组成原理课矩阵乘法优化实验报告

学号：2212046 姓名：王昱 专业：信息安全 班次：李涛老师班

（一）个人 PC 电脑实验

一、实验要求

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。如 PC 电脑有 Nvidia 显卡，建议尝试 CUDA 代码。

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。

二、实验原理

1. 向量化（Vectorization）

向量化是指使用 CPU 指令集（如 AVX）一次性处理多个数据。通过向量化，我们可以在单个指令中处理多个浮点运算，从而提高计算效率。

在代码 avx_dgemm 中，通过使用 AVX 指令集，将四个双精度浮点数打包到一个 256 位寄存器中，并同时进行运算。这使得每个循环迭代处理四个元素，而不是一个。

2. 循环展开（Loop Unrolling）

循环展开是指将循环体展开成多份，从而减少循环控制语句的开销，提高流水线效率。在代码 pavx_dgemm 中,循环展开减少了循环控制的开销,并且更好地利用了 CPU 的流水线.

3. 块分割（Blocking）

目标是在一个子矩阵（块）被 cache 替换出去之前，最大限度的对其进行数据访问。利用时间局部性,提高 cache 命中率。块分割技术是将大矩阵分割成较小的子矩阵块进行处理。这样做可以更好地利用缓存，提高数据的局部性，从而减少内存访问的开销。在代码 block_gemm 中，使用 do_block 函数对矩阵进行块分割。

4. 并行计算（Parallel Computing）

并行计算是通过多线程技术同时执行多个计算任务,以充分利用多核处理器的计算能力。在代码 omp_gemm 中，通过 OpenMP 实现了并行计算。

5. 利用 GPU 的高并行性

利用 GPU 进行计算可以极大地提高矩阵乘法的性能。CUDA（Compute Unified Device

Architecture）是 NVIDIA 推出的一种并行计算平台和编程模型。以下是 CUDA 矩阵乘法的核函数和主函数实现。

三、mmm.cpp 代码解析

（1）printFlops 函数，initMatrix 函数

```
1.  void printFlops(int A_height, int B_width, int B_height, clock_t start, cloc
    k_t stop ){
2.      REAL_T flops = ( 2.0 * A_height * B_width * B_height ) / 1E9 /((stop -
    start)/(CLOCKS_PER_SEC * 1.0));
3.      cout<<"GFLOPS:\t"<<flops<<endl;
4.  }
```

该函数的作用是计算和输出矩阵乘法的性能，以 GFLOPS（十亿次浮点运算每秒）为单位。首先计算总的浮点运算次数：2.0 * A_height * B_width * B_height，这是因为每个矩阵元素的乘法对应一个乘法和一个加法，共两次浮点运算；然后计算程序运行的总时间：(stop - start) / CLOCKS_PER_SEC。最后计算 GFLOPS 值，将总浮点运算次数除以运行时间，并转换为每秒十亿次浮点运算。

```
1.  void initMatrix( int n, REAL_T *A, REAL_T *B, REAL_T *C ){
2.      for( int i = 0; i < n; ++i )
3.          for( int j = 0; j < n; ++j ){
4.              A[i+j*n] = (i+j + (i*j)%100 ) %100;
5.              B[i+j*n] = ((i-j)*(i-j) + (i*j)%200 ) %100;
6.              C[i+j*n] = 0;
7.          }
8.  }
```

该函数的作用是初始化矩阵 A,B,C，其中，矩阵 A 和 B 被初始化为特定的模式，而矩阵 C 被初始化为 0。

（2）dgemm 基本矩阵乘法

```
1.  void dgemm( int n, REAL_T *A, REAL_T *B, REAL_T *C){
2.      for( int i = 0; i < n; ++i )
3.          for( int j = 0; j < n; ++j ){
4.              REAL_T cij = C[i+j*n];
5.              for( int k = 0; k < n; k++ ){
6.                  cij += A[i+k*n] * B[k+j*n];
7.              }
8.              C[i+j*n] = cij;
9.          }
10. }
```

该函数按照矩阵乘法的定义，实现了通常称为“ijk”排列的三重嵌套循环。这个算法的目的是将两个矩阵 A 和 B 相乘，并将结果存储在矩阵 C 中。

（3）avx_dgemm 子字并行的 GEMM

```
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);
        }
}
```

这段代码实现了子字并行，每次计算 4 个元素，利用 AVX 寄存器将 4 个双精度浮点数加载到寄存器中，进行乘法和加法操作，然后将结果存储回矩阵 C。使用 _mm256_load_pd 指令加载矩阵 C 的 4 个连续双精度浮点数到 AVX 寄存器 cij 中。然后使用 _mm256_store_pd 指令将 AVX 寄存器 cij 中的 4 个双精度浮点数存储回矩阵 C。利用 AVX 指令集，能够在单个指令中同时处理多个数据，从而显著提高矩阵乘法的性能。通过这种方式，可以更好地利用 CPU，每次运算 4 个数，理论上应有接近 4 倍的加速比。

（4）pavx_dgemm 指令集并行优化

```
#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]);
        }
}
```

这段代码实现了指令集并行优化，普通的 AVX 实现中，通常会逐行处理矩阵数据。改进后的代码通过循环展开和并行计算的方式，进一步提升矩阵乘法的性能。引入了宏定义 UNROLL (4)表示循环因子，将每次循 环展开成为 4 个迭代，每次处理 4 个元素，来实现并行执行的优化。在内层循环利用 AVX 指令来实现对应操作。其中 Unroll 对应循环展开次数，控制指令依次发射执行，理论上至少有 1 倍的加速比。

（5）do_block 分块运算

```
#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);
```

```
}
```

do_block 函数利用分块矩阵乘法的优化技术，通过分割大矩阵成小块进行计算，从而提高性能。分块矩阵乘法优化性能的关键在于提高缓存命中率和利用 SIMD 指令并行处理多个数据。外层循环控制行块范围,以步长 UNROLL * 4 遍历子矩阵块的行，每次处理一块大小为 BLOCKSIZE * UNROLL * 4 的子矩阵。中间循环控制列块范围,以步长遍历子矩阵块的列，每次处理一块大小为 BLOCKSIZE 的子矩阵；加载结果矩阵块即使用 _mm256_load_pd 指令将结果矩阵 C 的当前块加载到 AVX 寄存器 c 中。在内层的循环中,利用 AVX 指令来实现对应操作。

（6）block_gemm 优化分块 GEMM

```
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);
}
```

该函数综合利用多核处理器，实现了分块处理优化，让多个处理器同时计算不同的块。以 BLOCKSIZE 为步长扫描矩阵 C，并对每个 BLOCKSIZE * BLOCKSIZE 的子块调用 do_block 函数进行计算。do_block 函数保持不变，处理每个子块的计算。

（7）omp_gemm OpenMP 多处理器并行优化

```
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);
}
```

这段代码使用 OpenMP 并行化处理不同的子块，通过#pragma omp parallel for 代码让指令并行执行外层循环，从而将计算任务分配给多个处理器并行执行，使得计算过程可以同时在多个处理器上进行，从而大幅度提升计算速度。分块技术保证了数据的局部性，使得每个子块的计算在缓存中的命中率更高,减少了内存访问延迟使用 AVX 指令集进行向量化运算，每次处理多个数据，提高计算吞吐量。通过这种多处理器并行的分块技术，可以充分利用多核处理器的优势，加速大规模矩阵乘法运算。

（8）main 函数

首先初始化矩阵：

```
    REAL_T *A, *B, *C;
```

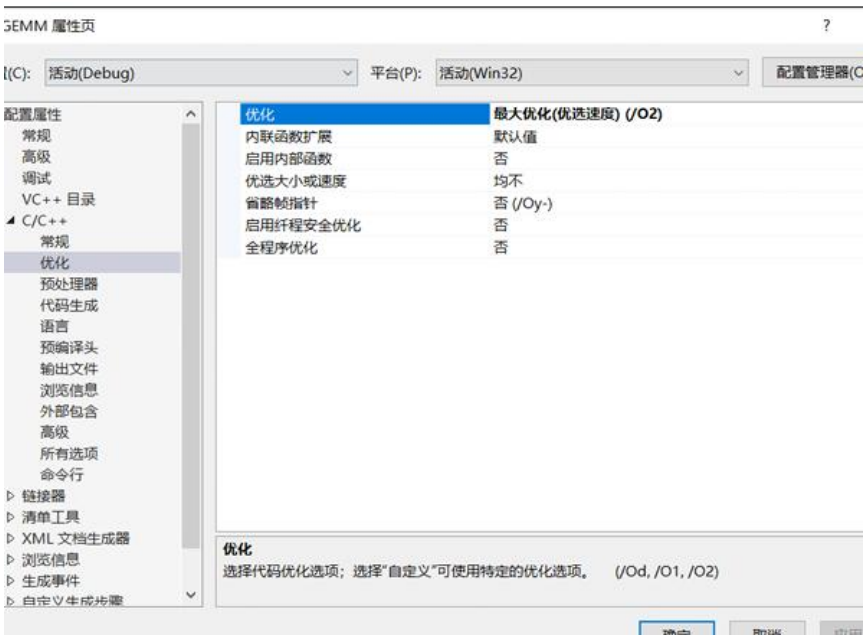
```
2.      clock_t start,stop;
3.      int n = 1024;
4.      A = new REAL_T[n*n];
5.      B = new REAL_T[n*n];
6.      C = new REAL_T[n*n];
```

随后分别对每种矩阵乘法方法进行测试，并输出执行时间和 GFLOPS。

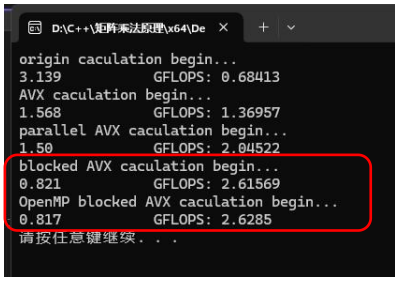
```
1.  cout<< "origin caculation begin...\n";
2.      start = clock();
3.      dgemm( n, A, B, C );
4.      stop = clock();
5.      cout <<(stop - start)/CLOCKS_PER_SEC<<"."<<(stop - start)%CLOCKS_PER_SEC
    <<"\t\t";
6.      printFlops(n, n, n, start, stop);
```

四、实验过程

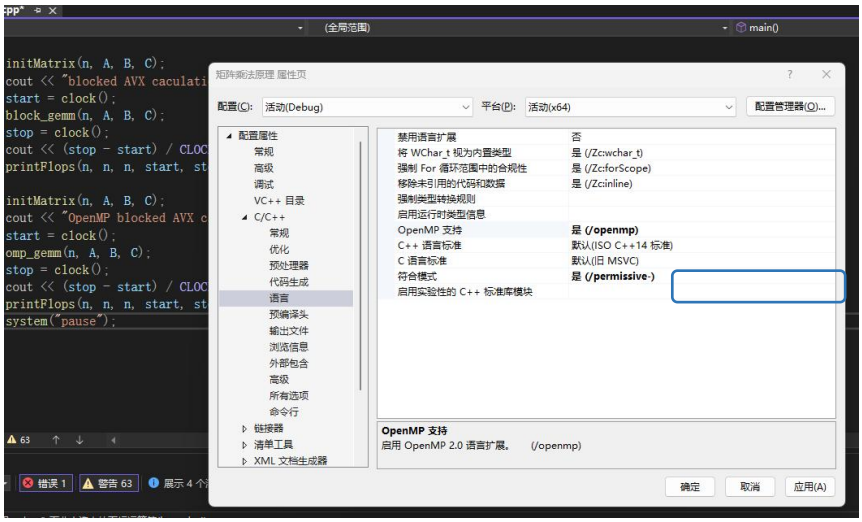
1.实验开始前，首先开启 visual studio 的 c++优化功能来提升整体的速度。



实验进行中，发现最后两种矩阵乘的性能基本相同



经过查询后得知，要想正确执行 OpenMP 多处理器并行优化，Visual Studio 编译需要将 OpenMP 支持打开，通过项目—属性—C/C++—语言来打开。



2.不同矩阵规模下实验结果分析

N=1024

N=2048

N=3072

N=4096

五、实验结果分析

1.计算性能指标

1.1. GFLOPS (Giga Floating Point Operations Per Second)

定义：GFLOPS 是衡量浮点计算性能的指标，表示每秒执行的十亿次浮点运算次数。

计算公式：GFLOPS = (2 * N * N * N) / (执行时间 * 1e9)

N 是矩阵的维度，执行时间是运行算法所花费的总时间（通常用秒表示）。

1.2. 执行时间

定义：执行时间是指从开始到结束完成计算所用的时间，通常用秒或毫秒表示。

测量方法：使用 clock() 函数测量。

1.3 加速比

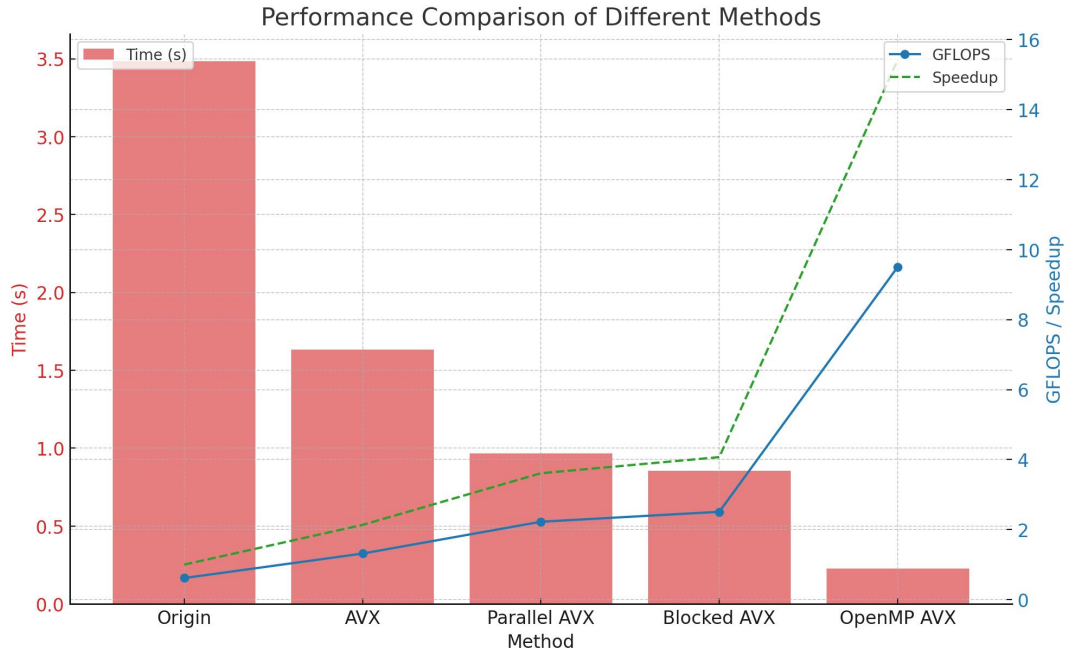
原始执行时间和优化后执行时间的比值。

2.表格以及可视化分析

N=1024

	Time	GFLOPS	Speedup
Origin	3.485	0.616	1.000
AVX	1.632	1.316	2.135
Parallel AVX	0.965	2.225	3.611
Blocked AVX	0.856	2.509	4.071
OpenMP AVX	0.226	9.502	15.420

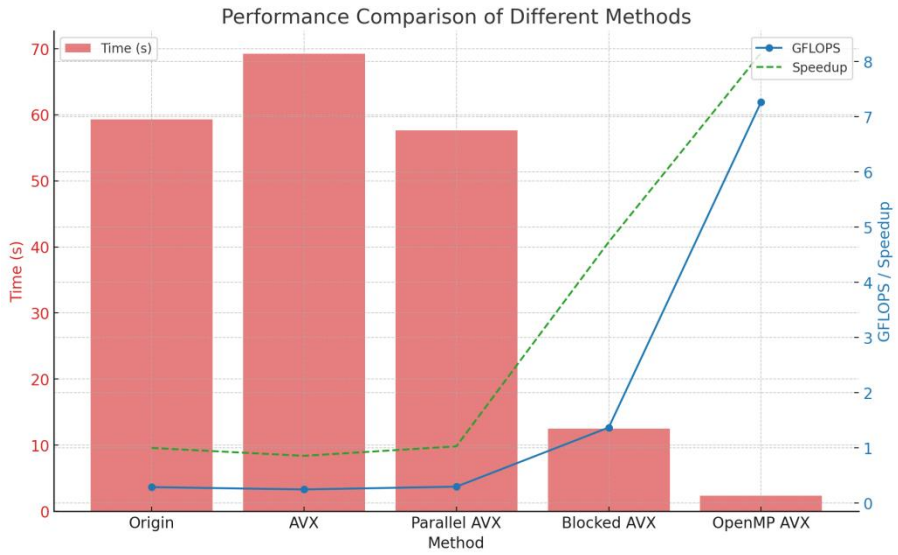
可视化图表：



N=2048

	Time	GFLOPS	Speedup
Origin	59.326	0.290	1.000
AVX	69.223	0.248	0.857
Parallel AVX	57.627	0.298	1.029
Blocked AVX	12.537	1.370	4.732
OpenMP AVX	2.363	7.270	8.160

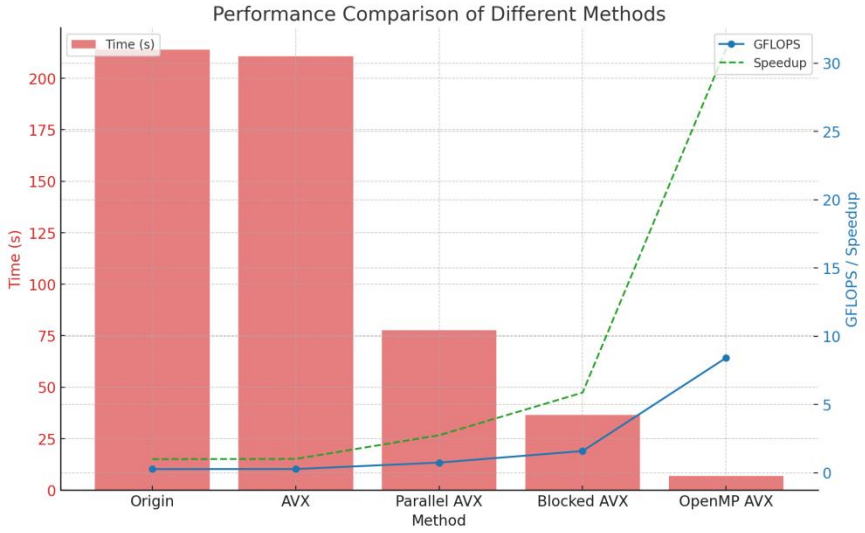
可视化图表：



N=3072

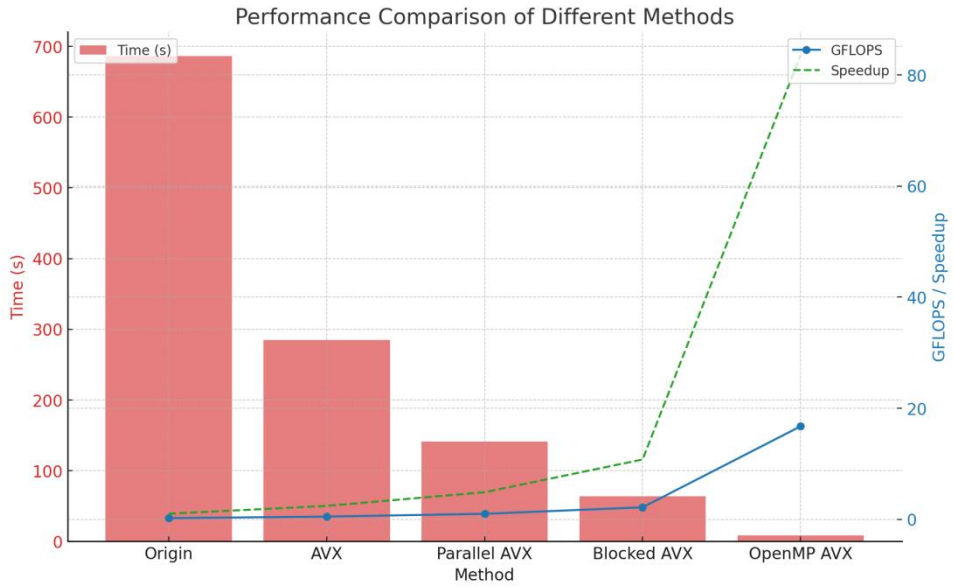
	Time	GFLOPS	Speedup
Origin	213.866	0.271	1.000
AVX	210.780	0.275	1.015
Parallel AVX	77.742	0.746	2.751
Blocked AVX	36.429	1.591	5.871
OpenMP AVX	6.888	8.418	31.049

可视化图表：



N=4096

	Time	GFLOPS	Speedup
Origin	686.322	0.200	1.000
AVX	284.913	0.482	2.409
Parallel AVX	140.830	0.981	4.873
Blocked AVX	63.771	2.155	10.762
OpenMP AVX	8.212	16.736	83.575



表格结果分析：

1.对于相同规模的矩阵：

每种优化方式相对于原始的矩阵相乘操作都有明显的优化，这种优化体现为运行时间变短、性能增大和加速比增大，且性能相比之下OpenMP>blockedAVX>blockedAVX>ParallelAVX>AVX。通过不同的优化技术（AVX 指令集、并行处理、分块技术和 OpenMP 并行）的应用，矩阵乘法的性能显著提升。特别是 OpenMP 并行 AVX 优化，表现出最好的性能，说明在高性能计算中，充分利用硬件并行计算能力和优化指令集能极大地提升计算效率。

2.对于不同规模矩阵，

2.1 运行时间：

- Origin：运行时间随着矩阵规模的增大呈指数增长，无法有效处理大规模矩阵。
- AVX 和 Parallel AVX：运行时间有所优化，但随着矩阵规模的增大，运行时间仍然显著增加，优化效果有限。

- Blocked AVX 和 OpenMP AVX：运行时间显著减少，随着矩阵规模的增大，运行时间几乎没有明显增加。这两种方法通过分块计算和并行优化，能够高效处理大规模矩阵，优化效果最为显著，特别是 OpenMP AVX 方法。
- 总结：随着矩阵规模的增大，原始矩阵相乘的方法的运算时间会变得很长，几乎呈现指数成长，所以需要进行优化，优化方法中的 AVX 和 PAVX 有所优化，但随着规模较大，运算时间也会不断变慢；然后 blocked 和 AVXOpenMP 方法通过分块计算让运行时间变快，且随着规模加大，二者几乎没有任何明显的计算耗时增加变化，仍十分稳定，优化效果最为明显。

2.2GFLOPS:

- Origin:GFLOPS 值较低，随着矩阵规模的增大，性能不佳。
- AVX 和 Parallel AVX：GFLOPS 有所提升，但提升效果有限。随着矩阵规模的增大，GFLOPS 值逐渐下降，性能优化不明显。
- Blocked AVX：GFLOPS 提升显著，优化效果明显。随着矩阵规模的增大，GFLOPS 值变化较小，性能较为稳定。在超大规模矩阵时，可能会出现性能骤降的情况。
- OpenMP AVX：GFLOPS 提升最为显著，优化效果最好。随着矩阵规模的增大，GFLOPS 值下降速度较快，但仍明显优于其他方法。

总结：Blocked AVX 和 OpenMP AVX 方法在 GFLOPS 上的优化效果显著，特别是 OpenMP AVX 方法，在各种规模下表现出最佳的性能优化效果。然而，随着矩阵规模的不断增大到 4096*4096，OpenMP AVX 的 GFLOPS 下降速度较快，尽管如此，它仍然保持了较高的性能。相比之下，AVX 和 Parallel AVX 方法的 GFLOPS 提升效果有限，性能优化不明显。这说明对性能的优化会随着矩阵规模的不断变大而变得越来越不显著。

2.3 加速比：

- AVX 和 Parallel AVX：加速比相较于原始方法有一定提升，但变化不大，优化效果有限。随着矩阵规模的增大，加速比变化不明显，优化效果逐渐接近原始方法。
- Blocked AVX：加速比提升显著，优化效果较为明显。随着矩阵规模的增大，加速比稳定，变化较小，性能优化持续有效。
- OpenMP AVX：加速比提升最为显著，优化效果最好。在各种规模下，加速比都保持在最高水平，相较于其他方法有显著提升。

总结：Blocked AVX 和 OpenMP AVX 方法在加速比上的优化效果显著，特别是 OpenMP AVX 方法，在各种规模下都表现出最佳的性能优化效果。相比之下，AVX 和

Parallel AVX 方法的加速比提升有限，随着矩阵规模的增大，优化效果逐渐接近原始方法。Blocked AVX 方法在处理大规模矩阵时，加速比较为稳定，而 OpenMP AVX 方法的加速比在所有优化方法中最高，且在一定范围内会随着矩阵规模增大而越来越高。

六、实验总结

1.在实验中，发现矩阵规模在 2048*2048 时，AVX 的方法反而低于 Origin 的方法，经过思考与查阅资料后，原因可能是：虽然 AVX 指令可以处理更多的数据，但它们的指令开销也更高。如果不能充分利用其并行能力，反而会增加额外的开销；也可能代码中对 AVX 指令的使用方式不当，导致性能未能发挥出优势。

2.实验开始时，发现 blockedAVX 优化方式和 OpenMP 优化方式竟然相同，经过资料查阅得知，是自己的 OpenMP 支持选项没有打开，因此才导致了二者优化速度相差无几，打开后代码可以正常运行。

3.对实验结果分析的过程中，发现规模越大，是否分块性能差异越来越明显，分块操作相较于不分块操作的优化效果有很大提高，并且随着矩阵规模的不断增大，分块操作对矩阵相乘的优化效果就越显著，比其他优化方式的优化效果提高好几倍，更能显示出其优越性。由此可以得出，大规模的矩阵乘法要进行分块来提升运行速度与性能。在高性能计算中，充分利用硬件的并行计算能力和优化指令集，特别是结合分块技术和多线程并行计算，是提升矩阵乘法性能的有效途径。

4.实验收获：本次实验通过对不同矩阵乘法优化方法的对比和分析，使我们深入理解了各种优化技术的原理和应用。在实际编程中，通过合理选择和组合这些优化技术，可以显著提升计算性能。这些知识和技能对于高性能计算和实际应用中的性能优化具有重要的指导意义。