只要现在的1次，可以从每个数可以利用4次看出）。结果比仅仅转置的版本V6快了一倍多，但还是比不上V5中的每次完成8个乘法的SIMD速度快（相差一倍，也比较符合直觉，这个版本相当于每次操作4次乘法）。

```
V5(tanspose+SIMD): duration = 363ms, result:
        48642.3 49058 48261.5 ... 49102.2 47922 47992.1
                                  ...
        49944.9 49760.8 48948.7 ... 49850 49418.1 47277.4

V6(transpose): duration = 1523ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4

V7(16registers+omp): duration = 620ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4
```

5.Compare my implementation with OpenBLAS

```
Test for two 2000*2000 matrix:
Cblas: duration = 79ms, result:
        48642.2 49058 48261.5 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418.1 47277.4

V1(naive)1: duration = 29649ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4

V1(naive)2: duration = 29621ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4

V2(change_to_ikj): duration = 2007ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4

V3(naive+openmp): duration = 8350ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4

V4(ikj+omp): duration = 373ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4

V5(tanspose+SIMD): duration = 363ms, result:
        48642.3 49058 48261.5 ... 49102.2 47922 47992.1
                                  ...
        49944.9 49760.8 48948.7 ... 49850 49418.1 47277.4

V6(transpose): duration = 1523ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4

V7(16registers+omp): duration = 620ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4
```

对于两个2000*2000的矩阵相乘：

速度方面OpenBlas用时79ms，自己实现的最优算法用时363ms，约为4.5倍，而计算结果方面有差异但是很细微：0.1级别的差别。

```
Cblas: duration = 79ms, result:
        48642.2 49058 48261.5 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418.1 47277.4

V1(naive)1: duration = 29649ms, result:
        48642.2 49058.1 48261.4 ... 49102.2 47922 47992.1
                                  ...
        49945 49760.8 48948.7 ... 49849.9 49418 47277.4
```

而当数据规模扩大时，OpenBlas的优势愈发明显，如10000*10000量级：

```
Test for two 10000*10000 matrix:
Cblas: duration = 7681ms, result:
        242033 242737 243319 ... 242446 243703 247281
                                 ...
        239478 240010 241123 ... 241323 245728 245686

V4(ikj+omp): duration = 96525ms, result:
        242033 242737 243319 ... 242446 243703 247281
                                 ...
        239478 240011 241124 ... 241323 245728 245686

V5(tanspose+SIMD): duration = 104894ms, result:
        242033 242737 243319 ... 242446 243703 247281
                                 ...
        239478 240010 241124 ... 241323 245728 245686
```

速度为自己实现算法最优时间的十多倍。（本次测试结果相同）

6.Github code link

https://github.com/apostlewang/CS205_CPP/blob/master/Proj1/matrixmulti.cpp

Part 3. Difficulties & Solutions, or others

刚开始用上openmp之后测量时间竟然发现时间变长了，后来查阅发现通过<time.h>中clock测量的时间并不是实际时间，对于多核多线程会累加它们的clock数目。

```
clock_t start,end;
start = clock();
//function
end = clock();
double(end-start)/CLOCKS_PER_SEC
```

后来通过老师给的https://en.cppreference.com/w/cpp/chrono/time_point解决。