CUDA微架构与指令集（4）-指令发射与warp调度

https://zhuanlan.zhihu.com/p/166180054

None Tue Jun, 01 03:14

这次聊一聊CUDA指令的发射和warp调度问题。这个问题牵涉到大量硬件设计的细节，我只是按我的理解来聊一聊，误解和错漏恐怕在所难免，请大家注意自行鉴别，也欢迎大家就感兴趣的点进行讨论。

今天好像又是长文，先列个大纲，大家如果对某些部分不感兴趣，可以跳着看。

首先我会介绍一下指令发射的基本逻辑，主要是指令发射需要满足的条件，几代架构发射指令的一些简单描述等等。然后会简单介绍一下control codes，主要是它与指令发射和warp调度的关系。再就是简单介绍下warp Scheduler的功能。最后，会略微讲一讲峰值算力的计算方法和达成条件。

指令发射的基本逻辑

首先，简单回顾一下CUDA程序的等级结构。每个Kernel有一个grid，下面有若干个block，每个block有若干个warp。同一个block的warp只能在同一个SM上运行，但是同一SM可以可以容纳来自不同block甚至不同grid的若干个warp。我们这里要聊的指令发射和warp调度的问题，就是指同一个SM内同一个warp或是不同warp的指令之间是按照什么逻辑来调度运行的。

然后，说一下指令发射的一些基本逻辑：

1. 每个指令都需要有对应的功能单元（Functional Unit）来执行。比如执行整数指令的单元，执行浮点运算指令的浮点单元，执行跳转的分支单元等等。功能单元的个数决定了这种指令的极限发射带宽（在没有其他资源冲突时）。
2. 每个指令都要dispatch unit经由dispatch port进行发射。不同的功能单元可能会共用dispatch port，这就意味着这些功能单元的指令需要通过竞争来获得发射机会。不同的架构dispatch port的数目和与功能单元分配情况会有一些差别。
3. 有些指令由于功能单元少，需要经由同一个dispatch port发射多次，这样dispatch port是一直占着的，期间也不能发射其他指令。比较典型的是F64指令和MUFU特殊函数指令。
4. 每个指令能否发射还要满足相应的依赖关系和资源需求。比如指令 `LDG.E R6, [R2] ;` 首先需要等待之前写入 `R[2:3]` 的指令完成，其次需要当前memory IO的queue还有空位，否则指令也无法下发。还有一些指令可能有conflict的情况，比如shared memory的bank conflict，register的bank conflict，atomic的地址conflict，constant memory在同一warp内地址不统一的conflict等等，这些都有可能导致指令re-issue（甚至cache miss也可能导致指令replay）。这些情况会不会重复占用dispatch port发射带宽我暂时还没仔细研究。

5. 在有多个warp满足发射条件的情况下，由于资源有限，需要排队等待发射，warp scheduler会根据一定的策略来选择其中的一个warp进行指令发射。
6. 当前CUDA的所有架构都没有乱序执行（Out of order），意味着每个warp的指令一定是按照运行顺序来发射的。当然，有的架构支持dual-issue，这样可以有两个连续的指令同时发射，前提是两者不相互依赖，而且有相应的空余资源（比如功能单元）供指令运行（对kepler来说不一定是不同类的功能单元，后面会具体分析）。另外一个显而易见的要求是双发射的第一个指令不能是分支或跳转指令。

接着我们要简单讲讲Kepler，Maxwell，Turing三代架构的指令发射和warp调度逻辑。先看看这三个微架构典型chip的SM示意图（图来自各自whitepaper，注意SM的名称有点变化）：

Kepler SMX(GK110)　Maxwell SMM(GM204)　Turing SM(TU102)

简单对比一下其中的warp scheduler和dispatch unit：

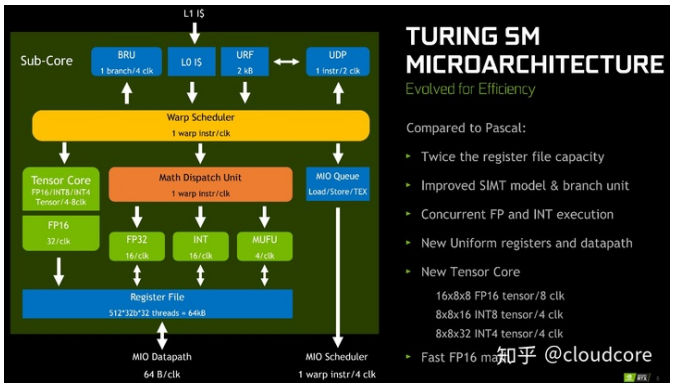
Architecture	Cuda Core	Warp Scheduler	Dispatch Unit
Kepler(GK110)	192	4	8
Maxwell(GM204)	128	4	8
Turing(TU102)	64	4	4

按我的理解，每个warp scheduler每cycle可以选中一个warp，每个dispatch unit每cycle可以issue一个指令。然后具体讲讲几个架构的区别（有些是我猜的，可能不准确）：

1. Kepler的SMX有192个core，但是core和warp没有固定的从属关系，相当于每个warp都可以运行在不同的32core组上（？这个需要进一步确认）。192=32*6，所以每个cycle最少需要发6个指令才能填满所有core。可是4个warp scheduler只能选中4个warp。所以Kepler必须依赖dual issue才能填满所有cuda core，而且由于core多且与warp没有对应关系，kepler的dual issue不一定是发给两个不同的功能单元，两个整数、两个F32或是混合之类的搭配应该也是可以的，关键是要有足够多的空闲core。每个warp scheduler配了两个dispatch unit，共8个，而发射带宽填满cuda core只要6个就够了，多出来的2个可以用来双发射一些load/store或是branch之类的指令。
2. Maxwell开始，SM的资源就做了明确的分区，每个warp都会从属于某个分区，各分区之间有些功能单元（比如cuda core和F64单元，SFU）是不共享的。Maxwell的SMM有128个

core，分成4组。每组有一个warp scheduler，2个dispatch unit，配32个CUDA core。这样每cycle发一个ALU指令就可以填满cuda core了，多出来的dispatch unit可以发射其他memory或是branch指令。由于功能单元做了分区，没有冗余了，这样Maxwell的dual-issue就不能发给同样的功能单元了。

3. Turing的SM的core数减半，变成64个，但还是分成4个区，每个区16个core，配一个warp scheduler和一个dispatch unit。这样两个cycle发一个指令就足够填满所有core了，另一个cycle就可以用来发射别的指令。所以Turing就没有dual-issue的必要了。从Volta开始，NV把整数和一些浮点数的pipe分开，使用不同的dispatch port。这样，一些整数指令和浮点数指令就可以不用竞争发射机会了。一般整数指令用途很广，即使是浮点运算为主的程序，也仍然需要整数指令进行一些地址和辅助运算。因此，把这两者分开对性能还是很有一些帮助的。但是，这里还是要澄清一下，不是所有的浮点和整数都是分开的。这里贴一张Hotchips里NV介绍Turing的图（图上没画F64，应该是和Tensor Core、F16在一块）：



据我所知，Volta和Turing的F64、F16和Tensor Core都是用的同一个dispatch port。然后F32一组（IMAD也放在这组，大概是要共享mantissa的那个乘法器），MUFU（就是特殊函数指令），其他ALU（包括整数算术运算和移位、位运算等等，但是不包括IMAD）一组，然后memory指令一组，branch指令一组，然后Turing的uniform datapath的指令一组，这些都各自有dispatch port。不同组的可以隔一个cycle发射，同组的就要看功能单元数目，最少是隔2个cycle。

Control codes

【补充】：感谢评论区 @彭飞 提醒和补充，这里Control codes里有些说法可能不太准确。不过我也没有很官方的输入，转述可能又会有我自己的观念带入，所以大家可以移步评论区自己体会~

前面讲SASS指令集分类的时候，就简单提到过control codes。但是由于control codes和指令本身的功能关系不大，更多的是影响指令的发射和warp调度，因此把它放到这里来讲。

从Kepler开始，指令中就带有Control codes。Kepler的架构略显久远，格式又不太一样，信息也不太全，所以这里就不聊了。Maxwell后control codes的功能更加丰富了，而且在Scott Gray的文章中非常具体详尽的描述了各个域的含义和功能。Pascal的指令集与Maxwell基本一致，control codes也没什么变化。Volta和Turing，包括最新的Ampere，只是把control codes编码到每条指令中，具体内容和bit对应关系其实也没有变，所以我这也不做架构的区分。这里主要复述一下Scott Gray的表述，但主要侧重它与指令发射和warp调度的关系。

这里我以Turing为例，选了一段程序，把对应的control codes 写在前面（这里我用的格式与Scott有点区别，主要是看起来更方便）：

```
1: [----:B------:R-:W0:-:S04]    S2R R113, SR_CTAID.Y ;
2: [----:B------:R-:W1:-:S04]    S2R R0, SR_CTAID.Z ;
3: [----:B------:R-:W3:-:S01]    S2R R106, SR_TID.X ;
4: [----:B0-----:R-:W-:-:S02]    IMAD.SHL.U32 R113, R113, 0x4, RZ ;
5: [----:B-1----:R-:W-:Y:S03]    IMAD.SHL.U32 R0, R0, 0x4, RZ ;
6: [R---:B------:R-:W-:-:S02]    IADD3 R107, R113.reuse, 0x1, RZ ;
7: [R---:B------:R-:W-:-:S01]    IADD3 R109, R113.reuse, 0x2, RZ ;
8: [R-R-:B------:R-:W-:-:S01]    IMAD R2, R113.reuse, c[0x0][0x1a8], R0.reuse ;
9: [----:B------:R-:W-:-:S01]    IADD3 R111, R113, 0x3, RZ ;
```

我这里采用的显示形式是类似 `R-R-:B------:R-:W-:-:S01` 这种，用冒号':'分隔开6个域：Register Reuse Cache（4bit，对应4个slot，有reuse就写R，没有就'-'），Wait Dependency Barrier（6bit，B+6个数，有等待就写上对应的barrier号0-5，否则写“-”），Read Dependency Barrier（3bit，R+设置的barrier号，不设置写“-”），Write Dependency Barrier（3bit，W+设置的barrier号），Yield Hint Flag（1bit，“Y”表示yield，否则写“-”），Stall Count（4bit，S+十进制的stall的cycle数）。

这里与scott的标记的区别一是显式的写了reuse，而不是只写在后面汇编文本里。这样看起来更直观且容易看到slot的对应关系，因为有的predicate夹在中间容易搞混。二是wait barrier的每个bit我拆开了，这样肉眼更容易与前面设置barrier的R#或是W#对应上。然后就是每个域前加了提示字符，BRWS之类，毕竟有的时候看久了也容易恍惚，这样区分比较明显，不容易搞混。

【更正1】：这里包括后文说的Dependency Barrier也许更合理的称呼是Scoreboard，在profiler里有short scoreboard和long scoreboard之分，应该指的就是这个。

【更正2】：scoreboard的编号我调整过一次。由于最开始我沿用了scott的1-6的编号方式，但后来我还是调整为更符合原语境的0-5。首先这是原编码的值，其次DEPBAR后会显式的接SB0~SB5这种数，所以后来我就统一改为0-5。

下面具体介绍control codes每个域的含义。

Register Reuse Cache

Register Reuse Cache有4bit。每个指令有4个slot，每个register的source operand的位置对应一个slot（predicate好像不算）。我暂时还没碰到4个source operand的指令，所以有一个bit好像一直没用到。Reuse的用法：如果当前指令某个slot的register还会被下一个指令的同一个slot读取，那就可以reuse当前指令读取到的register内容。Reuse的作用主要就是减少GPR的读取，一来可以减少register bank conflict，二来应该也能省一点功耗。比如前面代码中Line6的`IADD3`的第一个源操作数是`R113`，而Line7的第一个源操作数也是`R113`，所以可以reuse。同理，line7、line8同样位置的`R113`都可以reuse。但是line8的`R0`为什么reuse呢？我也没搞懂。Reuse是唯一在官方反汇编中出现的control codes，但也有很多疑点。

我的几个猜测：首先reuse是个cache，某种意义上是个hint，也就是说就算set了，reuse不了应该也不至于出错。第二，reuse cache的位置应该是位于所谓的operand collector。这个应该是很多功能单元公用的，所以不太可能是同种功能单元的指令才能reuse。另外，load store类的指令没看到reuse，好像是不用collector。第三，如果切换到别的warp，register是不同的空间，那reuse cache就失效了。所以reuse应该是需要当前warp的指令连续发射。第四，reuse既然是register的cache，那原来register的位置出现了constant memory或是immediate会不会使reuse失效呢？感觉好像不会。那`RZ`会不会呢？`UR`会不会呢？我也没仔细研究过。

Reuse这里还有很多奇怪的问题，有的我怀疑是编译器的bug。比如这种：

```
1: [R---:B------:R-:W-:-:S02]     FSETP.GT.AND P1, PT, R9.reuse, c[0x0][0x18c], PT ;
2: [----:B------:R-:W-:-:S02]     LEA.HI.X.SX32 R4, R4, c[0x0][0x164], 0x1, P3 ;
3: [R---:B------:R-:W-:-:S02]     LEA R5, P3, R0.reuse, R5, 0x2 ;
4: [----:B------:R-:W-:-:S02]     FSEL R9, R9, c[0x0][0x18c], !P1 ;
5: [----:B------:R-:W-:-:S02]     LEA.HI.X R4, R0, R4, R3, 0x2, P3 ;
```

其中Line1的`R9`只能在Line4中得到reuse，但之前的这个slot已经被Line2的`R4`用过了，应该是没法reuse的。Line3也是一样，`R0`只能在Line5中被reuse，但中间被Line4打断了。如果我们把Line4提到第2行前面去，这好像就顺理成章了：

```
1: [R---:B------:R-:W-:-:S02]     FSETP.GT.AND P1, PT, R9.reuse, c[0x0][0x18c], PT ;
4: [----:B------:R-:W-:-:S02]     FSEL R9, R9, c[0x0][0x18c], !P1 ;                 // 原先在第4行，依赖Line1的输出P1，如果放在这就要等
barrier
2: [----:B------:R-:W-:-:S02]     LEA.HI.X.SX32 R4, R4, c[0x0][0x164], 0x1, P3 ;
3: [R---:B------:R-:W-:-:S02]     LEA R5, P3, R0.reuse, R5, 0x2 ;
5: [----:B------:R-:W-:-:S02]     LEA.HI.X R4, R0, R4, R3, 0x2, P3 ;
```

由于Line4的`FSEL`用到了Line1的输出P1，为了把依赖指令移到后面以减少stall，编译器做了相应的指令调度，但是reuse却没有重新生成。所以我怀疑这个是编译器的bug，存疑中。

Wait Dependency Barrier

Wait Dependency Barrier有6bit，每个bit表示是否需要等待对应的dependency barrier。每个线程有6个dependency barrier，每个barrier都可以被后面的Read或Write操作设置上。设置wait dependency barrier是等待依赖的其中一种方式。SASS里面还有一个对应的指令，如 `DEPBAR.LE SB0, 0x0, {2,1} ;`，两者自由度不太一样。control codes里设置的barrier只能是bool形式，要么dependency resolved，要么就是not ready。而 `DEPBAR` 可以等待有计数的barrier。比如发了8个memory指令，其实设置的是同一个dependency barrier，每发出一个计数加1，每回来一个计数减1。这样 `DEPBAR` 可以等计数降到6就说明前面2个指令已经到位了，对应的load结果就能用了。而如果用control codes来等待，就只能等计数降到0，也就是8个都ready才行。

dependency barrier有一个和分支指令强相关的地方，比如不确定的跳转指令（带predicate的 `BRA`，或是 `BRX` 这种指令）需要等待当前所有已设置的dependency barrier都到齐才行。否则后面多个分支的代码可能用到这个barrier，但又不一定都会等待。这个可能是编译器在处理上有一些图方便的地方。有些情况是可以把wait后移到对应使用指令上的，这样延迟更容易被隐藏。只是有时候编译器拿不到足够的信息，为保证正确性就统一在跳转的时候等了。

Read Dependency Barrier

Read dependency barrier有3bit，表示需要设置的6个barrier中对应的索引（0~5，对应barrier 1-6，如果不需要设置barrier，就设置为0b111）。Read dependency barrier主要是一些指令不会在一开始就把所有操作数读进去，所以需要hold住GPR的值，防止后面的指令在它读取其内容之前把GPR改掉。使用Read dependency barrier的主要是memory类的指令，但是一些转换指令如 `F2I/I2F` 之类好像偶尔也能见到。

Write Dependency Barrier

Write dependency barrier与read dependency很类似，也是3bit，后面跟barrier索引。注意Read和Write两者用的dependency barrier资源是一样的，也都是上面wait的那6个。Write dependency barrier比较好理解，就是某个指令要把操作结果保存到某个GPR或是predicate中，使用barrier进行保序可以防止出现data race。不过这主要针对的是不定长latency指令。如果一个指令的latency是确定的（或者有不太长的上限），那用后面提到的stall cycle停足够长时间就可以保证没有race。

Yield Hint Flag

Yield hint flag是1bit。如果Yield，就表示下一个cycle会优先发射其他warp的指令。前面聊reuse的时候我们也说了，reuse是需要连续发射同一个warp的指令的。所以reuse和yield是不会联用的。另外，SASS有一个专门的指令 `YIELD`，感觉上是一样的功能（【注】：经评论区提醒，功

能还是不一样的，具体请移步评论区）。Yield的存在仿佛暗示warp scheduler不是round robin的选择warp，而是倾向于一直往同样的warp里发射指令（如果可以一直发射的话，有stall肯定就尽量切走了）。但是这个东西我也不太确定，没仔细测过。如果真是这样，yield的作用就是保持各个warp之间的进度均衡，否则在barrier之类的指令上会有较大的等待开销。而且如果退出时间差很大，也会导致一些资源不能尽快回收以容纳新的block。

【补充】：根据评论区所说，也许yield这个bit就是stall count的高位。只是假如这个bit不为0，那stall的cycle会>16，相当于warp被切换的概率也会大大增加。两者之间的具体含义应该还和具体指令有关，不同类型的指令也许是不一样的。这个问题我还没有具体研究过。先存疑。

Stall Count

Stall count有4bit，表示当前指令后需要stall指令发射的cycle数，然后再决定是不是要继续发射。这个cycle数受到极限发射带宽的约束，很多时候可以用来反推功能单元的分组和数目。比如Maxwell下有双发射，所以可以stall 0 cycle，在反汇编中会用大括号组合起来，比如这种：

```
/*0008*/        MOV R1, c[0x0][0x20] ;         /* 0x4c98078000870001 */
/*0010*/   {    MOV R5, c[0x0][0x148] ;        /* 0x4c98078005270005 */
/*0018*/        S2R R0, SR_TID.X         }
                                               /* 0xf0c8000002170000 */
```

Kepler的双发射不限定功能单元，Turing架构没有双发射，就都没有这种形式了。

我们再找个Turing的例子来看一下：

```
[----:B------:R-:W-:-:S01]       IADD3 R43, -R43, R28, R36 ;                  // STS与IADD3不同功能单元, stall 1 cycle
[----:B--2---:R0:W-:-:S04]  @!P3 STS.128 [R31.X16], R8 ;                      // 两个STS间至少stall 4 cycle,
[----:B---3--:R1:W-:-:S04]  @!P5 STS.128 [R31.X16+0x2100], R16 ;
[----:B----4-:R2:W-:-:S01]  @!P4 STS.128 [R51.X16+0x2200], R12 ;
[R-R-:B0-----:R-:W-:-:S01]       IADD3 R11, -R28.reuse, 0x20, R43.reuse ;     // IADD3是Integer pipe, IMAD是F32 pipe, 不共dispatch
port, stall 1 cycle
[----:B------:R-:W-:-:S01]       IMAD.IADD R8, R43, 0x1, -R28 ;
[----:B------:R-:W-:Y:S02]       IADD3 R9, -R28, 0x10, R43 ;                  // IADD3和ISETP同属普通Integer pipe, stall 2 cycle
[R---:B------:R-:W-:-:S01]       ISETP.GE.U32.AND P3, PT, R0.reuse, R11, PT ; // ISETP与LDG属不同功能单元, stall 1 cycle
[----:B-1----:R0:W5:-:S01]  @!P2 LDG.E.128.CONSTANT.SYS R16, [R24+0x200] ;
```

前面我们已经讲过，Turing的两个周期发一个ALU指令就可以用满cuda core了，所以同组ALU指令至少stall 2 cycle。如果两个指令间只stall 1 cycle，说明这两个指令应该分属不同的功能单元（或者说不共用dispatch port），可以分开发射。如果stall的时间更长，说明其发射带宽比较低，比如 `LDG/LDS/STG/STS` 这种memory指令都是4 cycle才能发一个。但是也有一些奇怪的例子：

```
[----:B------:R-:W-:-:S01]      IMAD.WIDE.U32 R2, R5, 0x3, R2 ;
[----:B--2---:R-:W0:-:S01]      I2F.S16 R4, R4 ;
[----:B---3--:R-:W1:-:S01]      I2F.S16 R6, R6 ;
[----:B----4-:R-:W2:-:S01]      I2F.S16 R7, R7 ;
[----:B0-----:R-:W-:-:S01]      FADD R8, R4, -c[0x0][0x194] ;
[----:B------:R-:W-:-:S01]      SHF.R.S32.HI R4, RZ, 0x1f, R5 ;
[----:B-1----:R-:W-:-:S02]      FADD R9, R6, -c[0x0][0x198] ;
```

`I2F`居然是挺机关枪，每cycle都可以发。可是按programming guide中的Instruction Throughput表，`I2F`如果不涉及F64，是1/4的throughput。难道是有特别的queue做buffer？这个也没太明白，有待进一步研究。

其他

Control codes一个比较容易忽视的问题是它与predicate是独立的。也就是说不管加不加predicate，control codes的作用是不会改变的。因为本身control codes很多东西是编译期决定的，如果按运行期的predicate来定是否启用control codes，有些代码的正确性就容易出问题。

Control codes是个很复杂的问题，有些东西真是纯逆向了。我也没仔细研究过。大家有兴趣可以看看scott gray的maxas的实现，或许会有一些启发。

Warp Scheduler

Warp scheduler的作用就是管理一系列的warp，在那些满足条件的warp中选中一个来发射指令。就绪可以发射指令的warp就是eligible，不满足发射条件的warp就是stalled。导致warp不能发射指令的原因有很多种。我根据NSight Visual Studio Edition中Issue Stall Reasons中的描述，大致搬运翻译一下：

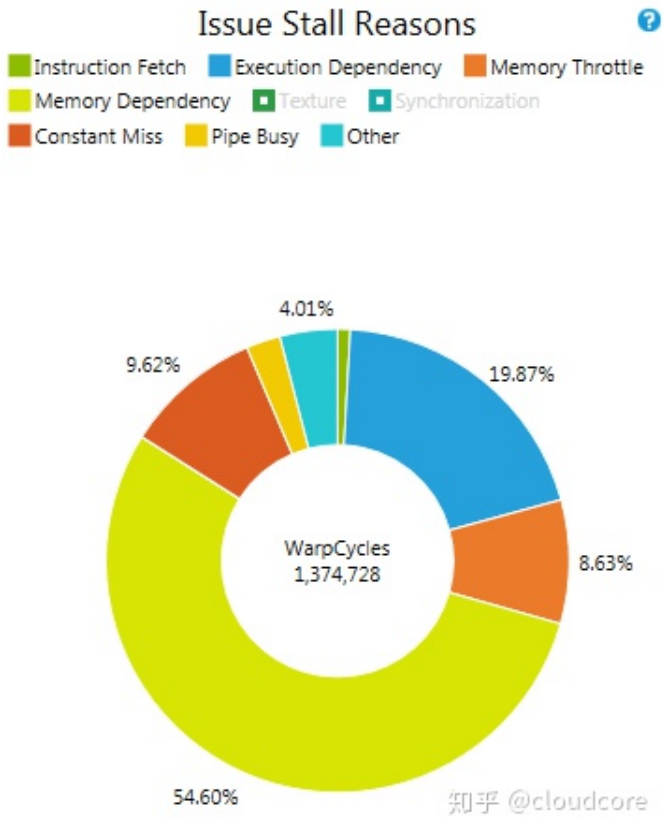
- **Pipeline Busy**：指令运行所需的功能单元正忙。
- **Texture**: Texture单元正忙，或者说已经下发的request过多。
- **Constant**：Constant cache的miss。一般说来，多数情况下constant cache的hit rate还是很高的，所以一般只会在第一次access的时候miss。
- **Instruction Fetch**：Instruction cache的miss。与constant cache的miss类似，一般只有第一次运行到的地方才容易miss。比如`BRA`新跳转到的地方，或是Instruction cache的cache line的边界处。
- **Memory Throttle**：有大量memory操作尚未完成，导致memory指令无法下发。可以通过合并memory transactions来缓解.
- **Memory Dependency**: 由于请求资源不可用或是满载导致load/store无法执行，可以通过内存访问对齐和改变access pattern来缓解。这个与memory throttle的细微差别我还没仔细研究过。

- **Synchronization**：warp在等待同步指令，如cuda C里的`_syncthreads()`，对应SASS里的`BAR`指令。
- **Execution Dependency**：输入依赖关系没解决。简单说，就是输入值未就绪，就是在等control codes里的dependency barrier。单个warp内通过增加ILP可以减少依赖型stall。如果ILP不够用，这个stall就会形成额外的latency，只能用TLP来隐藏了。

在Profiler里提供的Turing的performance counter里，还有两个当前warp不能发射指令的原因：

- `stall_not_selected`：warp当前虽然eligible，eligible的warp超过一个，当前的未被选中，所以不能发射。
- `stall_sleeping`：这个一般是用户自己调用sleep功能让该warp处于睡眠状态。

贴一个Nsight里的stall reason统计图：



Warp Scheduler的另一个关键功能是在eligible里选一个来发射。很多早期的书上都说选warp是round-robin，就是所谓的轮询，发一个换一个。当前的几代架构，至少maxwell之后，我觉得应该不是这个策略了。前面聊reuse，yield和stall的时候也提到了，如果是round-robin，这些东西都会显得很奇怪了。所以我感觉它应该是比较aggresive的往同一个warp发射指令，除非stall了。当然，中间如果没有yield，那stall 2个cycle的时候中间那个cycle能去发射别的warp吗？这个我也有点迷，有机会再仔细研究下。

【补充】：我仔细再想了想，stall 2cycle的时候中间那个cycle应该是可以发射其他不用operand collector的单元，比如memory和branch。网上看到一些说法，有些S2R的指令好像也是走的memory的pipe，那应该也可以发射。这个应该是可以写一个micro-benchmarking的case验证一下，只是我手上没有Turing的卡，也没法测了。但是我感觉只要它不影响reuse的cache，那就应该可以抽空发射。这也是一个比较符合性能需求的模式。

Eligible的warp数是影响峰值性能的关键表征之一，如果每个cycle都至少有一个eligible warp，那功能单元基本就会处于满载状态，性能一般也会比较好。这也是occupancy真正起作用的方式。

关于峰值算力的问题

最后要略微展开聊一下峰值算力的问题。NV的GPU经常用CUDA Core这个词来表示算力强弱，这其实是Int32/Float32功能单元的marketing叫法。算力评估常用的单位是**FLOPS**，表示**FL**oat **OP**erations per **S**econd（Flops有时候也用作Flop的复数，注意鉴别）。对于F32而言，`FADD/FMUL`都是一个指令一个flop，`FFMA`一个指令同时算乘加所以是两个flop。所以一般NV的GPU的F32峰值算力计算方法为：

```
SM数 * 每SM的Core数 * 2 * 运行频率
```

最后结果常用GFlops或是TFlops表示。其中乘以“2”是因为`FFMA`是两个flop。比如说Maxwell架构每个SM有128个CUDA core，每个SM每cycle可以发射128条Int32**或**Float32指令（两种指令不能同时发射，所以是或）。Maxwell架构的GTX 980有16个SM，共16 * 128=2048个CUDA core，每个cycle能做2048 * 2Flops/FFMA=4096 Flops。980的base clock是1.126GHz，相当于每个cycle是1/1.125G秒，每个Cuda Core每秒可以发射1.126G条指令，整个GPU就是1.126G*4096=4.5TFlops。放在一起算，就是峰值算力等于：

```
16 SM * 128 Core/SM * 2 Flop/Core/Cycle * 1.126G Cycle/second = 4.5TFlops.
```

当然，商家为了宣传，常用boost clock算峰值算力，非公版的频率也会有些差别，所以这个值会有些变化。另外，这里用的是F32浮点峰值做例子，如果你的任务不需要浮点运算或是精度不是F32，这个值就意义不大，需要转换成你需要的那个操作。现在AI处理器常常宣传峰值是多少FLOPs，或是多少IOPs，一般也会限定是F32，F16或是I8之类。因为每种操作对应的指令是不一样的，峰值当然也可能不一样。顶级HPC计算卡F64一般是F32的一半，但消费级显卡F64多数会有阉割。如果没有TensorCore而用packed F16（把两个F16塞到一个32bit GPR里同时运算），F16峰值性能通常是F32的两倍。有TensorCore时则要另行计算，要看具体TensorCore的数目和指令带宽，还有能不能和其他指令同时发射等。其实不看Tensor core的话，满血版一般有：F64:F32:F16=1:2:4，正好与占用的GPR成反比，这个其实是与GPR的带宽有很大的关联的，一般满血版的卡的功能单元配比就会尽量按极限的GPR带宽来设计。

这里贴一个Ampere的white paper中V100和A100的几种峰值性能对比：

	V100	A100	A100 Sparsity[1]	A100 Speedup	A100 Speedup with Sparsity
A100 FP16 vs V100 FP16	31.4 TFLOPS	78 TFLOPS	NA	2.5x	NA
A100 FP16 TC vs V100 FP16 TC	125 TFLOPS	312 TFLOPS	624 TFLOPS	2.5x	5x
A100 BF16 TC vs V100 FP16 TC	125 TFLOPS	312 TFLOPS	624 TFLOPS	2.5x	5x
A100 FP32 vs V100 FP32	15.7 TFLOPS	19.5 TFLOPS	NA	1.25x	NA
A100 TF32 TC vs V100 FP32	15.7 TFLOPS	156 TFLOPS	312 TFLOPS	10x	20x
A100 FP64 vs V100 FP64	7.8 TFLOPS	9.7 TFLOPS	NA	1.25x	NA
A100 FP64 TC vs V100 FP64	7.8 TFLOPS	19.5 TFLOPS	NA	2.5x	NA
A100 INT8 TC vs V100 INT8	62 TOPS	624 TOPS	1248 TOPS	10x	20x
A100 INT4 TC	NA	1248 TOPS	2496 TOPS	NA	NA
A100 Binary TC	NA	4992 TOPS	NA	NA	NA

要达到F32的峰值性能，需要满载发射 `FFMA` 指令，这是很苛刻的条件。首先，其他与 `FFMA` 共用dispatch port的指令，每发射一个都会挤占 `FFMA` 的发射机会。其次，由于多数情况下数据要从memory中来，而memory操作比ALU慢很多，常常导致指令操作数无法就绪，从而有些周期没有 `FFMA` 指令可发。同时还有其他一些overhead或是occupancy问题导致有些SM无法满载，从而无法达到峰值。一般说来，实际应用中，较大尺寸的矩阵乘法（GEMM）是难得的能接近峰值性能的程序，有些实现能到98%峰值的效率。但多数实际应用效率都远不及此，很多memory bound程序能到10%就很不错了。超算TOP500排名中，多数HPL效率都是50~60%左右，更接近实际应用的HPCG效率一般都在2~3%左右。虽有规模大导致的互联开销原因，但总体来讲实际应用的峰值性能离极限值还是差距很大的。

每种类型的指令都有一个峰值性能，那是不是能同时达到呢？基本是不能。对于共用dispatch port就不说了，要相互竞争发射机会，发射一条这种指令就少发射一条那种指令，所以显然不能同时到达峰值。如果是不同的dispatch port呢？理论上可以，但是实际上也会比较难。比如说Turing的F32和I32，首先I32的2IOP指令IMAD是和F32一伙的，相互竞争dispatch port，所以两个不能同时到达峰值。剩下的IADD3或是LEA之类的指令理论上可以与F32的并行，倒是有机会冲一冲。只不过多数实际应用中很难做到这么好的运算配比，而且register的bank conflict之类应该也会大大限制这两种指令的同时运行。另外，即使两者配合完美，它还是需要省出一些发射带宽给其他配套指令（比如memory load），不可能完全占满。

到今天这期就算是把指令集和微架构的总体叙述讲完了。之后会把我原来写的汇编器整理一下，然后零敲碎打的做一些micro benchmarking。先留个坑吧！做不做再说~

这期的内容颇有些不明确的地方，欢迎大家提出意见和反馈~