总体流程设计(3)-warp, block, grid

https://zhuanlan.zhihu.com/p/149817061

None Mon May, 24 03:18

之前讲了thread内的安排逻辑，后来写着写着有些无聊，就差点放弃了。想了想，还是接着写写，把warp，block和grid的安排也简单讲讲，也算是一个比较完整的收尾。

Warp

- warp内的所有线程尽量走同一分支（相当于if或else同时为true或同时为false），减少divergence。当然，一些短小的分支会用predicate的方式实现，不需要跳转，这样就相当于没有divergence，但一些线程空转的开销还是免不了的。在合适的情况下，可以通过交换线程的排列情况来使得同一个warp的线程尽量走同一分支。
- warp内的32个线程的仿存模式需要尽量满足最高效的模式，例如：

1. global load所需的transaction数目只与地址分布有关，而transaction数量与带宽的利用率是息息相关的。对于uncached load（针对L1而言），最小的load颗粒度是32B。而对于cached load，最小则是一个cacheline的大小（128B）。所以对于要在L1中缓存的数据（比如tex的load，用户通过 `__ldg` 内置函数的load等），尽量让地址落在对齐的128B内。而不用在L1中缓存的数据，尽量保证落在一系列对齐的32B内就可以了。
2. shared memory的访问不需要对齐。但尽量每个线程访问的是不同bank的数据。如果无法避免，则尽量减少同一个bank的不同地址数。因为每多一个地址就需要多传输一次。总传输次数由最多不同地址的bank决定。
3. shared memory也有broadcast模式。就是warp的不同线程访问同一地址其实只需要一次读取。这里有一个高级优化。假如warp的每个线程一个指令load 4*32bit=16B数据（有人称为**vector load**），那如果整个warp的地址分布仍然是每bank最多4B的话（相当于有线程load的是相同的数据），那仍然可以在一次传输内完成。这是GEMM中常用的一个优化shared memory读带宽的方式。
4. shared memory的atomic操作，不仅有bank的conflict，同一个地址也会有conflict。因为atomic不比write，单地址的多个操作也是全部有效的。而write就一定只有一个有效。global的atomic与shared memory类似，操作同一个地址时也是有conflict。当然，如果是global的reduction（就是不需要返回值的atomic），那conflict其实就没什么太大影响。因为它通常是“fire-and-forget”模式，指令发射后就由cache接管后面的事情，不用再管了。shared memory的atomic reduction暂时还不太清楚什么情况，应该也是类似。待研究。
5. local memory的读写一般是已经对齐好的，所以相对限制没那么明显。local memory在物理上也是global memory，但是索引方式是只在线程内算的。因为local memory在分配的时候就是按每线程4B的方式（vector load的情况没确认过，应该也是）对齐过的，每个线程只能读写自己的那些Byte，所以读同一个位置不存在不对齐的情况（相当每个线程执行load

0，实际上会load连续的128B）。但是如果每个thread读的是不同位置的local memory（相当于有的线程load 0，有的load 4之类），那就与一般global的load相当。当然，在某些情况下，一些thread也可能去访问其他thread的local memory，那就完全与global的情况是类似的。

6. constant memory尽量每warp访问同一个地址，否则会被序列化成多次操作。

- 一个warp如果是访问texture或surface memory，则会有一些更微妙的问题。不管是texture/surface的reference还是object，都是要先绑定到一段内存上才能访问。而绑定的这个内存就有两种形式：一种是原始的内存布局（linear，如果是2D则可能是pitched），另一种则是 `cudaArray` 。linear没有什么好说的，与前面的global访问模式是一致的。但如果是 `cudaArray` 则可能不一样。因为 `cudaArray` 的内存排布并不是原始的linear形式（这也就是 `cudaArray` 要单独copy一次的原因），而是所谓Z-order curve的形式。当然，因为texture对应的 `cudaArray` 当前只能通过tex/suf类的函数访问，所以用户不能直接控制访问地址。而只能通过X、Y、Z坐标的在同一个warp内的分布情况来控制。一般说来，如果是linear的texture，高性能的访问模式与global load也是一样的，warp内尽量访问连续的128B（texture一般会在L1缓存）。而对应Z-order Curve形式的 `cudaArray` ，则一般访问一个表面积最小的块，这样性能会比较好一些。比如一个warp访问8x4或是4x8的方块，会比32x1或是1x32的性能好一些。当然，这个问题也要具体分析，texture的访问模式往往与应用高度相关，还是要看具体需求。
- 如果warp内要做数据交换，可以用warp shuffle，也可以用shared memory。warp shuffle的好处是延迟短，不用block级别的同步，warp之间相对独立。但是shuffle只有全速ALU（比如 `FFMA` ）1/4的throughput，而且会抢占其他ALU指令的发射机会。shared memory则因为是load store指令，一般不占用ALU的发射机会。所以在不关心延迟或是延迟可以被有效隐藏的前提下，用shared memory也是不错的选择。另外一个功能上的差异是shuffle对数据交换的模式是有一些限制的，而shared memory是内存操作，地址访问的限制就会少很多。因而，用shared memory交换数据的操作自由度会更大一些。
- 除了shuffle以外，CUDA提供了许多其他warp级别的内置函数，比如warp vote functions，warp match functions，warp matrix functions，以及CUDA 11中为Ampere架构新加的**warp reduction**操作。这些操作都不再是通常的thread内部的独立操作，而是warp的所有线程都需要共同参与的过程。相当于还是有32个线程在一起工作，只不过这个执行结果是需要大家齐心合力得到的。由于涉及到warp内的同步问题，在volta引入independent thread scheduling后，这些函数都需要配合 `__syncwarp()` 来保证后续操作的正确性。

Turing开始提供Uniform Register用来做warp内的公共处理，是类似AMD的Scalar GPR的每个warp共用的执行路径。与前面warp指令不同的是，如果执行的是Uniform Datapath Instruction，那实际上是有另外的专门执行单元做工作，而不是由warp内的32个线程来共同完成。不过，当前NVIDIA还没有把这些相关的操作做为一个CUDA C的编程模型暴露给用户，相当于这些操作还只能由编译器来按需调用。PTX中倒是有一些可以有指定uniform路径的地方，

主要是各种跳转类指令比如bra.uni指令，还有一些指定特定单一对象的指令如ldu指令。其中有一些应该是可以用Uniform Register和Instruction来做的。这些可能都需要将来编译器不断跟进完善才能更好的被用户使用。

Block

- 尽量保证block的线程数是warpSize（当前是32）的整数倍。否则多出来的余数部分有些线程永远是闲置的，但仍会占用与其他线程等量的资源。
- 如果不需要Shared memory和同步，block的warp数可以不用太多。只要一个SM上运行的总warp数一样，效率就是接近的（前提是不要同步和shared memory），是不是属于同一个block区别不大。小block的好处是每个warp占用比较多资源时，每个SM有机会容纳更多warp。要同步的时候需要等待的block也更少。当然这个跟具体大小有关系。比如对于Turing，假设每线程用了128个GPR，每个warp是32*128个。每个SM总共有65536个GPR，所以最大可以容纳16个Warp（共512线程）。如果一个block就一个warp（或者2，4，8，16，能被16整除也一样），则可以用满16个warp。但如果一个block有6个warp，那一个SM就只能放两个block共12个warp。剩下的GPR就很可能被浪费了，因为剩下的4个warp的位置放不下一个完整的block。当然，如果上一个block有2个warp率先退出，那也许这两个warp的资源可以被提前回收，加上空置部分，就可以放下一个完整block了。这样就会出现三个block同时运行的情况。当然这个模式占总体时间的比例也许不会很大，可能大部分时间那些GPR都是空置的浪费状态。

但是，block太小也不一定就好。当前的几代SM架构下，一个SM能最多容纳的block数不如warp数多（一半）。所以一个block最好包含2个以上的warp。当然，也允许有一些例外。比如对sm75, 本来一个SM可以放下32个warp，但由于当前warp占用资源过多，一个SM最多也只能放下16个warp，那一个block就一个warp也无不可。 这是比较边际的情况。实际的最优配置要看具体资源占用。

| Compute Capability | 5.x, 6.x, 7.0 | 7.5 |

|--------------------|---------------|-----|

| Max blocks per SM | 32 | 16 |

| Max warps per SM | 64 | 32 |

| Max threads per SM | 2048 | 1024 |

block的warp数最好是2，4，8，16，32这种2的幂数，这样不容易产生余数。另外当前的SM一般都是**4**个warp scheduler，据我所知，每个warp scheduler分配的warp是确定的。所以只有同一个warp scheduler上的warp才能通过TLP相互cover latency。GPU一般会把每个block的warp尽量

均分到4个warp scheduler上。因为block的资源是整体分配的，同一个warp的生命周期或者说是运行流程上往往还是存在一定的同步性（更准确的说是有较大的关联）。这样就可能出现某段时间某个功能单元特别拥挤，另一段时间却特别空闲的情况。不同block间的运行则比较自由，更容易错开一些。

- 如果block内需要同步（`__syncthreads()`以及它的几个变种），则遵循最少原则：把需要同步的最小部分作为一个block。比如有8个warp，每4个warp内有数据交互或同步，两组4个warp间没有交互，则一个block就4个warp。因为有同步时，先到达同步点的warp需要等待其他warp。一个block中的warp越多，往往等待的开销就越大。如果有一个warp特别慢，则可运行的warp数会急剧减少。小的block到达同步点时只阻塞本block内的warp，而其他block的warp因为运行上可能是错开的，也许这段时间并不在同步点上，这样可以通过运行这部分的warp来cover另一个block同步的延迟。当然，运气不好的时候，不同block的warp可能都在等同步。但对于异步的block而言，这个概率会更小一些。
- 同一个block的warp的任务量尽量大致相当。尽量避免一个warp的任务显著多于其他warp的情况。在需要同步的情况下，这自不必说，相当于有一个远远拖在后面的人，拉其他warp的后腿。而即使是不需要同步的情况，单个warp的任务显著多也容易导致block占用的shared memory资源迟迟不能释放，使得后面的block进不来，最终导致eligible的warp减少而影响延迟隐藏的效果。当然，最糟糕的情况是warp内的线程就严重的任务不均衡，这相当于有大量的divergence，对整体效率的损伤是很大的。如果确实存在一些线程的任务要显著多于其他任务，尽量把这种不均衡分配到不同block去。
- block的size是一个很有趣的问题。实际上当前`blockDim`有`x`，`y`，`z`三个分量，`x`和`y`的范围都是1~1024，`z`的范围是1~64。block的总线程数是二者乘积，必须小于1024。虽然`blockDim`和`gridDim`都是三维向量，但是两者其实是各自摊平成一维，所以`blockDim.x`其实不一定要和`gridDim.x`对应。分成三维的实际好处主要还是减少一些可能的高开销整数取余或除法。当然，因为分成3个对应的维度，一些多维任务的划分上可以更简单直观一些。但这并不是绝对的。比如说有两个1024x1024的矩阵相加（线程间没有依赖关系），每个block算32x32的小矩阵，然后一共是32x32的grid，相当于`blockDim=(32,32,1), gridDim=(32,32,1)`。这是很合理的安排。但是如果是1000x1000的矩阵相加，还用这种方式就会导致所有边缘的block都有很多线程被mask了，相当于利用率为`1000x1000/(1024x1024)`。而如果我们让每个block计算一个1x1024的连续1024个元素，那只有最后一个block会有线程被提前mask掉，相当于利用率为`(1000x1000)/(ceil(1000x1000/1024)x1024)`。对于三维的情况，这个差异可能会更大一些。当然，如果是一个1024x1024的图像（绑定到2D的`cudaArray`），需要做局部卷积之类，那一个block算32x32就是一个很好的选择，因为这样可以用上shared memory做局部的缓存，texture cache的hit rate可能也会高一些。而即使图形的size变成了1000x1000，这个优势也许还是在的。所以这个问题也要辩证看待。

Grid

- 由于每个block只能完整的位于同一个SM上，那grid内的总block数尽量要大于总的SM数，如果可以的话，最好要是SM数的数倍。否则就会有SM得不到足够的任务而利用率下降。
- 对于block数很少的情况，要尽量要避免tail effect。所谓**tail effect**，就是说假如有M个SM，N个block，假设每个block的运行时间大致相当，那GPU会按每波M个block均匀发给M个SM各一个任务。如果N不是M的整数倍，那最后一波任务就有一些SM处于空转状态，从而影响效率。当N和M差别不大时，这个tail effect会尤为明显。这时，可以把block切分成更小的block，也可以通过调整GPR和shared memory的占用情况来改变每个SM可以容纳的block数。这其中的指导思想是尽量让所有的SM都可以得到足够的任务，而具体的表现形式还是要看任务具体怎么拆分。
- 每个kernel的启动多少是有overhead的。所以每个kernel的任务不宜太少。如果某几个很小的小kernel可以合并成一个大kernel，一般说来是划算的。因为这会让更多的功能单元参与到执行中而不是单纯等待其他单元的计算结果。当前AI的一个常用的优化手段就是底层算子的融合，其实就有一些把小kernel合成大kernel的操作。
- 多个kernel之间如果是有依赖关系的，要放在同一个stream上。没有依赖关系的（除kernel以外，包括`cudaMemcpy`，`cudaMemset`之类）可以放到不同的stream上。在GPU资源足够的情况下，这些操作也许可以并行执行，从而提高GPU的利用率而提高性能。
- 如果kernel之间有比较复杂的依赖关系，或者是kernel之间相互运行次序固定但需要运行很多次时，可以考虑使用`cudaGraph`。`cudaGraph`不仅可以显式的构造多个kernel（也包括一些memcpy之类的操作）的依赖关系，还可以通过运行上的合理安排减少每个kernel启动的overhead。对于一些需要一次定义但是反复多次运行的kernel来讲（比如多次迭代），这些开销加起来也是很可观的，特别是对于kernel本身耗时不太多的情况下，overhead的开销可能会与kernel运行相当甚至超过运行本身的时间。

暂时就想到这么多，先写到这吧…… 大部分规则其实就是个感觉，你一定可以找到一些场景是这个规则不适合的。所以要活学活用，关键是理解其中具体可能牵涉到哪些东西，然后具体问题具体分析。在没有合适优化思路的时候，套一套通常的路子，也不失为一种打开思路的方式。所以，仅供参考吧！