

矩阵乘法优化作业

学生姓名：胡博浩 学号：2212998 指导老师：董前琨 班级：李涛

一、实验要求

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。

二、实验原理

- 子字并行：通过使用更宽的寄存器一次性处理多个数据单元，从而提高计算速度。例如，使用128位宽度的字节寄存器可以一次性计算四个单位的数，理论上可以获得四倍的运算速度。
- 指令级并行：通过同时执行多条指令来提高计算性能。在矩阵乘法等计算密集型任务中，可以利用SIMD（单指令多数据）指令集，如AVX，同时处理多个数据元素。通过将矩阵元素打包成向量形式，并使用适当的指令集，可以在一次指令执行中完成多个乘法和累加操作。这种技术包括流水线、超标量、超长指令字等。
- Cache分块：当处理的数据尺寸过大时，数据访问会倾向于访问附近的数据，但是如果数据大小超过了Cache的容量，命中率会降低，导致处理器需要多次访问内存，从而造成时间浪费。通过将大的数据块分割成适当大小的块，然后按块进行计算，可以减少数据访问的跨越，提高Cache命中率。这利用了时间局部性和Cache的局部性原理来提高性能。
- 多核并行：利用多核处理器的并行计算能力，将大的计算任务分割成多个块，并分配给不同的处理器或计算节点，每个处理器或节点独立计算其分配的块，最后合并结果。这样可以充分利用并行计算资源，加速整个计算过程。

从CPU处理器、流水、处理器访存速度差异、多核四个方面优化，有效地加速计算任务的执行。

三、个人PC电脑实验

（一）实验要求

- 使用个人电脑完成，不仅限于visual studio、vscode等。
- 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。
- 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。
- 在作业中总结优化过程中遇到的问题和解决方式。

(二) 实验环境

在这个实验中，我使用visual studio2022作为编程环境，电脑为Windows11系统

(三) 代码编写

由于我的电脑没有Nvidia显卡，故不进行CUDA相关的编写。

复现老师提供的压缩包中的代码，将几个文件整合为一个文件

```
1 #include<iostream>
2 #include<time.h>
3 #include<omp.h>
4 #include<immintrin.h>
5
6 using namespace std;
7 #define REAL_T double
8
9 void printFlops(int A_height, int B_width, int B_height, clock_t start, clock_t
    stop) {
10     REAL_T flops = (2.0 * A_height * B_width * B_height) / 1E9 / ((stop -
        start) / (CLOCKS_PER_SEC * 1.0));
11     cout << "GFLOPS:\t" << flops << endl;
12 }
13
14 void initMatrix(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
15     for (int i = 0; i < n; ++i)
16         for (int j = 0; j < n; ++j) {
17             A[i + j * n] = (i + j + (i * j) % 100) % 100;
18             B[i + j * n] = ((i - j) * (i - j) + (i * j) % 200) %
                100;
19             C[i + j * n] = 0;
20         }
21 }
22
23 void dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
24     for (int i = 0; i < n; ++i)
25         for (int j = 0; j < n; ++j) {
26             REAL_T cij = C[i + j * n];
27             for (int k = 0; k < n; k++) {
28                 cij += A[i + k * n] * B[k + j * n];
29             }
30             C[i + j * n] = cij;
31         }
32 }
33
34 void avx_dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
```

```

35     for (int i = 0; i < n; i += 4)
36         for (int j = 0; j < n; ++j) {
37             __m256d cij = _mm256_load_pd(C + i + j * n);
38             for (int k = 0; k < n; k++) {
39                 //cij += A[i+k*n] * B[k+j*n];
40                 cij = _mm256_add_pd(
41                     cij,
42                     _mm256_mul_pd(_mm256_load_pd(A + i + k
43 * n), _mm256_load_pd(B + i + k * n))
44                 );
45             }
46             _mm256_store_pd(C + i + j * n, cij);
47         }
48 }
49 #define UNROLL (4)
50
51 void pavx_dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
52     for (int i = 0; i < n; i += 4 * UNROLL)
53         for (int j = 0; j < n; ++j) {
54             __m256d c1 = _mm256_load_pd(C + i + 0 * 4 + j * n);
55             __m256d c2 = _mm256_load_pd(C + i + 1 * 4 + j * n);
56             __m256d c3 = _mm256_load_pd(C + i + 2 * 4 + j * n);
57             __m256d c4 = _mm256_load_pd(C + i + 3 * 4 + j * n);
58
59             for (int k = 0; k < n; k++) {
60                 __m256d b = _mm256_broadcast_sd(B + k + j * n);
61                 c1 = _mm256_add_pd(
62                     c1,
63                     _mm256_mul_pd(_mm256_load_pd(A + i + 4
64 * 0 + k * n), b));
65                 c2 = _mm256_add_pd(
66                     c2,
67                     _mm256_mul_pd(_mm256_load_pd(A + i + 4
68 * 1 + k * n), b));
69                 c3 = _mm256_add_pd(
70                     c3,
71                     _mm256_mul_pd(_mm256_load_pd(A + i + 4
72 * 2 + k * n), b));
73                 c4 = _mm256_add_pd(
74                     c4,
75                     _mm256_mul_pd(_mm256_load_pd(A + i + 4
76 * 3 + k * n), b));
77             }
78             _mm256_store_pd(C + i + 0 * 4 + j * n, c1);
79             _mm256_store_pd(C + i + 1 * 4 + j * n, c2);
80             _mm256_store_pd(C + i + 2 * 4 + j * n, c3);
81             _mm256_store_pd(C + i + 3 * 4 + j * n, c4);
82         }
83     }
84 }

```

```

77         _mm256_store_pd(C + i + 3 * 4 + j * n, c4);
78     }
79 }
80
81 #define BLOCKSIZE (32)
82 void do_block(int n, int si, int sj, int sk, REAL_T* A, REAL_T* B, REAL_T* C) {
83     for (int i = si; i < si + BLOCKSIZE; i += UNROLL * 4)
84         for (int j = sj; j < sj + BLOCKSIZE; ++j) {
85             __m256d c1 = _mm256_load_pd(C + i + 0 * 4 + j * n);
86             __m256d c2 = _mm256_load_pd(C + i + 1 * 4 + j * n);
87             __m256d c3 = _mm256_load_pd(C + i + 2 * 4 + j * n);
88             __m256d c4 = _mm256_load_pd(C + i + 3 * 4 + j * n);
89
90             for (int k = sk; k < sk + BLOCKSIZE; ++k) {
91                 __m256d b = _mm256_broadcast_sd(B + k + j * n);
92                 c1 = _mm256_add_pd(
93                     c1,
94                     _mm256_mul_pd(_mm256_load_pd(A + i + 4
95 * 0 + k * n), b));
96                 c2 = _mm256_add_pd(
97                     c2,
98                     _mm256_mul_pd(_mm256_load_pd(A + i + 4
99 * 1 + k * n), b));
100                 c3 = _mm256_add_pd(
101                     c3,
102                     _mm256_mul_pd(_mm256_load_pd(A + i + 4
103 * 2 + k * n), b));
104                 c4 = _mm256_add_pd(
105                     c4,
106                     _mm256_mul_pd(_mm256_load_pd(A + i + 4
107 * 3 + k * n), b));
108             }
109
110             _mm256_store_pd(C + i + 0 * 4 + j * n, c1);
111             _mm256_store_pd(C + i + 1 * 4 + j * n, c2);
112             _mm256_store_pd(C + i + 2 * 4 + j * n, c3);
113             _mm256_store_pd(C + i + 3 * 4 + j * n, c4);
114         }
115     }
116 }
117
118 void block_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
119     for (int sj = 0; sj < n; sj += BLOCKSIZE)
120         for (int si = 0; si < n; si += BLOCKSIZE)
121             for (int sk = 0; sk < n; sk += BLOCKSIZE)
122                 do_block(n, si, sj, sk, A, B, C);
123 }

```

```

120 void omp_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
121     #pragma omp parallel for
122         for (int sj = 0; sj < n; sj += BLOCKSIZE)
123             for (int si = 0; si < n; si += BLOCKSIZE)
124                 for (int sk = 0; sk < n; sk += BLOCKSIZE)
125                     do_block(n, si, sj, sk, A, B, C);
126 }
127
128 void main()
129 {
130     REAL_T* A, * B, * C;
131     clock_t start, stop;
132     int n = 1024;
133     A = new REAL_T[n * n];
134     B = new REAL_T[n * n];
135     C = new REAL_T[n * n];
136     initMatrix(n, A, B, C);
137
138     cout << "origin caculation begin...\n";
139     start = clock();
140     dgemm(n, A, B, C);
141     stop = clock();
142     cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
143     printFlops(n, n, n, start, stop);
144
145     initMatrix(n, A, B, C);
146     cout << "AVX caculation begin...\n";
147     start = clock();
148     avx_dgemm(n, A, B, C);
149     stop = clock();
150     cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
151     printFlops(n, n, n, start, stop);
152
153     initMatrix(n, A, B, C);
154     cout << "parallel AVX caculation begin...\n";
155     start = clock();
156     pavx_dgemm(n, A, B, C);
157     stop = clock();
158     cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
159     printFlops(n, n, n, start, stop);
160
161     initMatrix(n, A, B, C);
162     cout << "blocked AVX caculation begin...\n";
163     start = clock();

```

```

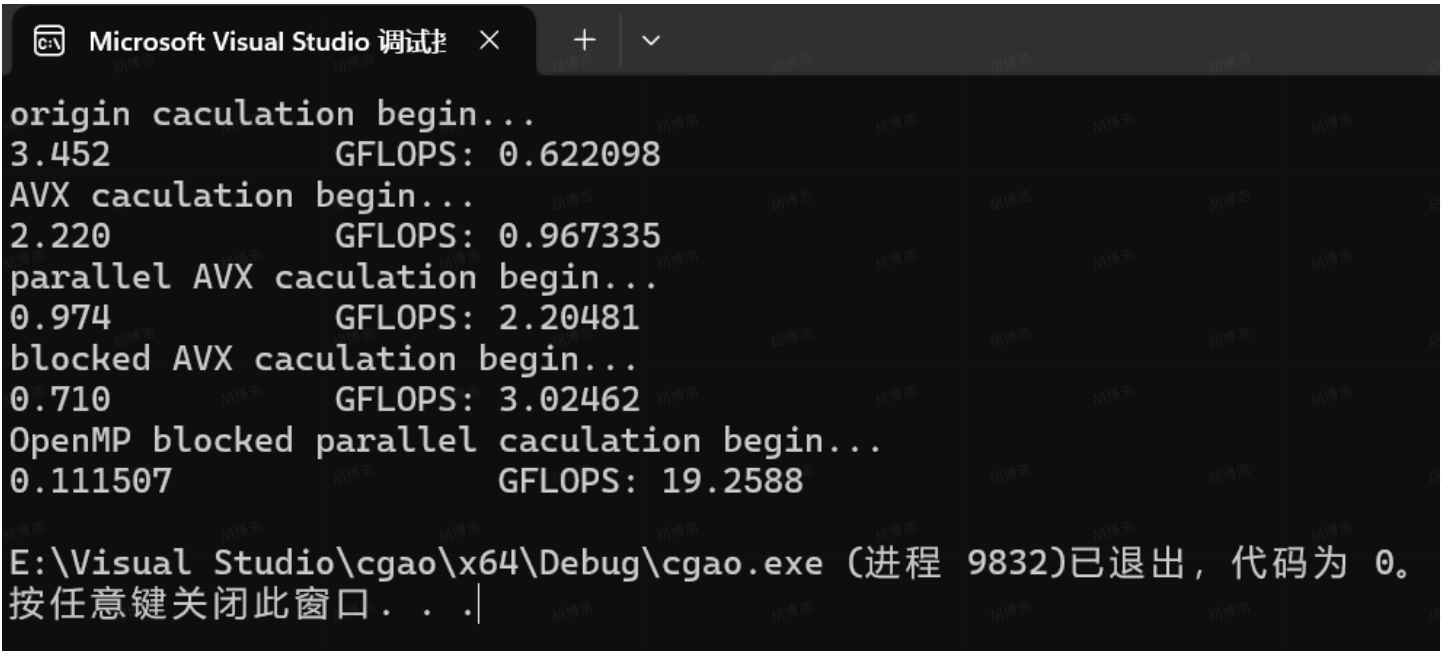
164     block_gemm(n, A, B, C);
165     stop = clock();
166     cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
167     printFlops(n, n, n, start, stop);
168
169     initMatrix(n, A, B, C);
170     cout << "OpenMP blocked parallel caculation begin...\n";
171     double begin = omp_get_wtime();
172     omp_gemm(n, A, B, C);
173     double end = omp_get_wtime();
174     cout << float(end - begin) << "\t\t";
175     REAL_T flops = (2.0 * n * n * n) / 1E9 / (end - begin);
176     cout << "GFLOPS:\t" << flops << endl;
177 }

```

(四) 运行结果

分别设置矩阵大小为1024、2048、3072、4096，运行程序，结果如下：

- n=1024



```

Microsoft Visual Studio 调试 × + v
origin caculation begin...
3.452          GFLOPS: 0.622098
AVX caculation begin...
2.220          GFLOPS: 0.967335
parallel AVX caculation begin...
0.974          GFLOPS: 2.20481
blocked AVX caculation begin...
0.710          GFLOPS: 3.02462
OpenMP blocked parallel caculation begin...
0.111507       GFLOPS: 19.2588
E:\Visual Studio\cgao\x64\Debug\cgao.exe (进程 9832)已退出，代码为 0。
按任意键关闭此窗口. . .|

```

- n=2048

```
Microsoft Visual Studio 调试 × + v
origin caculation begin...
96.607 GFLOPS: 0.177833
AVX caculation begin...
76.754 GFLOPS: 0.22383
parallel AVX caculation begin...
33.117 GFLOPS: 0.518763
blocked AVX caculation begin...
7.95 GFLOPS: 2.42141
OpenMP blocked parallel caculation begin...
0.997177 GFLOPS: 17.2285

E:\Visual Studio\cgao\x64\Debug\cgao.exe (进程 29444)已退出, 代码为 0。
按任意键关闭此窗口. . .|
```

• n=3072

```
Microsoft Visual Studio 调试 × + v
origin caculation begin...
326.769 GFLOPS: 0.177441
AVX caculation begin...
227.471 GFLOPS: 0.254899
parallel AVX caculation begin...
100.746 GFLOPS: 0.575527
blocked AVX caculation begin...
21.406 GFLOPS: 2.70868
OpenMP blocked parallel caculation begin...
2.89798 GFLOPS: 20.0077

E:\Visual Studio\cgao\x64\Debug\cgao.exe (进程 16688)已退出, 代码为 0。
按任意键关闭此窗口. . .|
```

• n=4096

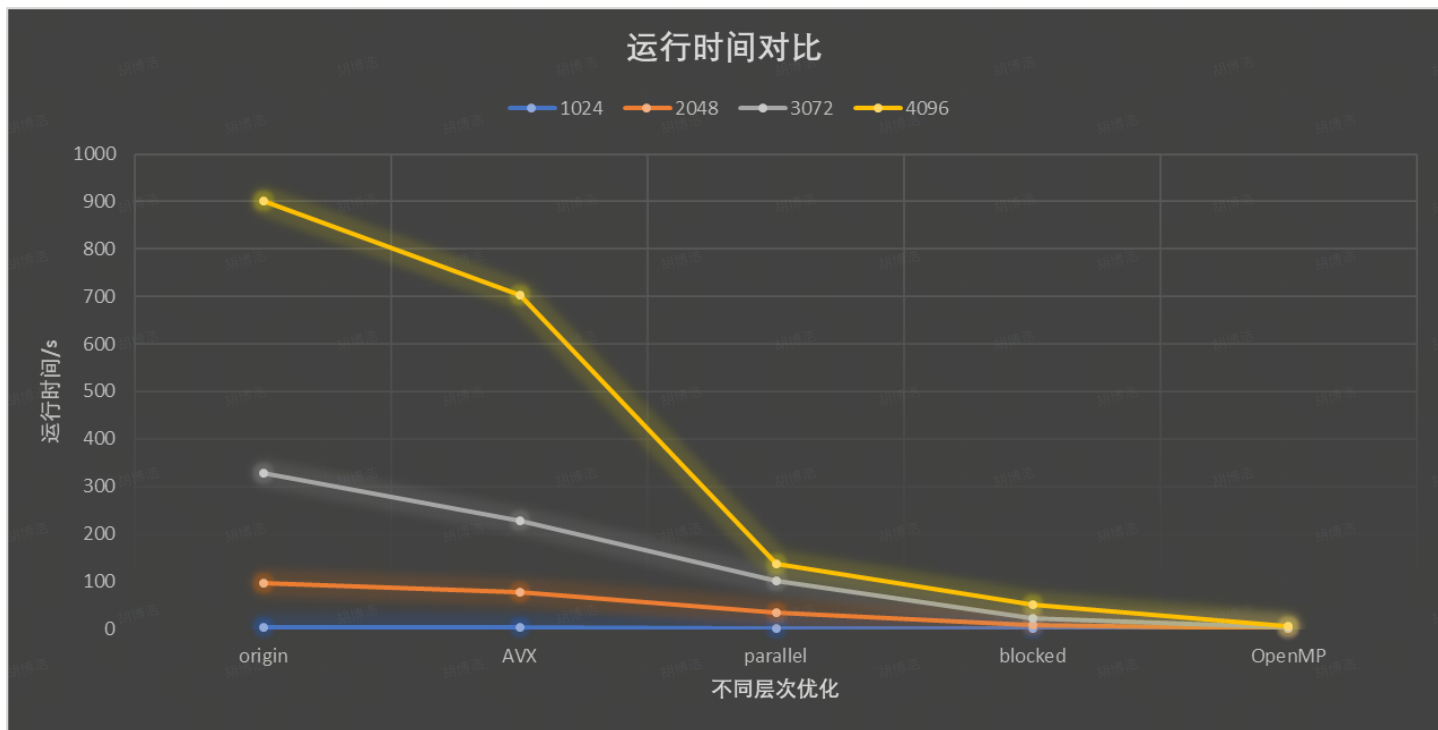
```
Microsoft Visual Studio 调试 × + v
origin caculation begin...
901.782      GFLOPS: 0.152408
AVX caculation begin...
701.945      GFLOPS: 0.195797
parallel AVX caculation begin...
137.761      GFLOPS: 0.997662
blocked AVX caculation begin...
49.85        GFLOPS: 2.80002
OpenMP blocked parallel caculation begin...
6.63841      GFLOPS: 20.7036

E:\Visual Studio\cgao\x64\Debug\cgao.exe (进程 22388)已退出，代码为 0。
按任意键关闭此窗口. . .
```

(五) 优化对比

1.计算耗时:

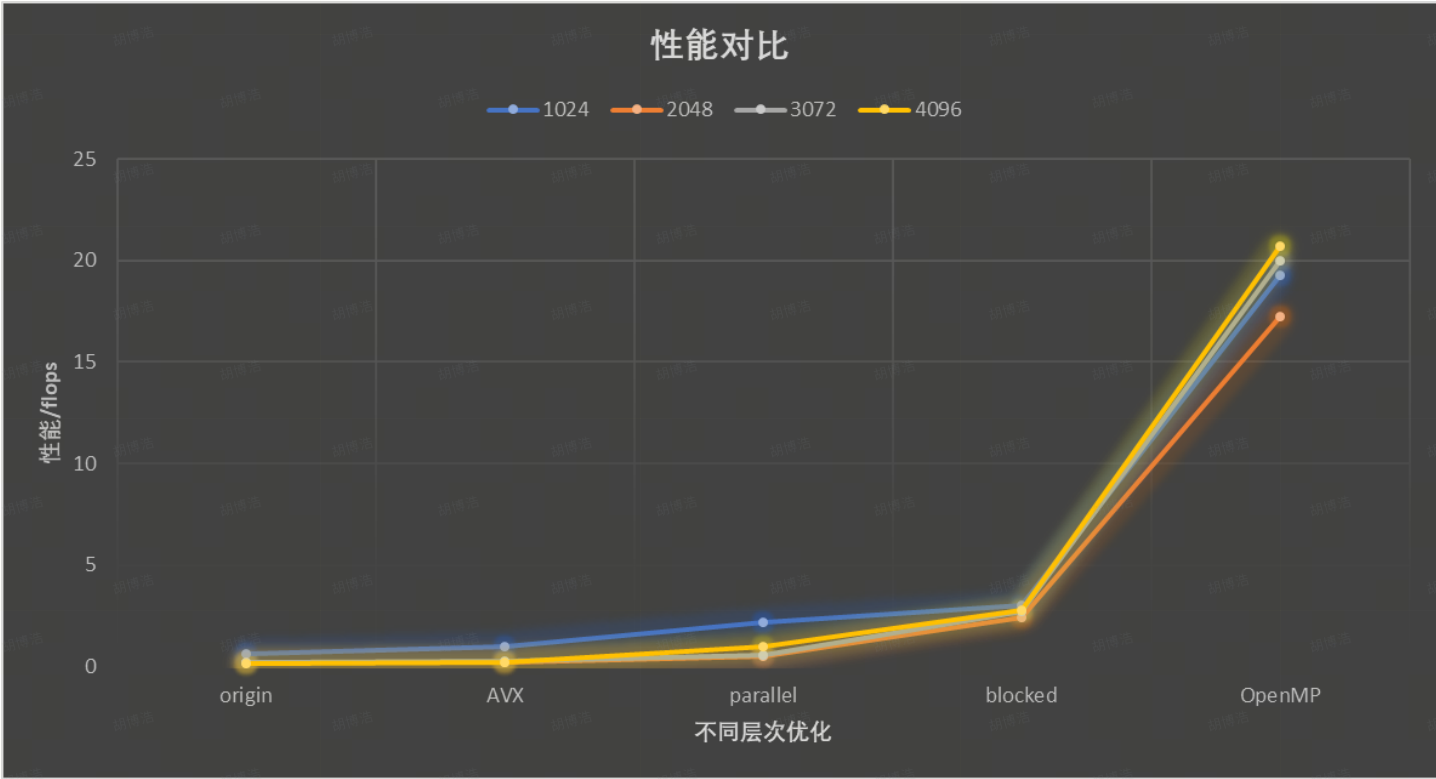
	1024	2048	3072	4096
origin	3.452	96.607	326.769	901.782
AVX	2.22	76.754	227.471	701.945
parallel	0.974	33.117	100.746	137.761
blocked	0.71	7.95	21.406	49.85
OpenMP	0.111507	0.997177	2.89798	6.63841



对于所有的N值，OpenMP并行计算的计算耗时都是最小的，这表明 OpenMP并行计算在处理大规模问题时具有显著的优势。其次，分块计算方式的计算耗时也相对较小，但仍然无法与OpenMP并行计算相比。AVX技术和并行AVX相比原始计算方式都能减少计算耗时，但效果不如分块计算和OpenMP并行计算明显。

2.运行性能：

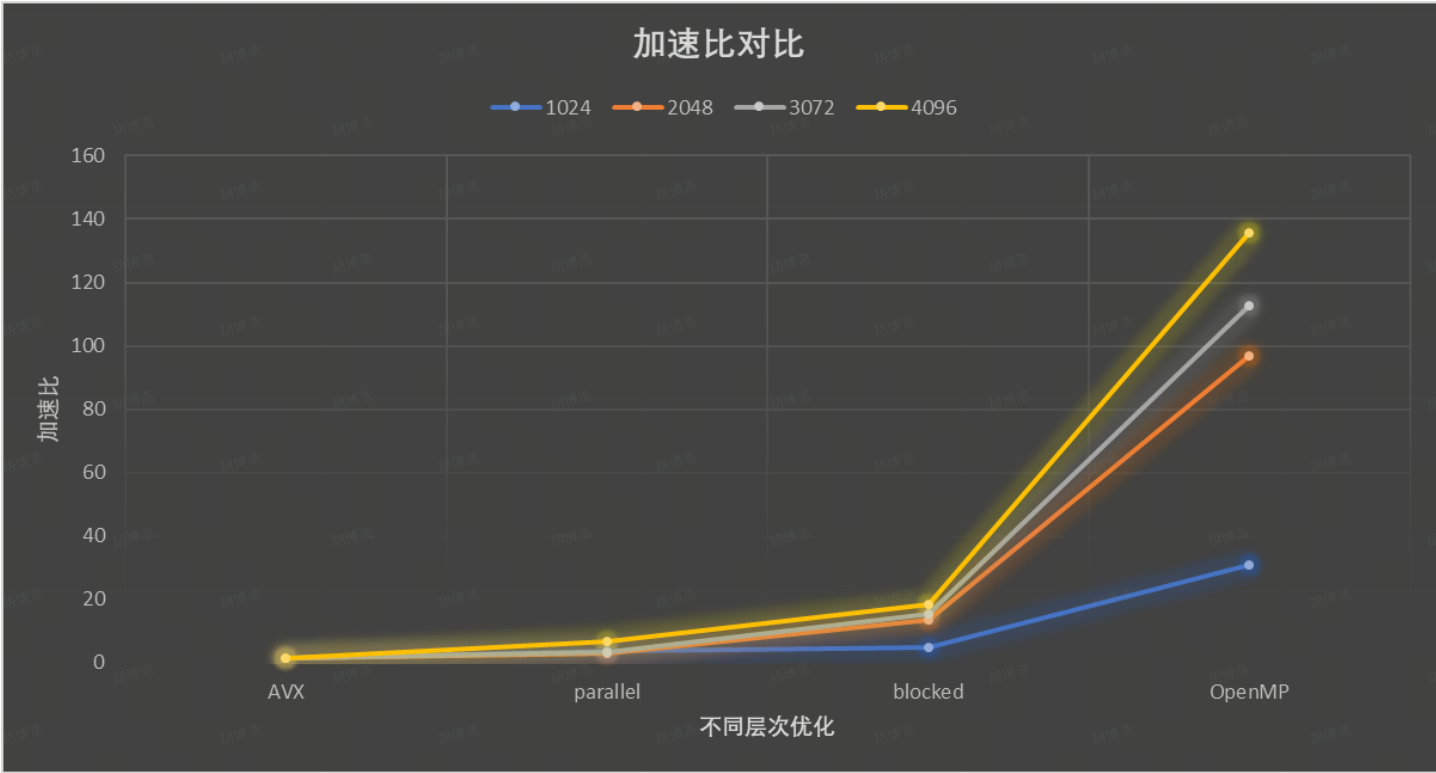
	1024	2048	3072	4096
origin	0.622098	0.177833	0.177441	0.152408
AVX	0.967335	0.22383	0.254899	0.195797
parallel	2.20481	0.518763	0.575527	0.997662
blocked	3.02462	2.42141	2.70868	2.80002
OpenMP	19.2588	17.2285	20.0077	20.7036



对于所有的N值，OpenMP并行计算的运行性能都是最高的，这再次证明了OpenMP并行计算在处理大规模问题时的优势。其次，分块计算的运行性能也相对较高，但仍然无法与OpenMP并行计算相比。AVX技术和并行AVX的运行性能相比原始矩阵乘法有所提高，但提升幅度不如分块计算和OpenMP并行计算大。

3.加速比：

	1024	2048	3072	4096
AVX	1.554956	1.258653	1.436528	1.28469
parallel	3.544152	2.917136	3.243484	6.545995
blocked	4.861967	13.6162	15.26524	18.37187
OpenMP	30.95782	96.88022	112.7569	135.8433



对于所有的N值，OpenMP并行计算的加速比都是最高的，这进一步证明了 OpenMP并行计算在处理大规模问题时的优势。其次，分块计算的加速比也相对较高，但仍然无法与OpenMP并行计算相比。AVX技术和并行AVX的加速比相比原始矩阵乘法有所提高，但提升幅度不如分块计算和OpenMP并行计算大。

• 总结

从实验数据中可以看出，所有的优化策略都成功地减少了计算的耗时和提高了运行性能，相比于原始未优化的计算方式，优化策略均能获得更好的计算效率。在所有优化策略中，使用OpenMP进行并行计算的方法表现出了最优秀的性能。在N=4096的时候，加速比更是达到了135！

尽管所有的优化策略都能显著提高计算性能，但它们在不同规模下的表现却各有不同。例如，AVX在N=1024的计算中加速比最大，在较大规模的计算中效率提升却并未如此显著。

总的来说，通过对比我们可以看出，OpenMP并行计算的优化策略在处理大规模问题时，优势更为明显。虽然其他的优化策略如AVX, PAVX, Block各有优势，但相比使用OpenMP并行计算的策略，它们的效果都稍显逊色。

(六) 遇到的问题和解决方式

1.开启OpenMp的时候报错：错误 C2338 C++/CLI、C++/CX 或 OpenMP 不支持两阶段名称查找

解决方法：在属性菜单中找到C/C++ --> 语言 -->符合模式下拉菜单中选择"否"

四、Taishan服务器上实验

(一) 实验要求

在Taishan服务器上使用vim+gcc编程环境，要求如下：

- 在Taishan服务器上完成，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。

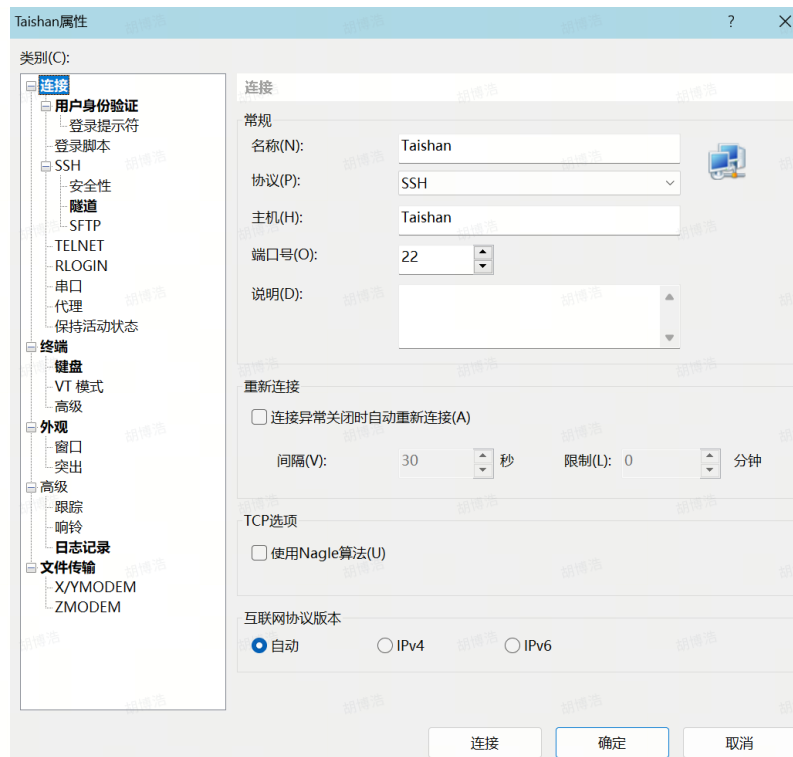
(二) 实验环境

在本次实验中，我使用XShell7远程连接Taishan服务器，利用vim和gcc进行代码的编写和编译

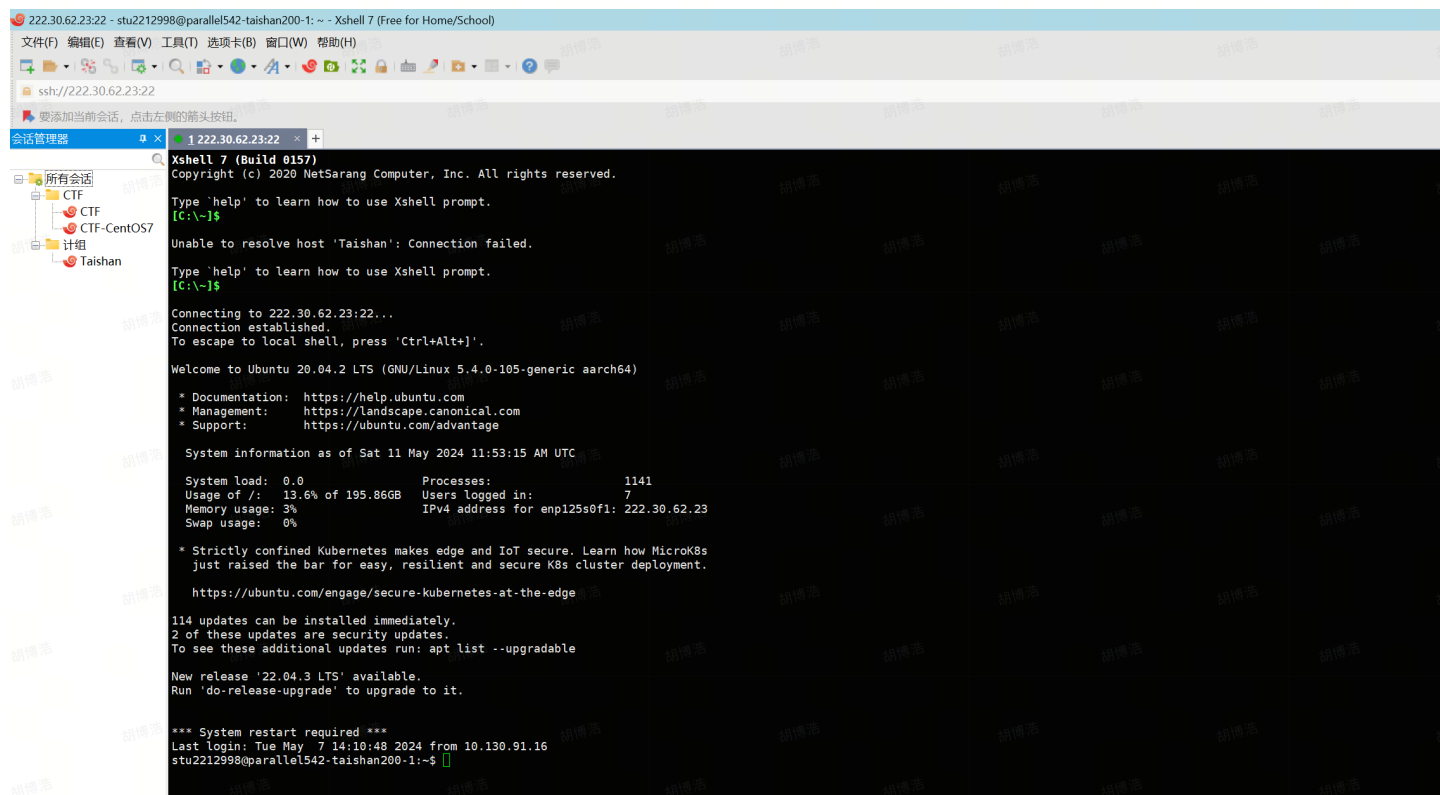
(三) 实验步骤

1.连接服务器

- (1) 打开XShell7，新建会话：输入名称、端口号

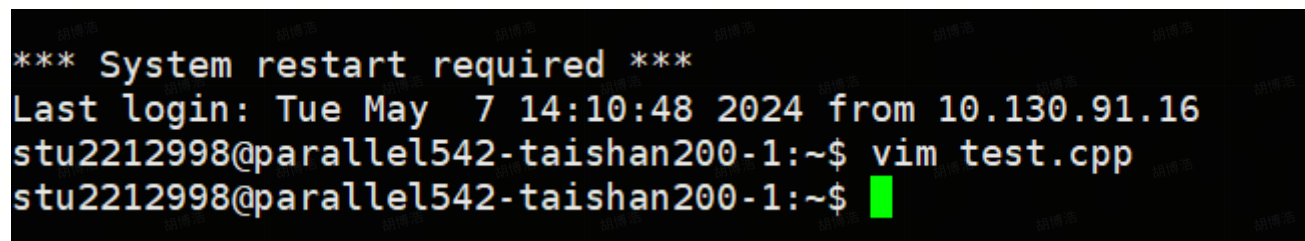


(2) 连接会话Taishan，依次输入IP：222.30.62.23，用户名：stu2212998，密码：123456



(3) 创建cpp文件：vim test.cpp

进入vim编辑器，然后就可以敲代码了



2.代码编写

由于使用AVX 库进行子字优化在 Taishan 服务器上的软件包环境不好配置，这里不进行子字并行优化，改动如下：

- 将子字并行函数avx_dgemm () 删除
- 将pavx_dgemm () 、do_block () 函数中相应部分修改
- 删除main函数中对avx_dgemm () 的测试

```
1  #include<iostream>
2  #include<time.h>
3  #include<omp.h>
4
5  using namespace std;
6  #define REAL_T double
7
8  void printFlops(int A_height, int B_width, int B_height, clock_t start, clock_t
    stop) {
9      REAL_T flops = (2.0 * A_height * B_width * B_height) / 1E9 / ((stop -
    start) / (CLOCKS_PER_SEC * 1.0));
10     cout << "GFLOPS:\t" << flops << endl;
11 }
12
13 void initMatrix(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
14     for (int i = 0; i < n; ++i)
15         for (int j = 0; j < n; ++j) {
16             A[i + j * n] = (i + j + (i * j) % 100) % 100;
17             B[i + j * n] = ((i - j) * (i - j) + (i * j) % 200) %
    100;
18             C[i + j * n] = 0;
19         }
20 }
21
22 void dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
23     for (int i = 0; i < n; ++i)
24         for (int j = 0; j < n; ++j) {
25             REAL_T cij = C[i + j * n];
26             for (int k = 0; k < n; k++) {
27                 cij += A[i + k * n] * B[k + j * n];
28             }
29             C[i + j * n] = cij;
30         }
31 }
32
33 #define UNROLL (4)
34 void pavx_dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
```

```

35     for (int i = 0; i < n; i += UNROLL)
36         for (int j = 0; j < n; ++j) {
37             REAL_T c1 = C[i + j * n];
38             REAL_T c2 = C[i + 1 + j * n];
39             REAL_T c3 = C[i + 2 + j * n];
40             REAL_T c4 = C[i + 3 + j * n];
41             for (int k = 0; k < n; k++) {
42                 c1 += A[i + k * n] * B[k + j * n];
43                 c2 += A[i + 1 + k * n] * B[k + j * n];
44                 c3 += A[i + 2 + k * n] * B[k + j * n];
45                 c4 += A[i + 3 + k * n] * B[k + j * n];
46             }
47             C[i + j * n] = c1;
48             C[i + 1 + j * n] = c2;
49             C[i + 2 + j * n] = c3;
50             C[i + 3 + j * n] = c4;
51         }
52     }
53
54 #define BLOCKSIZE (32)
55 void do_block(int n, int si, int sj, int sk, REAL_T* A, REAL_T* B, REAL_T* C) {
56     for (int i = si; i < si + BLOCKSIZE; i += 4)
57         for (int j = sj; j < sj + BLOCKSIZE; ++j) {
58             REAL_T c1 = C[i + j * n];
59             REAL_T c2 = C[i + 1 + j * n];
60             REAL_T c3 = C[i + 2 + j * n];
61             REAL_T c4 = C[i + 3 + j * n];
62
63             for (int k = sk; k < sk + BLOCKSIZE; ++k) {
64                 c1 += A[i + k * n] * B[k + j * n];
65                 c2 += A[i + 1 + k * n] * B[k + j * n];
66                 c3 += A[i + 2 + k * n] * B[k + j * n];
67                 c4 += A[i + 3 + k * n] * B[k + j * n];
68             }
69
70             C[i + j * n] = c1;
71             C[i + 1 + j * n] = c2;
72             C[i + 2 + j * n] = c3;
73             C[i + 3 + j * n] = c4;
74         }
75     }
76
77
78 void block_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
79     for (int sj = 0; sj < n; sj += BLOCKSIZE)
80         for (int si = 0; si < n; si += BLOCKSIZE)
81             for (int sk = 0; sk < n; sk += BLOCKSIZE)

```

```

82         do_block(n, si, sj, sk, A, B, C);
83     }
84
85 void omp_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
86     #pragma omp parallel for
87         for (int sj = 0; sj < n; sj += BLOCKSIZE)
88             for (int si = 0; si < n; si += BLOCKSIZE)
89                 for (int sk = 0; sk < n; sk += BLOCKSIZE)
90                     do_block(n, si, sj, sk, A, B, C);
91 }
92
93 int main()
94 {
95     REAL_T* A, * B, * C;
96     clock_t start, stop;
97     int n = 1024;
98     A = new REAL_T[n * n];
99     B = new REAL_T[n * n];
100    C = new REAL_T[n * n];
101    initMatrix(n, A, B, C);
102
103    cout << "origin caculation begin...\n";
104    start = clock();
105    dgemm(n, A, B, C);
106    stop = clock();
107    cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
108    printFlops(n, n, n, start, stop);
109
110    initMatrix(n, A, B, C);
111    cout << "parallel caculation begin...\n";
112    start = clock();
113    pax_dgemm(n, A, B, C);
114    stop = clock();
115    cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
116    printFlops(n, n, n, start, stop);
117
118    initMatrix(n, A, B, C);
119    cout << "blocked parallel caculation begin...\n";
120    start = clock();
121    block_gemm(n, A, B, C);
122    stop = clock();
123    cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
124    printFlops(n, n, n, start, stop);
125

```



```

126     initMatrix(n, A, B, C);
127     cout << "OpenMP blocked parallel caculation begin...\n";
128     double begin = omp_get_wtime();
129     omp_gemm(n, A, B, C);
130     double end = omp_get_wtime();
131     cout << float(end - begin) << "\t\t";
132     REAL_T flops = (2.0 * n * n * n) / 1E9 / (end - begin);
133     cout << "GFLOPS:\t" << flops << endl;
134     return 1;
135
136 }

```

3.编译运行代码

保存好代码并退出vim，输入以下命令对代码进行编译

```
1 g++ -fopenmp test.cpp -o test
```

其中，-fopenmp是一个编译选项，用于启用OpenMP并行化支持。

```

*** System restart required ***
Last login: Tue May  7 14:10:48 2024 from 10.130.91.16
stu2212998@parallel542-taishan200-1:~$ vim test.cpp
stu2212998@parallel542-taishan200-1:~$ g++ -fopenmp test.cpp -o test
stu2212998@parallel542-taishan200-1:~$ █

```

之后输入./test即可运行代码

(四) 运行结果

选取矩阵规模为1024、2048、3072和4096进行测试，结果如下：

- n=1024

```

stu2212998@parallel542-taishan200-1:~$ vim test.cpp
stu2212998@parallel542-taishan200-1:~$ g++ -fopenmp test.cpp -o test
stu2212998@parallel542-taishan200-1:~$ ./test
origin caculation begin...
17.873847          GFLOPS: 0.120147
parallel caculation begin...
13.9828           GFLOPS: 0.165066
blocked parallel caculation begin...
5.506051          GFLOPS: 0.390022
OpenMP blocked parallel caculation begin...
0.192068          GFLOPS: 11.1809
stu2212998@parallel542-taishan200-1:~$ █

```


- n=2048

```
stu2212998@parallel542-taishan200-1:~$ vim test.cpp
stu2212998@parallel542-taishan200-1:~$ g++ -fopenmp test.cpp -o test
stu2212998@parallel542-taishan200-1:~$ ./test
origin caculation begin...
219.120609          GFLOPS: 0.0784037
parallel caculation begin...
162.392894          GFLOPS: 0.105792
blocked parallel caculation begin...
45.291255           GFLOPS: 0.37932
OpenMP blocked parallel caculation begin...
0.800106            GFLOPS: 21.472
stu2212998@parallel542-taishan200-1:~$ █
```

- n=3072

```
stu2212998@parallel542-taishan200-1:~$ vim test.cpp
stu2212998@parallel542-taishan200-1:~$ g++ -fopenmp test.cpp -o test
stu2212998@parallel542-taishan200-1:~$ ./test
origin caculation begin...
816.392654          GFLOPS: 0.0710223
parallel caculation begin...
572.932818          GFLOPS: 0.101202
blocked parallel caculation begin...
150.902220          GFLOPS: 0.384236
OpenMP blocked parallel caculation begin...
1.84289             GFLOPS: 31.4626
stu2212998@parallel542-taishan200-1:~$ █
```

- n=4096

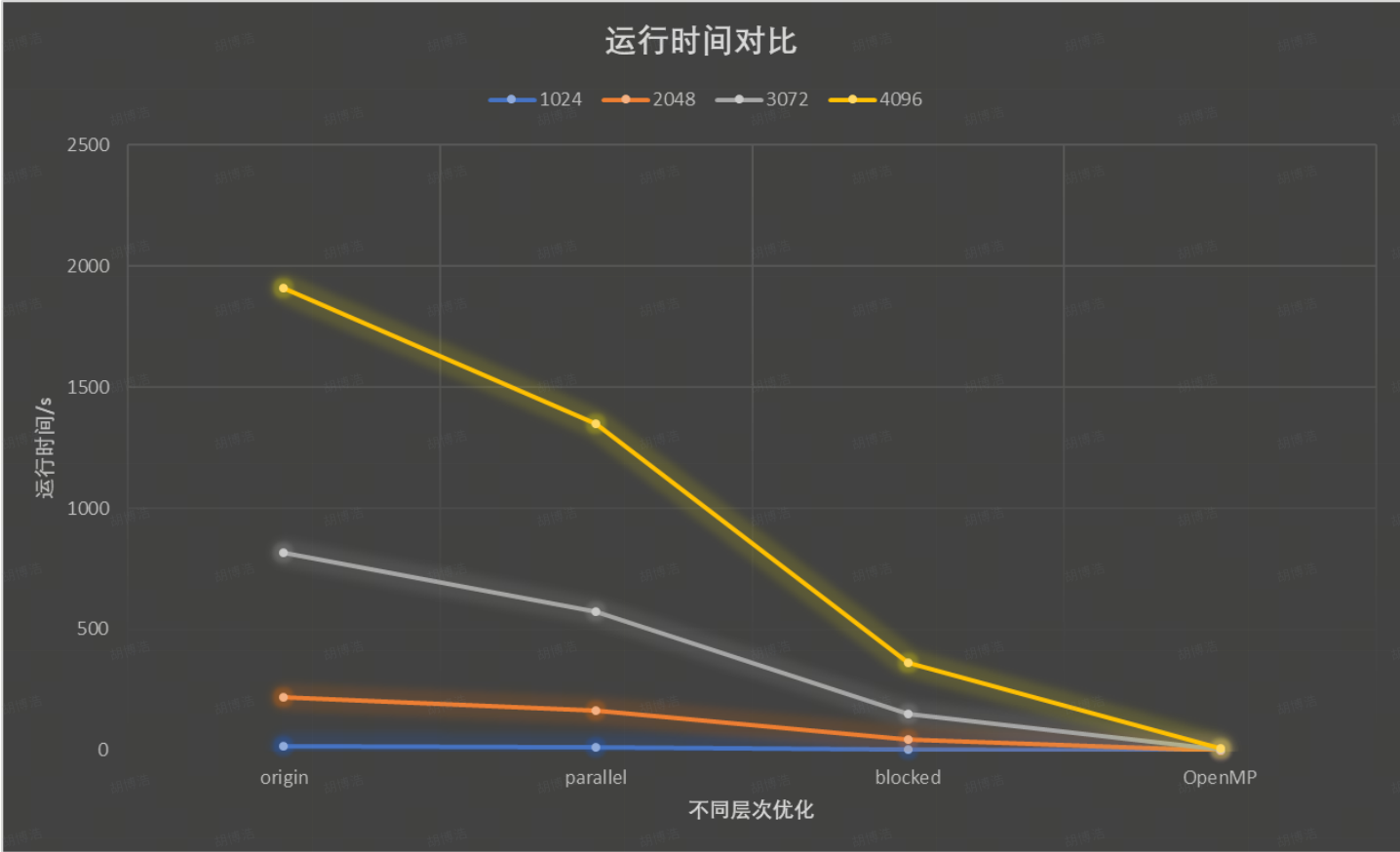
```
stu2212998@parallel542-taishan200-1:~$ ./test
origin caculation begin...
1923.446522         GFLOPS: 0.0714545
parallel caculation begin...
1442.910122         GFLOPS: 0.0952512
blocked parallel caculation begin...
369.398457          GFLOPS: 0.372062
OpenMP blocked parallel caculation begin...
5.90004             GFLOPS: 23.2946
stu2212998@parallel542-taishan200-1:~$ █
```

```
stu2212998@parallel542-taishan200-1:~$ vim test.cpp
stu2212998@parallel542-taishan200-1:~$ g++ -fopenmp test.cpp -o test
stu2212998@parallel542-taishan200-1:~$ ./test
origin caculation begin...
1909.937733          GFLOPS: 0.0719599
parallel caculation begin...
1348.315893          GFLOPS: 0.101934
blocked parallel caculation begin...
361.742676           GFLOPS: 0.379936
OpenMP blocked parallel caculation begin...
6.39239              GFLOPS: 21.5004
stu2212998@parallel542-taishan200-1:~$
```

(五) 优化对比

1.计算耗时：

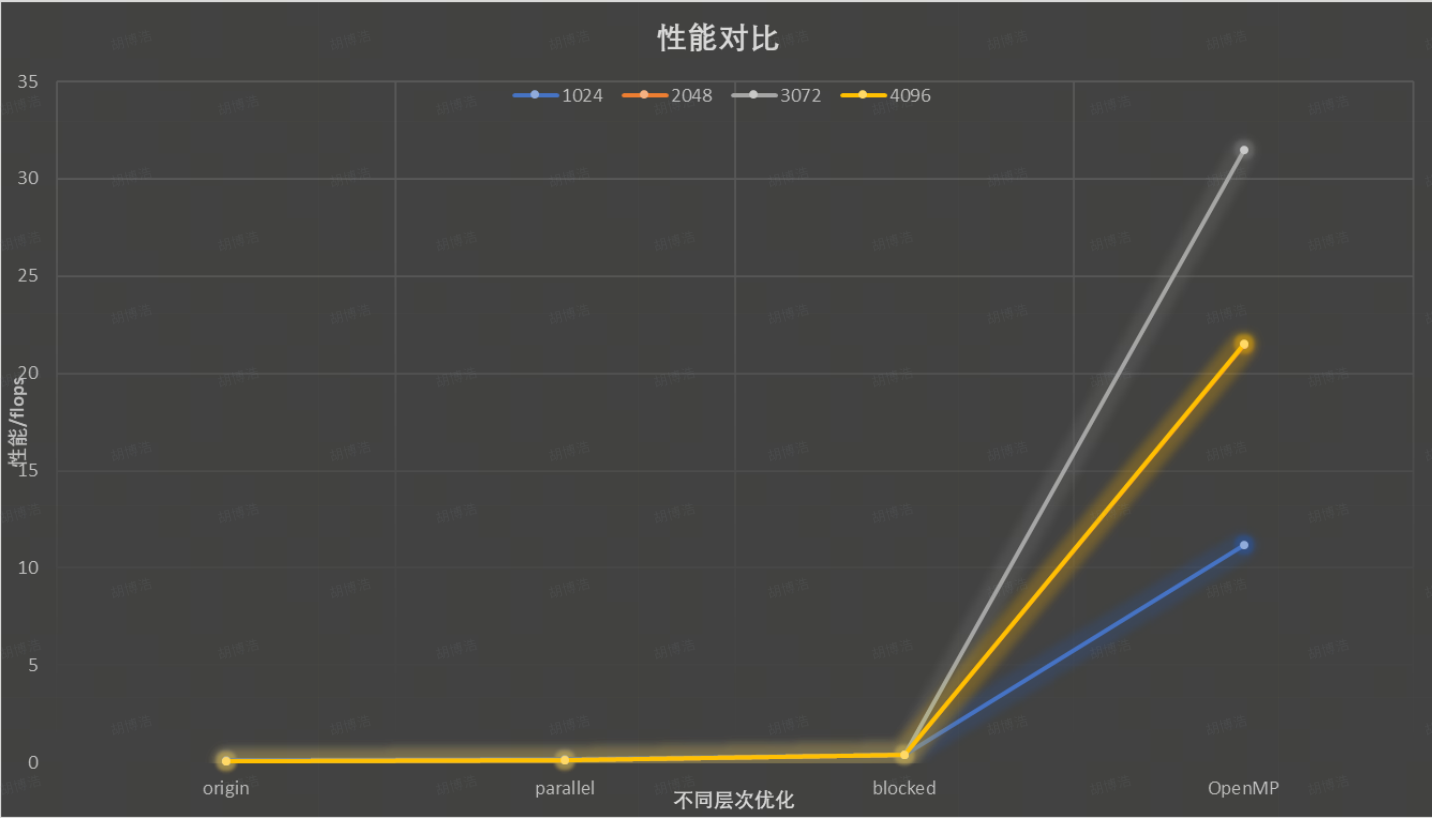
	1024	2048	3072	4096
origin	17.873847	219.120609	816.392654	1909.937733
parallel	13.9828	162.392894	572.932818	1348.315893
blocked	5.506051	45.291255	150.90222	361.742676
OpenMP	0.192068	0.800106	1.84289	6.39239



随着矩阵规模的增大，原始计算方法的耗时呈指数增长。不同层次的优化均有效降低了运行时间，特别是OpenMP方法，在大规模矩阵乘法中表现出更好的计算性能，耗时明显更短。

2.运行性能：

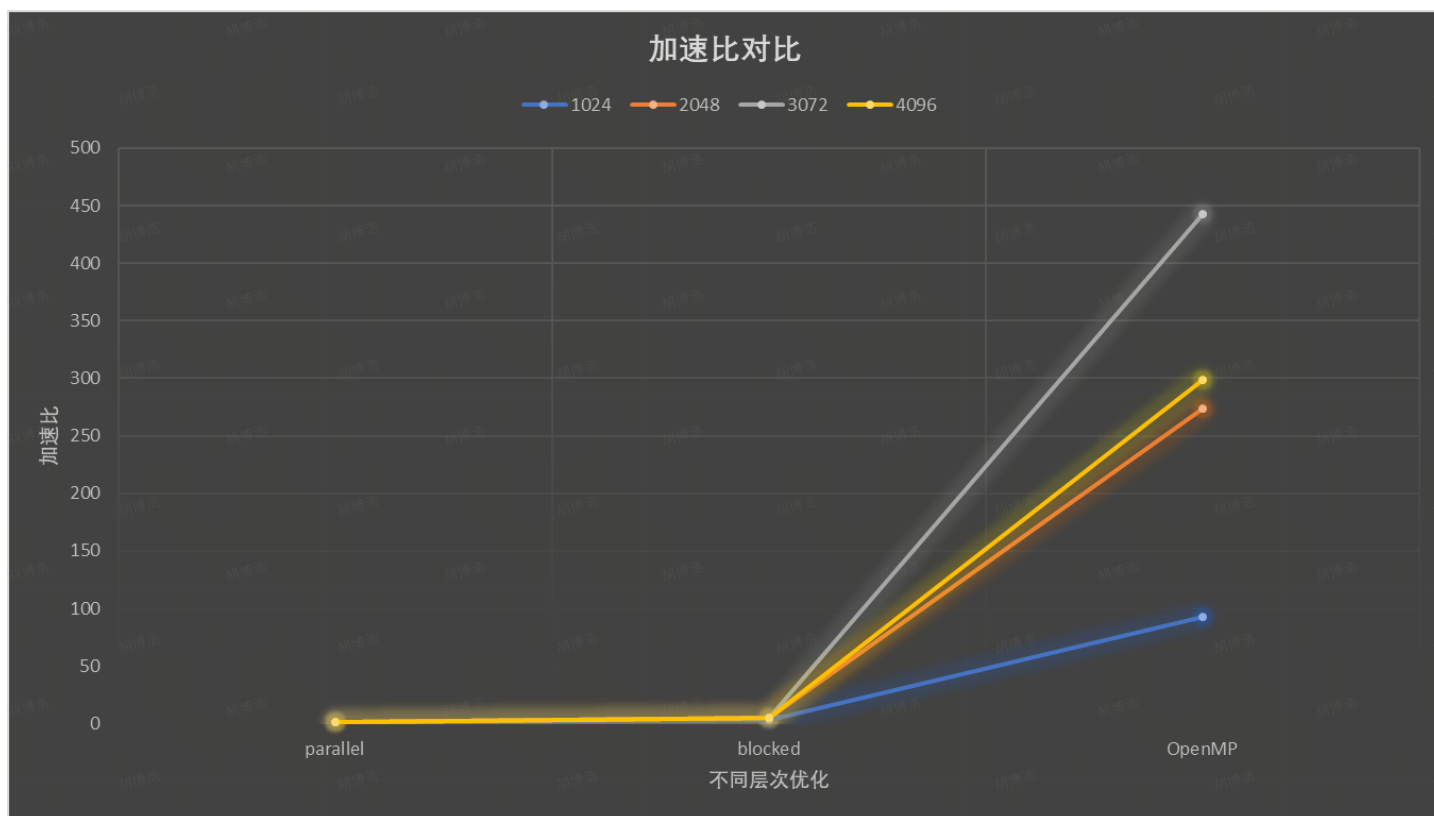
	1024	2048	3072	4096
origin	0.120147	0.078404	0.071022	0.07196
parallel	0.165066	0.105792	0.101202	0.101934
blocked	0.390022	0.37932	0.384236	0.379936
OpenMP	11.1809	21.472	31.4626	21.5004



原始计算方法的运行性能相对较低，而经过优化后的计算方法运行性能显著提升。OpenMP方法在不同规模下都获得了较高的GFLOPS值，表明其具有更优的运行性能。

3.加速比：

	1024	2048	3072	4096
parallel	1.373867	1.349324	1.424933	1.416539
blocked	3.246207	4.838037	5.410075	5.279829
OpenMP	93.06017	273.8646	442.9961	298.7831



经过优化后的并行计算方法相较于原始计算方法，实现了较好的加速效果。OpenMP方法的加速效果最好，加速比最高可以达到442，可见taishan服务器计算核心优势。

• 总结

从实验数据中可以看出，所有的优化策略都成功地减少了计算的耗时和提高了运行性能，相比于原始未优化的计算方式，优化策略均能获得更好的计算效率。在所有优化策略中，使用OpenMP进行并行计算的方法表现出了最优秀性能。在N=3072的时候，加速比更是达到了442！

尽管所有的优化策略都能显著提高计算性能，但它们在规模下的表现却各有不同。比如说OpenMP在较大规模的时候比小规模的时候高。

总的来说，通过对比我们可以看出，OpenMP并行计算的优化策略过利用多线程并行计算的特点，在处理大规模问题时，优势更为明显。虽然其他的优化策略如parallel, block各有优势，但相比使用OpenMP并行计算的策略，它们的效果都稍显逊色。

（六）遇到的问题 and 解决方式

1.代码编译为可执行程序的时候，一开始使用的是gcc，报错不能识别头文件

解决方法：更换为g++

2.一开始没有开启多核并行，导致运行结果不理想

解决方法：编译的时候添加-fopenmp，启用OpenMP并行化支持

五、不同电脑上的运行差异

1.计算耗时对比：

- 在本机上，使用AVX指令集的并行计算方法和分块计算方法的计算耗时相对于原始方法都有显著减少。尤其是OpenMp在各个矩阵规模上都表现出较低的计算耗时。
- 在Taishan服务器上，不优化的计算方法耗时较长，但最后OpenMP后、由于多核的优势，计算时间和本机上相差无几。

2.运行性能对比：

- 在本机上，使用AVX指令集的并行计算方法和分块计算方法相较于原始方法都取得了显著的运行性能提升。特别是在较大矩阵规模上，性能提升更为明显。
- 在Taishan服务器上，原始的计算方法运行性能低下，但由于服务器多核的优势明显，导致运行性能最后得到了显著提高，追上了本机上的运行性能。

3.加速比对比：

同理，从对比中可以看出服务器多核的优势，将加速比拉升了两个数量级。

4.本机和服务器上运行差异的原因：

- 服务器架构和资源限制：服务器通常设计用于处理大量请求和并发任务，因此在硬件设计上可能更注重稳定性和可靠性，而不是单个任务的计算性能。服务器可能采用较低主频的处理器和较小的缓存容量，导致相对较低的运行性能。
- 并行计算支持：非服务器计算机通常处理器核数量有限。而服务器核的数量较多，可以充分利用多核处理器的并行计算能力，从而大大提高运算效率。
- 内存和缓存：非服务器计算机通常配置较大容量的内存和高速缓存，能够更好地满足大规模矩阵计算的需求，减少数据访问延迟。而服务器可能配置相对较小的内存和缓存，无法提供与非服务器计算机相同的性能。
- 系统负载和资源竞争：服务器可能同时运行多个任务，存在系统负载和资源竞争的情况，导致矩阵乘法计算任务无法充分占用服务器资源，进而影响计算性能和加速比。

综上所述，本机相较于Taishan服务器具有更好的运行性能和计算能力，能够更充分地利用多核处理器和AVX指令集的优势，从而实现更快的计算速度和更高的运行性能。而Taishan服务器由于各种因素，限制了性能的发挥，导致计算耗时和运行性能相对较低。

六、总结感想

在进行矩阵乘法优化实验的过程中，我获得了许多宝贵的经验和体会。

- 1.从硬件上优化算法的重要性：实验验证了优化算法在提高计算性能和效率方面的重要性。通过采用AVX指令集、分块技术和OpenMP并行化等优化算法，我们显著提高了矩阵乘法的性能和效率。由此，我们设计程序时不能仅仅重视算法的复杂度，还要注意其硬件设备的适配！
- 2.矩阵规模对性能的影响：实验结果显示，随着矩阵规模的增加，原始算法的计算耗时和运行性能呈指数增长。这表明在处理大规模矩阵时，优化算法的应用尤为关键，能够显著提升计算效率。

3.硬件环境的影响：我注意到实验结果在不同硬件环境下的表现不同。在非服务器环境下，优化算法的运行性能和加速比更高，这归因于非服务器环境通常具备更强的硬件配置，能够更好地支持并行计算和高性能计算任务。

总的来说，这次矩阵乘法优化实验为我提供了宝贵的学习经验。我不仅掌握了优化算法的应用技巧，还深入地理解了矩阵乘法的优化原理和方法，并将其应用到实际的编程环境中。这对我今后优化代码、提高程序运行效率有很大的启发。