CUDA微架构与指令集（5）-Independent Thread Scheduling

https://zhuanlan.zhihu.com/p/186192189

None Sun May, 30 17:50

今天接着上期指令调度的话题，简单聊一聊volta后才有的新feature：independent thread scheduling。在聊这个feature之前，还是先聊一聊CUDA里warp divergence时分支执行的基本流程。

Warp Divergence的由来及处理

CUDA里的分支指令中，有两类可能导致warp divergence（注：我这里“分支”和“跳转”两个词经常混着用，反正它们界限也很模糊，理解就好）：

1. 条件跳转（Conditional branch/call），比如带predicate的`BRA`。条件跳转的目标是确定的，predicate只能决定跳或不跳。不跳就是继续执行下一条指令。CUDA C中的很多流控制指令（比如if-else, for, do-while等等）都有能力触发这种类型的跳转。当然有些流控制语句会用predicate来实现，不一定需要跳转。有的指令比如`IMNMX`、`FMNMX`和`SEL`、`FSEL`则直接有处理简单选择型分支的能力（有点类似C的三元运算符`p?a:b`）。编译器有时候也会把一些较复杂的分支分解为能用predicate或是内置分支指令处理的形式，从而尽可能的减少跳转开销。
2. 间接跳转（Indirect branch/call），如`BRX`指令，后面可以接一个GPR做目标PC地址。由于目标GPR可能是不同的值，所以不同线程可能会跳转到不同的目标去。CUDA C中能触发这种指令的场景比如switch-case，函数指针的调用，或者是类似函数指针的虚函数表之类。当然，是不是一定生成这种类型的指令，还要看编译器的具体处理。有的switch-case搞不好还是if-else方式实现的，所以这种类型的代码往往要检查最后生成的汇编。一般说来，只有可连续整数取值的switch-case才能转成brx，否则一般都是if-else。CUDA C应该是不支持char*或者string类型的switch（支持单个char，相当于整数），而且这个东西就算有，估计也只能用if-else做，字符串比较又是个慢活，性能会是个问题，实现起来也麻烦，实际使用时还是尽量避免为好。

两者造成的divergence略有一点区别：条件跳转的divergence是二重的，但是可以通过多重嵌套来创造更多的分支。间接跳转则是每个目标地址都有自己的分支，所以本来就可能是多重的，当然要嵌套也是可以的。

之前的专栏文章也多次提到过，CUDA的kernel里绝大部分的函数调用都是inline的。所以`BRA`、`JMP`、`CALL`这种指令也好，`BRX`、带GPR操作数的`RET`之类的指令也好，最主要的工作仍然是转移Program Counter（程序计数器，简称PC，可以简单的认为是当前执行的指令地址）。有一些架构可能在跳转时需要隐式或显式的保存一些PC的调用栈。CPU的跳转往往伴随着上下文保存，以及为了满足ABI做的一些准备工作（比如参数需要存到特定寄存器，而返回值也必须

存到某个特定寄存器）。GPU的跳转与之相比还是非常轻量级的。另外，CPU的跳转可能导致pipeline里很多工作要flush，断流对性能影响比较大，所以需要依靠分支预测和speculative execution来缓解影响。GPU跳转没有这些功能，跳转产生的延迟都要靠其他warp的运行来隐藏。不过，GPU的pipeline级数没有CPU那么多，而且指令Cache的命中率还是很高的，所以GPU跳转的开销并没有CPU分支预测失败那么大。GPU的跳转指令通常可以与ALU指令同时发射，加上TLP延迟隐藏的效果（跳转就没有ILP可用了），编译器还可以通过loop unroll和predicate代替跳转等优化来减少跳转指令的产生。所以总体来讲，单纯的跳转对程序整体性能的影响一般并不太大。

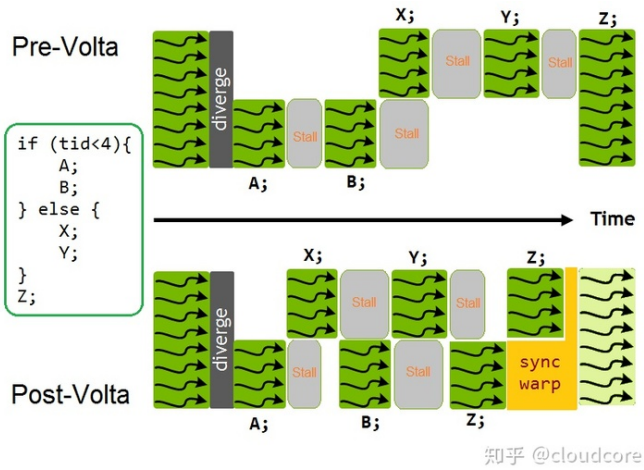
跳转对GPU性能影响比较大的一个情形就是warp divergence。由于SIMT的特性，导致每个warp只能同步执行相同PC处的指令（也可以同时双发射）。比如这样的一段代码：

```
if(threadIdx.x<4){
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

这里假设warp内共8个线程，前4个走if分支，执行语句A、B，后4个走else分支，执行语句X、Y，然后都需要执行Z。在volta之前，这段代码的执行是类似深度优先的一个遍历，具体执行顺序是A->B->X->Y->Z（**注**：其实也有可能X->Y->A->B->Z，编译器也许是有这个自由度做次序交换的，总之一个分支走完才能走下一个分支，后面我们都假定AB在XY前面）。这样相当于将分支图做了线性摊平，warp内所有线程都可以共用一个PC。当然在执行时，还会隐式的设置相应的mask，保证只有active的线程才真正执行。这带来的问题一是两者之间有了必然的先后关系，二是A、B中存在stall时，尽管X不依赖于A、B，但仍然不能运行。

Independent thread scheduling的工作方式

Independent thread scheduling的关键就在于，既然上面的A、B和X、Y没有相互依赖，那就完全可以交错执行。特别是当其中一个分支处于stall状态时，另一个分支仍然是可以执行的。简单示意如下（图修改自NV官网Blog Inside Volta: The World's Most Advanced Data Center GPU，这里加入了我自己的一些理解）：



从性能角度上讲，independent thread scheduling没有改变SIMT的基本属性，warp内的所有线程仍然只能同时执行同样PC的指令，不active的thread一样会被mask。因此，warp内出现divergence时的指令发射效率并没有变化。这样做对性能的关键影响在于：不同的分支之间，可以通过交错执行来一定程度上形成TLP的效果，从而达到相互帮助隐藏延迟的目的。上图中B和Y由于对A和X有依赖，pipeline出现了stall。对于volta前的架构，这个stall的延迟只能由其他warp来隐藏。但volta后，X可以在B等待A输出的时候运行，B可以在Y等待X输出的时候运行，等等。分支之间相互切换的方式应该与warp间相互切换的方式类似，比如有reuse的时候应该也是尽量不切，memory指令也许可以插在ALU的2 cycle发射间期中发射，等等。另外，这里要注意的一个点是，Z在volta前的架构里是没有divergence的。但是到了Volta，if分支中的Z可能会先运行（比如上图中Z和B、Y都有依赖，但是B完成的早），这样Z也可能出现divergence。要保证converge，需要显式的调用 `__syncwarp()`（`WARPSYNC` 指令）。

【注】：这里我其实仔细想了一下导致Z不能同步的机制。首先，我感觉表述依赖关系的scoreboard主要应该是warp level的东西，否则有个别线程先满足条件岂不先跑了，那这开销未免也太大了点。前面介绍control codes的时候也提到过，指令的predicate应该是不影响scoreboard，可能相当于predicate为否时设置scoreboard后马上就发acknowledge了。warp divergence的时候，隐式的active mask与predicate效果是不是不一样呢？有independent thread scheduling，感觉就不太一样了。也许scoreboard也按PC做了分组，每个分支都只检查active的线程的ack信号。我检查过一些带分支的Turing的代码，两个分支混用同样编号的scoreboard是很常见的行为。如果分支是并行运行，物理上用同一个值肯定会相互干扰，所以肯定是分开的。只是不知道这个分开具体是怎么个形式。当然这是硬件实现的问题，我们也不用太关心，反正看起来每个分支应该是可以独立地用scoreboard。那`DEPAR`等待计数的功能行不行呢？还没仔细研究过。另外，讲分支指令的时候我们也提到，跳转前一般会把当前scoreboard的dependency都先resolve掉。因为分支后的代码可能有多个跳转来源，如果设置了不同的scoreboard处理会很麻烦。所以我感觉这里的Z其实不太可能因为scoreboard的原因导致不能converge。不过由于代码排列是线性的，Z的两个来源B、Y必然至少有一个最后一个指令是跳转或分支指令，stall count一般更大，也许这个原因导致不能converge的概率大一些。当然，我也不知道它底层怎么实现的，这些都是我的猜测，仅供参考。

【20210305补注】：上面的猜测不太正确……从NV公开的专利来看，warp内的线程diverge是前面说的跳转指令造成的，但重新converge回来不是自动判断的，也需要相应的指令。比如`WARPSYNC`，`BSYNC`之类。warp内部有Convergence Barrier在维护这个分支mask。有兴趣的读者可以自行搜索相关专利。

从程序实现的需求上看，一些问题从算法上就很难避免divergence。Independent thread scheduling可以更好的帮助其隐藏延迟，从而减少divergence的performance penalty，但并不是减少divergence本身。当然，假如这些延迟本来就被隐藏得很好，或者说本来就几乎没什么stall，那它对性能的帮助就几乎没有了。而且由于PC输入变多，运行切换更加频繁，指令cache的压力自然更大，也许性能反而差一些。

Independent thread scheduling的另一个好处是，并行的分支之间并没有先后关系了，多个分支可以同时参与同步或是进行类似lock-unlock操作。分支间其实是可以进行一些信号传递甚至是数据交互，比如操作mutex，比如一个写内存一个去读等等，也可以与Cooperative Group结合，多个分支做同步等等。当然，这种用法在性能上未必就好，多数情况只是说提供了这种灵活度，对于一些性能不太敏感的应用，这种方式会让一些功能更容易实现而已。这里需要注意的是，不管是内存操作还是其他操作，要保证结果相互可见都需要`__syncwarp()`，否则可能出现data race。

对Warp Shuffle和Vote函数的影响

Independent thread scheduling带来的另一个影响是warp shuffle和vote类的操作方式发生了变化。之前的这类函数(`__shfl*, __any, __all, __ballot`)全部都需要加上`_sync`，并且要显式的输入一个32bit mask，指定具体哪些lane必须参与操作。比如Programing Guide中给出了一个例子：

```
if (tid % warpSize < 16) {
    float swapped = __shfl_xor_sync(0xffffffff, val, 16);
    //...
} else {
    float swapped = __shfl_xor_sync(0xffffffff, val, 16);
    //...
}
```

这个操作对volta之前的架构是invalid，因为`0xffffffff`表示32个lane必须同时参与操作，而volta前的架构在分支中有些lane被mask后并不支持这种操作。文档原文解释如下：

> for Pascal and earlier architectures, all threads in mask must execute the same warp intrinsic instruction in convergence, and the union of all values in mask must be equal to the warp's active mask.

这里对于volta前架构的正确写法应该是分别传入`0xffff0000`和`0x0000ffff`的mask。volta后，尽管if和else分支中都有一半被mask了，但它们仍然可以参与`__shfl_xor_sync`操作（但是根据文档，从inactive的lane中取数据是undefined）。

第二个例子是本来在volta前的架构是converge的代码，但到了volta后不保证converge了。比如：

```
// Sets bit in output[] to 1 if the correspond element in data[i]
// is greater than 'threshold', using 32 threads in a warp.

for(int i=warpLane; i<dataLen; i+=warpSize) {
    unsigned active = __activemask();
    unsigned bitPack = __ballot_sync(active, data[i] > threshold);
    if (warpLane == 0)
        output[i/32] = bitPack;
}
```

上面代码在dataLen不是warpSize的整数倍时会有divergence。但是`__activemask()`只针对当前active的lane，并不一定包括所有要经过这个地方的lane。比如说有mask为`0xffff0000`的16个线程要执行`__activemask()`，volta前的架构会对这16个lane统一返回`active=0xffff0000`。但到了volta，可能这16个lane是分两组过去的，比如说前8个得到`active=0xff000000`，后8个得到`active=0x00ff0000`。那这个代码就有问题了。文档中给出的一个正确写法如下：

```
for(int i=warpLane; i-warpLane<dataLen; i+=warpSize) {
    unsigned active = __ballot_sync(0xFFFFFFFF, i < dataLen);
    if (i < dataLen) {
        unsigned bitPack = __ballot_sync(active, data[i] > threshold);
        if (warpLane == 0)
            output[i/32] = bitPack;
    }
}
```

`__ballot_sync` 就保证了只要是要执行这段代码的分支都必须同时vote，这样就避免了上面说的分组通过各自都只拿到一部分mask的问题。

这里面其实还涉及很多细微的问题，大家有兴趣的话可以研究一下programming guide中的相关章节。比如Warp Vote Functions，Warp Shuffle Functions等。

小结

Independent thread scheduling是一个初看起来好像没什么大用，但底层却会发生翻天覆地大变化的feature。它对整个程序运行流程的影响其实是方方面面的，这里只是简单的列了一部分。分支的处理其实一直是GPGPU的一个难点，NV愿意推动这样一个feature，确实是展示了其在技术上的野心。对当前GPGPU擅长的高并行度的处理任务，这个feature意义很有限，似乎是有些得不偿失。但是，如果你想要把一些CPU程序移植到GPU上，这个feature在很多时候就会显得雪中送炭。很多非数值程序都有复杂而密集的分支，比如常见的检索、排序、图的遍历等等。当然，其实也有很多数值程序也涉及到分支，比如常见的sparse矩阵操作中就存在很多这种情况。另外，很多CPU的多线程程序都涉及到比较复杂的加锁和同步操作，这也是GPU不擅长的地方。其实在Cooperative Group以前，block间是没有peer-to-peer的同步操作的，因为不同block的生命周期就不确定有overlap，做同步没意义。Cooperative Group通过保证一些block一定是同时运行的，这才有同步和数据交互的可能。而Independent thread scheduling使得Cooperative Group不仅可以上到block间做同步，还可以下到warp内的部分线程同步。这在灵活性上是一个很大的进步。

通过引入Independent thread scheduling，GPU对于一些问题的处理能力还是能有不小的提升，从某种意义上说是尽力补齐性能上的一些短板，或者是让一些程序的实现变得更简单，从而让GPU的应用面更宽，适应性和通用性更强，进而为开拓新的市场提供了必要的技术基础。