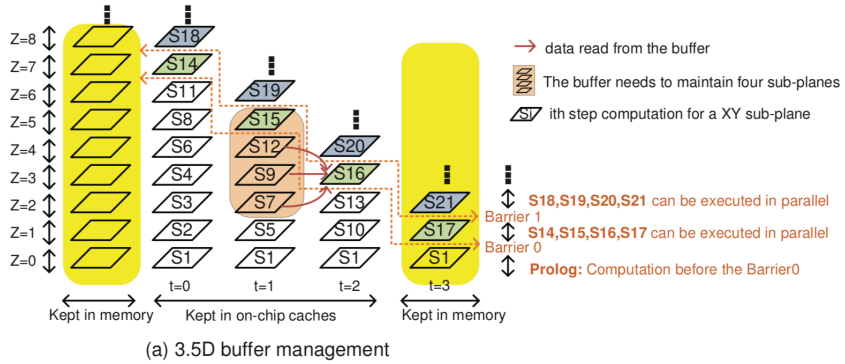
Stencil 在各种平台的实现

摘要：

和矩阵乘法类似，Stencil 的性能很大程度上也依赖于利用 Blocking 提高计算-访存比例。除了在空间维度上的 Blocking 之外，由于 Stencil 有迭代过程，因此在时间维上也可以进行 Blocking。针对 CPU 和 KNL，实现了 3.5D Blocking 和 Naïve 的算法。针对 GPU，实现了 2.5D Blocking。针对 CPU+GPU，在 GPU 上进行 3.5D Blocking 的计算，然后用 CPU 计算边界。针对 CPU 队列，把 Stencil 按照 z 轴划分到不同机器上，每台机器用 3.5D Blocking 计算。

1. CPU

3.5D Blocking 参考自论文[1]。简要 3.5D Blocking 介绍之后将介绍我的改进。3.5D Blocking 建立在 2.5D Blocking 之上的。2.5D Blocking，是把 Stencil 在 XY 平面上切成小平面。然后在每一个时间 t，依次对每一个小平面沿 Z 方向计算。比起 Naïve 地对每个 Z 遍历整个 XY 平面的方法，这样做的好处是每一个小平面在计算它上面的小平面的时候都可以被重复利用。如果对每个 Z 遍历整个 XY 平面，则一个小平面读入缓存之后很快就会被踢出。



(a) 3.5D buffer management

图 1　3.5D Blocking

3.5D Blocking 在 2.5D Blocking 的基础上增加了时间维的 Blocking。对每一个小平面，一次性计算 k 步。由于每计算一步，需要用到上一步的结果，一次性计算 k 步的好处是上一步的结果都还在缓存里。而 2.5D Blocking 中，每个小平面在计算的时候，该小平面上一步的结果一般已经不在缓存中。因此 3.5D Blocking 进一步提高了访存的局部性。

1.1 改进原始的 3.5D Blocking

我认为[1]中提出的 3.5D Blocking 主要存在两个问题。

1. [1]的 3.5D Blocking 的线程级并行是利用在计算同一个小平面上的。如图 1 所示，在计算 t=2, Z=3 小平面的时候，只会用到 t=1 时 Z=2,3,4 的小平面。因此 t=2,Z=3 和 t=1,Z=5 的小平面没有依赖关系，可以并行计算，分

配给不同线程。如果每个小平面沿 Z 和 t 维度都是串行计算的，对每个小平面只需要维护 3k 份历史结果，其中 k 是连续计算的步数。而[1]中为了开发小平面在时间维的线程并行，需要维护 4k 份历史结果。

2. [1]的方法在计算下一个 Z 的时候，都需要同步一下线程（如图 1 右下角所示）。

如果直接把每个 XY 子平面分配给各个线程就可以避免以上两个问题。每个 XY 小平面在 Z 轴和时间轴上的计算都是串行的，只需要在所有 XY 小平面每计算完 k 步之后同步一下即可。此外，每个小平面只需要维护 3k 个历史结果：

以图 1 为例，依次计算(t=1, S2)，(t=2, S1)，(t=1, S3)，(t=2, S2)，(t=3, S1)，(t=1, S4)，(t=2, S3)，(t=3, S2)，(t=4, s1)，…，这些计算都是串行的，容易检验对于每一个 t，只在 Z 轴上只需要维护 3 个子平面所需要的存储空间。

1.2 性能测试

实现了 3.5D Blocking 和 Naïve 的依次沿 t, Z, Y, X 轴迭代的方法。正确性检验是将 3.5D Blocking 和 Naïve 方法的输出逐个元素比较，误差设置为 1e-6。实验中连续计算的步数 k=5。当 X 方向的规模小于 100 时，XY 子平面 Block 的长度设为 20，超过 100 时设为 50。对 XY 子平面的 Block 宽度根据 Y 的规模也做同样的设定。

单机上 CPU 的性能如下表。由于权重各不相同，在更新一个元素时，需要 7 次乘法和 6 次加法，因此 Gflops 的计算方式为$13 \times X \times Y \times Z \times \frac{Steps}{Time}$.

表 1： CPU Stencil 性能

规模(x, y ,z, steps)	3.5D Gflops	Naïve Gflops	3.5D Time	Naïve Time
100,100,100,100	14.54	32.69	0.0894101	0.0397533
200,200,200,200	25.67	17.03	0.810211	1.22194
300,300,300,300	21.62	21.59	4.8692	4.87591
500,500,100,100	29.25	18.88	1.11147	1.72188
500,500,500,100	31.75	13.74	5.11785	11.8247

可以看出，但问题规模增大的时候，Naïve 实现的表现在下降，而 3.5D Blocking 的表现逐渐提高。

2. GPU

考虑到 GPU 计算边界不太方便，GPU 上只实现了利用 shared memory 的 2.5D Blocking。具体算法如下：将 Stencil 在 XY 平面上切割成长宽为 32 x 32 的小方柱。在每一步迭代中，每个线程块负责计算一个小方柱。在计算时刻 t 的高度为 z 的小平面（记为(t, z)）时，需要用到时刻为 t-1 时高度为 z-1, z, z+1 三个小平面(分别记为(t-1, z-1)，(t-1, z)，(t-1, z+1))。把这三个小平面读入

shared memory。在计算(t, z+1)时，shared memory 中的(t-1, z),(t-1,z+1)还可以重复利用，并用(t-1,z+2)替换掉 shared memory 中的(t-1,z-1)。这样，每个 shared memory 中的小平面都可以被用于计算三个下一时刻的小平面。每迭代一个时间点所有的线程块同步一次。

性能测试结果如下，其中时间是纯计算部分的时间。正确性检验是将 3.5D Blocking 和 Naïve 方法的输出逐个元素比较，误差设置为 1e-6。

表 2： GPU Stencil 性能

规模(x, y ,z, steps)	2.5D Gflops	2.5D Time
100,100,100,100	48.99	0.0265345
200,200,200,200	72.29	0.287703
300,300,300,300	71.87	1.46491
500,500,100,100	82.58	0.393516
500,500,500,100	73.93	2.19793

3. GPU+CPU

时间维的 Blocking 通常需要单独计算 Stencil 边界的元素。单个 GPU 不适合这种计算。而加上 CPU 之后，就可以利用 CPU 进行边界元素的计算，用 3.5D Blocking 在 GPU 上计算 Stencil 的中间部分，把边界留给 CPU。这样做的代价是需要频繁地把边界元素拷贝到 GPU 上，也需要从 GPU 上拷贝计算边界需要的那些元素到 CPU 上。经验证发现整个地拷贝 Stencil 比只拷贝许多不连续的边界元素要快，因此实现中都是完整地拷贝整个 Stencil。性能结果如下：

表 3： GPU + CPU Stencil 性能

规模(x, y ,z, steps)	2.5D Gflops	2.5D Time
100,100,100,100	3.60	0.361163
200,200,200,200	3.48	5.97998
300,300,300,300	3.48	30.2486
500,500,100,100	3.42	9.51282
500,500,500,100	3.53	46.0308

4. CPU 队列

CPU 队列的实现是基于单机 CPU 的 3.5D Blocking 实现。把 Stencil 沿 Z 轴方向划分给各个 CPU。在每个 CPU 上分别计算 k 步，之后再通信进行同步，这里的 k 和 3.5D Blocking 中的 k 相同。使用多个节点的时候通信代价较高，使用两个节点差不多可以达到单 CPU 性能的两倍。下面是使用两个节点的测试结果。正确性检验是将 3.5D Blocking 和 Naïve 方法的输出逐个元素比较，误差设置为 1e-6。

表 4: CPU 队列 Stencil 性能

规模(x, y ,z, steps)	2.5D Gflops	2.5D Time
100,100,100,100	12.56	0.103467
200,200,200,200	37.06	0.561305
300,300,300,300	52.87	1.99157
500,500,100,100	42.00	0.773809
500,500,500,100	57.20	2.8411

5. KNL

KNL 的实现和 CPU 上的相同, 使用 72 个线程。表 5 列出了在 KNL 上的 3.5D Blocking 和 Naïve 实现的性能。

表 1： KNL Stencil 性能

规模(x, y ,z, steps)	3.5D Gflops	Naïve Gflops	3.5D Time	Naïve Time
100,100,100,100	12.75	49.06	0.101993	0.0264978
200,200,200,200	12.65	25.74	1.64402	0.808136
300,300,300,300	17.81	24.82	5.9115	4.24188
500,500,100,100	14.78	33.30	2.19817	0.975913
500,500,500,100	14.74	25.28	11.025	6.4281

参考资料：

[1] Nguyen A, Satish N, Chhugani J, et al. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs[C]//High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for. IEEE, 2010: 1-13.