

# 组成原理矩阵乘法作业报告

实验名称	矩阵乘法			班级	李涛老师
学生姓名	曹瑜	学号	2212794	指导老师	董前琨
实验地点	实验楼 A 区 306		实验时间	2024.05.16	

## 1、实验目的

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、其他系统也可以)以及 Taishan 服务器上使用相关编程环境，完成不同层次的矩阵乘法优化作业。

## 2、实验内容说明

- 在自己电脑上完成矩阵乘法编程，不仅限于 visual studio、vscode;
- 在 Taishan 服务器上完成矩阵乘法编程，使用 Putty 等远程软件在校内登录使用，服务器 IP: 222.30.62.23，端口 22，用户名 stu+学号，默认密码 123456，登录成功后可自行修改密码;
- 在完成矩阵乘法优化后（使用 AVX 库进行子字优化在 Taishan 服务器上的软件包环境不好配置，可以不进行此层次优化操作，注意原始代码需要调整），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;
- 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;
- 在作业中需对比 Taishan 服务器和自己个人电脑上程序运行时间等相关指标，分析一下不同电脑上的运行差异的原因，总结在优化过程中遇到的问题和解决方式。

## 3、实验原理

### 1、子字并行 (avx)

利用 128 位宽的字节寄存器，可一次性处理四个数据单元，理论上实现四倍的运算效率提升;

### 2、指令级并行 (pavx)

通过 SIMD（单指令多数据）技术，如 AVX 指令集，在矩阵乘法中并行执行多个数据元素的运算。一次指令执行即能完成多个乘法和累加操作，显著提高计算性能;

### 3、分块处理 (block)

面对大型矩阵，为避免 Cache 命中率下降导致的性能损失，采用分块策略。将矩阵切分成小块后独立计算，以此减少内存访问的跨度，并利用局部性原理提升 Cache 的命中率和计算效率;

### 4、多处理器并行的分块处理 (cache)

利用多核处理器或多计算节点的优势，将大型矩阵划分为多个小块，并分配给不同的处理器或节点进行计算。这种分布式处理方式能够充分利用并行计算资源，大幅提高矩阵乘法的整体计算速度。

## 4、实验步骤

[本机 visualstudio 编译代码：](#)

Main.cpp

```
#include "basic.h"

int main() {
    srand( int( time(0) ) );
    REAL_T *A, *B, *C, *a, *b, *c;
    clock_t start, stop;
    int n = 1024; // 矩阵规模

    A = new REAL_T[n*n]; a = new REAL_T[n*n];
    B = new REAL_T[n*n]; b = new REAL_T[n*n];
    C = new REAL_T[n*n]; c = new REAL_T[n*n];
    initMatrix(n, A, B, C); //构造原始矩阵

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout<< "origin caculation begin...\n";
    start = clock();
    origin_gemm( n, a, b, c );
    stop = clock();
    printFlops( n, n, n, start, stop );

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout<< "avx caculation begin...\n";
    start = clock();
    avx_gemm( n, a, b, c );
    stop = clock();
    printFlops( n, n, n, start, stop );

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout<< "pavx caculation begin...\n";
    start = clock();
    pavx_gemm( n, a, b, c );
    stop = clock();
    printFlops( n, n, n, start, stop );

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout<< "block caculation begin...\n";
    start = clock();
    block_gemm( n, a, b, c );
    stop = clock();
    printFlops( n, n, n, start, stop );
```

## Basic.h

```
#include<iostream>
#include<time.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 );
void initMatrix( int n, REAL_T *A, REAL_T *B, REAL_T *C );
void copyMatrix(int n, REAL_T *S_A, REAL_T *S_B, REAL_T *S_C, REAL_T *D_A, REAL_T *D_B,
REAL_T *D_C);

// 原始 GEMM
void origin_gemm( int n, REAL_T *A, REAL_T *B, REAL_T *C);
// 子字并行, AVX 版 GEMM
void avx_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C);
// 指令级并行, parallel AVX 版 GEMM
void pavx_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C);
// 考虑 cache 缓存的分块矩阵乘法, blockGEMM
void block_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C);
// 使用 openMP 的多核分块并行矩阵乘法
void omp_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C);
```

## Block.cpp

```
#include "basic.h"
#include<immintrin.h>
#define UNROLL (4)
#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);
}

```

## Avx.cpp

```

#include "basic.h"
#include <immintrin.h>

void avx_gemm(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);
        }
}

```

## Pavx.cpp

```
#include "basic.h"
#include <immintrin.h>

#define UNROLL (4)

void pavx_gemm(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]);
        }
}
```

## Origin.cpp

```
#include "basic.h"

void origin_gemm( 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;
        }
}
```

## Basic.cpp

```
#include "basic.h"

//计算并输出计算时间和每秒浮点运算次数
void printFlops(int A_height, int B_width, int B_height, clock_t start, clock_t stop ){
    cout<<"SECOND:\t"<<(stop - start)/CLOCKS_PER_SEC<<". "<<(stop -
start)%CLOCKS_PER_SEC<<"\t\t";

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

// 随机生成浮点数构造原始矩阵
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;
            A[i+j*n] = rand() / REAL_T(RAND_MAX);
            B[i+j*n] = rand() / REAL_T(RAND_MAX);
            C[i+j*n] = 0;
        }
}
```

[taishan 服务器 vim+gcc 编译代码:](#)

由于 avx 库不便在 Taishan 服务器上进行配置编译，故不作此层子字优化，在代码中使用普通的循环和标准数据类型来代替 avx 指令，将\_\_m256d 类型改为 double，以便能在 Taishan 服务器上运行代码;

## GEMM.cpp

```
#include <iostream>
#include<cstring>
#include <time.h>

using namespace std;

#define REAL_T double
```

```

void avx_gemm(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) {
            double 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;
        }
}

#define UNROLL (4)
void pavx_gemm(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) {
            double cij[4];
            for (int x = 0; x < UNROLL; ++x)
                cij[x] = C[i + j * n];

            for (int k = 0; k < n; k++) {
                double b = B[k + j * n];
                for (int x = 0; x < UNROLL; ++x)
                    cij[x] += A[i + 4 * x + k * n] * b;
            }
            for (int x = 0; x < UNROLL; ++x)
                C[i + x * 4 + j * n] = cij[x];
        }
}

void origin_gemm(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) {
            double 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;
        }
}

#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) {
        double c[4];
        for (int x = 0; x < UNROLL; ++x)
            c[x] = C[i + 4 * x + j * n];

        for (int k = sk; k < sk + BLOCKSIZE; ++k) {
            double b = B[k + j * n];
            for (int x = 0; x < UNROLL; ++x)
                c[x] += A[i + 4 * x + k * n] * b;
        }

        for (int x = 0; x < UNROLL; ++x)
            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 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);
}

//计算并输出计算时间和每秒浮点运算次数
void printFlops(int A_height, int B_width, int B_height, clock_t start, clock_t stop)
{
    cout << "SECOND:\t" << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
    CLOCKS_PER_SEC << "\t\t";

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

```



```

// 随机生成浮点数构造原始矩阵
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] = rand() / REAL_T(RAND_MAX);
            B[i + j * n] = rand() / REAL_T(RAND_MAX);
            C[i + j * n] = 0;
        }
}

// 拷贝矩阵
void copyMatrix(int n, REAL_T* S_A, REAL_T* S_B, REAL_T* S_C, REAL_T* D_A, REAL_T* D_B,
REAL_T* D_C) {
    memcpy(D_A, S_A, n * n * sizeof(REAL_T));
    memcpy(D_B, S_B, n * n * sizeof(REAL_T));
    memcpy(D_C, S_C, n * n * sizeof(REAL_T));
}

int main() {
    srand(int(time(0)));
    REAL_T* A, * B, * C, * a, * b, * c;
    clock_t start, stop;
    int n = 3072; // 矩阵规模

    A = new REAL_T[n * n]; a = new REAL_T[n * n];
    B = new REAL_T[n * n]; b = new REAL_T[n * n];
    C = new REAL_T[n * n]; c = new REAL_T[n * n];
    initMatrix(n, A, B, C); //构造原始矩阵

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout << "origin caculation begin...\n";
    start = clock();
    origin_gemm(n, a, b, c);
    stop = clock();
    printFlops(n, n, n, start, stop);

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout << "pavx caculation begin...\n";
    start = clock();
    pavx_gemm(n, a, b, c);
    stop = clock();
    printFlops(n, n, n, start, stop);

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据

```

```

cout << "block caculation begin...\n";
    start = clock();
    block_gemm(n, a, b, c);
    stop = clock();
    printFlops(n, n, n, start, stop);

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout << "openmp caculation begin...\n";
    start = clock();
    omp_gemm(n, a, b, c);
    stop = clock();
    printFlops(n, n, n, start, stop);
}

```

## 5、实验结果分析

### (一) 本机 visual studio 编程:

1024\*1024

```

int n = 1024; // 矩阵规模

```



Microsoft Visual Studio 调试控制台

```

origin caculation begin...
SECOND: 3.401          GFLOPS: 0.631427
avx caculation begin...
SECOND: 1.977          GFLOPS: 1.08623
pavx caculation begin...
SECOND: 1.8            GFLOPS: 2.13044
block caculation begin...
SECOND: 0.945          GFLOPS: 2.27247
openmp caculation begin...
SECOND: 0.153          GFLOPS: 14.0358
F:\桌面\24年组成原理GEMM资料\testGEMM\Debug\testGEMM.exe (进程 20840)已退出, 代码为 0。
按任意键关闭此窗口。 . . .


```

2048\*2048

```

int n = 2048; // 矩阵规模

```



Microsoft Visual Studio 调试控制台

```

origin caculation begin...
SECOND: 53.140         GFLOPS: 0.323294
avx caculation begin...
SECOND: 32.936         GFLOPS: 0.521614
pavx caculation begin...
SECOND: 14.44          GFLOPS: 1.22329
block caculation begin...
SECOND: 7.593          GFLOPS: 2.26259
openmp caculation begin...
SECOND: 1.139          GFLOPS: 15.0833
F:\桌面\24年组成原理GEMM资料\testGEMM\Debug\testGEMM.exe (进程 17268)已退出, 代码为 0。
按任意键关闭此窗口。 . . .

```

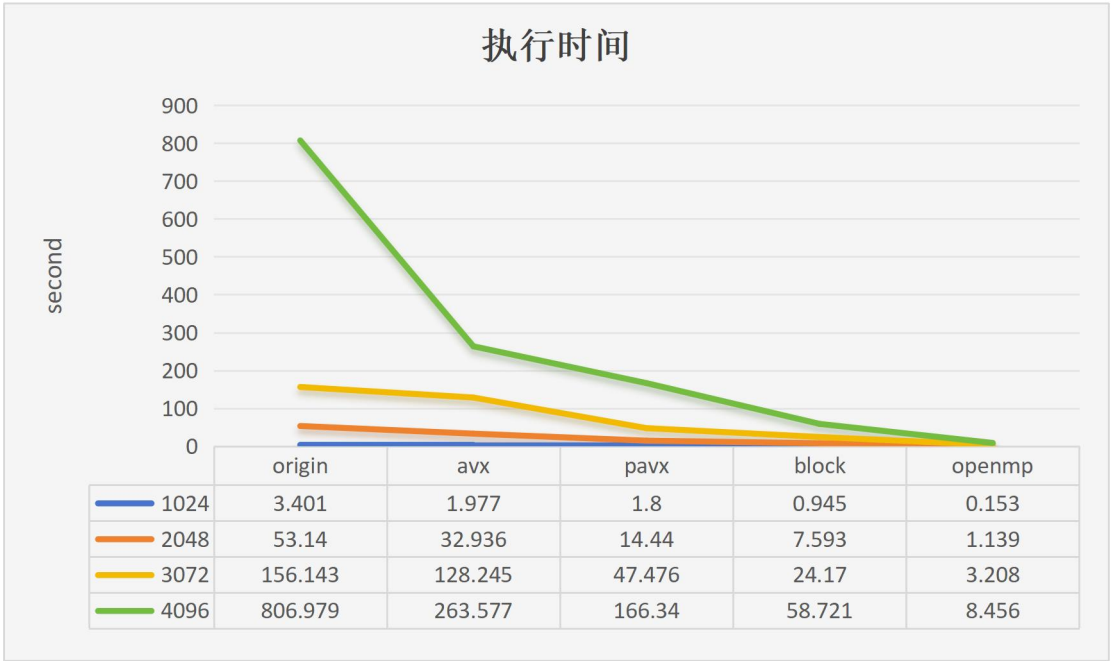
3072\*3072

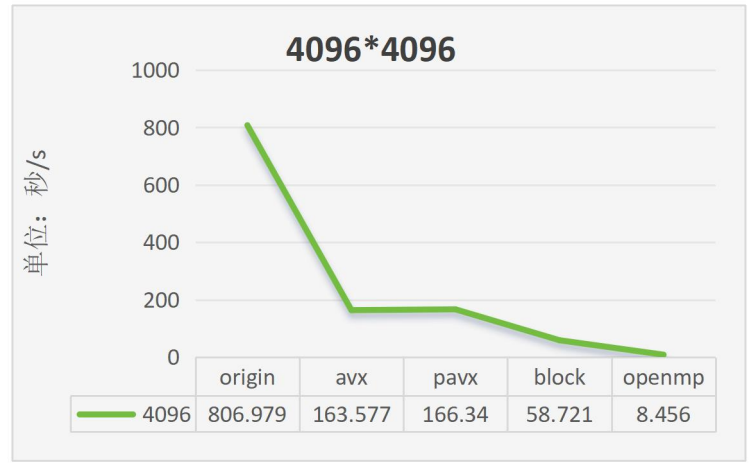
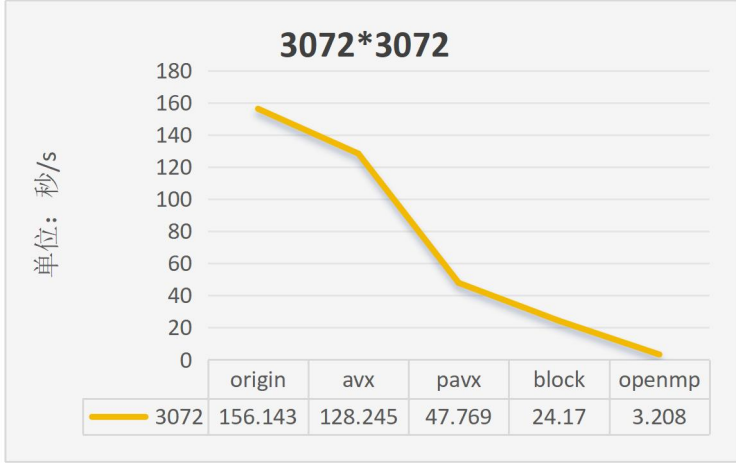
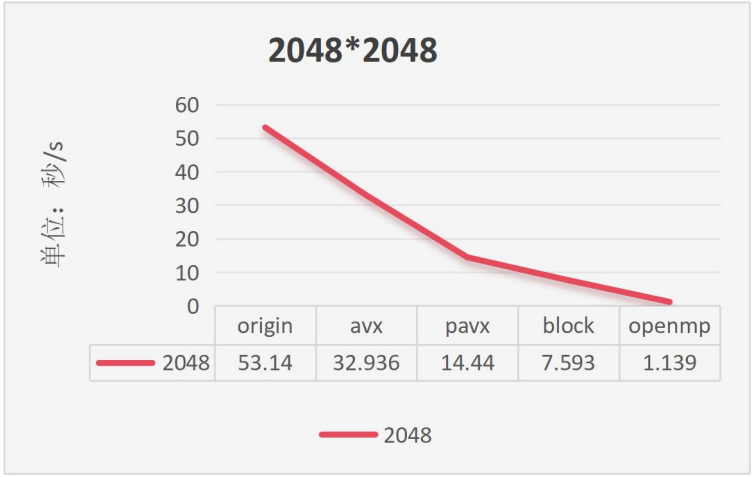
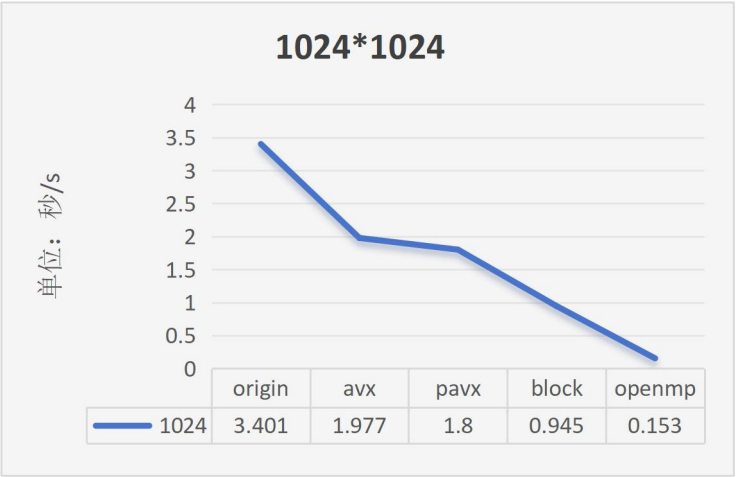
```
int n = 3072; // 矩阵规模
Microsoft Visual Studio 调试控制台
origin caculation begin...
SECOND: 156.143 GFLOPS: 0.371339
avx caculation begin...
SECOND: 128.245 GFLOPS: 0.452119
pavx caculation begin...
SECOND: 47.769 GFLOPS: 1.2138
block caculation begin...
SECOND: 24.17 GFLOPS: 2.41421
openmp caculation begin...
SECOND: 3.208 GFLOPS: 18.0742
F:\桌面\24年组成原理GEMM资料\testGEMM\Debug\testGEMM.exe (进程 21448)已退出, 代码为 0。
按任意键关闭此窗口. . .
```

4096\*4096

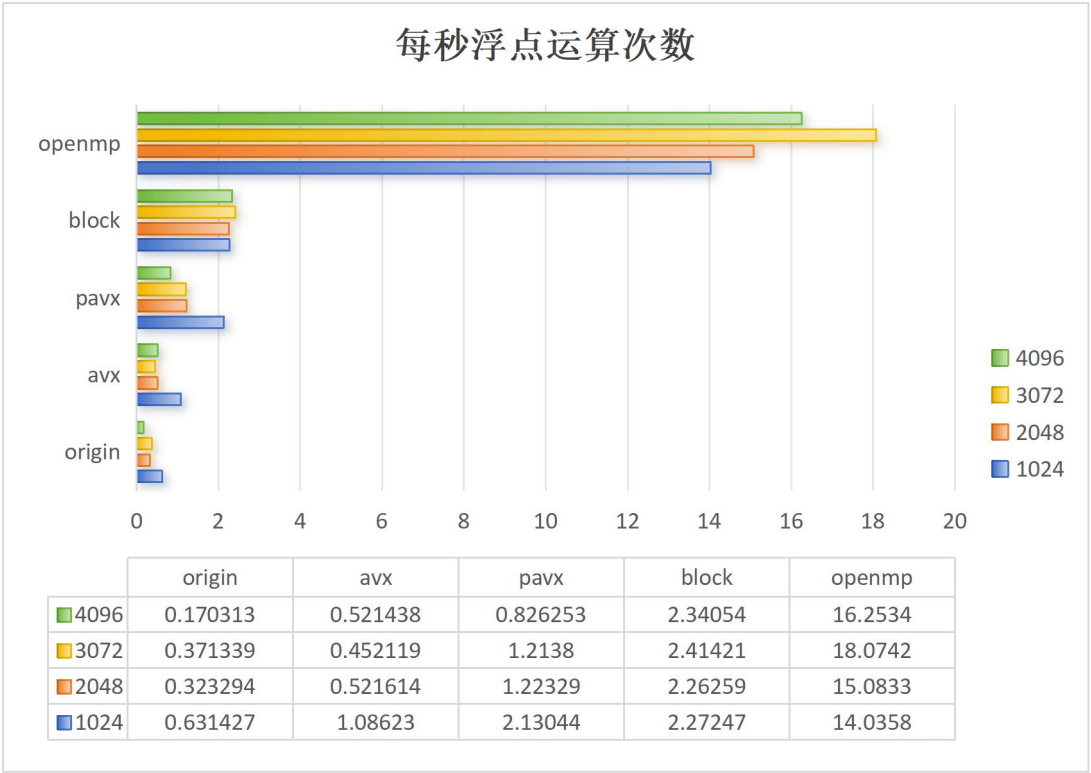
```
int n = 4096; // 矩阵规模
Microsoft Visual Studio 调试控制台
origin caculation begin...
SECOND: 806.979 GFLOPS: 0.170313
avx caculation begin...
SECOND: 263.577 GFLOPS: 0.521438
pavx caculation begin...
SECOND: 166.340 GFLOPS: 0.826253
block caculation begin...
SECOND: 58.721 GFLOPS: 2.34054
openmp caculation begin...
SECOND: 8.456 GFLOPS: 16.2534
F:\桌面\24年组成原理GEMM资料\testGEMM\Debug\testGEMM.exe (进程 10568)已退出, 代码为 0。
按任意键关闭此窗口. . .
```

执行时间对比：

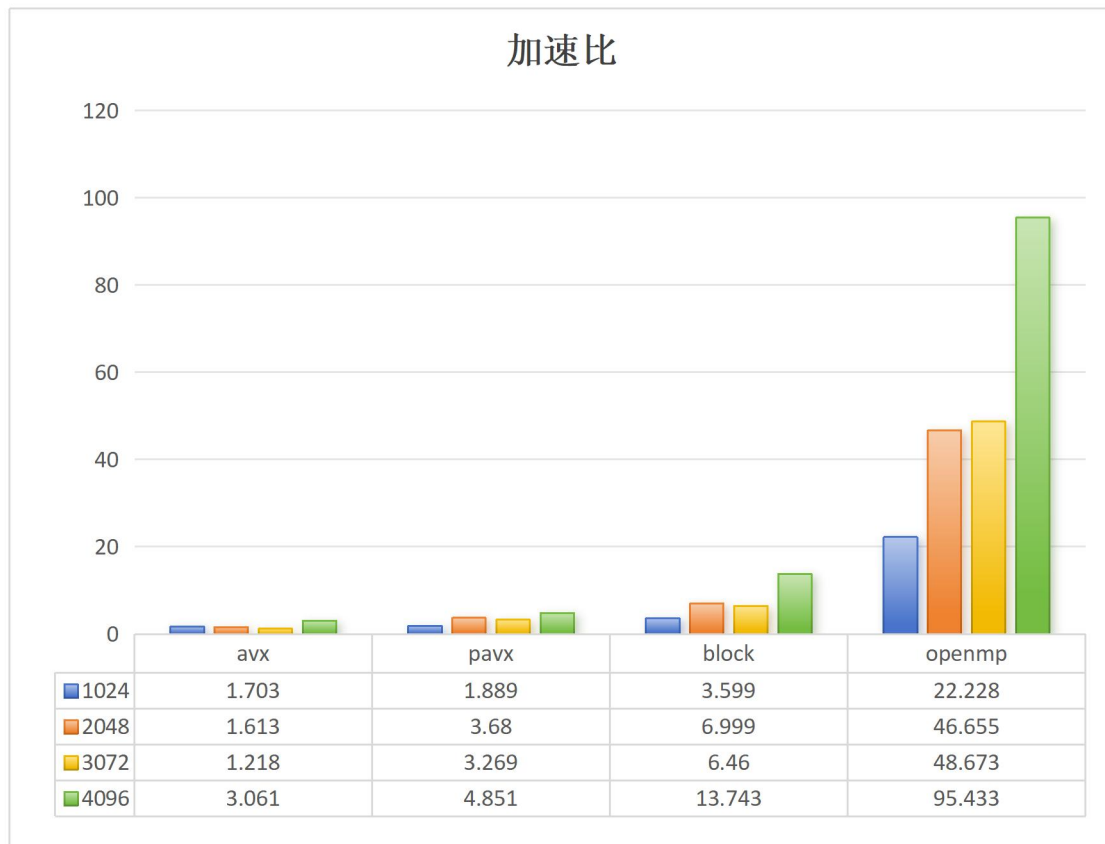




每秒浮点运算次数对比：



加速比对比



由实验结果可知：（本机 vs 编译环境）

- 1、原始矩阵乘法的执行时间随着矩阵 size 的增长有近似  $n^3$  的增幅，而进行优化后执行时间有明显的大幅下降；
- 2、大规模矩阵上的优化效果更加明显，4096 矩阵上的优化加速比和其他规模矩阵不在一个数量级；
- 3、子字并行优化的加速比理论值为 4，事实上却达不到 4，尤其是小规模矩阵，分析原因可能为函数调用和循环控制的开销占比较大，而实际计算的比例相对较小，在小规模矩阵上，循环展开和向量化的收益较小，计算量不足以掩盖这些优化所需的开销，即使每次能并行处理多个元素，但整体的计算时间仍然受到其他因素（如加载、存储、缓存访问）的限制，因此小规模矩阵上的加速比不理想；
- 4、分块操作在大规模矩阵上的优化效果更明显，分析原因可能为在小规模矩阵上，缓存未命中的代价较小，数据量不大时缓存可以较好地容纳全部数据，因此分块优化的效果不如在大规模矩阵上明显；
- 5、多核处理器的优化效果是稳定且显著的；

## (二) Taishan 服务器 vim+gcc 编程:

1024\*1024:

```
stu2212794@parallel542-taishan200-1:~$ ./test
origin caculation begin...
SECOND: 17.566540          GFLOPS: 0.122249
pavx caculation begin...
SECOND: 3.215660          GFLOPS: 0.66782
block caculation begin...
SECOND: 1.831374          GFLOPS: 1.17261
openmp caculation begin...
SECOND: 1.831449          GFLOPS: 1.17256
```

2048\*2048:

```
stu2212794@parallel542-taishan200-1:~$ ./test
origin caculation begin...
SECOND: 219.980439        GFLOPS: 0.0780973
pavx caculation begin...
SECOND: 42.189087         GFLOPS: 0.407211
block caculation begin...
SECOND: 15.200529         GFLOPS: 1.13022
openmp caculation begin...
SECOND: 15.192316         GFLOPS: 1.13083
stu2212794@parallel542-taishan200-1:~$
```

3072\*3072:

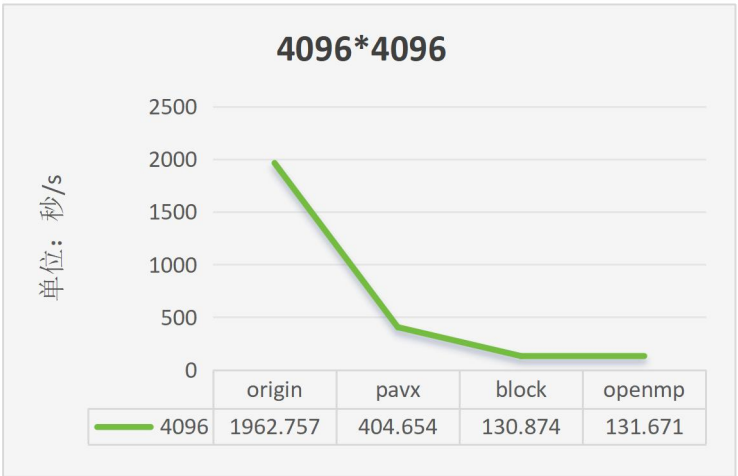
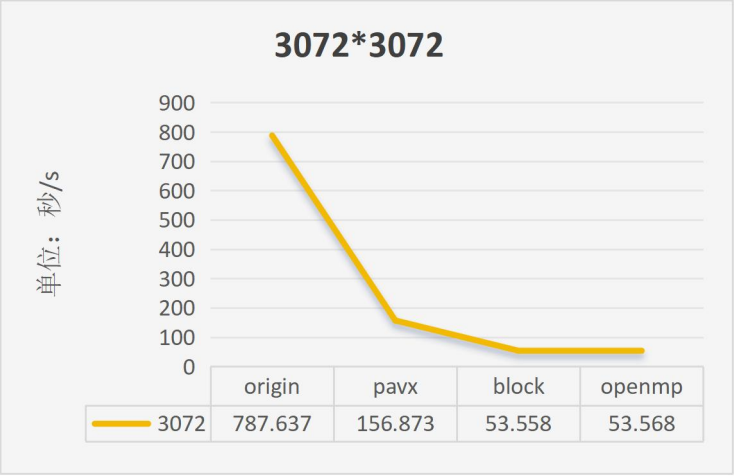
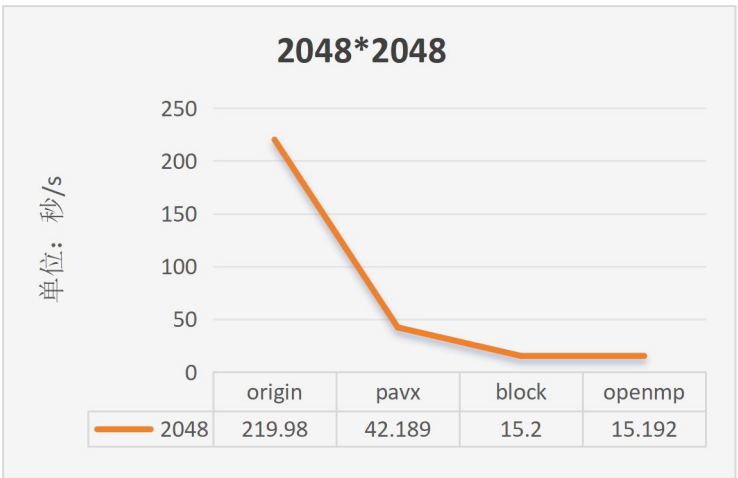
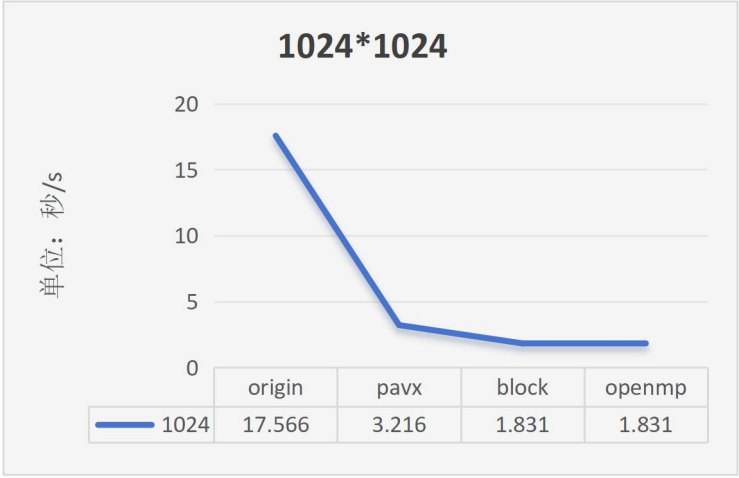
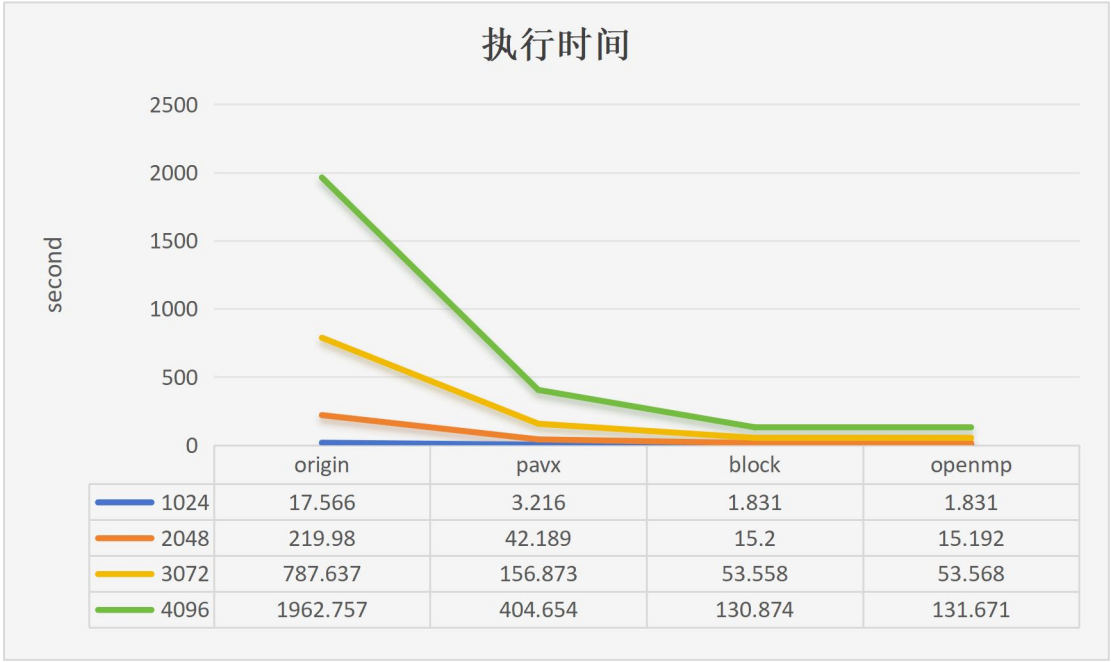
```
stu2212794@parallel542-taishan200-1:~$ ./test
origin caculation begin...
SECOND: 787.637996        GFLOPS: 0.0736151
pavx caculation begin...
SECOND: 156.873131        GFLOPS: 0.369611
block caculation begin...
SECOND: 53.558613         GFLOPS: 1.08259
openmp caculation begin...
SECOND: 53.568835         GFLOPS: 1.08238
stu2212794@parallel542-taishan200-1:~$
```

4096\*4096:

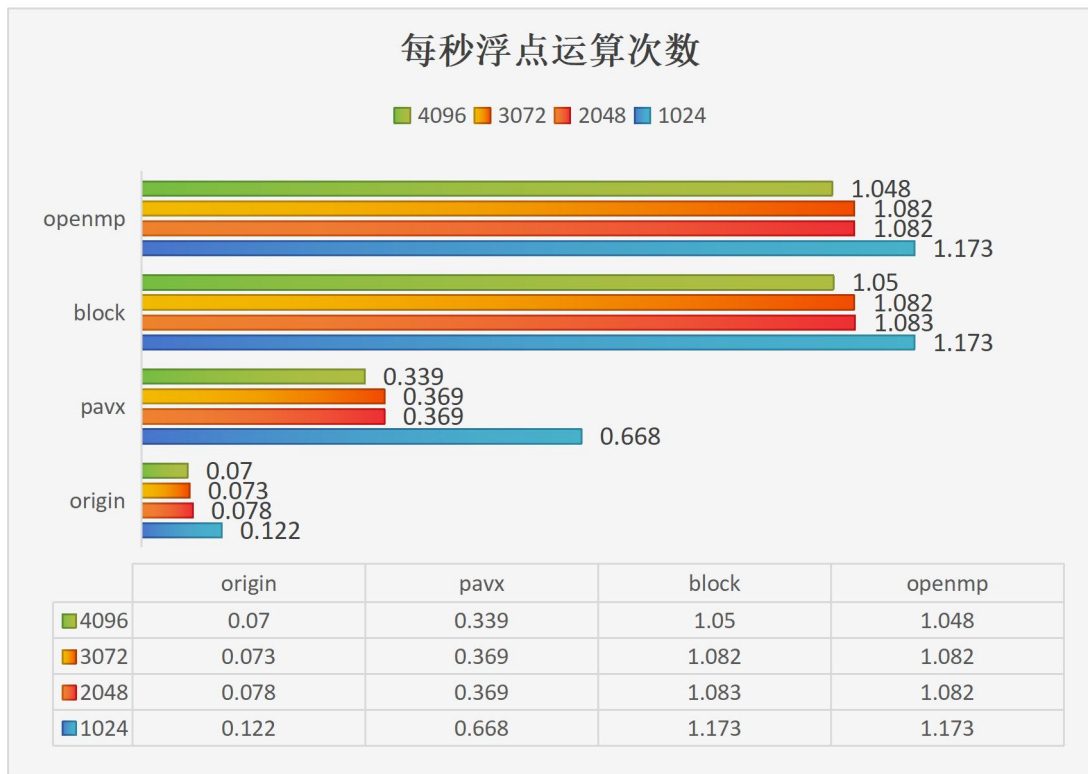
```
stu2212794@parallel542-taishan200-1:~$ g++ T.cpp -o tes
stu2212794@parallel542-taishan200-1:~$ ./test
origin caculation begin...
SECOND: 1962.757394       GFLOPS: 0.0700234
pavx caculation begin...
SECOND: 404.654218        GFLOPS: 0.339645
block caculation begin...
SECOND: 130.874251        GFLOPS: 1.05016
openmp caculation begin...
SECOND: 131.67151         GFLOPS: 1.04861
stu2212794@parallel542-taishan200-1:~$
```



执行时间对比：



每秒浮点运算次数：



加速比对比：





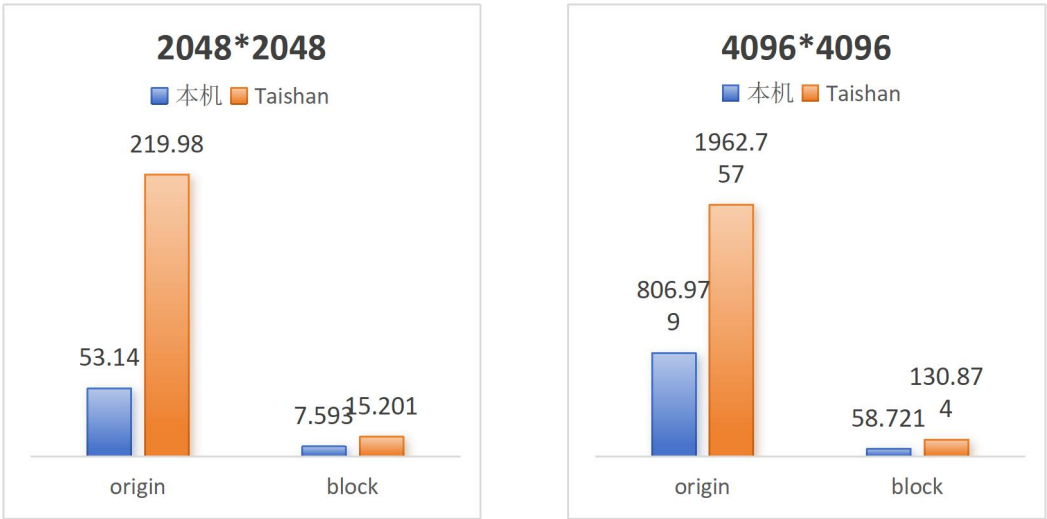
由实验结果可知：（Taishan 服务器 vim+gcc 编译环境）

- 1、原始矩阵乘法的执行时间随着矩阵 size 的增长有近似  $n^3$  的增幅，而进行优化后执行时间有明显的下降；
- 2、矩阵规模对优化效果的影响不是特别显著，不同规模的矩阵优化加速比差不多在同一个数量级；
- 3、多核处理器并行操作几乎没有优化效果，实验时我已经使用 `g++ -fopenmp -o test test.cpp` 命令启用 openmp，但最终多处理器和单处理器的并行分块时间是相同的，不知道是否为未分配多核处理器或分配的 cache 数不够的问题；

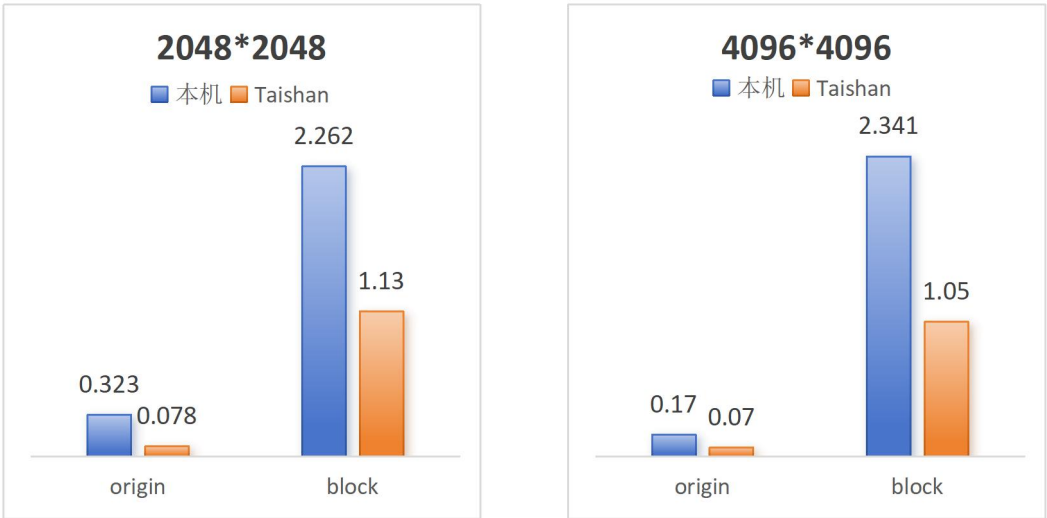
（三）对比分析：

以矩阵规模为 2048 和 4096 为例：对比 origin 和 block 的执行时间及 GFLOPS

执行时间对比：



GFLOPS 对比：



由对比结果可知：

1、Taishan 服务器的计算速度远不及本机，在矩阵乘法优化代码的编译性能上明显比本机要差，而且似乎没有实现多核并行的优化效果；

2、我的计算机设备配置如下：

处理器 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz

机带 RAM 16.0 GB (15.7 GB 可用)

系统类型 64 位操作系统，基于 x64 的处理器

Taishan 服务器规格如下：

处理器：支持 2 路处理器，处理器包含 64 核，48 核，40 核，32 核三种配置，频率均为 2.6GHz，L3 Cache 容量最大为 64MB；

内存：最多 32 个 DDR4 内存插槽，支持 RDIMM。

内存设计速率最大可达 2933MT/s。

内存保护支持 ECC、SEC/DED、SDDC、Patrol scrubbing 功能。

单根内存条容量支持 16GB/32GB/64GB/128GB。

3、从处理器和内存的角度来看，泰山服务器在核心数、内存容量和内存速率方面性能都更好，但不知为何运算速度不如本机，猜测可能是本机 vs 编译器调试后支持多核并行使得优化能高效进行；

## 6、实验总结

通过在本机 visualstudio 和 Taishan 服务器 vim+gcc 环境下完成不同层次的矩阵乘法优化，对比不同矩阵规模、不同优化方式及不同硬件配置对优化效果的影响，更深入地理解了处理思路和硬件配置对程序优化的决定性作用。