# CS205 C/C++ Program Design Project5


> Name:章志轩
>
> SID:12010526


> Time:2023/05/27


***Abstract: BLAS（Basic Linear Algebra Subprograms）是数值计算软件必须具备的核心库之一，被称为高性能计算、仿真、数据处理、人工智能的基石，是当之无愧的根技术。本次项目的目标就是实现BLAS接口规范的GEMM函数，并在与OpenBLAS库函数的性能比较中尽可能提高运算效率，从中学习BLAS的方法和设计理念。***


# I. 项目介绍

基础线性代数运算是指矢量的线性组合， 矩阵乘以矢量， 矩阵乘以矩阵等，是科学计算(程序)中的重要方法。它被用来计算大量数据的运算。为提高它的运算效率，1979年，编程世界出现了一个标准Basic Linear Algebra Subprograms（简称BLAS），为向量和后来的矩阵数学定义了一系列基本程序。BLAS实际上就是将复杂的矩阵和向量运算简化成类似加减法一样基础的简单计算单元。如今BLAS 被广泛用于科学计算和工业界， 已成为业界标准。 在更高级的语言和库中， 即使我们不直接使用 BLAS 接口， 它们也是通过调用 BLAS 来实现的（如 Matlab 中的各种矩阵运算）。

OpenBLAS是一个开源的BLAS库。本次项目就是基于OpenBLAS提供的代码规范设计我自己的GEMM函数完成双精度浮点数矩阵运算，通过各种手段提高性能并与OpenBLAS实现的函数对比运算效率。从中我将学习到加速矩阵运算效率的各类方法和BLAS的设计理念。

# II. 环境配置&语言

系统环境：Win10，AMD 锐龙 5 4600U处理器；

IDE：VScode-x64-1.75.1

语言：C++（g++编译器，版本C++11）

# III. 代码设计与实现

## 申明

本次项目目标是OpenBLAS中的`cblas_dgemm()`函数，它的接口规范是：

```
#ifndef OPENBLAS_CONST
# define OPENBLAS_CONST const
#endif

typedef enum CBLAS_ORDER {CblasRowMajor=101, CblasColMajor=102} CBLAS_ORDER;
typedef enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113,
CblasConjNoTrans=114} CBLAS_TRANSPOSE;
typedef int blasint;

void cblas_dgemm(
    OPENBLAS_CONST enum CBLAS_ORDER Order,
```

```
    OPENBLAS_CONST enum CBLAS_TRANSPOSE TransA,
    OPENBLAS_CONST enum CBLAS_TRANSPOSE TransB,
    OPENBLAS_CONST blasint M,
    OPENBLAS_CONST blasint N,
    OPENBLAS_CONST blasint K,
    OPENBLAS_CONST double alpha,
    OPENBLAS_CONST double *A,
    OPENBLAS_CONST blasint lda,
    OPENBLAS_CONST double *B,
    OPENBLAS_CONST blasint ldb,
    OPENBLAS_CONST double beta,
    double *C,
    OPENBLAS_CONST blasint ldc);
```

`OPENBLAS_CONST`：可以简单理解成const

`CBLAS_ORDER Order`：指的是layout，也就是指定矩阵的排列方式，按照行主序(CblasRowMajor)还是列主序(CblasColMajor)。因为矩阵在blas接口中都是一维数组表示的，行主序就是将一维数组分成M份，每份就是一行对应的值(C/C++默认方式)；列主序就是分成N份，每份就是一列对应的值(fortran默认方式)。根据OpenBLAS设置成{`CblasRowMajor`=101, `CblasColMajor`=102}

`CBLAS_TRANSPOSE TransA, TransB`：转置操作，根据OpenBLAS设置成{`CblasNoTrans`=111, `CblasTrans`=112, `CblasConjTrans`=113, `CblasConjNoTrans`=114}

- CblasNoTrans：不转置，$op(X) = X$
- CblasTrans：转置，$op(X) = X^T$
- CblasConjTrans：共轭转置，$op(X) = (\overline{X^T}) = X^*$
- CblasConjNoTrans：共轭，$op(X) = \overline{X}$

`alpha`：即$\alpha$

`beta`：即$\beta$

`*A`：矩阵A的头指针

`*B`：矩阵B的头指针

`*C`：输出矩阵(本身也作为常数项)的头指针，$C := \alpha * op(A) * op(B) + \beta * C$

`blasint`：OpenBLAS中可以是64位的整型数，也可以是32位的，这里我简单设为int

- `M`：矩阵A的行，矩阵C的行，如果A转置，则表示转置后的行数
- `N`：矩阵B的列，矩阵C的列，如果B转置，则表示转置后的列数
- `K`：矩阵A的列，矩阵B的行，如果A/B转置，则表示转置后的列数
- `lda`：若是行主序，就是每行要跳过的元素，若是列主序就是每列要跳过的元素。以行主序A(M×K)为例子，若是想取A[i][j]就是在一维数组中取A[i*lda+j]。因为A可能是某个矩阵的子矩阵，所以lda不一定等于M，需要单独保存。(列主序A[i][j]就是取A[i+j*lda],当然以上都是默认没转置)。
- `ldb`：同上，指的是矩阵B每行or列要跳过元素个数
- `ldc`：同上，指的是矩阵C每行or列要跳过元素个数

## 定义

### cblas_dgemm

根据阅读文档，我得出以下结论：

1. `M`、`N`是转置后最终结果的形状，所以参数输入时矩阵形状已固定
2. `lda`、`ldb`、`ldc`根据的不是`Order`显示的如何，应该是算上转置后的中和结果，也就是说列主序=行主序+转置；行主序 = 列主序+转置。两者可以以一定规则合并。
3. 由于我们使用的是双精度浮点矩阵，所以数据一定是实数，所以共轭不共轭结果一样。
4. 看似入参可以很随意，但错误输入在函数里都处理了。

根据上述结论，我设计了我的`cblas_dgemm`，其中我使用了`assert`库来辅助我处理异常数据。

```
void cblas_dgemm(OPENBLAS_CONST enum CBLAS_ORDER Order, OPENBLAS_CONST enum
CBLAS_TRANSPOSE TransA, OPENBLAS_CONST enum CBLAS_TRANSPOSE TransB, OPENBLAS_CONST
blasint M, OPENBLAS_CONST blasint N, OPENBLAS_CONST blasint K,
         OPENBLAS_CONST double alpha, OPENBLAS_CONST double *A, OPENBLAS_CONST
blasint lda, OPENBLAS_CONST double *B, OPENBLAS_CONST blasint ldb, OPENBLAS_CONST
double beta, double *C, OPENBLAS_CONST blasint ldc)
{
    assert(Order >= CblasRowMajor);
    assert(Order <= CblasColMajor);
    assert(TransA >= CblasNoTrans);
    assert(TransA <= CblasConjNoTrans);
    assert(TransB >= CblasNoTrans);
    assert(TransB <= CblasConjNoTrans);
    char TA, TB, TC;  // T:按列展开  N:按行展开
    TC = (Order == CblasRowMajor) ? 'N' : 'T';
    if(Order == CblasRowMajor)
    {
        TA = (TransA == CblasTrans || TransA == CblasConjTrans) ? 'T' : 'N';
        TB = (TransB == CblasTrans || TransB == CblasConjTrans) ? 'T' : 'N';
    }
    else
    {
        TA = (TransA==CblasNoTrans || TransA==CblasConjNoTrans) ? 'T' : 'N';
        TB = (TransB==CblasNoTrans || TransB==CblasConjNoTrans) ? 'T' : 'N';
    }
    assert(lda >= ((TA=='T') ? M : K));
    assert(ldb >= ((TB=='T') ? K : N));
    assert(ldc >= ((TC=='T') ? M : N));
    mat_mul(TC, TA, TB, alpha, beta, M, N, K, A, lda, B, ldb, C, ldc);
}
```

### mat_mul

`mat_mul`即矩阵乘法，能够实现$C := \alpha * op(A) * op(B) + \beta * C$。其中由于按列展开(C/C++下)的数据地址是不连续的，不能使用SIMD进行运算加速，但为了追求性能，我又特意分出来个`mat_mul_quick`来运算恰好`A`和`B`取数据地址都是连续的情况。此时我使用了__mm256来取数据，而一个双精度浮点数是64位，因此我一次可以取4个数。具体实现如下：

```
void mat_mul_quick(...参数部分省略)
{
    long long cycle1 = M;
    long long cycle2 = N;
    long long cycle3 = K;
    long long remainder = (K >> 2) << 2; // 除4再乘4，相当于减去余数
    // #ifdef __AVX512F___
    __m256d a;
    __m256d b;
    __m256d c;
    __m256d _alpha = _mm256_set1_pd(alpha);
    double result[4] = {0};
    for(long long i = 0; i < cycle1; i++)
    {
        for(long long j = 0; j < cycle2; j++)
        {
            if(remainder != 0)
            {
                #pragma omp parallel for schedule(dynamic)
                for (long long k = 0, length=cycle3-4; k <=length; k+=4)
                { // C+=αA*B
                    a = _mm256_load_pd(A + k);
                    b = _mm256_load_pd(B + k);
                    a = _mm256_mul_pd(a, _alpha);
                    c = _mm256_mul_pd(a, b);
                    _mm256_storeu_pd(result, c);
                    set_item(TC, C, ldc, result[0]+result[1]+result[2]+result[3], i,
j);
                }
            }
            for (long long k = remainder; k < cycle3; k++)
            {
                // set_item(Order, C, ldc, alpha*A[i*lda+k]*B[j*ldb+k], i, j);
                set_item(TC, C, ldc, alpha*(*get_item(TA, A, lda, i, k))*
(*get_item(TB, B, ldb, k, j)), i, j);
            }
        }
    }
}

void mat_mul(...参数部分省略)
{
    // 先把β*C给计算了，同样使用SIMD加速
    int size = M*N;
    int remainder = (size >> 2) << 2; // 除4再乘4，相当于减去余数
    __m256d c;
    __m256d _beta = _mm256_set1_pd(beta);
    #pragma omp parallel for schedule(dynamic)
    for (int k = 0, length=size-4; k <= length; k+=4)
    {
        c = _mm256_load_pd(C + k);
        c = _mm256_mul_pd(c, _beta);
```

```
        _mm256_storeu_pd(C + k, c);
    }
    for (int k = remainder; k < size; k++)
    {
        C[k] *= beta;
    }
    if(TA=='N' && TB=='T') {
        mat_mul_quick(TC, TA, TB, alpha, A, B, C, M, N, K, lda, ldb, ldc);
        return;
    }

    int cycle1 = M;
    int cycle2 = N;
    int cycle3 = K;
    remainder = (K >> 3) << 3; // 除4再乘4，相当于减去余数
    double result = 0;
    for(int i = 0; i < cycle1; i++)
    {
        for(int j = 0; j < cycle2; j++)
        {
            // #pragma omp parallel for schedule(dynamic)
            for (long long k = 0, length = cycle3-8; k<=length; k+=8)
            {
                result = 0;
                result += alpha*(*get_item(TA, A, lda, i, k))*(*get_item(TB, B, ldb,
k, j));
                result += ...连续加8次
                set_item(TC, C, ldc, result, i, j);
            }
            for (long long k = remainder; k < cycle3; k++)
            {
                // set_item(Order, C, ldc, alpha*A[i*lda+k]*B[j*ldb+k], i, j);
                set_item(TC, C, ldc,
                alpha*(*get_item(TA, A, lda, i, k))*(*get_item(TB, B, ldb, k, j)),
i, j);
            }
        }
    }
}
```

### get_item/set_item

在实现函数的过程中，我对参数并不熟悉，对数据如何取十分苦恼(转置居然不改形状只改展开方向！)。为了避免频繁修改，我将get函数和set函数都额外拎出来，如此在主函数中我就不用在意地址到底是怎么取的。

```
void set_item(TX是展开方式，N代表行主序，x，y是坐标，ld是lda/ldb/ldc,value是值，X是C)
{
    if(TX == 'N')
        X[x*ld+y] += value;
    else
        X[y*ld+x] += value;
```

```
}

OPENBLAS_CONST double *get_item(参数省略)
{
    OPENBLAS_CONST double *p = X;
    if(TX == 'N')
        p += x*ld+y;
    else
        p += y*ld+x;
    return p;
}
```

# IV. 实验

由于没领到ARM架构的开发板，所以我只在本机进行测试。不同环境运行结果可能不同。

## i. 准确性实验

代码跑得再快，也得要结果准确。因此我设计了几组小实验，对比OpenBLAS和我自己的代码的输出结果。由于测试组数较多，我这里仅展示一组(实际上就是在测试的时候发现结果不一样才意识到我对参数的理解有误)。

```
double A[6] = {1.0,2.0,1.0,-3.0,4.0,-1.0};
double B[6] = {1.0,2.0,1.0,-3.0,4.0,-1.0};
double C[9] = {.5,.5,.5,.5,.5,.5,.5,.5,.5}; // 分别运行一次，对比结果
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasTrans,3,3,2,1,A, 3, B, 3,2,C,3);
...打印...
我的结果：// 仅保留6位小数
11.000000 -9.000000 5.000000 -9.000000 21.000000 -1.000000 5.000000 -1.000000
3.000000
OpenBLAS运行结果：
11.000000 -9.000000 5.000000 -9.000000 21.000000 -1.000000 5.000000 -1.000000
3.000000
```

## ii. 效率实验

根据以往project的经验，本次我在设计时就为了后期加速做准备。首先最重要的一点就是绝对不要对数据进行任何复制，复制数据造成的时间空间浪费是得不偿失的，且容易引发内存问题。本来我是计划使用__mm512一次性提取8个双精度浮点数进行运算，但在编译过程中出了问题，我的电脑并不能很好的支持__mm512。且据我了解，现在主流的AVX也只是512位的寄存器，当两个数进行运算的时候也刚好可以占满设备，达到较好的利用率。实际上根据我所查阅到的资料，似乎512在实现上有所不同，在某方面做了优化使得效率比256高出一部分，尽管看上去硬件资源早就饱和。（注：若是使用命令行，需要在末尾添加 -mavx。）除此之外，我还用上omp和O3进行优化，并与OpenBLAS进行横向对比。然后就是设计实验，实现我实现了随机数生成的简单方法，并将它们用文件存储起来，为的是给数据提供多种输入途径。当然为追求效率，我在实验时是直接往矩阵里面写生成的随机数的，这样就省去文件读写的时间。以下是实验结果(多次测量取平均)：

| ms | my_dgemm | my_dgemm+omp | my_dgemm+O1 | my_dgemm+O2 | my_dgemm+O3 | my_dgemm+omp+O3 | OpenBLAS |
|---|---|---|---|---|---|---|---|
| 128*128 | 16.20645 | 15.60437 | 9.69792 | 7.20774 | 6.40996 | 7.15269 | 13.39384 |

| ms | my_dgemm | my_dgemm+omp | my_dgemm+O1 | my_dgemm+O2 | my_dgemm+O3 | my_dgemm+omp+O3 | OpenBLAS |
|---|---|---|---|---|---|---|---|
| 256*256 | 200.844 | 143.163 | 76.2058 | 57.6391 | 60.9899 | 78.5009 | 20.2529 |
| 512*512 | 1572.054 | 1617.181 | 705.214 | 590.208 | 602.681 | 653.988 | 27.7741 |
| 1024*1024 | 82182.8 | 79841.3 | 34167.2 | 30637.9 | 30381.2 | 29174.9 | 35.6162 |

（注：行是矩阵形状，A、B、C的形状均相同；列是优化方案，单位是ms）

根据实验结果，我的算法远逊于OpenBLAS！且O2有着和O3相近的性能，而omp的加成不明显。首先思考我的代码效率缓慢的原因，我发现我前面设计的快速矩阵乘法并没用被使用上！根据前面的说明，我特例化了地址连续的情况，即1）行主序+A不转置+B转置； 2）列主序+A转置+B不转置，有着对应的函数`mat_mul_quick`。因此，我重新进行了实验，得出了以下新的结果：

| ms | my_dgemm+O1 | my_dgemm+O2 | my_dgemm+O3 |
|---|---|---|---|
| 128*128 | 1.54985 | 1.43305 | 1.39325 |
| 256*256 | 12.5643 | 12.0285 | 11.5693 |
| 512*512 | 94.8278 | 93.7276 | 95.825 |
| 1024*1024 | 774.767 | 758.766 | 771.455 |

效率提升比例：

| | my_dgemm+O1 | my_dgemm+O2 | my_dgemm+O3 |
|---|---|---|---|
| 128*128 | 6.257328129 | 5.029650047 | 4.600724924 |
| 256*256 | 6.065264281 | 4.791877624 | 5.271701832 |
| 512*512 | 7.436785415 | 6.297056577 | 6.289392121 |
| 1024*1024 | 44.09996812 | 40.37858839 | 39.38168785 |

效率最高可比原版快44倍！这就体现出连续取址的含金量，在连续取址+SIMD汇编级加速的情况下代码效率可以在低数据量的情况下与OpenBLAS比肩(高数据量时又被甩开了)。且通过分析O1、O2、O3，发现三者在这种情况下效率十分接近！这就说明了O3已经没法比O1，O2再额外提高更多速度，也说明我的代码经过充分优化。

# V. 总结&缺陷&收获的经验

通过这次项目，我发现取址的连续性十分重要，它带给代码性能上的提升是极为显著的。也因此在进行矩阵、向量等并行运算的同时，应该尽可能保证取址的连续，通过如交换计算顺序，或者用SIMD同时取值，同时运算等手段可以极大的提高代码的运行效率。同时我也发现我代码中的缺点，比如当取址不连续时，运行效率就会低到难以忍受的程度。且omp并没用对我代码起到帮助，可能是我使用的地点有问题，应当暴露其它适合并行的循环让omp进行优化。还有一点是当我运算小矩阵时代码不会有任何问题，但当矩阵大到2000*2000以上，不进行On优化的代码就会出现错误(原因未知？)，只有开启On优化，代码才会正常执行。

以上是我本项目收获的经验。感谢老师和学助一个学期的精心教导，我在C/C++程序设计这门课受益良多，不仅仅是掌握一门编程语言，更是学到了许多代码设计的理念，和优化程序的方法以及确保内存安全的各种要点与习惯。再次感谢老师和学助们的帮助与教诲。

# VI. 源码

https://github.com/15775011722/CS205-Sustech-Cpp/tree/main/project/project5/code