组成原理矩阵乘法作业1实验报告

学号：2111460 姓名：张洋 指导老师：李涛老师、董前琨老师

**一、实验要求**

1、实现矩阵乘法

2、完成矩阵乘法的优化，并在1024~4096规模的矩阵上进行测试

3、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、总结优化过程中遇到的问题和解决方式

**二、矩阵乘法的优化**

**1、矩阵乘法的原始代码**

|  |
| --- |
| 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;  }  } |

**2、子字并行的矩阵乘法**

|  |
| --- |
| 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);  }  } |

·数据类型·YMM寄存器，256位（32字节，可存放4double数据）·\_m265d 类型，保存4个双精度浮点值

·X86 CPU 子字并行库

·操作指令·\_mm256\_load\_pd() \_mm256\_store\_pd()·\_mm256\_add\_pd() \_mm256\_mul\_pd()

·每次运算4个数，理论上应有接近4倍的加速比

**3、指令级并行的矩阵乘法**

|  |
| --- |
| 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]);  }  } |

·原始的是多发射乱序执行·循环展开，复制4份后，gcc编译器优化时会将后面的三次循环进行指令级并行·B[i][j]可以在四次循环中反复使用，因此broadcast一份即可。·指令流水·Unroll 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**4、考虑cache的分块矩阵乘法**

|  |
| --- |
| 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);  } |

·存储器层次为【主存】-【Cache】-【CPU】·cache很难保证高命中率·分块算法（分成小块子矩阵运算）

**5、多处理器并行的分块矩阵乘法**

|  |
| --- |
| void omp\_gemm(int n, double\*A, double\*B, double\*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);  } |

·综合利用多核处理器·#pragma omp parallel for // 当前主线程将此for循环开多线程并行运行·线程数一般默认为4，也可以自行修改配置·此处采用默认的4个处理器开4个线程并行

**6、其他代码**

**（1）Flops ：每秒浮点运算次数**

|  |
| --- |
| //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;  } |

**（2）初始化矩阵A，B，C**

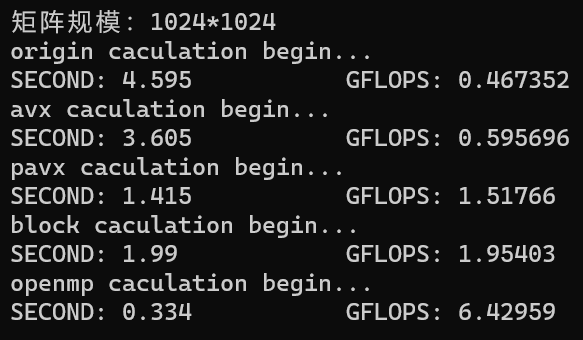
|  |
| --- |
| // 随机生成浮点数构造原始矩阵  void initMatrix( int n, REAL\_T \*A, REAL\_T \*B, REAL\_T \*C ){  for( int i = 0; i < n; ++i )  for( int j = 0; j < n; ++j ){  //A[i+j\*n] = (i+j + (i\*j)%100 ) %100;  //B[i+j\*n] = ((i-j)\*(i-j) + (i\*j)%200 ) %100;  A[i+j\*n] = rand() / REAL\_T(RAND\_MAX);  B[i+j\*n] = rand() / REAL\_T(RAND\_MAX);  C[i+j\*n] = 0;  }  } |

**（3）main.cpp**

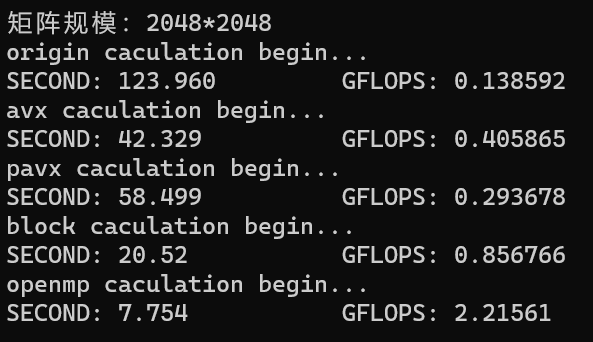
|  |
| --- |
| int main(){  srand( int( time(0) ) );  REAL\_T \*A, \*B, \*C, \*a, \*b, \*c;  clock\_t start,stop;  int n = 2048; // 矩阵规模  cout << "矩阵规模：" << n << "\*" << n << endl;  A = new REAL\_T[n\*n]; a = new REAL\_T[n\*n];  B = new REAL\_T[n\*n]; b = new REAL\_T[n\*n];  C = new REAL\_T[n\*n]; c = 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 );  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 );  } |

**三、运行结果**

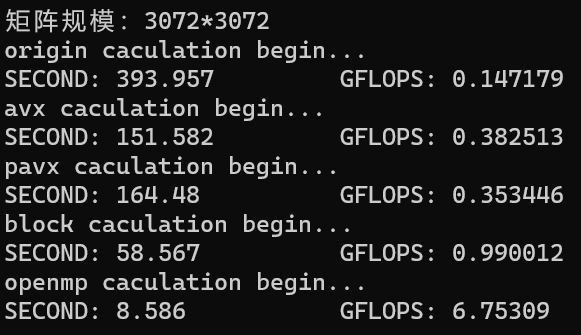
1、n=1024



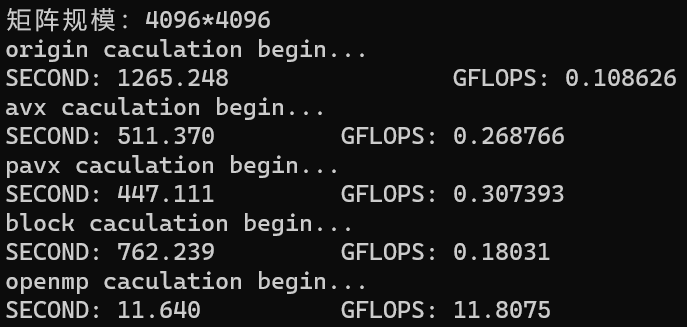
2、n=2048



3、n=3072



4、n=4096



**四、对比实验效果**

1、执行时间

2、加速比

3、运行性能

由运行截图可知，优化前的矩阵乘法计算耗时最长，运行性能最差。

执行时间的计算：按照优化前计算的矩阵乘法耗时最长，因为它需要执行三重循环来逐个元素进行计算。不同规模优化后的执行时间都有所下降，但是在测试的四种规模中，执行时间最快的优化方式都是多处理器并行的分块矩阵乘法。

加速比：优化的矩阵乘法和并行计算矩阵乘法相对于按照定义计算的矩阵乘法都能够取得较好的加速比。优化的矩阵乘法通过减少不必要的访存操作和利用分块矩阵乘法的优化策略，加速比相对较高。并行计算矩阵乘法通过利用多线程并行计算的特点，能够进一步提高计算速度，加速比最高。

运行性能：按照优化前计算的矩阵乘法的性能最低，因为它使用了三重循环的嵌套，导致计算复杂度较高。优化的矩阵乘法通过采用分块矩阵乘法的优化方法，减少了不必要的访存操作，提高了运行性能。并行计算矩阵乘法利用多线程实现并行计算，充分利用多核处理器的计算能力，因此具有更好的运行性能。

**五、遇到的问题及解决办法**

1. 有一些代码具体步骤不太明白，我通过仔细阅读董老师的ppt，发现ppt上给出了很详细的解释，再加上到csdn上搜索相关的概念，明白了优化代码的每一步是如何写出的，并自己尝试重新编写代码以实现相关功能。

2. 分块矩阵乘法算法时代码使用了默认的个数，我试着更改块的个数来找到合适的优化策略，以达到最佳性能。

3. 并行计算矩阵乘法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。可以通过使用线程同步机制，如互斥锁、信号量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。