并行计算第四次作业

by mo kanglong

目录

Table of contents

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

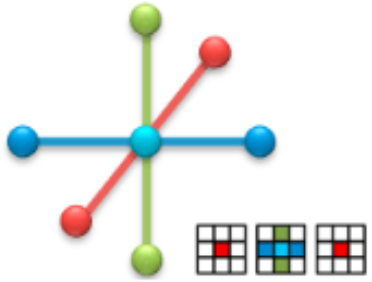
1.分析程序

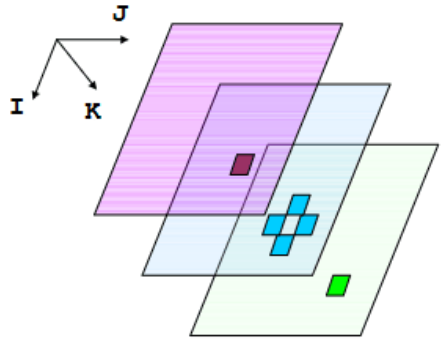
3维张量里的每一个点的下一时间步的值，受到本时间步其上下前后左右的点的影响。

时间步之间存在直接的强烈依赖关系，时间步的并行很困难。

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Z轴的2个邻居点想被复用，必须等到XY这个平面的点都计算完，通常难以被复用，即便是XY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XYZ，记[x, y, z]为第z层第y行第x列。





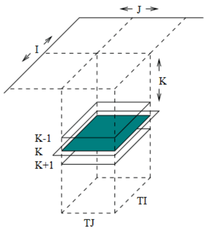
2.论文思路

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

本论文提出block思想，沿着Z轴，将3维张量分块，不仅可以使得Z轴的2个邻居点复用，而且XY平面的4个邻居点也能更快得到复用，**提高cache命中率**，从而得到加速。

同时粗化了并行粒度，使得并行更加易懂易用。

block的加速是理想的，现实还要考虑block会导致循环时指针不再像naive一样连续等问题。



[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2SP模板计算代替7P，一个2SP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。

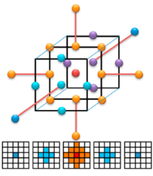


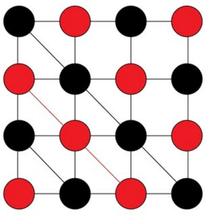
Figure 2: The 25-point stencil corresponding to the composition of two 7-point kernels.

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并行难度。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。

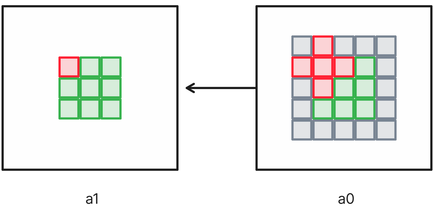


[4]Implicit and Explicit Optimizations for Stencil Computations Shoaib Kamil †, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, Katherine Yelick

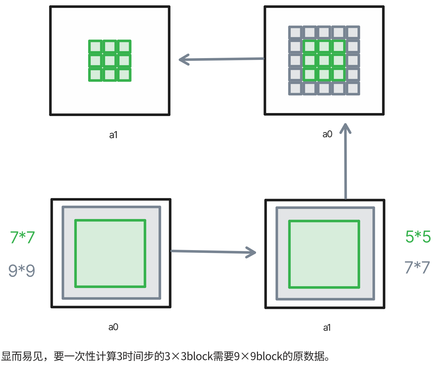
本论文提出诸多优化方法，比如，隐式block-与缓存无关，显式block-时间偏移，显式block-软件管理内存，这里只介绍**time skew algorithm（TSA）**，时间偏移算法，文中实验了**timestep（block time,以下简称BT）**从1到4，这里选BT = 3作为介绍，并且不考虑Z轴。

TSA最大的优势在于，对**时间步轴上复用数据**，突破了传统只能在空间上做复用的局限。

如下图，计算a1里一个3×3的block，当BT=1时（naive的BT就是1），需要a0里4×4的block数据。



那么计算a1里一个3×3的block，BT=3时，需要a0多大的block数据呢?



显而易见，要一次性计算3时间步的3×3block需要9×9block的原数据。

由此可以退出**普遍性公式**：一次性计算BT时间步的N*N的block，需要(N+2*BT)*(N+2*BT)的原数据。

TSA看似使得时间步缩短了BT倍，但是并不能使得程序加速BT倍，理想上也不行。因为如上两图所示，单次内，计算某一block的时间步从1变成了BT，所以总的时间步的是不变的。TSA实际的优势在于单次内，计算某一block的BT步用的数据都存在内存的同一位置上，**理想下，这使得计算量不变的情况下，使得内存同一位置的数据复用次数变成原来的BT倍。**

(同时TSA本身继承了block思想，四舍五入一下，简直就是在cache上编程啊)

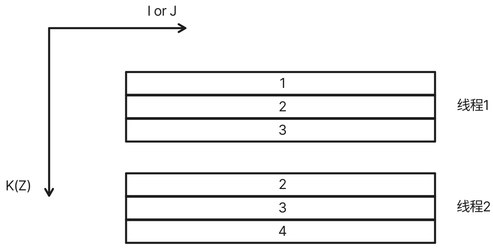
3.优化方法

[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

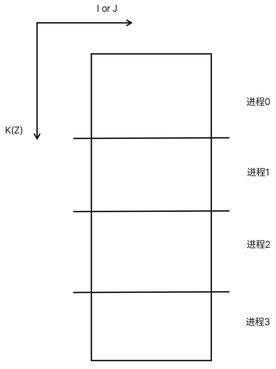
在K轴方向启用OMP，使得，每个线程自己在XY平面上复用数据，而相邻线程在Z方向上复用数据。如图，线程1计算第2层，需要第1,2,3层的数据，而线程2计算第3层时，需要第2,3,4层的数据，以此类推，可以提高Z轴方向的数据cache命中率。带来大量加速。



[3]MPI

如图，为了不影响OMP以层为并行单位的优化下，MPI也在Z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Z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。

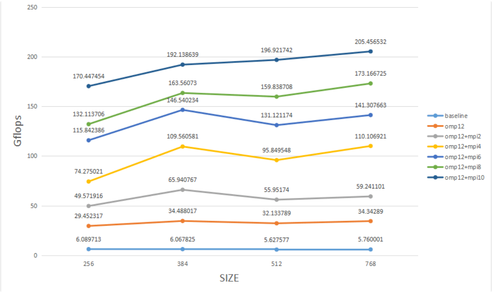
```
if(id != size - 1){
    MPI_Sendrecv(&a1[INDEX(0, end-1, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id+1, 0, &a1[INDEX(0, 0, end, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id+1, 1, active_procs, &status);
    MPI_Sendrecv(&b1[INDEX(0, 0, end-1, ldx, ldy)], ldx*ldy, MPI_DOUBLE, id+1, 0, &b1[INDEX(0, 0, end, ldx, ldy)], 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, &buffers[nt%2][INDEX(0, 0, 0, ldx, ldy)], gathersv_recv, gathersv_shift, MPI_DOUBLE, 0, active_procs);
}else{
    MPI_Gatherv(&buffers[nt%2][INDEX(0, 0, start, ldx, ldy)], (end-start)*ldx*ldy, MPI_DOUBLE, NULL, gathersv_recv, gathersv_shift, MPI_DOUBLE, 0, active_procs);
}
```

[4]目前为止结果

参数调优后，OMP取12时，效果更好。之后每结点1进程，每进程12线程，绘制进程数在不同SIZE下性能表现的折线图。



[5]Time Skew

结合论文1和论文4的思想，在XYZ空间轴分块后，再对T时间轴进行分块。一个block的计算经历以下几个步骤。

- 确认block的空间大小为BX*BY*BZ，时间大小为BT。
- 防止污染原数据，从原数据复制所需的(BX+2*BT)*(BY+2*BT)*(BZ+2*BT)多的数据到block本地内存，之后的操作都在本地内存进行。
- 使用本地内存计算BT时间步，得到大小为BX*BY*BZ的结果。
- 将此结果复制回原数据。

[6]参数调优

经历以下循环迭代后，对比结果，得出当**BT=5,BX=24,BY=BZ=16**时，在各SIZE拥有更好的性能。

```
for BT in [3,7] step 2 :
for BZ in [8,32] step 8 :
for BY in [8,32] step 8 :
for BX in [8,32] step 8 :
  calculate()
```

[7]OMP

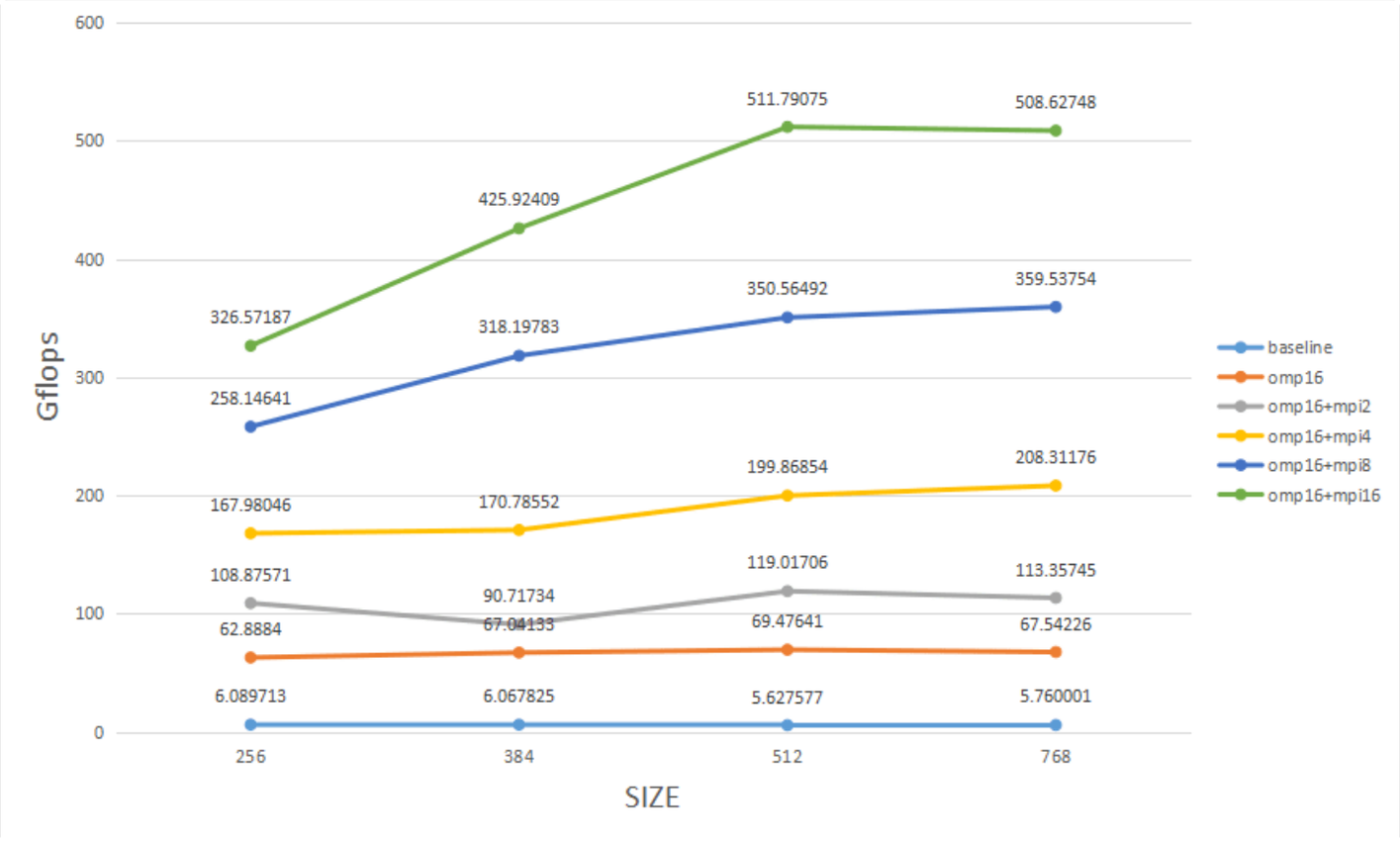
time skew将并行粒度粗化到block，omp以此为并行单位进行并行计算，邻近的block在复制数据时也能更好的复用数据。

[8]MPI

mpi处理方式和上面的MPI一样，不一样的是，为了发挥NUMA架构的优势，决定每结点2进程。

4.结果

参数设置为BT=5,BX=24,BY=BZ=16, 每结点2进程，每进程16线程，绘制进程数在不同SIZE下性能表现的折线图。



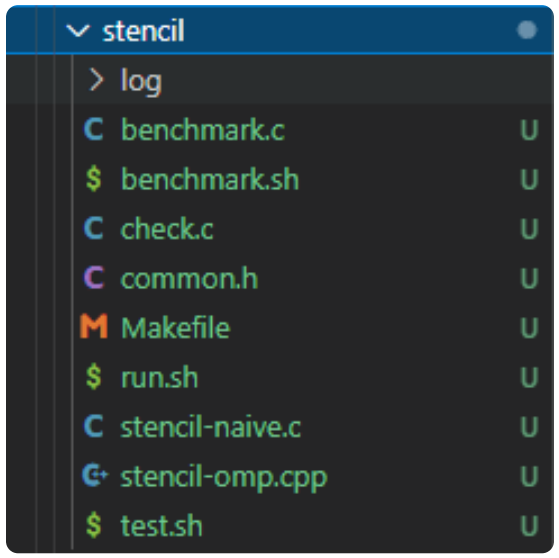
5.验证

以omp16+mpi4为例，验证通过。

```
MPI id:1 of size:4
MPI id:0 of size:4
MPI id:3 of size:4
MPI id:2 of size:4
        MPI - 3 : [192, 257)
        MPI - 2 : [128, 192)
        --INFO--
        BX = 24
        BY = 16
        BZ = 16
        BT = 5
        nt = 16
        x : [1 - 257)
        y : [1 - 257)
        z : [1 - 257)
        cut : 1 64 128 192 257
        MPI - 0 : [1, 64)
        MPI - 1 : [64, 128)
MPI - 0 : calculate T = 1
MPI - 0 : calculate T = 6
MPI - 0 : calculate T = 11
MPI - 2 : OVER
MPI - 3 : OVER
MPI - 0 : OVER
MPI - 1 : OVER
MPI - 0 : ALLOVER
MPI - 0 0: checking
errors:
    1-norm = 0.0000000154359539
    2-norm = 0.0000242293816073
 7-point stencil - A naive base-line:
Size (256 x 256 x 256), Timestep 16
Preprocessing time 0.000042s
Computation time 0.206531s, Performance 66.286394Gflop/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
        MPI - 3 : [384, 513)
        MPI - 2 : [256, 384)
        --INFO--
        BX = 24
        BY = 16
        BZ = 16
        BT = 5
        nt = 16
        x : [1 - 513)
        y : [1 - 513)
        z : [1 - 513)
        cut : 1 128 256 384 513
        MPI - 0 : [1, 128)
        MPI - 1 : [128, 256)
MPI - 0 : calculate T = 1
MPI - 0 : calculate T = 6
MPI - 0 : calculate T = 11
MPI - 3 : OVER
MPI - 2 : OVER
MPI - 0 : OVER
MPI - 1 : OVER
MPI - 0 : ALLOVER
MPI - 0 0: checking
errors:
    1-norm = 0.0000000908416214
    2-norm = 0.0002782093125611
 7-point stencil - A naive base-line:
Size (512 x 512 x 512), Timestep 16
Preprocessing time 0.000011s
Computation time 1.742590s, Performance 62.849950Gflop/s
```

6.代码使用

- 文件夹结构



- 运行一个程序只需要改动run.sh,设置如图八个参数，分别代表BT,BX,BY,BZ ，其中start=end。bash run.sh 即可得到5-24-16-16可执行文件和./log/5-24-16-16.txt记录文件。start≠end时脚本用于迭代最优参数。

```
t_start=5
t_end=5
z_start=16
z_end=16
y_start=16
y_end=16
x_start=24
x_end=24
```

- 调整资源申请和mpi相关修改benchmark.sh和test.sh。尽量不要手动sbatch，是benchmark运行还是test.sh检验，修改run.sh的sbatch行。