矩阵乘法优化作业 1

2112060-孙蕗-信息安全

一、 作业要求

【矩阵乘法优化作业 1】参考课程中讲解的矩阵乘法优化机制和原理，在自己电脑上(windows 系统)使用相关编程环境，完成不同层次的矩阵乘法优化作业，要求如下：

1、使用个人电脑完成，不仅限于 visual studio、vscode 等。

2、在完成矩阵乘法优化后，测试矩阵规模在 1024~4096，或更大维度上，至少进行 4 个矩阵规模维度的测试。

3、在作业中需总结出不同层次，不同规模下的矩阵乘法优化对比，对比指标包括计算耗时、运行性能、加速比等。

4、在作业中总结优化过程中遇到的问题和解决方式。

5、作业无固定模板，以附件形式提交，应为 word 或 pdf 文件，文件名为："学号_姓名_组成原理矩阵乘法作业 1.pdf"

二、 源码

```
#include<iostream>
#include<time.h>
//#include<x86intrin.h>
#include<immintrin.h>

using namespace std;
#define REAL_T double

void printFlops(int A_height, int B_width, int B_height, clock_t start, clock_t stop ){
    REAL_T flops = ( 2.0 * A_height * B_width * B_height ) / 1E9 /((stop - start)/(CLOCKS_PER_SEC * 1.0));
    cout<<"GFLOPS:\t"<<flops<<endl;
}

void initMatrix( int n, REAL_T *A, REAL_T *B, REAL_T *C ){
    for( int i = 0; i < n; ++i )
        for( int j = 0; j < n; ++j ){
            A[i+j*n] = (i+j + (i*j)%100 ) %100;
            B[i+j*n] = ((i-j)*(i-j) + (i*j)%200 ) %100;
            C[i+j*n] = 0;
        }
}

void dgemm( int n, REAL_T *A, REAL_T *B, REAL_T *C){
    for( int i = 0; i < n; ++i )
        for( int j = 0; j < n; ++j ){
```

```
            REAL_T cij = C[i+j*n];
            for( int k = 0; k < n; k++ ){
                cij += A[i+k*n] * B[k+j*n];
            }
            C[i+j*n] = cij;
        }
    }

    void avx_dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
       for( int i = 0; i < n; i+=4 )
           for( int j = 0; j < n; ++j ){
               __m256d cij = _mm256_load_pd( C+i+j*n );
               for( int k = 0; k < n; k++ ){
                   //cij += A[i+k*n] * B[k+j*n];
                   cij = _mm256_add_pd(
                          cij,
                          _mm256_mul_pd(       _mm256_load_pd(A+i+k*n),
_mm256_load_pd(B+i+k*n) )
                          );
               }
               _mm256_store_pd(C+i+j*n,cij);
           }
    }

    #define UNROLL (4)

    void pavx_dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
       for( int i = 0; i < n; i+=4*UNROLL )
           for( int j = 0; j < n; ++j ){
               __m256d cij[4];
               for( int x = 0; x < UNROLL; ++x)
                   cij[x]= _mm256_load_pd( C+i+j*n );

               for( int k = 0; k < n; k++ ){
                   //cij += A[i+k*n] * B[k+j*n];
                   /*cij = _mm256_add_pd(
                          cij,
                          _mm256_mul_pd(       _mm256_load_pd(A+i+k*n),
_mm256_load_pd(B+i+k*n) )
                          );*/
                   __m256d b = _mm256_broadcast_sd( B+k+j*n );
                   for( int x = 0; x <UNROLL; ++x)
                       cij[x] = _mm256_add_pd(
                           cij[x],
```

```
                        _mm256_mul_pd(   _mm256_load_pd(A+i+4*x+k*n),
b ) );
                }
                for( int x = 0; x < UNROLL; ++x)
                    _mm256_store_pd( C+i+x*4 +j*n, cij[x]);
            }
        }

        #define BLOCKSIZE (32)
        void do_block( int n, int si, int sj, int sk, REAL_T *A, REAL_T *B,
REAL_T *C){
           for( int i = si; i < si + BLOCKSIZE; i+=UNROLL*4 )
               for( int j = sj; j < sj + BLOCKSIZE; ++j){
                   __m256d c[4];
                   for( int x = 0; x < UNROLL; ++x )
                       c[x] = _mm256_load_pd( C+i+4*x+j*n );

                   for( int k = sk; k < sk + BLOCKSIZE; ++k ){
                       __m256d b = b = _mm256_broadcast_sd( B+k+j*n );
                       for( int x = 0; x <UNROLL; ++x)
                           c[x] = _mm256_add_pd(
                               c[x],
                               _mm256_mul_pd(   _mm256_load_pd(A+i+4*x+k*n),
b ) );
                   }

                   for( int x = 0; x < UNROLL; ++x)
                       _mm256_store_pd( C+i+x*4+j*n, c[x]);
               }
        }


        void block_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
           for( int sj = 0; sj <n; sj+=BLOCKSIZE)
               for( int si = 0; si <n; si+=BLOCKSIZE)
                   for( int sk = 0; sk <n; sk+=BLOCKSIZE)
                       do_block( n, si, sj, sk, A, B, C);
        }

        void omp_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
        #pragma omp parallel for
           for( int sj = 0; sj <n; sj+=BLOCKSIZE)
               for( int si = 0; si <n; si+=BLOCKSIZE)
                   for( int sk = 0; sk <n; sk+=BLOCKSIZE)
```

```
                do_block( n, si, sj, sk, A, B, C);
    }

    void main()
    {
       REAL_T *A, *B, *C;
       clock_t start,stop;
       int n = 1024;
       A = new REAL_T[n*n];
       B = new REAL_T[n*n];
       C = new REAL_T[n*n];
       initMatrix(n, A, B, C);

       cout<< "origin caculation begin...\n";
       start = clock();
       dgemm( n, A, B, C );
       stop = clock();
       cout    <<(stop    -    start)/CLOCKS_PER_SEC<<"."<<(stop    -
start)%CLOCKS_PER_SEC<<"\t\t";
       printFlops(n, n, n, start, stop);

       initMatrix(n, A, B, C);
       cout<< "AVX caculation begin...\n";
       start = clock();
       avx_dgemm( n, A, B, C );
       stop = clock();
       cout    <<(stop    -    start)/CLOCKS_PER_SEC<<"."<<(stop    -
start)%CLOCKS_PER_SEC<<"\t\t";
       printFlops(n, n, n, start, stop);

       initMatrix(n, A, B, C);
       cout<< "parallel AVX caculation begin...\n";
       start = clock();
       pavx_dgemm( n, A, B, C );
       stop = clock();
       cout    <<(stop    -    start)/CLOCKS_PER_SEC<<"."<<(stop    -
start)%CLOCKS_PER_SEC<<"\t\t";
       printFlops(n, n, n, start, stop);

       initMatrix(n, A, B, C);
       cout<< "blocked AVX caculation begin...\n";
       start = clock();
       block_gemm( n, A, B, C );
       stop = clock();
```

```
        cout    <<(stop    -    start)/CLOCKS_PER_SEC<<"."<<(stop    -
start)%CLOCKS_PER_SEC<<"\t\t";
        printFlops(n, n, n, start, stop);

        initMatrix(n, A, B, C);
        cout<< "OpenMP blocked AVX caculation begin...\n";
        start = clock();
        omp_gemm( n, A, B, C );
        stop = clock();
        cout    <<(stop    -    start)/CLOCKS_PER_SEC<<"."<<(stop    -
start)%CLOCKS_PER_SEC<<"\t\t";
        printFlops(n, n, n, start, stop);
    }
```

#include<immintrin.h>: 引入 immintrin.h 库，该库提供了对 AVX 指令集的支持。

void printFlops(int A_height, int B_width, int B_height, clock_t start, clock_t stop): 打印每秒执行的浮点运算次数（GFLOPS）。

void initMatrix(int n, REAL_T *A, REAL_T *B, REAL_T *C): 初始化矩阵 A、B 和 C，n 为矩阵的维度。

void dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C): 矩阵乘法的原始实现。

void avx_dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C): 使用 AVX 指令集进行矩阵乘法的优化实现。

void pavx_dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C): 使用 AVX 指令集和循环展开进行矩阵乘法的优化实现。

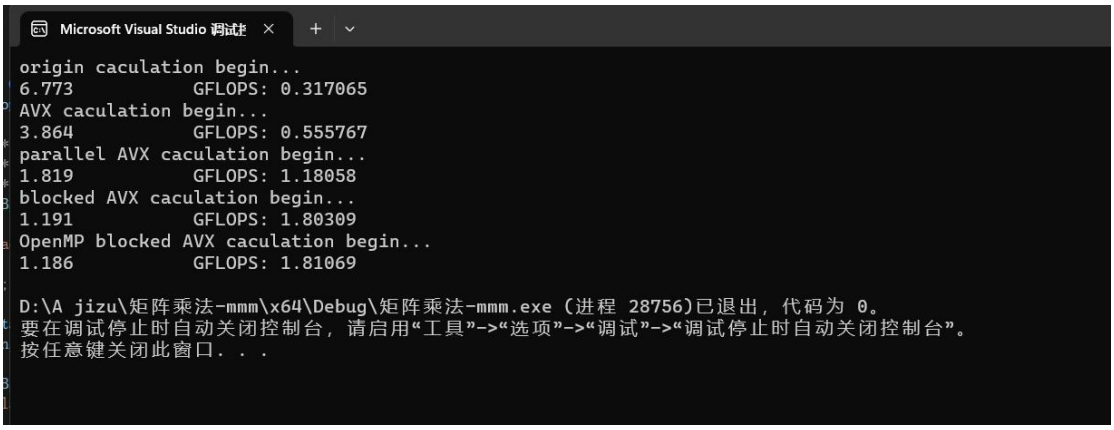
void do_block(int n, int si, int sj, int sk, REAL_T *A, REAL_T *B, REAL_T *C): 用于分块计算矩阵乘法的辅助函数。

void block_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C): 使用分块计算的方式进行矩阵乘法的优化实现。

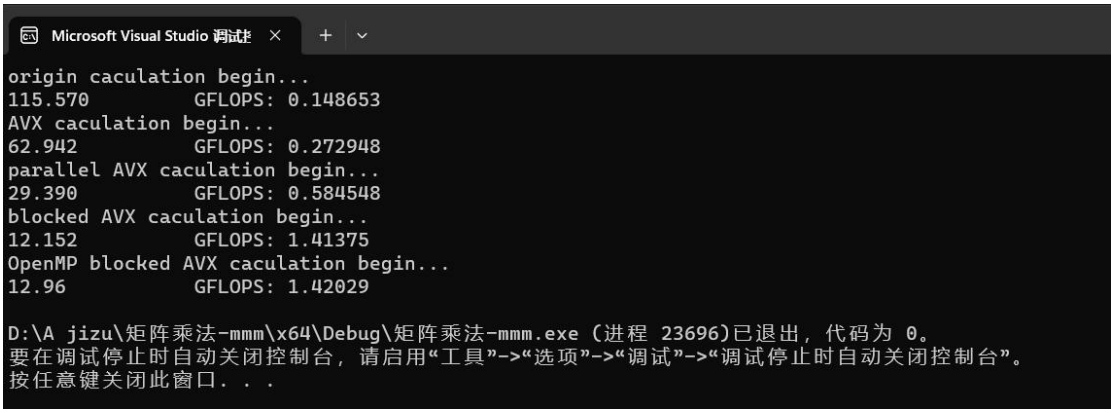
void omp_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C): 使用 OpenMP 进行并行计算的矩阵乘法优化实现。

三、 运行结果

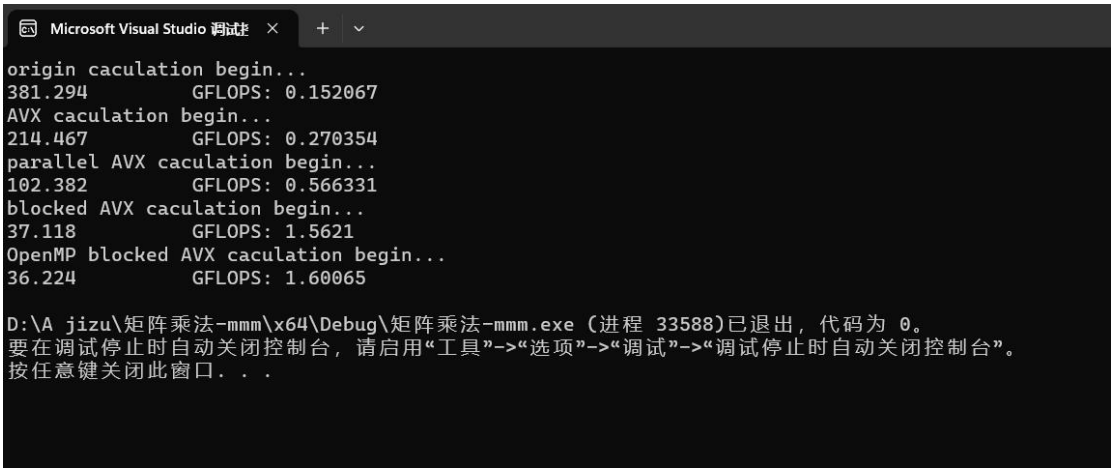
1. N=1024



2. N=2048



3. N=3072



4. N=4096

	N=1024	N=2048	N=3072	N=4096
原始矩阵乘法执行时间	6.773	115.570	381.294	1085.582
原始矩阵乘法每秒执行的浮点运算次数（GFLOPS）	0.317065	0.148653	0.152067	0.126604
使用 AVX 指令集进行矩阵乘法优化的执行时间	3.864	62.942	214.467	565.913
使用 AVX 指令集进行矩阵乘法优化的每秒执行的浮点运算次数（GFLOPS）	0.555767	0.272948	0.270354	0.242862
使用 AVX 指令集进行矩阵乘法优化的加速比	1.753	1.836	1.778	1.918
使用 AVX 指令集和循环展开进行矩阵乘法优化的执行时间	1.819	29.390	102.382	251.159
使用 AVX 指令集和循环展开进行矩阵乘法优化的每秒执行的浮点运算次数（GFLOPS）	1.18058	0.584548	0.566331	0.547219
使用 AVX 指令集和循环展开进行矩阵乘法优化的加速比	3.723	3.932	3.724	4.322
使用分块计算的方式进行矩阵乘法优化的执行时间	1.191	12.152	37.118	98.874
使用分块计算的	1.80309	1.41375	1.5621	1.39004

方式进行矩阵乘法优化的每秒执行的浮点运算次数（GFLOPS）				
使用分块计算的方式进行矩阵乘法优化的加速比	5.687	9.510	10.272	10.979
使用 OpenMP 进行并行计算的矩阵乘法优化的执行时间	1.186	12.96	36.224	99.902
使用 OpenMP 进行并行计算的矩阵乘法优化的每秒执行的浮点运算次数（GFLOPS）	1.81069	1.42029	1.60065	1.37574
使用 OpenMP 进行并行计算的矩阵乘法优化的加速比	5.711	8.917	10.526	10.866

四、 数据分析

1. 计算耗时

通过比较不同优化方式的执行时间，可以评估优化的效果。从数据中可以观察到，在不同规模的矩阵上，优化后的方法都比原始的矩阵乘法执行时间更短，这意味着优化方法能够加快计算速度，其中 OpenMP 并行计算的效果最好。

2. 运行性能

通过每秒执行的浮点运算次数（GFLOPS）来评估不同优化方法的运行性能。从数据中可以看出，使用 AVX 指令集进行矩阵乘法优化的性能比原始矩阵乘法有显著提升。而结合 AVX 指令集和循环展开的优化方法，在性能上又进一步提升。分块计算和使用 OpenMP 进行并行计算的方法也能获得较高的性能。

3. 加速比

加速比是评估优化方法相对于原始方法的性能提升程度的指标。计算加速比的方法是将原始方法的执行时间除以优化方法的执行时间。从数据中可以看到，不同规模下的矩阵乘法优化方法都获得了明显的加速比。加速比越高表示优化方法对于提升性能的效果越好。所有优化方法都获得了显著的加速比，其中分块计算和 OpenMP 并行计算的加速比较高。

4. N=2048 时、N=3072 时、N=4096

在这些情况下，与 N=1024 时相比，计算耗时、运行性能和加速比都呈现类似的趋势：优化方法的执行时间更短，运行性能更高，加速比更大。这表明这些优化方法在不同规模的矩阵上都具有一致的优化效果，在大规模矩阵运算中更为明显。

矩阵乘法执行时间、每秒执行的浮点运算次数和加速比随着 N 值的增加而增加。这是因为随着矩阵规模的增大，执行时间增加，每秒执行的浮点运算次数和加速比也随之增加。

对比不同优化方法在相同 N 值下的执行时间、每秒执行的浮点运算次数和加速比，可以发现优化方法对性能的提升效果随着 N 值的增加而增强。

五、 遇到的问题和解决方式

1. 编译错误：当使用不同的优化方法时，可能会遇到编译错误或链接错误。解决方式是检查编译器选项、库依赖关系和版本兼容性，确保正确地配置和链接所需的优化库。

2. 性能提升不明显：有时候在应用优化方法后，性能提升可能不如预期。这可能是因为优化方法对特定硬件架构或数据大小更加适用。解决方式是分析优化方法的原理，了解其适用条件，针对不同情况进行测试和调整，以获得更好的性能提升。

3. 内存限制：对于大规模矩阵乘法，可能会遇到内存限制问题，导致程序崩溃或性能下降。解决方式可以采用分块计算的方法，将大规模矩阵划分为更小的块，在每个块上执行矩阵乘法操作，以降低内存需求。

4. 并行计算问题：当使用并行计算方法（如 OpenMP）时，可能会遇到线程同步、数据竞争或负载平衡等问题，导致性能下降或错误结果。解决方式包括正确使用同步机制（如互斥锁、原子操作等）、合理划分任务和数据、调整线程数等，以确保并行计算的正确性和高效性。