并行计算第四次作业

by mo kanglong

一、多线程矩阵乘

1.block

单线程时分块有助于cache命中，带来少量加速。

2.SIMD

充分利用intel cpu的simd，可以一次性处理更多的数据。

结合AVX512 + FMA，带来大量加速。

并且展开循环，一次性计算8行24列的矩阵C，更好的复用数据。

3.理智抛弃block

block并不会对算法的并行可行性带来太大影响，但是以block为并行单位时，多线程的cache命中将会大大下降，因为block之间使用相同的数据不一定会很多。

抛弃block，而直接在IJK最外层套上OMP，可以得到大量加速。

4.结果

通过验证程序。在24线程时，得到较好性能。

```
66   Size: 673   Gflop/s: 307 (64 iter, 0.127 seconds)
67   Size: 703   Gflop/s: 324 (64 iter, 0.137 seconds)
68   Size: 704   Gflop/s: 327 (64 iter, 0.137 seconds)
69   Size: 705   Gflop/s: 325 (64 iter, 0.138 seconds)
70   Size: 735   Gflop/s: 339 (64 iter, 0.150 seconds)
71   Size: 736   Gflop/s: 347 (64 iter, 0.147 seconds)
72   Size: 737   Gflop/s: 345 (64 iter, 0.149 seconds)
73   Size: 767   Gflop/s: 202 (32 iter, 0.143 seconds)
74   Size: 768   Gflop/s: 207 (32 iter, 0.140 seconds)
75   Size: 769   Gflop/s: 361 (64 iter, 0.161 seconds)
76   Size: 799   Gflop/s: 380 (64 iter, 0.172 seconds)
77   Size: 800   Gflop/s: 378 (64 iter, 0.174 seconds)
78   Size: 801   Gflop/s: 378 (64 iter, 0.174 seconds)
79   Size: 831   Gflop/s: 403 (64 iter, 0.182 seconds)
80   Size: 832   Gflop/s: 404 (64 iter, 0.182 seconds)
81   Size: 833   Gflop/s: 403 (64 iter, 0.183 seconds)
82   Size: 863   Gflop/s: 415 (64 iter, 0.198 seconds)
83   Size: 864   Gflop/s: 432 (64 iter, 0.191 seconds)
84   Size: 865   Gflop/s: 425 (64 iter, 0.195 seconds)
85   Size: 895   Gflop/s: 439 (32 iter, 0.105 seconds)
86   Size: 896   Gflop/s: 456 (32 iter, 0.101 seconds)
87   Size: 897   Gflop/s: 453 (32 iter, 0.102 seconds)
88   Size: 927   Gflop/s: 476 (32 iter, 0.107 seconds)
89   Size: 928   Gflop/s: 477 (32 iter, 0.107 seconds)
90   Size: 929   Gflop/s: 466 (32 iter, 0.110 seconds)
91   Size: 959   Gflop/s: 482 (32 iter, 0.117 seconds)
92   Size: 960   Gflop/s: 484 (32 iter, 0.117 seconds)
93   Size: 961   Gflop/s: 480 (32 iter, 0.118 seconds)
94   Size: 991   Gflop/s: 497 (32 iter, 0.125 seconds)
95   Size: 992   Gflop/s: 497 (32 iter, 0.126 seconds)
96   Size: 993   Gflop/s: 497 (32 iter, 0.126 seconds)
97   Size: 1023  Gflop/s: 522 (32 iter, 0.131 seconds)
98   Size: 1024  Gflop/s: 524 (32 iter, 0.131 seconds)
99   Size: 1025  Gflop/s: 522 (32 iter, 0.132 seconds)
100
```

二、3D-7P模板计算的多进程多线程优化

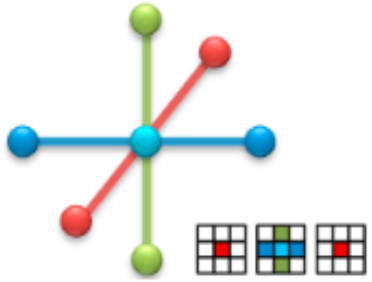
1.分析程序

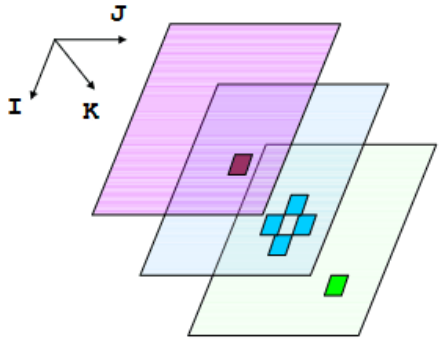
3维张量里的每一个点的下一时间步的值，受到本时间步其上下前后左右的点的影响。

时间步之间存在直接的强烈依赖关系，所以并行时间步不可行。

并且7点模板计算占用大量cache但命中率却不理想，如图，主要因为K轴的2个邻居点想被复用，必须等到IJ这个平面的点都计算完，即便是IJ面的4个邻居被容易被复用，但复用率却也不高。

之后假设循环顺序是KIJ，记[j, i, k]为第k层第i行第j列。



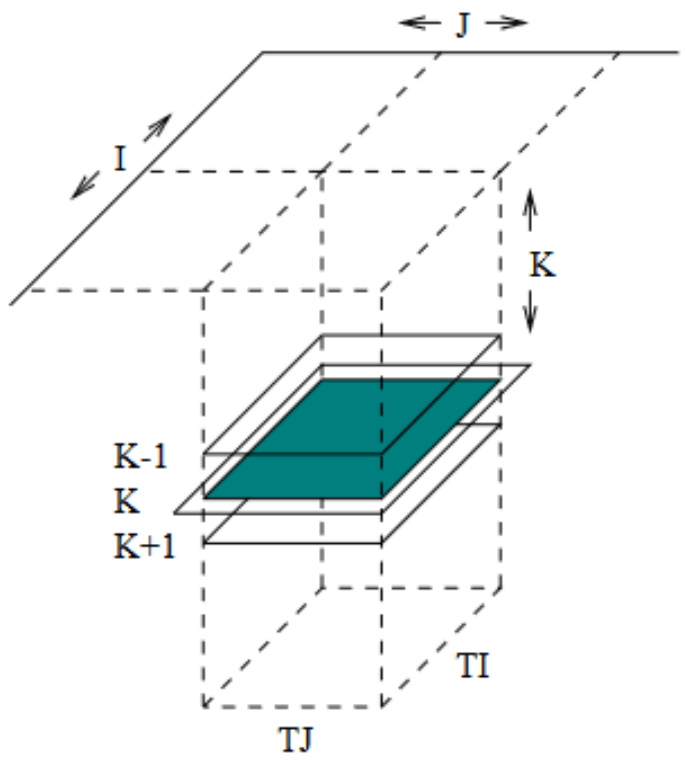


2.论文思路

[1]Tiling Optimizations for 3D Scientific Computations Gabriel Rivera, Chau-Wen Tseng

本论文提出block思想，沿着K轴，将3维张量分块，可以使得K轴的2个邻居点尽快得到复用，提高cache命中率，从而得到加速。

但是这需要3维张量在一开始就已经切分成多块，不然一个块里的[J-1, i, k]和[0, i+1, k]在物理内存上并不连续，同样的，[J-1, I-1, k]和[0, 0, k+1]也不连续，跨度也更大，在访存上没有带来理应多的加速。同时，本次程序的数据读入后已是一个完整的3维张量，再划分需要时间。



[2]A Multi-level Optimization Strategy to Improve the Performance of Stencil Computation Gauthier Sornet, Fabrice Dupros, Sylvain Jubertie

本论文提出使用25P模板计算代替7P，一个25P时间步等于7P的2个时间步，在减少时间步的同时，提高了数据的复用率。

但是这不能通过本次程序的验证。

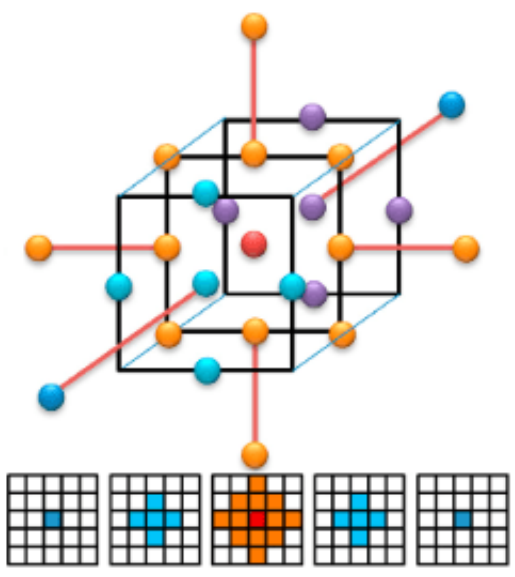


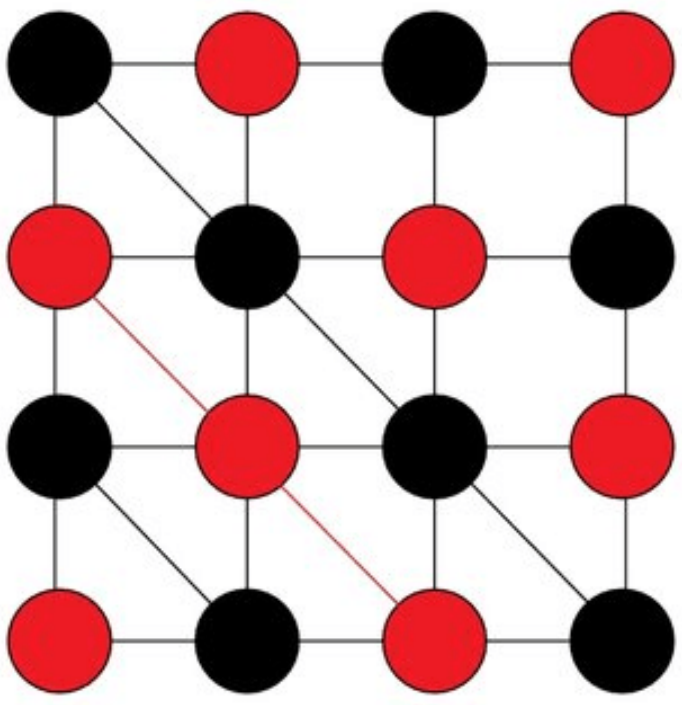
Figure 2: The 25-point stencil corresponding to the composition of two 7-point kernels.

additions by iteration (from 6 to 24 additions). However, one iteration of the 25-point stencil is equivalent to 2 iterations of the 7-point stencil. Thus, to obtain the same numerical results,

[3]JI Ying-rui, YUAN Liang, ZHANG Yun-quan. Parallelization and Locality Optimization for Red-Black Gauss-Seidel Stencil[J]. Computer Science, 2022, 49(5): 363-370.

和上一篇类似，本论文使用的红黑Gauss-Seidel Stencil，相比于7Pstencil降低了时间步带来的难度。首先是并行性提高，因为同颜色的点互相无依赖。其次是收敛速度更快，因为是先算红色的点，再算黑色的点，并且算得的结果即刻更新，这就使得一个点总是使用邻居的最新值。

但是本次程序依旧用不上。



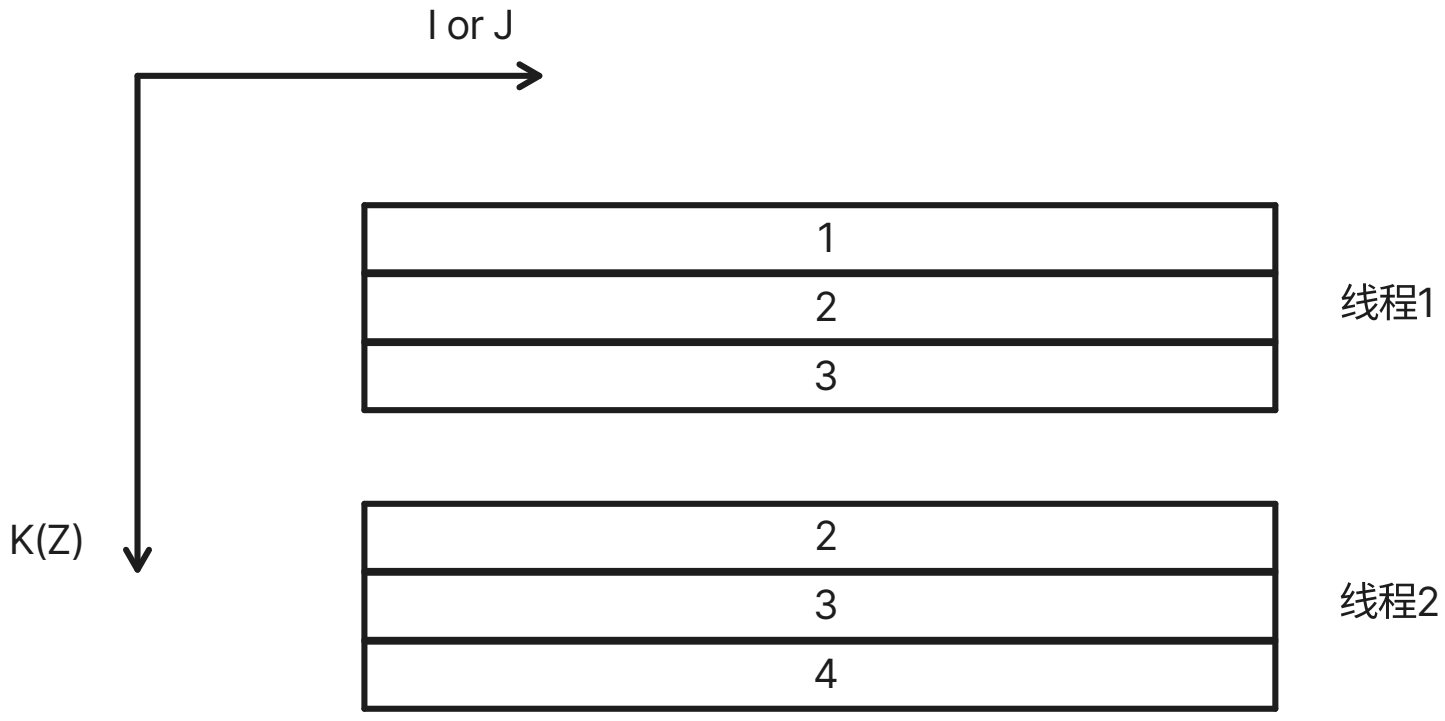
3.优化方法

[1]SIMD

使用AVX512，一次计算8个连续的点，同时读取它们所需的邻居也使用SIMD，可惜效果不佳，没有加速。

[2]OMP

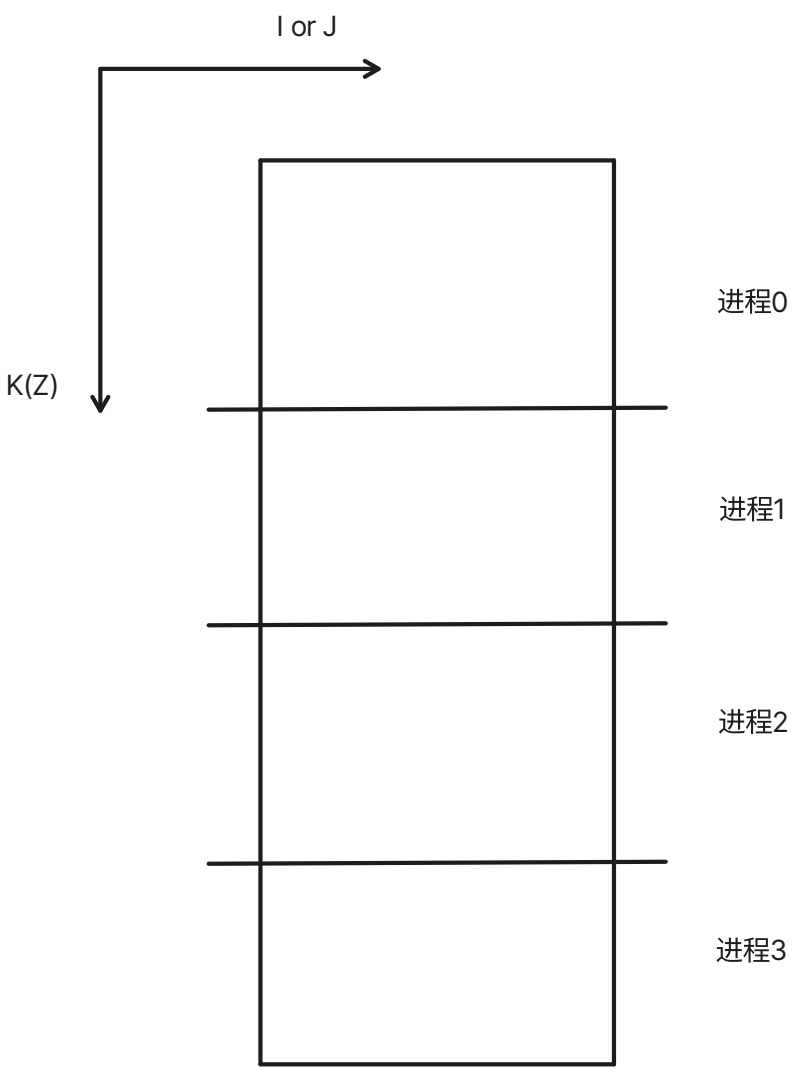
在K轴方向启用OMP，使得，每个线程自己在IJ平面上复用数据，而相邻线程在K方向上复用数据。如图，线程1计算第2层，需要第1,2,3层的数据，而线程2计算第3层时，需要第2,3,4层的数据，以此类推，可以提高K轴方向的数据cache命中率。带来大量加速。



[3]MPI

如图，为了不影响OMP以层为并行单位的优化下，MPI也在K轴方向划分任务，每个结点一个进程，每个进程内再多线程，最后进程0收集结果，返回验证。带来大量加速。

需要注意的是，虽然3维张量在K轴上切分成了4份，但每个进程还是存在少量的数据依赖，具体体现在相邻进程的边界上，比如进程0的最下层，在计算时，需要进程1的最上层，而且，这个最上层还必须在每个时间步都更新一次，确保进程0计算最下层时，拿到的是上一时间步的数据。



所以在每个时间步的循环最后，进程之间交换边界层以更新数据。

```
if(id != size - 1){
    MPI_Sendrecv(&a1[INDEX(0, 0, end-1, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id+1, 0, &a1[INDEX(0, 0, end, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id+1, 1, active_procs, &status);
    MPI_Sendrecv(&b1[INDEX(0, 0, end-1, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id+1, 0, &b1[INDEX(0, 0, end, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id+1, 1, active_procs, &status);
    MPI_Sendrecv(&c1[INDEX(0, 0, end-1, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id+1, 0, &c1[INDEX(0, 0, end, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id+1, 1, active_procs, &status);
}
if(id != 0){
    MPI_Sendrecv(&a1[INDEX(0, 0, start, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id-1, 1, &a1[INDEX(0, 0, start-1, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id-1, 0, active_procs, &status);
    MPI_Sendrecv(&b1[INDEX(0, 0, start, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id-1, 1, &b1[INDEX(0, 0, start-1, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id-1, 0, active_procs, &status);
    MPI_Sendrecv(&c1[INDEX(0, 0, start, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id-1, 1, &c1[INDEX(0, 0, start-1, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id-1, 0, active_procs, &status);
}
```

时间步都迭代完成之后，进程0收集其他进程的计算结果，最后返回整体结果，验证。

```
MPI_Barrier(active_procs);
if(id == 0){
    MPI_Gatherv(MPI_IN_PLACE, (end-start)*ldx*ldy, MPI_DOUBLE, &bufferx[nt%2][INDEX(0, 0, 0, ldx, ldy)], gatherv_revc, gatherv_shift, MPI_DOUBLE, 0, active_procs);
}else{
    MPI_Gatherv(&bufferx[nt%2][INDEX(0, 0, start, ldx, ldy)], (end-start)*ldx*ldy, MPI_DOUBLE, NULL, gatherv_revc, gatherv_shift, MPI_DOUBLE, 0, active_procs);
}
```

4.结果

Gflop/s	256	384	512	768
baseline	6.089713	6.067825	5.627577	5.760001
omp12	29.452317	34.488017	32.133789	34.342890
omp12+mpi2	49.571916	65.940767	55.951740	59.241101
omp12+mpi4	74.275021	109.560581	95.849548	110.106921
omp12+mpi6	115.842386	146.540234	131.121174	141.307663
omp12+mpi8	132.113706	163.560730	159.838708	173.166725
omp12+mpi10	170.447454	192.138639	196.921742	205.456532

以mpi4为例，验证通过。

```
MPI id:0 of size:4
MPI id:3 of size:4
MPI id:1 of size:4
MPI id:2 of size:4
cut : 1 64 128 192 257
MPI - 0 : [1, 64)
MPI - 1 : [64, 128)
MPI - 3 : [192, 257)
MPI - 2 : [128, 192)
MPI - 3 : OVER
MPI - 2 : OVER
MPI - 0 : OVER
MPI - 1 : OVER
MPI - 0 : ALLOVER
MPI - 0 0: checking
errors:
    1-norm = 0.0000000054475285
    2-norm = 0.0000066829895576
 7-point stencil - A naive base-line:
Size (256 x 256 x 256), Timestep 16
Preprocessing time 0.000015s
Computation time 0.203243s, Performance 67.358799Gflop/s
MPI id:0 of size:4
MPI id:1 of size:4
MPI id:3 of size:4
MPI id:2 of size:4
cut : 1 128 256 384 513
MPI - 0 : [1, 128)
MPI - 2 : [256, 384)
MPI - 3 : [384, 513)
MPI - 1 : [128, 256)
MPI - 3 : OVER
MPI - 2 : OVER
MPI - 0 : OVER
MPI - 1 : OVER
MPI - 0 : ALLOVER
MPI - 0 0: checking
errors:
    1-norm = 0.0000000865054423
    2-norm = 0.0002774738924372
 7-point stencil - A naive base-line:
Size (512 x 512 x 512), Timestep 16
Preprocessing time 0.000241s
Computation time 1.427195s, Performance 76.739118Gflop/s

```

5.代码使用

- sbatch sub.sh ./myproc运行程序
- sbatch check.sh ./myproc验证程序
- 线程数和进程数都需要在sh文件里设置