组成原理课程矩阵乘法实报告

实验名称：矩阵乘法

学号：2213041 姓名：李雅帆 班次：李涛老师

一、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。

二、实验内容说明

1.个人 PC 电脑实验要求如下：

（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。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。

2.在 Taishan 服务器上使用 vim+gcc 编程环境，要求如下：

（1）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。

（2）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。

（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。

3.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。

三、实验步骤

1.在所给的代码中，有几个不同的层次和规模：

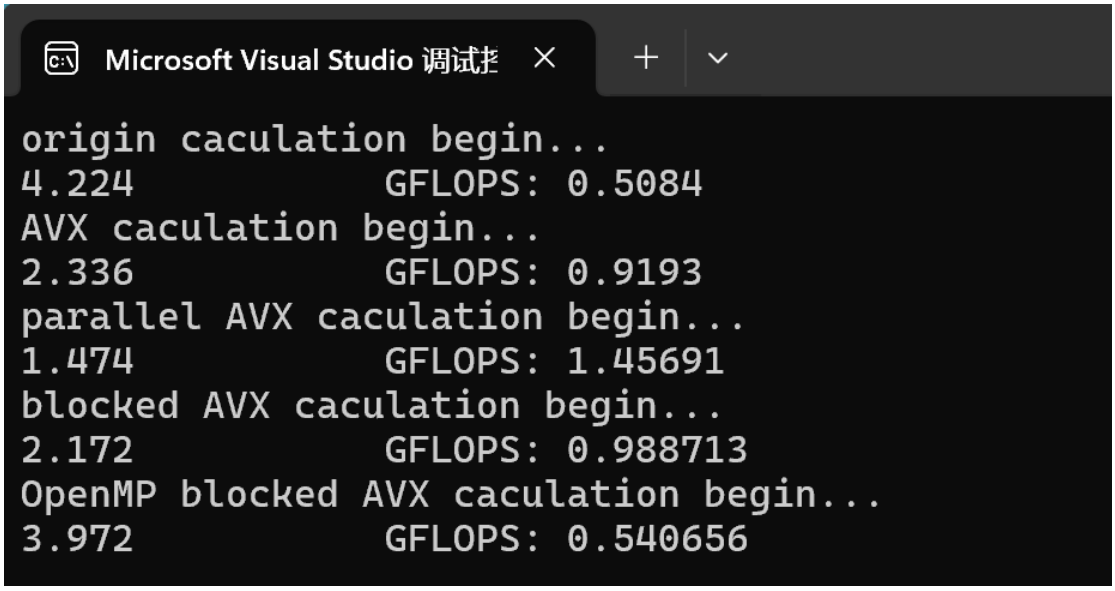
(1) 基本矩阵乘法（dgemm)：这是最简单的矩阵乘法实现，没有任何优化。它是其他优化方法的基准。

(2) AVX 优化矩阵乘法(avx_dgemm)：这个版本利用了 AVX(Advanced Vector Extensions)指令集，通过并行计算多个双精度浮点数，提高了计算性能。

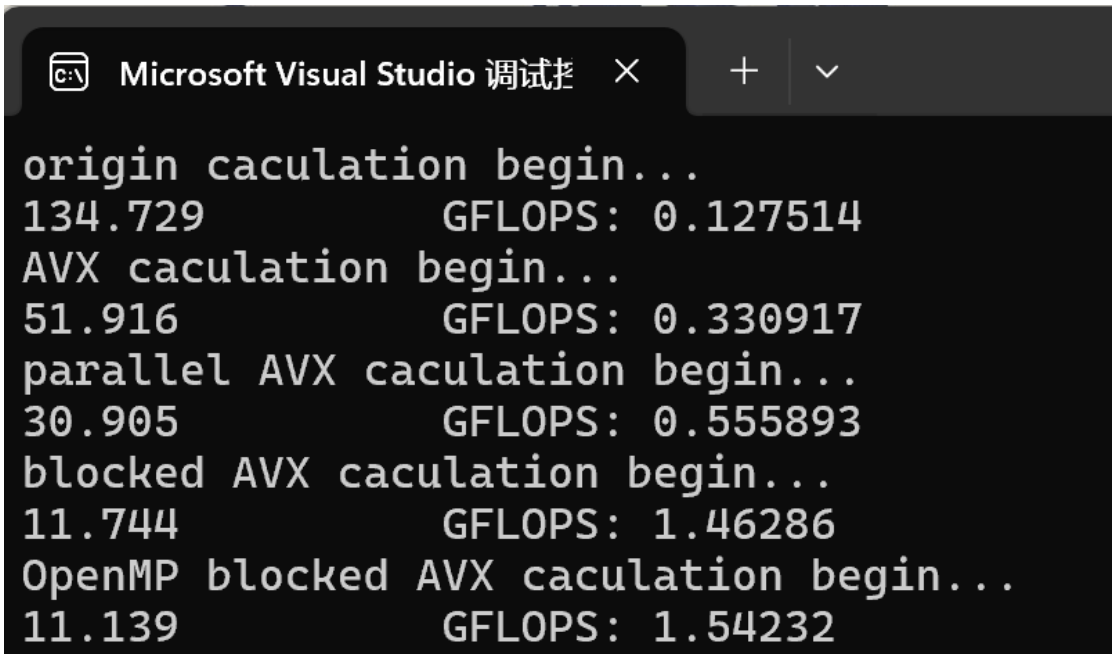
（3）并行 AVX 优化矩阵乘法（pavx_dgemm）：这个版本在 AVX 优化的基础上，进一步利用了并行化，将矩阵乘法的计算任务分配到多个线程中执行，以提高整体计算速度。
（4）块状优化矩阵乘法（block_gemm）：这个版本将矩阵乘法任务划分为较小的块，通过优化内存访问模式和缓存利用，减少了缓存未命中的次数，提高了计算效率。
（5）使用 OpenMP 并行化的块状 AVX 优化矩阵乘法（omp_gemm）：这个版本在块状优化的基础上，使用了 OpenMP 来实现并行化，进一步提高了多核处理器的利用率。

2.在个人 PC 电脑端 visual studio 中分别在 1024、2048、3072、4096 的矩阵规模下进行测试，测试结果如下：

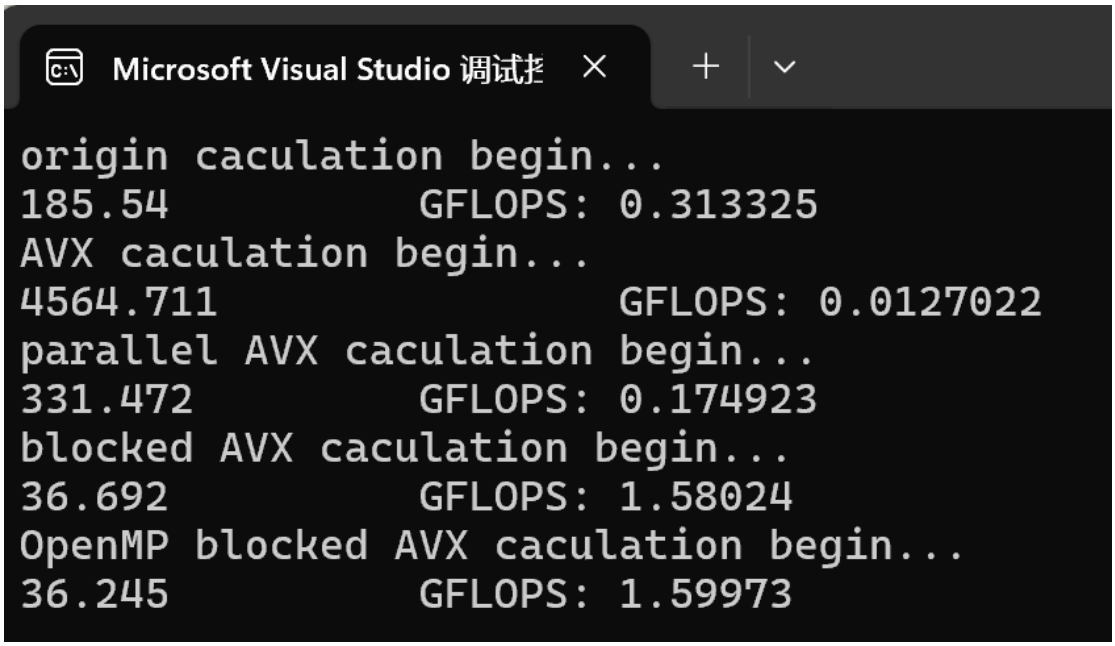
（1）n=1024;



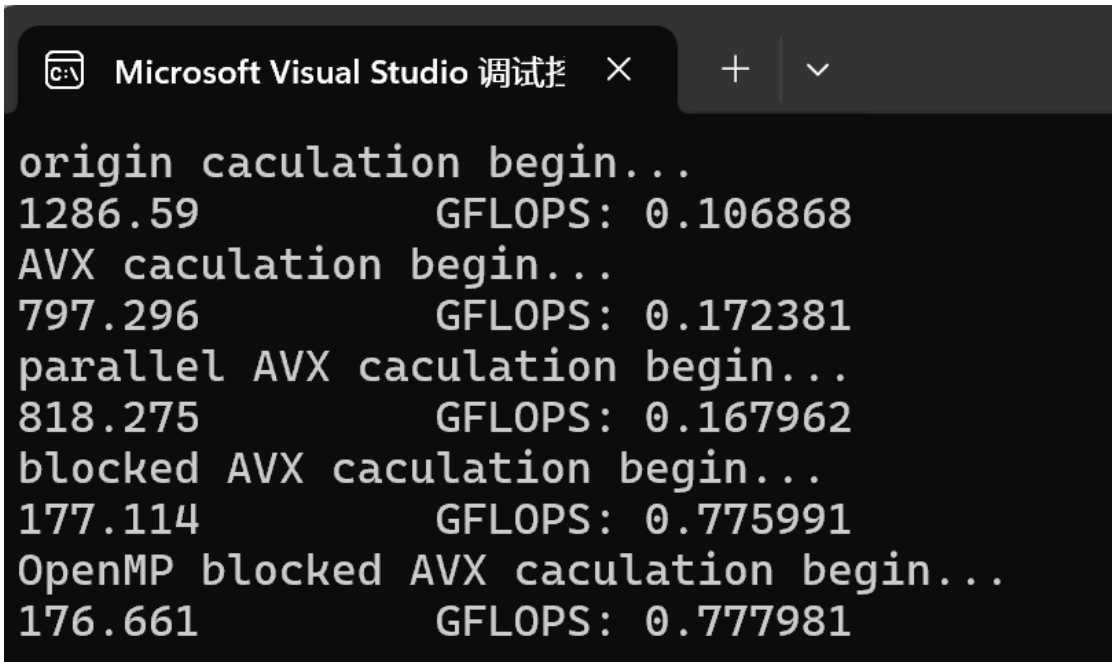
（2）n=2048;



(3)n=3072;



（4）n=4096;



随着矩阵规模的增加，最简单的计算方法的效率会逐渐下降，可能是因为算力资源的饱和。

随着矩阵规模加大，并行的优化效率保持稳定，分块的优化效率有所上升，其他的优化效率基本都有所下降。这是因为算力资源饱和，单核压力较大，导致计算速度下降，而并行和分块就很好的解决了这个问题，优化的效果是最好的

3.在泰山服务器上进行测试。

由于泰山服务器上没有 iostream 和_mm256，于是实现余下的几个包。修改后的代码如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <immintrin.h> // For AVX instructions

#define REAL_T double
#define UNROLL (4)
#define BLOCKSIZE (32)

void printFlops(int A_height, int B_width, int B_height, double start, double stop)
{
double flops = (2.0 * A_height * B_width * B_height) / 1E9 / (stop - start);
printf("GFLOPS:\t%f\n", flops);
}

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 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)
for (int j = sj; j < sj + BLOCKSIZE; ++j)
for (int k = sk; k < sk + BLOCKSIZE; ++k)
for (int ii = 0; ii < UNROLL; ++ii)
C[i + (j + ii) * n] += A[i + (k + ii) * n] * B[k + j * n];
}

void omp_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
#pragma omp parallel for collapse(2)
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);
}

int main()
{
REAL_T* A, * B, * C;
double start, stop;
int n = 1024;
A = (REAL_T*)malloc(n * n * sizeof(REAL_T));
B = (REAL_T*)malloc(n * n * sizeof(REAL_T));
C = (REAL_T*)malloc(n * n * sizeof(REAL_T));
initMatrix(n, A, B, C);

printf("Original calculation begins...\n");
start = omp_get_wtime();
dgemm(n, A, B, C);
stop = omp_get_wtime();
printf("%f\n", stop - start);
printFlops(n, n, n, start, stop);

initMatrix(n, A, B, C);
printf("OpenMP blocked AVX calculation begins...\n");
start = omp_get_wtime();
omp_gemm(n, A, B, C);
stop = omp_get_wtime();
```

```
printf("%f\n", stop - start);
printFlops(n, n, n, start, stop);

free(A);
free(B);
free(C);
return 0;
}
```

在泰山服务器上分别在 1024、2048、3072、4096 的矩阵规模下进行测试，测试结果如下:

(1)n=1024;

```
origin caculation begin...
SECOND: 16.295398                GFLOPS: 0.131785
pavx caculation begin...
SECOND: 3.265413                 GFLOPS: 0.657645
block caculation begin...
SECOND: 1.832111                 GFLOPS: 1.17214
openmp caculation begin...
SECOND: 1.834559                 GFLOPS: 1.17057
```

(2)n=2048;

```
origin caculation begin...
SECOND: 217.423308               GFLOPS: 0.0790158
pavx caculation begin...
SECOND: 43.560106                GFLOPS: 0.394395
block caculation begin...
SECOND: 14.925702                GFLOPS: 1.15103
openmp caculation begin...
SECOND: 14.929177                GFLOPS: 1.15076
```

(3)n=3072;

```
origin caculation begin...
SECOND: 793.767015               GFLOPS: 0.0730467
pavx caculation begin...
SECOND: 156.329952               GFLOPS: 0.370895
block caculation begin...
SECOND: 55.485432                GFLOPS: 1.045
openmp caculation begin...
SECOND: 55.466944                GFLOPS: 1.04534
```

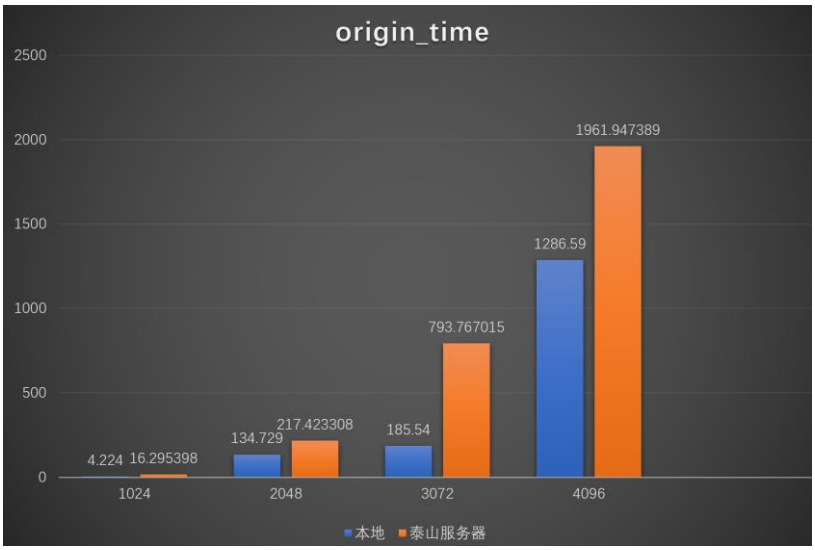
(4)n=4096;

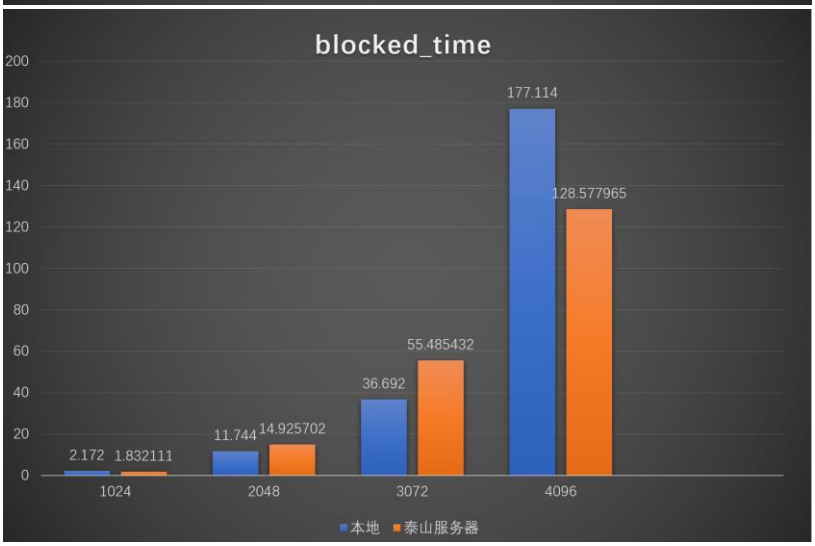
```
origin caculation begin...
SECOND: 1961.947389              GFLOPS: 0.0700523
pavx caculation begin...
SECOND: 378.331586               GFLOPS: 0.363276
block caculation begin...
SECOND: 128.577956               GFLOPS: 1.06892
openmp caculation begin...
SECOND: 130.112704               GFLOPS: 1.05631
```

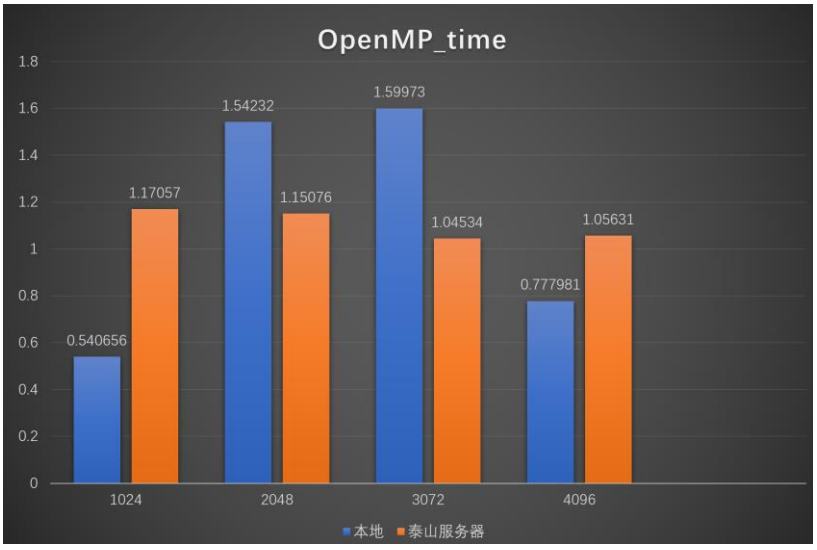
加速比是随着矩阵的规模加大而加大了，但是算力却在下降。

四、实验结果分析

1. 从 origin、blocked 和 OpenMP 对本地和泰山服务器的运算速度进行比较。从实验数据上看，泰山服务器的运算速度在绝大多是时候远不及我们本地计算机。

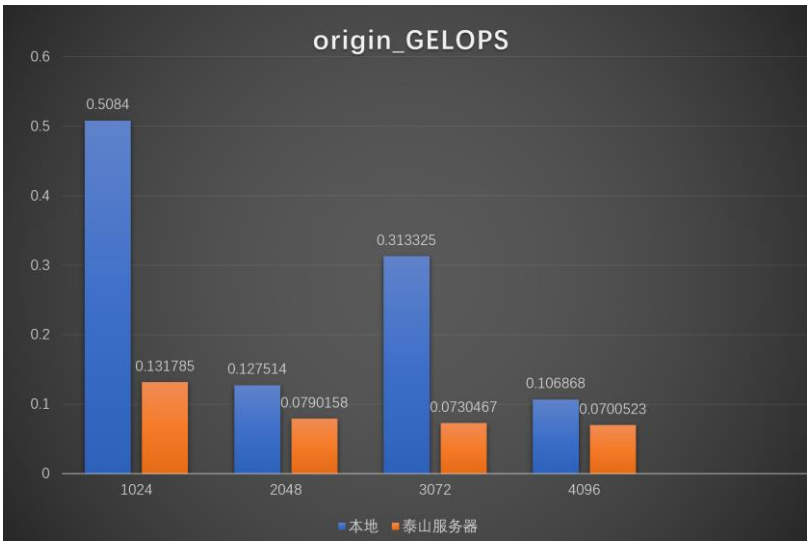


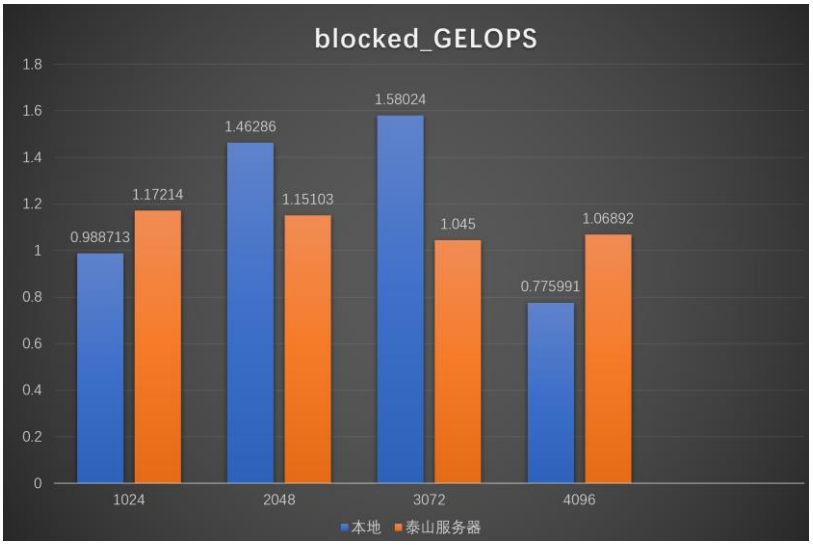


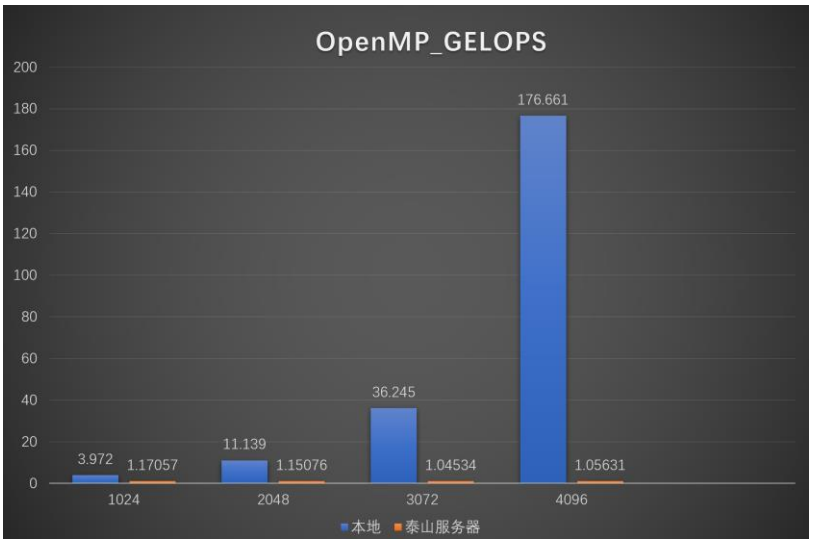


2.从 origin、blocked 和 OpenMP 对本地和泰山服务器的算力进行比较。

分块的算力基本保持稳定，原始的运算也和电脑上一样有所下降。在四个规模下进行测试，相差的值基本可以归于波动和降频，因为 OpenMP 在应对大矩阵的时候运算速度不应该有大幅度的下降，可能与 CPU 有关。







3.遇到的问题及解决方法

（1）按照定义计算矩阵乘法

问题：三重循环嵌套导致计算复杂度高，耗时较长。

考虑使用分块矩阵乘法等优化方法来减少不必要的计算和访存操作，从而提高计算效率。

（2）分块矩阵乘法算法

问题：实现分块矩阵乘法时，需要确定分块大小和合适的优化策略。

解决方法：通过实验和测试来选择合适的分块大小，以达到最佳性能。

（3）并行计算矩阵乘法

问题：并行计算可能会引发数据竞争和同步问题，导致结果错误或性能下降。

解决方法：使用线程同步机制，如互斥锁（mutex）、信号量（semaphore）等来解决数据竞争和同步问题。确保每个线程访问共享资源时的互斥和同步操作，以保证正确的计算结果和高效的并行计算。

五、总结感想

在完成实验过程中，我深刻理解了矩阵乘法优化对于提高计算性能的重要性。通过对不同层次、不同规模下的优化对比，可以清晰地看到优化带来的性能提升。

通过实际编程实现了基本矩阵乘法以及 AVX、并行化、块状优化等不同层次的矩阵乘法优化方法。通过调整代码结构、利用硬件特性以及并行化技术等手段，提高了计算效率。

在个人 PC 和 Taishan 服务器上进行了实验，并对比了它们在不同规模下的矩阵乘法运行时间等指标。从中可以分析出不同硬件环境对程序性能的影响，例如处理器类型、内存带宽等因素。

最后，在实验总结中可以探讨不同优化方法的适用场景和优缺点，以及在实际应用中如何选择合适的优化策略来提高程序性能。

通过这次实验，不仅加深了对矩阵乘法优化原理的理解，还提升了编程能力和对计算机体系结构的认识，为今后的科研和工程实践打下了良好的基础。