组成原理实验课程矩阵乘法优化本地版报告

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验名称 | 矩阵优化 | | | 班级 | 李涛 |
| 学生姓名 | 艾明旭 | 学号 | 2111033 | 指导老师 | 董前琨 |
| 实验地点 | A306 | | 实验时间 | 2023.6.1 | |

1. **实验目的**

参考课程中讲解的矩阵乘法优化机制和原理，在本地上使用 vs 编程环境，完成不同层次的矩阵乘法优化作业。

1. **实验内容说明**

使用自己电脑用基本矩阵乘法、指令集并行矩阵乘法、分块矩阵乘法、多处理器矩阵乘法，在 1024、2048、3072、4096 数量级下比较他们执行速度

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

加速原理

1 子字并行

使用 128 位宽字节寄存器，一次性计算四个单位的数，理论上得到四倍的运算速度。

2 指令级并行

通过同时执行多条指令来提高计算性能的技术。在矩阵乘法中，可以使用 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。如流水线、超标量、超长指令字等。

3 分块处理

当矩阵尺寸过大时，由于 Cache 的局部性原理，数据访问会倾向于访问附近的数据，但数据的大小大大超过 Cache，使得命中率过低，导致处理器需要多次访问内存造成时间浪费。通过将矩阵分割为适当大小的块然后按块计算，可以减少数据访问的跨越，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。利用时间局部性与缓存的局部性原理来提高性能，提高 cache 命中率。

4 多处理器并行的分块

调度多核处理器，将矩阵分割为多个块，并分配给不同计算节点，每个处理器或计算节点可以独立地计算其分配的块，最后合并结果。这样可以充分利用并行计算资源，加快整个矩阵乘法的计算速度。

1. **实验步骤**

复现老师提供的压缩包中的代码

1.cu\_gemm.cu.txt

#include <stdio.h>

#include <stdlib.h>

#include <cuda\_runtime.h>

#define THREAD\_X\_PREBLOCK 4

#define PELE(ARR, X, Y, i, j) ((ARR) + (i) \* (Y) + (j))

\_\_global\_\_ void kernel(int \*A, int \*B, int s1, int s2, int s3, int \*C) {

int i = blockDim.x \* blockIdx.x + threadIdx.x;

int j = blockDim.y \* blockIdx.y + threadIdx.y;

if (i < 0 || i >= s1 || j < 0 || j >= s3)

return;

\*PELE(C, s1, s3, i, j) = 0;

for (int t = 0; t < s2; t++) {

\*PELE(C, s1, s3, i, j) += \*PELE(A, s1, s2, i, t) \* \*PELE(B, s2, s3, t, j);

}

}

void print\_matrix(int \*A, int a, int b) {

for (int i = 0; i < a; i++) {

for (int j = 0; j < b; j++) {

if (j != b - 1) {

printf("%d ", \*PELE(A, a, b, i, j));

} else {

printf("%d\n", \*PELE(A, a, b, i, j));

}

}

}

}

void read\_matrix(int \*A, int a, int b) {

for (int i = 0; i < a; i++) {

for (int j = 0; j < b; j++) {

scanf("%d", PELE(A, a, b, i, j));

}

}

}

int main1(void) {

int s1, s2, s3;

scanf("%d%d", &s1, &s2);

int \*A = (int\*)malloc(s1 \* s2 \* sizeof(int));

read\_matrix(A, s1, s2);

scanf("%d%d", &s2, &s3);

int \*B = (int\*)malloc(s2 \* s3 \* sizeof(int));

read\_matrix(B, s2, s3);

int \*dA, \*dB, \*dC;

cudaMalloc((void\*\*)&dA, s1 \* s2 \* sizeof(int));

cudaMalloc((void\*\*)&dB, s2 \* s3 \* sizeof(int));

cudaMalloc((void\*\*)&dC, s1 \* s3 \* sizeof(int));

cudaMemcpy(dA, A, s1 \* s2 \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dB, B, s2 \* s3 \* sizeof(int), cudaMemcpyHostToDevice);

// cudaMemcpy(dC,c,size,cudaMemcpyHostToDevice);

// int threadPerBlock = dim3(16, 16);

// int blockPerGrid = (s1 \* s3 + threadPerBlock - 1)/threadPerBlock;

dim3 threadPerBlock(THREAD\_X\_PREBLOCK, THREAD\_X\_PREBLOCK);

dim3 blockPerGrid((s1 + THREAD\_X\_PREBLOCK - 1) / THREAD\_X\_PREBLOCK, (s3 + THREAD\_X\_PREBLOCK - 1) / THREAD\_X\_PREBLOCK);

kernel <<< blockPerGrid, threadPerBlock >>> (dA, dB, s1, s2, s3, dC);

int \*C = (int\*)malloc(s1 \* s3 \* sizeof(int));

cudaMemcpy(C, dC, s1 \* s3 \* sizeof(int), cudaMemcpyDeviceToHost);

//for (int i = 0; i < s1; i++) {

// for (int j = 0; j < s3; j++) {

// kernel(A, B, s1, s2, s3, C, i, j);

// }

//}

print\_matrix(C, s1, s3);

cudaFree(dA);

cudaFree(dB);

cudaFree(dC);

free(A);

free(B);

free(C);

return 0;

}

int main(void) {

main1();

}

Mmm.cpp:

#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();

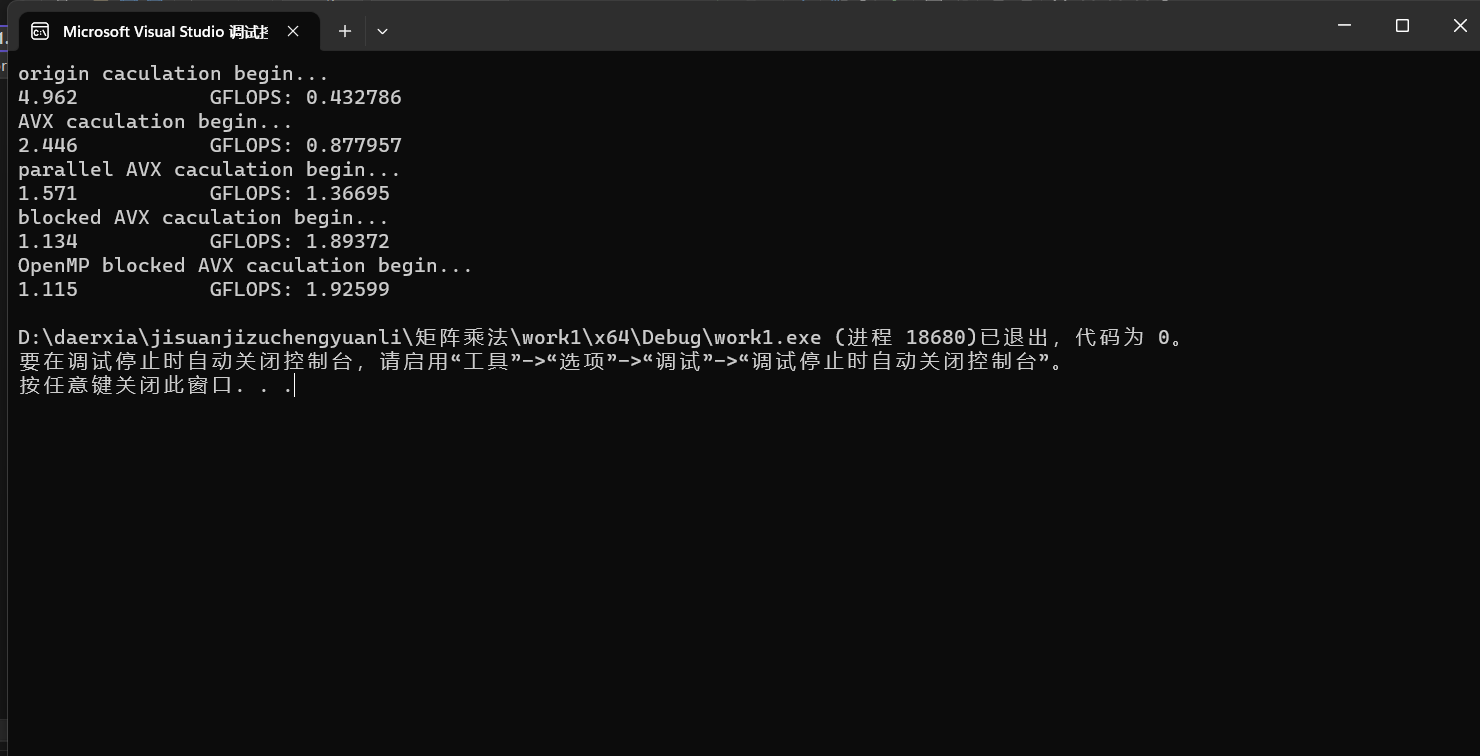
cout <<(stop - start)/CLOCKS\_PER\_SEC<<"."<<(stop - start)%CLOCKS\_PER\_SEC<<"\t\t";

printFlops(n, n, n, start, stop);

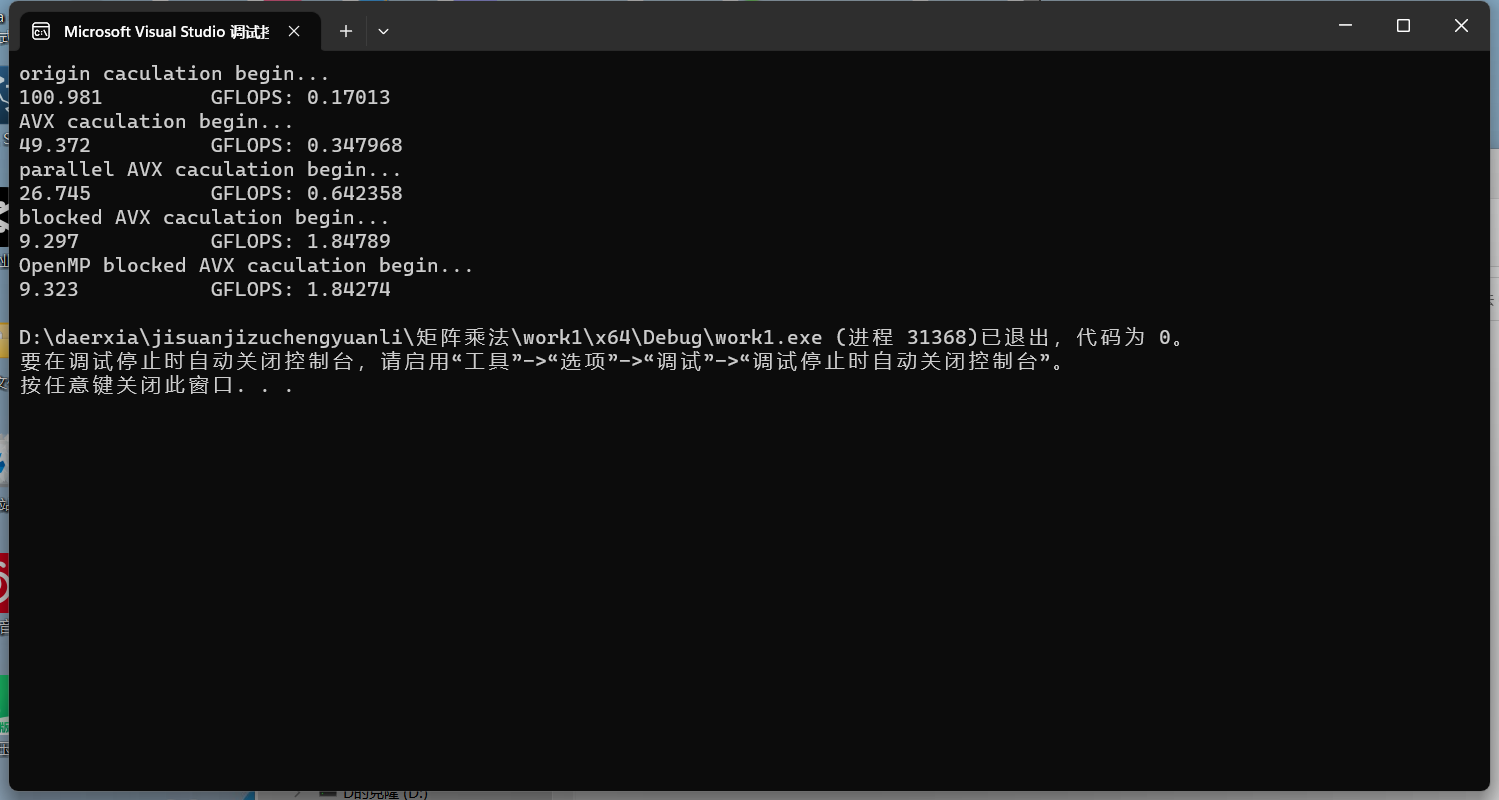
}

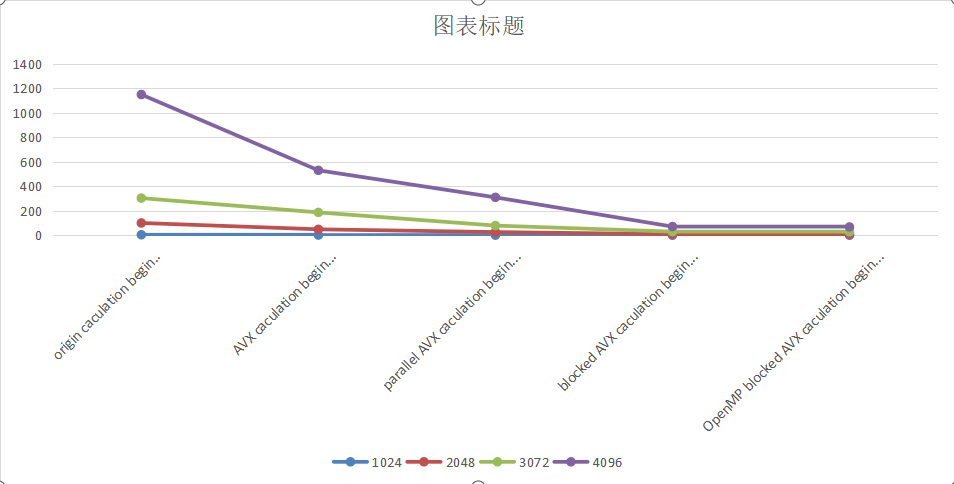
1. **实验结果分析**

**1 1024**

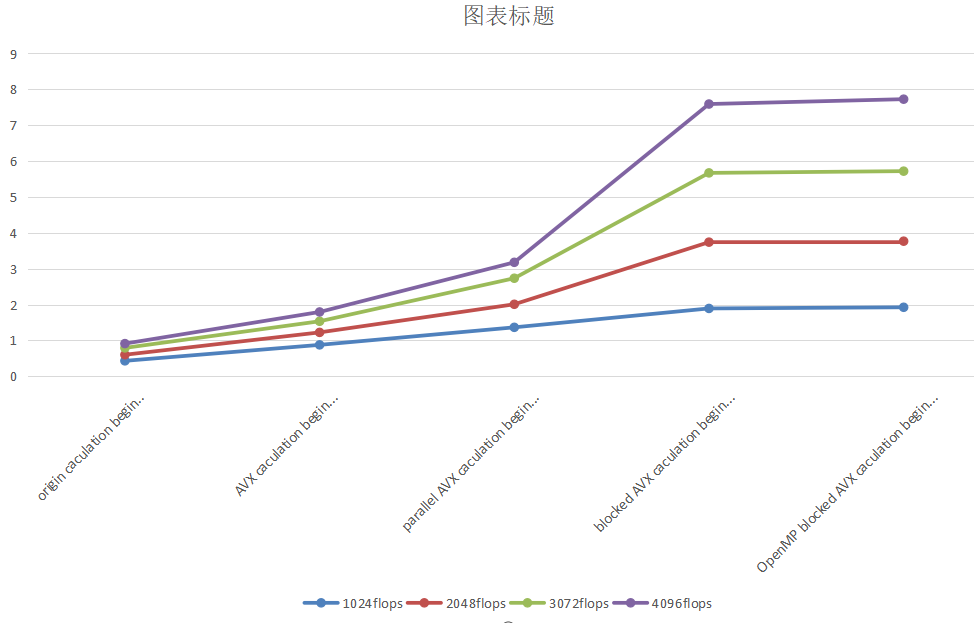


**2048:**





可以看到，随着矩阵 Size 的增长，原始方法的时间增幅近似为 n 的三次方，而根据上述加速方法后，时间显然大幅降低。OpenMP Blocked AVX 方法，在大规模矩阵乘法中时间几乎不增长。



OpenMP Blocked AVX 方法的计算性能和其他四种方法不在一个数量级别

**一些思考**

1. 首先发现，在使用子字并行时，我们使用 128 的宽字节，一次算 4 个数字，理论上加速比应为 4，但在小规模是，加速比却不及 4，原因可能是小规模时，数据 io 时间对总时间影响较大，因此加速效果较低。

2. 其次，使用分块加速矩阵运算，在数据量较小时也加速不明显，原因可能是因为分块目的是增加 cache 的命中率，而矩阵小规模时本身命中率就很高。

3. 而对于多核处理器的使用，则较稳定地提高计算性能

1. **总结感想**

可以发现，多处理器下分块 AVX 的性能提升是及其显著的，而原始 ORigin 的 n 立方级别的算法是几乎不能适应大规模的运算环境的。而这次矩阵乘法优化的实验，给我们提出了从宽字节提升每一次的运算量，到使用分块提升 Cache 的命中率，到使用指令并行降低数据通路的阻塞，最后使用多处理器分布式计算，这四种提升方法实际都是从硬件上的改进，归根结底未时间复杂度仍为 n3̂, 但是性能差异却如此大。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！