CUDA随笔

- Linux Perf

1.写的时候获得的知识

- 有的循环展开会占用微量寄存器，并且限制SM上的block数，这会表现在Block Limit Registers和Block Limit SharedMem指标上。很奇怪。
  - 这部分循环展开可能就是编译器没有自动帮我做的。
- 如果寄存器的值异常大，可能是因为寄存器没有重新初始化，导致随机初始化了。
- 选择一定数量的寄存器数组，首先考虑的是对 活跃warp数/SM 的影响。
- CUDA的存储层级：Global Mem -> L2 cache, L2 cache -> texture cache，texture cache -> registers，registers -> Shared Mem，Shared Mem -> registers，registers -> instruction operand cache， registers -> Global Mem
- GEMM选择SMEM_Y是128的原因是：我们首先选择寄存器块为8，这样可以控制单线程总寄存器用量在128以内，从而控制活跃warp数。然后一个线程处理8 * 8的C矩阵块，如果启动16 * 16的线程块，则一个block处理 (16*8) * (16*8) 的C矩阵块。
- 展开后循环的大小，最好不要超过指令缓存的大小。
  - maxwell里指令缓存的size为8KB，这是测得的。
  - maxwell的指令好像是每三个有用指令跟一条control code，都是8字节。
  - 还要考虑指令对齐
  - 考虑循环展开因子是8的话，64线程的block，会产生64*8=512条FFMA指令，再加一些内存和整型算术指令（约40条），就肯定小于3/4 * 1024
- GEMM是计算密集型算子，所以，宏观思路应该是用计算隐藏内存加载的耗时，要算到512条FFMA能不能隐藏访存耗时。
  - 如果是访存密集型，则应该考虑如何减少访存耗时。
- 利用一切机会提高计算指令和非计算指令的比率。

2.SASS和PTX的知识

SM_52和Compute_52

```
nvcc cudatest.cu -o cudatest -gencode=arch=compute_52,code=\"sm_52,compute_52\"
-gencode=arch=compute_75,code=\"sm_75,compute_75\"  -
gencode=arch=compute_86,code=\"sm_86,compute_86\"
```

这里写了很多`-gencode=*`，用来控制具体要生成哪些PTX和SASS代码。`arch=compute_30`表示基于`compute_30`的**virtual GPU architecture**，但它只是我们前面提到的控制使用feature的子集，并不控制是否生成具体PTX代码。后面的`code=\"sm_30,compute_30\"`才表示代码生成列表。其中`sm_30`**表示基于**`sm_30`**的架构生成相应SASS代码**，`compute_30`**表示基于**`compute_30`**的虚拟架构生成相应PTX代码，这个必须与前面**`arch=*`**一致。**

所有的代码生成后会被打包成FatBinary，内嵌在程序中供调用。程序运行时driver会去判断是否有编译好的对应架构的SASS版本，如果没有就从可选的PTX中JIT编译一个。

PTX 指令集小tips

H = half = fp16

F = float = fp32

D = double = fp64

I = interger

B = bit

P = Predicate

U = Uniform

S = Special

SASS指令集

- RZ就是R255，常0。寄存器的编码是8bit的，也就是从R0~R255。
- SR：Special Register。SR_TID.X是threadIdx.x。 SR_CTAID.X是blockIdx.x。SR_LANEID是lane_id。
- S2R就表示把SR载入到Regiter。
- blockDim和gridDim会存在常量寄存器里，自动映射。
- ULDC：load from ConstantMem into a Uniform Register。
  - UR*就表示Uniform 寄存器。
  - Uniform的reg和predicate是公用的。Uniform的计算是scalar，也就是一个warp只执行一次，而不是32个线程都执行一次。
- c[0x0][0x160]是核函数的第一个参数，依次从c[0x0][0x164]是第二个参数（假设4字节）。
  - 160 v0, 164 v1, 168 v2, 170 n_unroll,

```
// 后面那个!PT好像是指PT寄存器是否取反的意思
ISETP.GE.AND P0, PT, R0, 0x1, PT ;        // res = R0>=1? P0=PT&res
IADD3 R2, R0.reuse, -0x1, RZ ;            // R2 = n_unroll-1
LOP3.LUT R0, R0, 0x3, RZ, 0xc0, !PT ;     // R0 = R0&3
ISETP.GE.U32.AND P1, PT, R2, 0x3, PT ;    // res = R2>=3? P1=PT&res
ISETP.NE.AND P0, PT, R0, RZ, PT ;         // P0 = R0!=RZ
IADD3 R4, -R0, c[0x0][0x170], RZ ;        // R4 = n_unroll-R0
```

3.MicroBenchmark

3.1 写microbenchmark时注意的点

- 内核代码跑两次，忽略第一次，以此避免指令缓存冷启动造成的影响。
- 内核代码要小到可以完全装入L1 cache？
- 计时代码围绕内核代码，写在首尾。计时值先存在寄存器里，最后写回global mem，以避免全局内存访问干扰时序测量。
- 片外资源的latency可能取决于执行代码的TPC（Thread Processing Cluster），要在所有TPC上测量，再取均值。
-

- 3060，L2cache, 3072KB。L1指令cache，64KB。L0指令
-

4.TensorCore加速原理

> NVIDIA Tensor Core微架构解析 - 知乎 (zhihu.com)

- TensorCore可以在一个周期内做一次4*4矩阵的乘法

90.SIMT核心

> 从GPU编程到SIMT核心 - 知乎 (zhihu.com)

- SIMT核心被分为前端（绿色）、后端（黄色），并配有三个独立的调度器，分别位于I-Cache、Issue、Operand Collector部分。
- 下图的SIMT架构对应于一个SM，而CUDA core(SP)则是一组ALU流水线。

SIMT核心前端

- **取指调度器**：负责将取到的指令送到**I-Cache**中。

- **指令缓存模块(I-Buffer)** 用于缓存从指令Cache中取出的指令。I-Buffer被静态划分，使得运行在SIMT核心上的所有warp在其中都有专门的空间存储指令。在当前的模型中，每个warp有两个I-Buffer条目。
- 每个I-Buffer条目有一个有效位（valid bit），一个就绪位 (ready bit) 以及一个对应于该warp当前指令的译码后的指令。有效位表示当前I-Buffer 的该条目中还有一个未发射的指令（该条目上的指令有效）。 就绪位表示该Warp当前指令已经准备好被发射到执行流水线中。
- 通常情况，没有结构冒险、没有WAW冲突的时候，就绪位置1。
    - 在**Scoreboard**算法中，发射阶段对指令进行解码，解码单元会“问”记分牌当前指令是否有“ WAW 冒险”和“ Structure 结构冒险”，如果没有，指令就可以顺利渡过当前的阶段。
- 如果在I-Buffer中没有任何有效指令，所有需要取指的warp会以轮询的方式访问I-Cache 。一旦被选中，一个读请求以及下一条指令的地址被送入到I-Cache 。默认情况下，两条连续的指令被取出。
- 只要一个Warp被取指调度器调度进行取指操作，对应的I-Buffer条目有效位即为1， 直到该 Warp 内所有指令均执行完毕。
- 只有当一个线程执行完所有指令，并且没有未完成写回存储器、写回寄存器请求时，才能说一个线程执行完毕；只有当一个Warp内所有线程均执行完毕，该Warp才被认为执行完毕，并且不再受取指调度器调度；只有一个线程块内所有Warp执行完毕，该线程块才被认为执行完毕；只有所有块执行完毕，该内核函数函数才算执行完毕。
- **Decode指令译码：**当前被取出的指令被译码，确定指令种类（算术/分支/访存）和要被使用的寄存器。之后便存储到 I-Buffer 相应的条目中等待被发射。译码的同时也会检查寄存器的记分牌，以确定可能有相关性的冲突。一旦检测到冲突，将清空译码阶段的输入流水线寄存器，使正在译码的指令失效。若没有冲突，将在记分牌入口设置标识，表示这些指令流出的寄存器正被使用。
- **发射调度器：**功能是从I-Buffer中选择一个Warp发射到后续流水线中。此调度器独立于取指调度器，调度方式是循环优先级策略（指不同Warp）。
- 被发射的指令必须满足以下条件：
    - 该warp没有处于栅栏同步的等待状态
    - I-Buffer对应条目中的有效位为1
    - 通过记分板检测
    - 指令流水线中的取操作数阶段（operand access stage)不是挂起状态
- 指令发射的目的地
    - 存储器相关指令（load, store , memory barriers etc）被发射到存储流水线（MEM PIPELINE）
    - 运算指令被发射到ALU计算单元，其包括多个SP流水线、SFU流水线。
    - 分支指令将清空I-Buffer内所有与该Warp相关指令（参看下文SIMT STACK）
    - SIMT STACK：SIMT指令栈
- **SIMT-stack**：每个Warp均有一个SIMT-stack来解决Warp内的线程指令分歧问题。分支分化带来的串行路径执行会降低硬件效率，最简单的解决方案是PDOM(post-dominator stack-based reconvergence)机制。

SIMT核心后端

- **Operand Collector**：由一组缓冲器和一个**取数调度器**组成。（OP.COL.）

- 每当一条指令被译码后，OP.COL.便为该指令分配空间，用于取数。OP.COL.单元并没有通过寄存器换名技术来消除寄存器名字依赖，而是通过另一种方式：**确保每个周期内，对一个 Bank 的访问，不得超过一次。**
- 四个Collector Units，每个Unit包含三个操作数条目，和一个标志符，用于指示当前该Unit属于哪个Warp的哪条指令。
- 每个操作数条目包含四个字段：
  - 就绪位
  - 有效位
  - 寄存器识别符
  - 操作数：该域包含128字节，可以存放32个4字节数，可以满足一个Warp内的32个线程。
    - 注意：每个Thread 有自己的寄存器，因此仅需一个寄存器标志符即可
    - 另外，调度器为每个Bank均保留了一个请求读队列，直到所有Unit对该Bank的访问均已完毕
- 当一条指令经过译码阶段并且存在Collector Unit可用，则该Collector Unit被分配给该指令，相应的Warp标志、有效位、寄存器识别符被设置，操作数域被初始化为 0 。此外，操作数的读请求被排队到调度相应Bank队列。
- 实际上，执行单元写回的数据的优先级总是高于读请求。调度器每周期选择一组至多4个无bank冲突的数据发送到寄存器堆。
- 每个Collector Unit每个周期只接收一个操作数。
- 当每个操作数被从寄存器堆读出并放入到相应的OP.COL.单元中，该操作数条目就绪位被设置为1。最终，当一个指令的所有操作数都就绪后，该指令被发射到一个SIMD执行单元。
- **对于每种不同的SIMD执行单元（SP,SFU,MEM）,均有各自独立的 Collector Units，同时也有一组共享的 Collector Units。**
- **Mem Pipeline**：处理线程访问全局内存与共享内存发出的请求。
- 每个 SIMT 核心有 4 种不同的片上一级存储器：共享存储器（ Shared Memory ）， 一级数据缓存（ L1-data-cache ）， 常量缓存 （ constant cache ）以及纹理缓存（ Texture Cache ）。虽然上述四个存储器物理上独立，但由于其均为是存储器流水线 (LDST unit) 的组成部分，因此它们共享同一个写回阶段。
- **ALU**：两种ALU计算单元
  - SP计算单元通常每周期执行一个Warp的一条指令
  - SFU计算单元执行周期视指令不同而不同：sin指令需要4个周期，取倒数指令需要两个周期。

ScoreBoard算法

> 计算机体系结构-记分牌ScoreBoard - 知乎 (zhihu.com)

- 记分牌是一个信息存储单元：存储了**功能单元状态**和**寄存器结果状态。**
  - 功能单元状态包括：部件是否正在忙、部件执行的指令类型、部件现在需要的源寄存器、部件现在的目的寄存器、源寄存器是否准备好和如果源寄存器没准备好部件该向哪里要数据
  - 寄存器结果状态主要记录对于某一个寄存器，是否有部件正准备写入数据。
- 在CDC6600处理器中，一条指令分四个阶段执行：发射、读数、执行、写回。

- **发射**：对指令进行解码，并观察记分牌信息，主要观察各个功能部件的占用情况，和寄存器堆的写情况，以此来判断是否可以把解码得到的信息存进对应的部件寄存器。
    - **如果指令对应的功能部件空闲，且指令要写的目标寄存器没有别的指令将要写**（这是为了解决 *WAW* 冒险），那么阶段结束的时候，就可以把指令信息存进部件寄存器，同时改写记分牌，把指令相关信息进入记分牌。
- **取数**：观察记分牌，了解当前指令需要哪些寄存器的数值、该数值是否准备好、如果数值没有准备好的话数值正在哪个功能部件进行运算。
    - 如果数值没有准备好（这是为了解决 *RAW* 冒险），那么指令就卡在部件寄存器中，无法读取数据。
    - **如果寄存器都可以读取**，那么阶段结束的时候，对应的寄存器数值会被存进操作数寄存器中。
    - 这里不会改写记分牌。
- **执行**：执行计算过程，计算过程可能维持很多个周期。在第一个周期结束时，记分牌的读取寄存器部分内容（即 Rj、Rk ）会被修改，表明指令不再需要读寄存器。
    - 在得到计算结果的那个周期结束时，结果会被存进结果寄存器。
- **写回**：观察记分牌，**如果别的指令不需要读当前计算结果即将写入的寄存器**（这是为了解决 *WAR* 冒险，需要观察所有 Rj、Rk，如果有指令要读当前要写入的寄存器，就要先等前序指令读完寄存器再写回），那么周期结束时，就会把结果写回到寄存器堆，同时会清空记分牌中的信息。

91.GA10x SM架构

- GeForce RTX 3060是采用规模较细的「GA106」核心，相比上代 「TU106」，性能差异主要来自 FP32 运算单元提升了 1 倍、升级第 2 代 RT处理器、升级第 3 代 Tensor处理器，使传统光栅图形运算提高了 1.7 倍，同时在光线追踪性能上提升近 2 倍。

- 三代Tensor Core、并行执行FP32和INT32。
- SM被分区为4个processing blocks（partition），它们共享128KB的 L1 data cache/shared mem
    - 所以Shared Mem最大100KB，至少有28KB的L1 cache。
    - 一个Processing Block有64KB的寄存器文件，有L0指令cache，有一个warp scheduler，一个dispatch unit，16个FP32/INT32 core和16个FP32core

- 每个processing block(partition)有自己的寄存器文件、L0指令cache。
  - 也就是说L0指令缓存对一个warp scheduler/partition专用
- 每个SM有L1 data cache/shared mem，L1 constant cache，L1.5constant cache/L1 instruction cache。
  - L1指令缓存专属于SM。
- 每个GPU有L2 data cache/L2 constant cache/L2 instruction cache， GDDR。
  - L2缓存是统一的（即缓存指令和数据），所有SM共享。

-

- Ampere的内存访问latency（CPI）：
  - Global Mem：290Cycles(HBM2 for A100)
  - L2 cache：200cycles
  - L1 cache：33cycles
  - SMem(ld/st)：23/19cycles

92.Maxas/ControlCodes

简介

- Kepler每8条指令（64bit/指令）中会有一个Control Code，Maxwell是每4条一个。Turing则是一条指令就有128bit，其中会编码一个control code。每条控制代码都是一组的第一条，作用于后续的三条代码。
- 64bit里 最高位不管它，63bit分连续的三份儿，一份儿21bit，前4bit是reuse，后17bit分为watdb(6)、readb(3)、wrtdb(3)、yield(1)、stall(4)
- Dependency Barrier也许更合理的称呼是Scoreboard

```
; 这显示了如何将控制代码拆分为 3 个部分的细分，每个部分对应一个关联的指令。我还将其进一步分为控制代码和重用代码。
push @ctrl,  ($code & 0x000000000001ffff) >> 0;
push @ctrl,  ($code & 0x0000003fffe00000) >> 21;
push @ctrl,  ($code & 0x07fffc0000000000) >> 42;

push @reuse, ($code & 0x00000000001e0000) >> 17;
push @reuse, ($code & 0x000003c000000000) >> 38;
push @reuse, ($code & 0x7800000000000000) >> 59;
```

```
# 17 bits @ctrl are broken down into 5 sections
sub printCtrl
{
    my $code = shift;

    my $stall = ($code & 0x0000f) >> 0;
    my $yield = ($code & 0x00010) >> 4;
    my $wrtdb = ($code & 0x000e0) >> 5;  # write dependency barier
```

```
    my $readb = ($code & 0x00700) >> 8;  # read  dependency barier
    my $watdb = ($code & 0x1f800) >> 11; # wait on dependency barier

    $yield = $yield ? '-' : 'Y';
    $wrtdb = $wrtdb == 7 ? '-' : $wrtdb + 1;
    $readb = $readb == 7 ? '-' : $readb + 1;
    $watdb = $watdb ? sprintf('%02x', $watdb) : '--';

    return sprintf '%s:%s:%s:%s:%x', $watdb, $readb, $wrtdb, $yield, $stall;
}
```

- `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）。

```
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 ;
```

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）。
- 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。

Stall Counts

- 大多数指令的流水线深度是6个时钟周期。

```
# 所有合法的stall counts数
f : 15 clocks*
e : 14 clocks*
d : 13 clocks*
c : 12 clocks*
b : 11 clocks
a : 10 clocks
9 :  9 clocks
8 :  8 clocks
7 :  7 clocks
6 :  6 clocks
5 :  5 clocks
```

```
4 :  4 clocks
3 :  3 clocks
2 :  2 clocks
1 :  1 clocks
0 :  0 clocks, dual issue
```

- 用片上不同资源的两条相邻指令可以dual issue，这样stall count就是0。
- 为了stall12-15个clocks，yield hint必须被额外设置上。不设置就会造成更少的stall clocks。
- 别的指令设置了的predicate，当前指令需要等13个stall count才能用。
- 一些没有流水线的指令需要最少x个stall count才能正常运行。这方面的示例包括 BAR、BRA、CAL、RET 和 EXIT，它们需要至少5个stall。
- 共享内存读的延迟是2个时钟周期，全局内存读的延迟是4个时钟周期。
- 通过使用 maxas SCHEDULE_BLOCKs 您可以在很大程度上忽略设置停顿计数，因为这是为您处理的，能获得最高吞吐量的最佳排序指令。

Yeild Hint Flag

- 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。
- 只要有其他活跃warp，那yield就不会有时钟开销，不管是在一个block里还是跨block。

Write Dependency Barriers

- Write 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）。
- 有许多类型的指令会输出结果，但不会以固定的流水线延迟运行。可变延迟指令分为两类:内存操作和共享资源操作，如双精度操作、特殊功能单元操作或其他低吞吐量操作。由于延迟是可变的，我们不能可靠地使用暂停计数来等待结果。相反，我们设置并等待一个障碍。Maxwell每个线程有6个屏障资源。原始控制代码使用值0-5，但我将它们重新映射到1-6。我发现这些值更容易阅读。因此，对于任何写入寄存器并以可变延迟进行操作的指令，您需要设置其中一个屏障，或者至少将这条指令置于另一条指令设置了屏障的指令之后(尽管您必须小心处理)
- 在不写出寄存器的指令上设置写依赖屏障是非法的。
- 障碍需要在执行设置的指令消耗的时钟周期之上 再加一个额外的时钟周期 才能变为活动状态。
- 因此，如果您打算等待后续指令，您可能还需要使用 2个 stall count。

Read Dependency Barriers

- 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）。
- 这种类型的屏障用于不使用operand collector的指令，因此不缓冲其操作数（主要是内存操作）。这些指令也以可变延迟运行，因此为了避免WAR hazard，我们需要设置并等待屏障，然后才能覆盖操作数值。用于此目的的屏障资源与写依赖项屏障共享，也编号为 1-6。
- 也额外消耗一个clock来变活跃。

- 读取屏障的延迟通常约为 20 个时钟（the time it takes to set the memory instruction in flight）。完整的指令本身可能需要比这更多的时钟。

Wait Dependency Barriers

- 等待之前设置的屏障，可以同时等一个以上的屏障，所以用的是bit mask，而不是barrier num。
- 6个bit，每个bit表示等待对应的Dependency Barrier

```
01 : 1
02 : 2
04 : 3
08 : 4
10 : 5
20 : 6
```

- 等待未设置或已signaled的障碍似乎不会增加任何额外费用。
- SASS里面还有一个对应的指令，如`DEPBAR.LE SB0, 0x0, {2,1} ;`。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。
- 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。

- 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。

93.指令执行吞吐和指令集设计

- RISC 五级流水
  - Instruction fetch cycle (IF)：主要是获得Program Counter对应的指令内容。
  - Instruction decode/register fetch cycle (ID)：解码指令，同时读取输入寄存器的内容。由于RISC类指令的GPR编码位置相对固定，所以可以直接在解码时去读取GPR的值。
  - Execution/effective address cycle (EX)：这是指令进入执行单元执行的过程。比如计算memory地址的具体位置（把base和offset加起来），执行输入输出都是GPR或输入含立即数的ALU指令，或者是确认条件跳转指令的条件是否为真等。
  - Memory access (MEM)：对于load指令，读取相应的内存内容。对于store指令，将相应GPR的值写入到内存地址处。RISC类指令集通常都是load-store machine，ALU指令不能直接用内存地址做操作数（只能用GPR或立即数），因而通常ALU指令没有memory access。
  - Write-back cycle (WB)：将相应的ALU计算结果或memory load结果写入到相应的GPR中。

- 由于GPU运行模型的复杂性，在Decode后Execution前，还有大量其他的步骤：比如scoreboard的判断（主要用来保证每个指令的执行次序，防hazard），warp Scheduler的仲裁（多个eligible warp中选一个），dispatch unit和dispatch port的仲裁（发射或执行资源冲突时需要等待），还可能有读取作为立即数操作数的constant memory，读取predicate的值生成执行mask，等等。在执行过程中，也有很多中间步骤，比如输入GPR操作数的bank仲裁，跳转指令要根据跳转目标自动判断divergence并生成对应的mask，访存指令要根据地址分布做相应的请求广播、合并等等。在写回这一级的时候，由于一些指令的完成是异步的（比如一些内存指令），所以也可能需要GPR端口的仲裁，等等

93.1 GPGPU指令执行吞吐的影响因素

- **通常GPU的指令吞吐用每个SM每周期可以执行多少指令来计量。**对于多数算术逻辑指令而言，指令执行吞吐只与SM内的单元有关，整个GPU的吞吐就是每个SM的吞吐乘以SM的数目。
- **GPU的FMA指令（通常以F32计算）往往具有最高的指令吞吐，其他指令吞吐可能与FMA吞吐一样，或是只有一半、四分之一等等。**所以很多英文文档会说FMA这种是full throughput，一半吞吐的是half rate，四分之一的是quarter rate等。
- 指令吞吐主要受以下因素影响：
  - **功能单元**的数目。绝大多数指令的功能都需要专用或共享的硬件资源去实现，设计上配置的功能单元多，指令执行的吞吐才可能大。显然，只有最常用的那些指令，才能得到最充分的硬件资源。而为了节约面积，很多指令的功能单元会相互共享，所以他们的吞吐往往也会趋于一致。
  - 指令**Dispatch Port**和**Dispatch Unit**的吞吐。Turing、Ampere是一个Scheduler带16个单元，每个指令要发两cycle，从而空出另一个cycle给别的指令用。**首先要eligible**，**其次要被warp scheduler选中**，**最后要求Dispatch Port或其他资源不被占用。**
  - **GPR读写吞吐。**绝大部分的指令都要涉及GPR的读写，由于Register File每个bank每个cycle的吞吐是有限的（一般是32bit），如果一个指令读取的GPR过多或是GPR之间有bank conflict，都会导致指令吞吐受影响。GPR的吞吐设计是影响指令发射的重要原因之一，有的时候甚至占主导地位，功能单元的数目配置会根据它和指令集功能的设计来定。
    - 比如NV常用的配置是4个Bank，每个bank每个周期可以输出一个32bit的GPR。这样FFMA这种指令就是3输入1输出，在没有bank conflict的时候可以一个cycle读完。其他如DFMA、HFMA2指令也会根据实际的输入输出需求，进行功能单元的配置。
  - 很多指令有**replay**的逻辑（参考Greg Smith在StackOverflow上的一个回答）。这就意味着有的指令一次发射可能不够。这并不是之前提过的由于功能单元少而连续占用多轮dispath port，而是指令处理的逻辑上有需要分批或是多次处理的部分。
    - 比如constant memory做立即数时的cache miss，memory load时的地址分散，shared memory的bank conflict，atomic的地址conflict，甚至是普通的cache miss或是TLB的miss之类。
    - 根据上面Greg的介绍，Maxwell之前，这些replay都是在warp scheduler里做的，maxwell开始将它们下放到了各级功能单元，从而节约最上层的发射吞吐。不过，只要有replay，相应dispath port的占用应该是必然的，这样同类指令的总发射和执行吞吐自然也就会受影响
- 注意的点：
  - 指令发射吞吐和执行吞吐有时会有所区别。有些指令有专门的Queue做相应的缓存，这样指令发射的吞吐会大于执行的吞吐。这类指令通常需要访问竞争性资源，比较典型的是各种访存指令。但也有一些ALU指令，比如我们之前提过的Turing的I2F只有1/4的吞吐，但是可以每cycle连发（也就是只stall 1cycle）。不过多数ALU指令的发射吞吐和执行吞吐是匹配的。

- Turing把普通ALU和FFMA（包括FFMA、FMUL、FADD、IMAD等）的PIPE分开，从而一般ALU指令可以与FFMA从不同的Dispatch Port发射，客观上是增加了指令并行度。NVIDIA对CUDA Core的定义是F32核心的个数，所以Turing的一个SM是64个Core。Ampere则把一般ALU PIPE中再加了一组F32单元，相当于一个SM有了128个F32单元（CUDA Core），但是只有64个INT32单元。也就是说SM86的F32类指令的吞吐是128/SM/cycle，但其中有一半要与INT32的64/SM/cycle共享。或者说，Turing的F32和INT32可以同时达到峰值（包括A100），而SM86的INT32和F32不能同时达到峰值。
- 要注意区分指令吞吐与常说的FLOPS或是IOPS的区别。通常的FLOPS和IOPS是按乘法和加法操作次数计算，这样FMUL、FADD是一个FLOP，FFMA是两个FLOP，但都是一个指令吞吐。在TensorCore出现以前，通常的Tesla卡HFMA2、FFMA的指令吞吐一样，DFMA吞吐是一半，而看峰值FLOP就是H:F:D=4:2:1的关系。TensorCore出现后，指令（比如HMMA）本身的吞吐和指令入口的GPR输入量没有变化，但由于同一个warp的指令可以相互共享操作数数据，一个指令能算的FLOP更多了，因而峰值又提高了。

93.2 GPGPU指令执行的特点

静态资源的分配

- GPU有一个很重要的设计逻辑是尽量减少硬件需要动态判断的部分。GPU的每个线程和block运行所需的资源尽量在编译期就确定好，在每个block运行开始前就分配完成。GPU没有CPU通用寄存器的动态映射逻辑(renaming)，每个线程的GPR将一一映射到物理GPR。由于每个线程能用的GPR通常较多，加上编译器的指令调度优化，这种假依赖（映射固定导致依赖于特定的硬件寄存器）对性能的影响通常可以降到很低的程度。
- 每个block在运行前还会分配相应的shared memory，这也是静态的。这里需要明确的是，每个block的shared memory包括两部分，写kernel时固定长度的静态shared memory，以及启动kernel时才指定大小的动态shared memory。虽然这里也分动静态，但指的是编译期是否确定大小，在运行时总大小在kernel启动时已经确定了，kernel运行过程中是不能改变的。
- 其实block还有一些静态资源，比如用来做block同步的barrier，每个block最多可以有16个。
- 另一种是Turing后才出现的warp内的标量寄存器Uniform Register，每个warp 63个+恒零的URZ。
- 另外每个线程有7个predicate，每个warp有7个Uniform predicate。这三种都是足额，不影响Occupancy。
- GPU里还有一种半静态的stack资源，local memory。多数情况下每个线程会用多少local memory也是确定的。不过，如果出现一些把local memory当stack使用的复杂递归操作，可能造成local memory的大小在编译期未知。这种情况编译器会报warning，但是也能运行。不过local memory有最大尺寸限制，当前是每个线程最多512KB。

顺序执行

- GPU主要通过Warp切换的逻辑保持功能单元的吞吐处于高效利用状态，这样总体性能对单个warp内是否stall就不太敏感。虽然GPU一般是顺序执行，但指令之间不相互依赖的时候，可以连续发射而不用等待前一条指令完成。在理想的情况下，一个warp就可以把指令吞吐用满。当然，实际程序还是会不可避免出现stall（比如branch），这时就需要靠TLP来隐藏这部分延迟。

显式解决依赖

- 既然是顺序执行，但同时又可以连续发射，那怎么保证不出现数据冒险呢？NV GPU现在主要有两类方式：
- 第一种是固定latency的指令，**通过调节control codes中的stall count，或者插入其他无关指令，保证下一条相关指令发射前其输入已经就位**；
- 第二种是不固定latency的指令，就需要**通过显式的设置和等待scoreboard来保证结果已经可用。**

- GPU本身运行就是多线程的，同一个warp内也是通过scoreboard来保证次序。但多个warp之间，GPU也需要维护相应的coherence和memory consistency model。参考PTX文档: Memory Consistency Model。

93.3 指令设计中的一些原则和思路

指令长度问题

- SASS是定长的。Volta、Turing、Ampere都是每条指令128bit，每条都自带control code。
    - 定长的好处之一是解码器可以提前解码，且一般解码开销小。因为首先指令等长，每个指令的范围是确定的，指令定界不依赖于前一个指令的解码。其次RISC在编码的时候一般各个域的位置和长度比较整齐，解码相对说来自然也更简单一些。

93.4 指令集设计和ILP的一些相关性

- GPU两个最重要的并行逻辑，ILP（Instruction Level Parallelism）和TLP（Thread Level Parallelism），两者在隐藏延迟中都有重要作用。ILP的逻辑主要是靠前一条指令不需要执行完成就能发射下一条无关指令，而TLP则是通过warp之间切换来隐藏延迟。
- ILP是线程内（更准确的说是Warp内）的并行逻辑，影响ILP的主要因素有两种，一是指令之间的**依赖性**，二是指令的**资源竞争或冲突**。依赖分显式和隐式。显式依赖主要是数据的相关性，隐式依赖则与资源竞争很相似，主要是两个指令都要使用某个特定含义的公共资源。

--待续

94.各级存储的吞吐问题

94.1 对程序性能有直接影响的吞吐

- **指令执行**的吞吐：主要与执行的功能单元数目和功能单元本身的吞吐能力有关。通常说的峰值计算能力（Peak FLOPS/IOPS），就是一些ALU指令执行吞吐的体现。
- **指令发射**的吞吐：主要与Warp Scheduler、Dispatch Unit、Dispatch Port等的吞吐有关。当然，指令发射其实也要满足很多其他条件，包括上面提到的功能单元数目的限制等。毕竟每个指令的执行都需要相应的资源支持，而很多指令在条件不满足时不能发射。注意：指令发射吞吐与执行吞吐高度相关，但也可能有差别。首先发射吞吐肯定不能比执行吞吐小，否则执行吞吐一定用不满，单元就浪费了。例如一些内存访问的共享单元指令，发射吞吐可以执行吞吐大一些，这类指令一般会有对应的队列去解决排队和竞争问题，这样更有利于减少单元的空闲。另外有些架构有多发射的逻辑，执行单元有富余的时候可以同时发射相应的指令，这样发射吞吐也会相应变化。
- **Register File**的读写吞吐：GPGPU的RF一般由若干块有若干读写端口的SRAM组成，是GPR的物理存储空间。RF的吞吐是限制指令发射和执行吞吐的关键因素之一。

- **Shared Memory和各级cache**的吞吐：Shared Memory和各级cache其实一般也是大小不一的SRAM，各自也有对应的吞吐设计。GPU的Cache种类非常多。**一般L2 Cache是片上所有SM的共享资源，可以缓存几乎所有数据、指令等；L1 Cache通常是SM私有，包括独立的数据缓存（有的也叫Read-only Cache, Texture Cache，Unified L1 Cache等等）和指令缓存；**SM内还有专门的constant cache可以缓存constant memory，据说有些立即数操作数也会放在特殊的缓存里；有的架构可能每个SM的sub-partition（对应每个warp scheduler）有独立的L0指令缓存；片上还有虚拟地址和物理地址转换的页表缓存TLB，但是不太确定是否是独立的缓存空间。
- **Device Memory**（主要是GDDR或是HBM，核芯显卡当然可能也用DDR）的吞吐：一般DRAM的运行频率和处理器的频率不一样，两者通过内存控制器（Memory Controler，简称MC）连接起来。所以它的吞吐通常不以处理器周期为单位，一般用绝对带宽是多少GB/s之类。DRAM吞吐的表现形式与片上的资源很不一样，不仅因为工作频率不一样，也由于它的访问模式比较复杂。
- **Host Memory或其他互联资源**的吞吐：GPGPU多数时候都作为一个数据处理加速器来用，无论是初始数据的载入还是结果数据的存回，都需要与外部Host memory进行交互，这样Host memory的吞吐就会很重要。在有些情况下，还要通过NVLink等互联结构与其他GPU或CPU交互。

- 还有一些比较隐式的，或者说是编程中没有直接体现的吞吐。比如间隔多久能启动一个kernel，同时能执行多少copy任务，kernel的任务分发中每周期可以下发多少block或warp到SM上，等等。这些也和性能有关，只不过一般不显著，编程的时候也很少关注而已。

94.2 GPGPU中吞吐处理的几个思路

94.2.1 增加缓存：

- 增加缓存是常见的减少ALU与内存之间延迟差距的方法。
- 缓存除了因为离ALU近具有延迟和吞吐的优势外，另一个重要作用就是拦截部分向下一层的访问，从而减少下层存储单元的吞吐压力。
    - 比如NV GPU中的Register File就有reuse cache。用不用reuse其实并不影响指令的延迟。但是它通过减少对Register File的访问，可以省下宝贵的Register Bank访问吞吐，从而部分缓解Register Bank Conflict，间接提高性能。
- 当然，缓存也有很多局限性，也会带来额外的开销。
    - 首先，缓存只有在hit的情况下才有收益，miss时肯定是开销变大了（多数情况下latency会变长，除非它同时往下一层缓存发送了请求）。如果hit rate太低，缓存在延迟上的收益可能是负的，不过吞吐的节约仍然有效。
    - 其次，缓存的容量往往比较有限，增加容量不仅仅会增加面积，同时也会增加访问的开销。
    - 再次，缓存的一致性一直是个复杂的问题，为了维持缓存一致性，往往需要各种特殊处理，这也是缓存的主要负担之一。
    - 最后，缓存由于功能上的限制，并不能缓存所有内存操作。比如NV GPU的很多写入操作对L1都是write-through到L2，意味着每次写可能都会同时占用L1和L2的写吞吐。再比如由于GPU的L1一般是不保一致性的，那所有的atomic操作都必须送到保一致性的L2上去做。这些都是缓存存在局限的地方。

94.2.2 分而治之

- RAM设计时，为了扩大容量，减小粒度，提供并行性，维持吞吐，常用的思路是把存储 分成多个块，“块”就是常说的Bank。

- 分Bank的好处是可以把RAM模块化，单个RAM块的设计比较固定，用增加bank的方式增加吞吐。另外每个RAM需要的地址线也少了，因为有几个bit地址被用做Bank-Select，Bank内的寻址就不用它了。即使要对各个bank扩容，它的开销也会比单独一块RAM的扩容成本小。这样无论是设计难度还是面积开销，都会更有优势。
- 但是，分Bank也有一个很大的问题，就是它的极限吞吐是建立在所有Bank负载均匀的情况下。如果有多个请求同时落在同一个Bank内，那它一次只能响应其中一个（实际上输入的时候就只能输入一个），导致这些请求会被序列化，这就是**Bank Conflict**。
- 从设计上讲，缓解Bank Conflict的方法除了尽量均匀分配请求，还有一种方法是打散同步访问，用前后的异步请求来相互错开Bank。
- Bank Conflict在GPGPU中几乎无处不在。Register File有，Shared Memory有，各级Cache有，连HBM和GDDR都一样有。

94.2.3 请求的前处理和后处理：广播、合并、重组、重排等

- 第一种操作是**广播**（Broadcast）。广播指的是访问请求中如果需要读相同的地址，则只需要读一次后warp内部复制分发即可。常见于constant mem的读（只有LDC指令才支持在访问constant mem是用GPR索引）、一个指令的多个操作数用同一个GPR、读shared mem的同一个地址。
  - 广播一般只对读有效，不能针对往同一个地址的写，也不能用于atomic的read-modify-write。
- 第二种是**合并**（Coalesce）。落在同一个cacheline内的请求就会合并成一次传输。合并不仅可以针对读，也可以针对写。理论上应该也可以用于atomic操作，只是在执行的时候仍然需要按地址做序列化。NV GPU的global memory访问逻辑可能还更复杂一些，它访问的颗粒度并不是一个cacheline，而是32B。具体合并的逻辑还与是否被L1 Cache有关。简单说就是过L1 Cache的时候，按128B的颗粒度做合并。如果不过L1 Cache，按32B的颗粒度做合并。
- 第三种是**重组**。比如说有两个请求，各需要32B的数据。而传输单元的吞吐是每次64B数据，那就可以把两次传输合并为一次，从而减少传输带宽的浪费。当然，这也要求每个请求有完备的header信息和payload信息，否则接收端收到数据后不知道如何分发。多数的处理器内部都有数据传输总线，总线的位宽决定了吞吐上限。通过这种重组，可以最大限度的利用传输带宽，减少吞吐浪费。
- 第四种是**重排**。将一系列同步操作打散成异步操作，然后又重新组合成一系列同步操作的过程。
  - NV GPU用operand collector进行ALU的输入操作数准备时采用了这个逻辑。

94.3 各级存储单元的吞吐设计

Register File

- GPR的访问颗粒度一般较高。因为一个Warp的32个线程是同步运行的，比如读取32bit的`R0`，其实需要读32*32bit=1024bit，在空间分配上是可以做成连续的。因此，尽管GPR看起来访问颗粒度是32bit甚至更小，但GPR所在的Register File的访问颗粒度可以做得很大。
  - Turing的SM每个sub-partition的ALU其实最多是16的宽度，一个warp指令的操作数分2 cycle去读应该也可行。
- GPR的吞吐是指令发射和执行的核心资源，也是竞争性最强的资源。因为GPR的读写是绝大多数指令的共享资源，发生Bank Conflict时会挤占其他指令读写GPR的机会，导致其他指令不能发射或完成。所以**GPR的bank Conflict对程序性能的影响是最直接的，很难靠ILP或TLP缓解。**此外访存指令也需要异步读写GPR，也可能会与ALU竞争RF的端口。当前GPR的吞吐处理逻辑除了前面说的异步重排以外，基本都是依赖于编译器生成比较好的代码：包括怎么错开GPR的bank，指令的调度重排，以及什么时候要reuse等等。因为当前GPU几乎都是顺序执行的，也没有听说会用register renaming之类的方式，所以感觉上GPR的bank conflict不会有太多的花招可以玩。

Shared Mem和L1 Cache

- Shared Memory是block内用户配置的编程资源。同一个SM上的不同block会使用同一块物理空间，只是相互独立互不可见而已。由于用户具有完全的控制权，它的访问颗粒度其实可以小到一个字节。当然，4B的颗粒度是日常编程比较常用的情况（一个int或float的大小，也是一个GPR的大小），每个warp的32个线程常常独立访问4B数据。因此，N卡的shared memory一般32个bank，每个bank是4B。Kepler曾经弄出过4B和8B可配的Bank Size，但估计觉得意义不大，之后就弃用了。
- 一些常见的减少Bank Conflict的方法比如padding（主要是高维），地址swizzle之类。
- N卡的L1主要用来缓存只读数据，例如texture数据，或是带const修饰符的数组和`__ldg()`的访问。L1也可能可以缓存部分写操作，比如register-spill或用户写入的local memory数据，文档说是write-through。一般的内存访问默认可能是bypass L1的（这个各代架构似乎会不太一样）。L1通常的访问颗粒度是128B，正好是4B*32，算是一个cache line的size，也是shared memory无Bank Conflict的峰值吞吐。
- Shared Memory和L1的空间配置不是任意的，而是有一个最小颗粒度。比如Turing有96K，可以配成64+32或32+64。
- L1一般说来容量不大（以前每个SM几十K，现在有的一百多K），需要当前SM所有block共享。因此，在大量warp读写压力下，hit rate多数情况下是比较低的。减轻L1压力的方式之一是把用户能控制的可共享数据都用shared memory共享。当然，这只能是block内共享，但block间能共享的数据其实是很少的。何况block分配的SM也不确定，能共享也很难控制。因此，L1能bypass还是一个挺好的逻辑，至少L2访问的latency能省一点，同时也减少对一些只读数据进行的不必要的evict。
- L1还可以做coalesce buffer，相当于把所有访问数据集齐了再统一送给warp处理。地址Coalesce是很有用的优化。不能或者不方便coalesce的情况，比如图形图像同时有横向和纵向的tile型访问方式，还是建议使用texture。因为Texture可以使用Z-Order Curve之类的排列方式减小这种小tile的地址分散度。如果是非常散的访问，coalesce无望，用L1可能还不如用L2的32B小颗粒读。
- 总体来讲，L1的负担还是很重的，设计上的吞吐不能太小，至少不能拉L2的后腿。不过因为有GPR和Shared Memory的存在，L1的压力还是可以分摊一些。最近几代N卡的L1 hit latency一般在20~30cycles左右。
- 在指令发射上，Shared Mem和L1是共享一个port的（`LDS` vs `LDG/LDL`，也包括一些atomic的操作，至少都要隔4 cycle）。

L2 Cache

- L2 Cache是所有SM共享的缓存资源，同时也是几乎所有device memory公共的缓存区。除了通常说的用户数据外，还包括指令及相关模块的数据，constant memory，甚至包括页表，GPU自身硬件相关代码，等等。L2的容量一般在MB量级，所以分bank是很顺理成章的事情（当然也许不都叫bank，最上层好像叫Slice或Channel的比较多，一个Slice应该是包括若干个Bank）。
- N卡的设计上做得比较细，L2的最小读写颗粒度比cacheline要小。L1、L2每个cacheline都是128B，只是L2在访问的时候有特定的mask，指定具体需要128B中的哪个或哪几个32B的sector。然后把相应的数据拼到data bus上去一起传输。但是L2到L1数据传输的颗粒度是32B。
- 由于有coherence的限制，内存地址到L2 bank的映射应该是固定的。L2需要支持很多SM同时访问，负载都会比较重。所以这个地址映射关系还是比较关键的。如果说很多访问被映射到同一组，那会导致比较严重的负载不均衡，从而造成吞吐的浪费。我之前做过一些测试，能够测到它的地址映射不是简单用二进制表示的前缀，而是有一些复杂的映射算法（各种异或hash之类）。

Device Mem

- DRAM是几乎所有需要在GPU上运行的数据的最初来源。当前GPGPU最常用的DRAM有GDDR和HBM，一般GDDR的传输率高但是位宽低，总带宽还是HBM更有优势。HBM位宽大，功耗低，体积小，但是线很多（位宽大啊！），价格也贵，所以一般只有高端卡会使用HBM。核芯显卡当然可能用DDR，不过它应该会与CPU共用一套Cache（一般叫LLC，Last Level Cache，不叫L1、L2之类是因为它相对CPU可能是L3，但GPU这可能是L2）和内存控制器。
- DRAM的时钟频率就和GPU可能不一样。所以一般说来GPU对DRAM的访问是异步的，它吞吐的表现形式和GPU内的各种同步的SRAM还是有不小的差别。当然，DRAM内部机制挺复杂的，有很多都对极限吞吐有影响。比如prefetch，refresh，page policy等等。一般说来，DRAM的峰值带宽还是很难达到的，即使是非常理想的访问模式，能到80%~90%就算是很不错了。
- NV最新的Ampere架构支持一个新Feature叫Multi-Instance GPU（MIG），可以根据用户设置对DRAM、L2还有SM进行资源分区，相互之间互不影响。这里应该不完全是软件层的封装，应该也用到上前面说的DRAM、MC、L2的映射关系。这样DRAM和L2的吞吐才能真正隔离开，互补影响。至于SM，因为与L2间有CrossBar，也许用虚拟地址的映射就能隔离开（当然这里我说的是内存访问隔离，block分配肯定还有对应的分配隔离逻辑）。这个其实有一点像以前的GPU分时间片的虚拟化方法，只是把资源分片从时间维度转成了空间维度，资源隔离更严谨了，也减少了context-switch的开销，感觉上是要更先进一些。

Host Mem

- Pinned Mem

95.GPGPU上层架构

95.1共享单元与私有单元

- 单元都有分组的层次结构，那就会有一个放置位置的问题。比如某种ALU单元，可以让它只接受SM的某个partition的请求（当前一般是4个partition)），也可以让它接受所有partition的请求，甚至是接受多个SM的请求。从上面的就近原则来讲，当然应该尽量放在最底层的位置，这样离得最近，运行开销最小。按这个逻辑，单元会尽量向私有的方向发展。
- 但是排队论的基本结论是：Server数目一定时，所有的Customer排同一个队，会比每个Server单独排一个队效率要更高。因为，多个队列会有负载不均衡的问题。按这个逻辑，单元会尽量朝共享的方向发展。
- 所以实际设计会是两种模式的组合。有些单元做成私有的，有些做成共享的。
- 私有的好处是距离近速度快，硬件实现也更简单。坏处是不均衡时容易有闲置，导致浪费。如果一个单元利用率通常都很高，或是对延迟特别敏感，那就应该尽量做成私有的。NV最近架构（比如Turing、Ampere）的SM里的多数ALU单元都是私有的。也就是说，一个partition的warp应该是用不上其他partition的ALU单元。注意：这里说的ALU单元与指令并不一一对应，因为很多指令会复用同样的ALU单元，或者说是占用同一个dispatch port。
- 共享的好处是则是利用率会更高。不是所有的功能单元在程序中都会频繁使用，为了增加面积的利用效率，不同的ALU指令会设计成不同的throughput。对于低throughput的指令（比如消费卡的double指令，格式转换指令，超越函数指令等等），本身使用就不太频繁，那就比较适合做成共享单元。因为这类指令占比不大，本身空置率就高，共享可以让不同组之间相互插空，提高利用率。类似的，像Cache中用来做Atomic操作的ALU，感觉上也是可以共享的。

95.2 GPU核心的互联和通信

- SM的L1一般只缓存只读数据（比如对const指针或`__ldg()`函数的load，texture等，外部L2被改了它不会立即更新，所以这些read-only cache有时候也叫non-coherent cache），而写入一般是write-through。所以SM之间数据交互不会通过L1，而是要通过它们共同连接的L2。只要支持合适的memory consistency model，位于不同SM的block是可以通过L2进行相应的同步和交互的（有时候要借助atomic操作），这也是上面说的Cooperative Group的block间同步的逻辑。

96.Independent Thread Scheduling

Warp Divergence的由来及处理

- 条件跳转：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。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。
  - 一般由if-else/do-while触发
- 间接跳转：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。
  - switch-case/函数指针调用/虚函数
- 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。
- 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）。

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

- 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。
- 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)。
- 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。
- 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。

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。

```
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。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）。
```

```
// 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。

// 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。
```

- 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。

97.指令发射和warp调度

97.1指令发射的基本逻辑

- 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）。
- 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。
- 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。
- 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）。
- 在有多个warp满足发射条件的情况下，由于dispatch port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。
- 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。

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

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：

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。

- dispatch port组别：
    - Volta和Turing的F64、F16和Tensor Core都是用的同一个dispatch port；
    - 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；
    - memory指令一组；
    - branch指令一组；
    - 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。
- 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。

97.x 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是 eligible（合格的，有资格的）；
  - 不满足发射条件的warp就是 stalled；
- warp不能发射的原因：
  - **Pipeline Busy**：指令运行所需的功能单元正忙
  - **Instruction Fetch**：Instruction cache的miss。一般是第一次运行到的地方容易miss：BRA跳转到的地方或cacheline的边界处。
  - **Synchronization**：warp在等待同步指令。
  - **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。
  - **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。
  - `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。
  - **Texture**：Texture单元正忙，下发的request过多
  - **Constant**：Constant cache的miss。一般只会在第一次access的时候miss。
- warp scheduler会选择eligible队列里的一个warp来发射。应该是比较aggressive的往同一个warp发射指令，除非stall了。
  - 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。

97.x+1 峰值算力

- 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：
  - 乘2是因为 `FFMA` 是两个FLOP

> SM数 * 每SM的Core数 * 2 * 运行频率

- 不看Tensor core的话，满血版的卡一般有：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，一般满血版的功能单元配比就会尽量按极限的GPR带宽来设计。
- 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。

98.PTX

- PTX是NVIDIA官方支持的最底层，有相关的文档（见Parallel Thread Execution ISA）和完善的工具链（NVCC，cuobjdump，PTXAS等等），也可以在driver api中load，甚至支持cuda C中inline PTX assembly。而SASS这层只有非常简略的介绍SASS Instruction Set Reference，虽然其中也提供了一些工具如nvdisasm和cuobjdump做一些分析，但也非常局限。Debug上两者倒是差别不大，NSight功能比较完善了，现在应该是可以支持cuda C/PTX/SASS三个层级的debug。

99.SASS指令集

```
Opcode dst, src0, src1, src2, ... srcn  ;
;
FFMA R2, R4, R2, R5      ; float fused multiply-add R2 = R4*R2+R5
;
@!P0 FADD R2, RZ, -R2;      ; P0是1bit的bool谓词Predicate，!表示取反， @!P0表示P0为否时真正执行该操作，否则什么也不做
; -R2的负号是operand modifier 相当于取源操作数的相反数
;
FSETP.NEU.AND P0, PT, |R10|, +INF, PT   ; FSETP通过比较两个float来设置predicate，NEU是float比较中的Unordered Not-Equal，如果操作数中有NAN则返回True，AND表示比较完再加个AND后得到最终结果， PT是恒为True的Predicate，|R10|是R10加绝对值的modifier， +INF是一个float32的立即数。SETP是CAS操作
```

- SASS ISA的总体特征
  - 指令长度是定长的。Volta以前的几代都是64bit，Volta、Turing、Ampere都是128bit。有些指令是control code（编译器用来做线程控制的逻辑代码）
  - 是load/store型的ISA，但是也有例外。除了constant mem外，其他mem只能在load/store指令里当操作数，也就是必须load到GPR里才能使用。Ampere之前，没有直接mem到mem的操作，Ampere加了global到shared mem的操作。
  - 是较复杂的RISC，虽然是load/store型ISA，但很多复杂SASS指令。
  - 控制、debug、trap指令性能不高
- Predicate
  - 每个线程有8个predicate register P0~P7， 其中P7=PT。每个指令有4bit的编码来指定每个predicate，3bit用来索引，1bit用来表示是否取反。
  - 控制某个线程是否执行某指令的另一种方式：conditional branch。条件分支延迟和指令cache问题导致短分支更适合predicate，因为predicate可以让warp内的线程名义上走同一路径而省去跳转开销。

99.1指令分类

- Float指令：
  - float64，以D开头，如加法`DADD`，乘法`DMUL`，乘加`DFMA`等。
  - float32，以F开头，如`FADD`，`FFMA`，最大最小值`FMNMX`等
  - float16，以H开头（Half），如`HADD2`，`HFMA2`，比较`HSET2`等。

    - `MUFU` 指令，包括所有的特殊函数指令（SFU中执行的指令）。比如倒数 `rcp`，倒数平方根 `rsq`，对数 `lg2`，指数 `ex2`，正弦 `sin`，余弦 `cos` 等等。
    - FMA比MUL+ADD速度更快，精度更高。
    - 没有直接的除法指令，x/y用x*rcp(y)来算的。为了精度，浮点除需要多步迭代，所以慢。
    - 一个通用寄存器是32bit的，可以放俩F16，所以H开头的指令一般是H*2的，可以同时算两路。
    - 多数float指令都支持一些opcode modifier，比如四种round的模式（最近偶数 `RN`，趋0 `RZ`，趋负无穷 `RM`，趋正无穷 `RP`），是否flush subnormal（`FTZ`，flush to zero，把指数特别小的subnormal数变为0），是否饱和（`SAT`，指saturate，将结果clamp到[0,1]之间），等等。
- Integer指令：
    - 算术指令：如加法 `IADD`，`IADD3`，乘法 `IMUL`，32bit乘加 `IMAD` 或是16bit的乘加 `XMAD`（Maxwell或Pascal）。还用一些特殊算术指令，如 `ISCADD` 和 `LEA`，两者与 `IMAD` 很相似，但语义不同。还有如dot-product的 `IDP/IDP4A` 等等。
    - 移位指令：如左移 `SHL`，右移 `SHR`，漏斗移位（Funnel Shift）`SHF` 等等。
    - 逻辑操作指令：现在多数逻辑操作都用3输入逻辑指令 `LOP3` 来实现，它支持三输入的任意按位逻辑操作。
    - 其他位操作指令：如计算1的个数 `POPC`，找第一个1的位置 `FLO`，按位逆序 `BREV` 等等。这些指令在一些特殊的场合下会收到奇效，特别是在一些warp内的相互关联操作，能形成很精妙的配合。
    - 其他：比如 `IMMA`，`BMMA` 这种Tensor指令。
    - Turing的IMAD和LEA分属不同的发射端口，可以独立发射。Turing的IMAD带有MOV和SHIFT模式，很多时候用来做MOV操作，如IMAD.MOV.U32 R1, RZ, RZ，R0；相当于MOV R1, R0; Turing应该是把Float32和普通ALU的发射端口分开了，IMAD用float32的pipe，这样就能同时发射了。
- 格式转换指令：
    - cuda里带格式的GPR其实就两大类，整形(I)和浮点型(F)，所以格式转换主要就四个指令：`I2F, I2I, F2F, F2I`。然后opcode modifier会具体指定整形和浮点型的位数（主要是16，32，64），整形还分有符号无符号等。
    - `I2F` 和 `F2I` 中opcode modifer中I类型如果不写就默认是 `S32`，F类型不写就默认是 `F32`。
    - `FRND` 指令，是浮点到浮点的取整转换
- 数据移动指令：
    - GPR到GPR的 `mov` 多数情况下是不需要的
    - `MOV` 指令多数时候的用法是把一些立即数或是constant memory甚至是0，移动到GPR里。但是后面也会讲到，首先这些大多数可以直接做operand，二来往往有一些特殊的指令可以帮忙，比如设置0可以用 `CS2R R0, SRZ;`。搬运constant memory的方法就更多了。而且前面也讲了，`IMAD` 也可以做 `MOV`，还可以不占ALU的dispatch端口。
    - warp shuffle指令SHFL，适用场景scan(prefix sum)。Ampere里加了一个 `REDUX` 指令，可以直接做warp的reduction。
- Predicate操作指令：
    - 逻辑操作 `PLOP3`；
    - Predicate指令转存GPR：P2R和R2P；

- 内存操作指令：
  - memory的load操作：根据memory所属域的不同，Generic就用LD，global用LDG，local用LDL，shared用LDS，constant用LDC。如果有tensor load的功能，还有LDSM可以直接load matrix。
  - memory的store操作：与load对应，Generic用ST，global用STG，local用STL，shared用STS。由于constant memory是只读
  - memory的atomic操作：所谓的atomic操作一般都遵循read-modify-write的流程，常见操作有Compare-And-Swap（CAS），Exchange，Add/Sub（或者加减一Inc/Dec?），Min/Max，And/Or/Xor等等。根据对象的不同，Generic用`ATOM`，global用`ATOMG`，shared用`ATOMS`。constant只读，所以没有atomic操作。local memory是私有的，没有线程竞争，所以也没有atomic操作。
  - Cache control指令：这个主要意义是我知道我将要访问的某个元素位于某个cache line，但是又不确定具体要哪一个值，所以没法先load。所以可以先把整个cacheline放到cache里，等到要用的时候从cache里取。这样load的latency可以更好的被隐藏。
  - Texture操作指令
  - Surface操作指令
- 跳转和分支指令：

```
BRA 0x210            ; 定向跳转， 跳转目的地是确定的，可以跳或者不跳，由predicate实现   BRA for branch
BRX R10 -0xe50       ; 不定跳转， 由GPR确定跳转位置
BSYNC B1             ;
BSSY B0, 0x3c00      ;是否出现warp divergence不被当前线程所知，所以SASS里提供了一些分支同步点，显式的做branch的sync和converge(汇聚、合流)
; 跳转目标管理，Turing里面，直接显式的用GPR保存回跳位置，然后RET或是BREAK的时候把这个GPR作为操作数就行了
```

- 其他
  - BAR，barrier同步指令。
  - S2R和CS2R，把特殊register的值载入到GPR。
  - `blockIdx`对应`SR_CTAID`，`threadIdx`对应`SR_TID`，两者的xyz都有special register，那`blockDim`和`gridDim`呢？其实很简单，只有变动或是不确定的数才有放到special register内的价值，`blockDim`和`gridDim`在kernel运行时都是全局定值，现在都会被放在constant memory里。编译器会自动把相应访问转为对constant memory相应地址的访问
  - `VOTE`，这是warp的投票指令。一个warp有32个线程，每个都`True`或`False`，正好可以组成一个32bit的数
  - `NOP`，啥也不做，就占个位置，让指令section满足对齐要求。
- Uniform DataPath指令：
  - 从Turing开始，SM里加入了一类新的ALU功能单元，用来进行warp内的一些公共计算。与此相配套的，自然就有uniform datapath的指令，还有Uniform的register和preidcate。如果说其他指令是32 lane的vector指令，这个就是1 lane的scalar指令。Uniform datapath的指令是针对warp而言，相当于每个warp只需要单个执行即可。
  - Uniform datapath指令只支持int的ALU指令，并没有float，也没有跳转，也没有memory访问（constant memory除外）。
  - Uniform datapath也支持guard predicate，一般写@UP0。

99.2 Operand 和 Operand Modifier

- GPR：通用寄存器是thread内最常用也是量最大的资源。因此绝大多数的指令的操作对象都是GPR。
  - `R`加一个十进制数来表示某个GPR， 8bit，最大就是R255。R255称为RZ，表示它一定为0。
  - GPR是32bit的，对于需要更多位数的指令，会占用连续多个GPR，但在反汇编里只会写第一个。
    - LDG.E.128.SYS R4, [R26] 中 R26其实是64bit的地址，实际上是R[26:27]；R4实际上是R[4:7]
  - 这里还有一个隐式的对齐要求，连用两个GPR的话就需要第一个是偶数（GPR编号从0开始），连用四个的话第一个就一定是4的倍数。
  - 一个指令最多可以支持4个GPR操作数（64bit或128bit也算一个操作数），三个src，一个dst。3个GPR的src中有一个GPR可以被替换成后面的Constant memory、Immediate或是Uniform register，但只能替换一个。
  - GPR支持很多种形式的operand modifier，比如前面提到的float的取相反数（`-R0`）和绝对值（`|R0|`）。印象中整形也可以取相反数，绝对值相对少见，但是有按位取反（如`~R0`）。这些operand modifier都可以用到constant memory和Uniform register上。
  - Opcode有一些modifier其实是dst的operand modifier，比如是否饱和（`SAT`）；有些是src和dst通用的modifier，比如`FTZ`。

```
FFMA R0, R1, R2, R3 ;                                  // R0 = R1*R2 + R3, 全是普通GPR输入
FFMA R5, R10, 1.8446744073709551616e+19, RZ ; // a或b是立即数，a、b等价，可互换，所以一个支持就够了
FFMA R2, R10, R3, -1 ;                                 // c是立即数
FFMA R5, R5, c[0x0][0x160], R6 ;                       // a或b来自constant memory
FFMA R0, R5, R6 , c[0x0][0x168];                       // c来自constant memory
FFMA R14, R19, UR6, R0 ;                               // a或b来自uniform register
FFMA R18, R16, R13, UR6 ;                              // c来自uniform register
```

- Predicate Register
  - predicate做操作数的场景还是比较多的，比如用做carry，用做比较操作的结果，用作`SEL、FSEL、IMNMX、FMNMX`等指令的条件判断输入
- Constant memory
  - 多数ALU指令都支持把其中的一个本来是GPR的操作数换成Constant memory
  - constant memory也支持32bit和64bit的模式
  - 一般说来，取代原GPR操作数的constant memory可以支持相同的operand modifier
  - 一般constant memory的写法类似`c[0x0][0x10]`，前面中括号表示bank，后面表示地址。现在constant memory主要有两个来源，一是用户显式的用`__constant__`声明的，二是driver或编译器自动生成的，包括kernel的参数，blockDim，gridDim等等。这两个一般会放在不同的bank里。还可能有一类是编译器自动把一些编译期常数放到constant memory里
- 立即数
  - SASS主要有两种立即数，整形和浮点型。
  - 整形的位数和是否有符号是由指令决定的。

    - 浮点型比较有意思，当前支持三种浮点型F16，F32，F64。但是立即数最多支持32bit编码，所以F64的浮点数是放不下的。一般F64的立即数只保留了前32bit（注意还是用F64的格式，不是F32格式）。这样有些浮点值就不能精确表示了。
- Uniform Register和Uniform Predicate
    - UR和UP是Turing架构开始才有的操作数。当前UR编码是到64个，`UR63=URZ`也是恒0，UP和普通predicate的编码是一样的。
    - UR因为资源开销并不大，感觉并没有按总量分，应该是见者有份，每个warp都可以用满63个。
    - UR和UP可以作为Uniform datapath指令的操作数，也可以做普通指令的操作数。但普通的GPR和predicate不能做Uniform datapath的操作数，因为用了可能就不Uniform了。
- 地址操作数
    - global的地址其实是虚拟地址，base一般是一个不确定的值。但LDS一般用的是实地址，base从0开始（相当于当前block分配到的第一个byte）。所以LDS支持`R0.X4`、`R0.X8`、`R0.X16`这种模式，相当于`R0`可以不用乘element的byte数了。
    - 一般global的最终地址肯定要是64bit，`LDG.E.U16.CONSTANT.SYS R6, [R6.U32+UR4+0x2200] ;`里应该UR4是64bit，R6是32bit。
    - shared memory因为空间小，用32bit就足够了，所以肯定都是32bit的运算。
- 其他

```
CS2R R4, SRZ ;
S2R R20, SR_CTAID.X ; blockIdx.x
S2R R5, SR_LANEID ;
BMOV.32 B6, R2 ;
DEPBAR.LE SB0, 0x0 ; 依赖barrier
DEPBAR.LE SB0, 0x0, {2,1} ;

R2P PR, R98.B1, 0x18 ;
P2R.B1 R98, PR, R98, 0x18 ;
```

- Control Code
    - Control code主要有reuse、read barrier、write barrier、wait barrier、yield hint、stall count等几个域。reuse是唯一能在反汇编文本里看到的。它可以有限的解决一些GPR的bank conflict问题，也许同时还能减少GPR的读写，节省一点功耗。关于barrier的几个主要控制thread内的依赖性问题，这会影响指令发射的仲裁过程。yield和stall count主要影响的是warp调度的逻辑。

论文

200.Img2Col

200.1为什么要加速卷积?

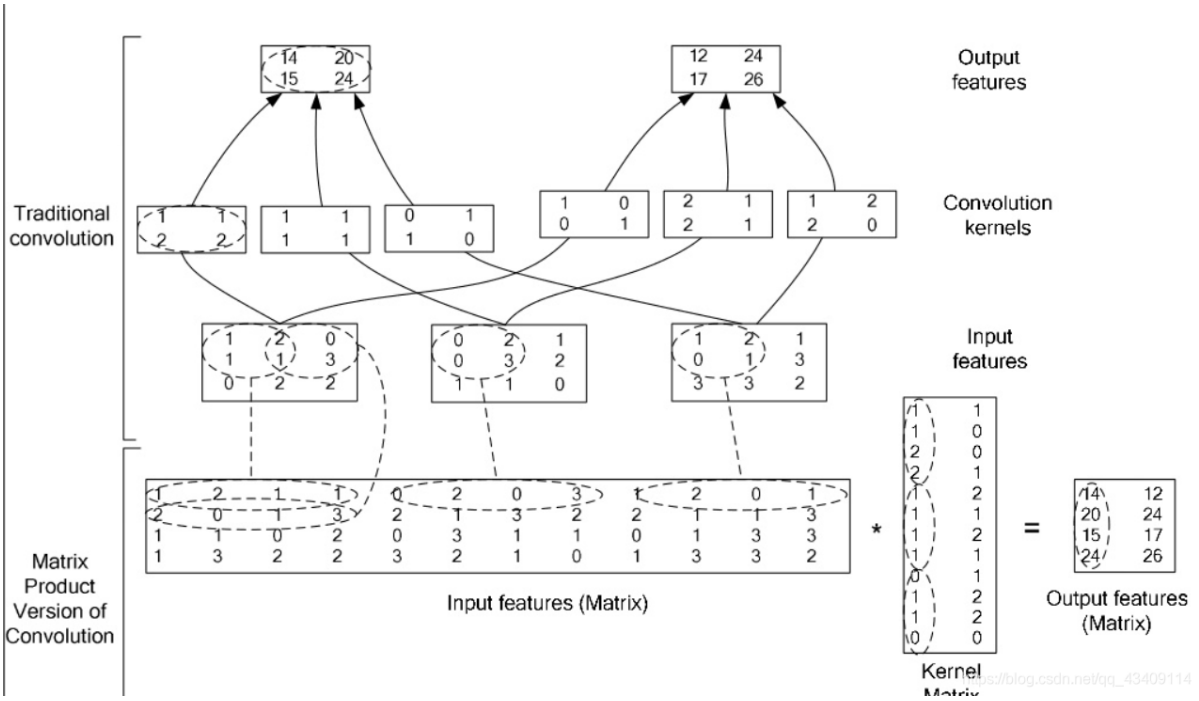
- 计算量随着卷积核length的增大而呈二次方增长。因为4重循环的内2重是遍历卷积核的channel。
- 小的kernelsize会造成过多的 JMP 指令
- 前向和反向传播需要列式访问数据，这违背了缓存的空间局部性原理。

200.2 CNN基础

- 一个特征图像的3个channel具有相同的感受野，共享参数的是不同的感受野，不是一个感受野的3个channel。
- 卷积就是图像特征和卷积核的对应元素乘，然后累加成标量。如果是多channel的话，多个channel的标量结果，再加成单值，就是卷积结果元素。
- 要想结果也是多通道的，那么需要多个卷积核。

200.3 Img2Col

- stride决定了数据元素被复制的次数。但复制的成本与计算成本相比，几乎可以忽略不计。



201.Roofline

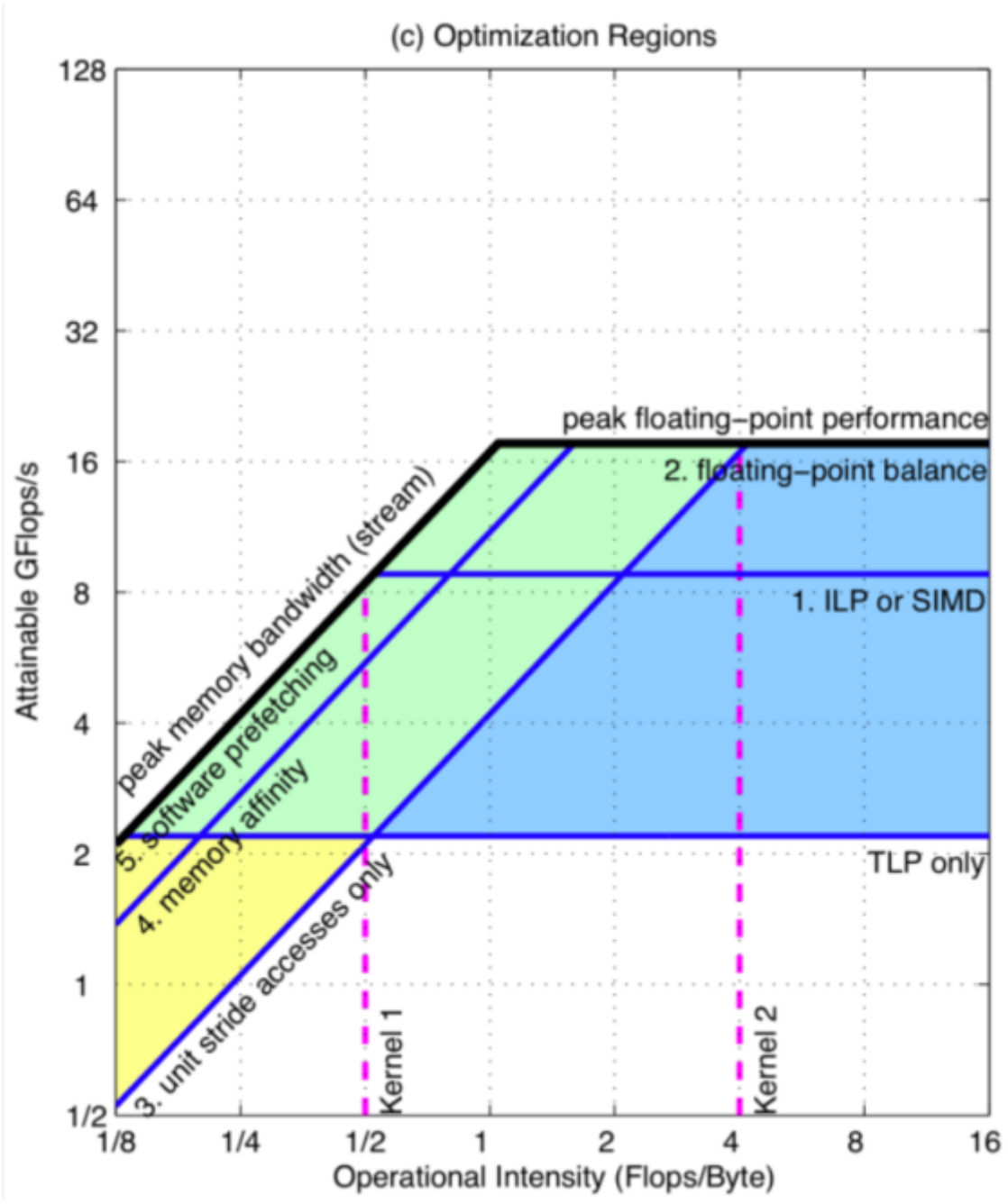
201.1 Roofline模型

- Roofline模型：将处理器性能和片外存储器流量关联起来的模型
- **operational intensity（运算强度）**：DRAM流量的每字节运算数**Flops/Byte**；我们将访问的总字节数定义为经过缓存层次结构过滤后进入主内存的字节数。也就是说，我们测量缓存和内存之间的流量，而不是处理器和缓存之间的流量。因此，运算强度 表明特定计算机上的kernel所需的DRAM带宽。
- 所提出的模型将浮点性能、运算强度和内存性能在二维图中联系在一起。可以使用硬件规格或微基准测试来找到峰值浮点性能。
- 但在这项工作中，我们编写了一系列逐步优化的微基准测试，旨在**确定可持续的 DRAM 带宽**。它们包括**获得最佳内存性能的所有技术，包括预取和数据对齐。**
- Roofline是多核计算机的属性，不是kernel的属性，对于确定的计算机，只需要做一次。
    - 横轴代表Operational Intensity是 Flops/Byte
    - 纵轴代表可达的浮点运算性能是 Flops/s
    - 首先画一条水平线表示硬件限制的峰值浮点运算性能

    - 峰值内存带宽 Byte/s 就是 下图中的45度角的斜线
        - 如果是自己画，在知道峰值浮点性能和峰值内存带宽的情况下，定交叉点（山脊点）的运算强度为单位1，然后根据下述公式画图？
    - 可达到的 GFlops/s = min （峰值浮点性能、峰值内存带宽 x 运算强度）
- 对于给定的kernel，我们可以根据其运算强度在 X 轴上找到一个点。如果我们通过该点画一条（粉红色虚线）垂直线，则该计算机上kernel的性能必须位于该线的某个位置。
- Roofline 根据kernel的运算强度设置kernel性能的上限。如果我们将运算强度视为一根撞击屋顶的柱子，那么它要么撞击屋顶的平坦部分，这意味着性能受到计算限制，要么它撞击屋顶的倾斜部分，这意味着性能最终受到内存限制。
- 山脊点的相对靠左或靠右，表明了实现峰值性能的难度。

201.2 给模型添加天花板

- 我们向 Roofline 模型添加多个上限，以指导执行哪些优化。我们可以将这些优化中的每一项视为低于相应 Roofline 的“性能天花板”，这意味着如果不执行相关优化，就无法突破天花板。
- 例如，为了减少计算瓶颈，有两种对任何kernel都适用的策略：
    - 提高ILP(instruction level parallelism)并利用SIMD。
        - 对于超标量架构，当每个时钟周期获取、执行和提交最大数量的指令时，性能最高。这里的目标是改进编译器的代码以增加 ILP。最高的性能来自于完全覆盖功能单元的延迟。一种方法是 **展开循环**。对于基于 x86 的架构，另一种方法是尽可能使用浮点 SIMD 指令，因为 **SIMD** 指令对相邻操作数对进行操作。
    - 平衡浮点运算组合。
        - 最佳性能要求指令组合的很大一部分是浮点运算。峰值浮点性能通常还需要相同数量的同时浮点加法和乘法，因为许多计算机具有乘加指令或者因为它们具有相同数量的加法器和乘法器。
- 例如，为了减少内存瓶颈，有三种策略：
    - 重构单位步长访问的循环。单位步长内存访问的优化涉及硬件预取，这显着增加了内存带宽。
    - 确保内存亲和力。这种优化将数据和负责该数据的线程分配给同一对内存-处理器，因此处理器很少需要访问连接到其他芯片的内存。
    - 使用软件预取。
- 与计算 Roofline 一样，计算上限可以来自优化手册，尽管很容易想象从简单的微基准收集必要的参数。内存上限需要在每台计算机上运行实验来确定它们之间的差距。



201.3 将3C和运算强度联系起来

- kernel的运算强度并不是恒定的，它可能随问题规模的增加而增加，如稠密矩阵和FFT（快速傅里叶变换）问题。
- 显然，缓存会影响对内存的访问次数，因此提高缓存性能的优化会增加运算强度。因此，我们可以将 3C 模型连接到 Roofline 模型。强制未命中设置了最小内存流量，从而设置了最高可能的操作强度。来自冲突和容量miss的内存流量可以大大降低内核的操作强度，因此我们应该尝试消除此类miss。
    - 可以通过填充数组来更改缓存行寻址来减少冲突未命中造成的流量。
    - 某些计算机具有不分配存储指令，因此存储直接进入内存并且不会影响高速缓存。这种优化可以防止加载带有要覆盖的数据的缓存块，从而减少内存流量。它还可以防止用不会读取的数据替换缓存中的有用项目，从而避免冲突miss。
- 建议 **先提高内核的运行强度再进行其他优化。**

201.4 案例

202.Turing T4 via MicroBenchmark

- float4加载能一定程度提升ld的效率
- 128bit的指令中 指令信息至少91位，控制信息至少23位，其余14位在turing和volta中似乎未被使用
- 当一条可变延迟指令写入一个寄存器时，汇编程序会通过设置相应的 写入屏障索引 字段，将其与 6 个可用屏障之一相关联。当后面的指令使用该寄存器时，汇编程序通过设置 等待屏障掩码 中与该屏障对应的位，将该指令标记为在对应屏障上等待。硬件将停止后一条指令的执行，直到前一条指令的执行结果出来。一条指令可能会在多个屏障上等待，所以waitdb是位掩码
- readdb 用来防止写后读冒险。后序指令在写相同寄存器前，需要等待前序指令的readdb。
- stall cycles：这个 4 位字段表示调度程序在发出下一条指令前应等待的时间，从 0 到 15 个周期不等。
- Turing没有dual issue
- yield：当该位被置位时，调度程序倾向于从当前处理器发出下一条指令。当该位清零时，调度器倾向于切换到另一个 warp，使得下一条指令的所有寄存器重用标志无效。这种情况下，切换到另一个 warp 需要一个额外的周期。
- turing里，warp被映射到SM分区（或scheduler）的规则是 scheduler_id = warp_id % 4;
- 这说明 对于有4个分区的SM来说，每个block至少要有128个线程才能用满处理单元（scheduler或cuda core）
- 谓词由 4 个比特位调节：第一个比特位是否定标志，其余 3 个比特位编码谓词寄存器索引。

203.通过MicroBench解析GPU内存层次结构

- 细粒度p-chase microbenchmark来揭示GPU缓存参数
  - L2 TLB的 unequal sets
  - 纹理L1缓存的 2D空间局部性优化组关联映射
  - L1数据缓存的非传统替换策略（也即不是LRU）
- 对GPU 全局内存和共享内存的吞吐量和访问延迟进行定量基准测试。
  - 影响内存吞吐量的各种因素
  - 共享内存存储体冲突对内存访问延迟的影响
  - 验证：为了避免共享存储体冲突下的长延迟，Maxwell进行了高度优化。
- GPU用page table来做虚拟地址到物理地址的映射，页表存在global mem里。TLB是页表的cache。
- compute3.5以上的只读数据缓存还可以缓存内核整个生命周期内只读的全局内存数据。

L1 cache策略

204.TuringAs

- 在Turing和Volta中，四个32位寄存器组已被两个64位寄存器组取代，奇数索引寄存器驻留在一个组中，偶数索引寄存器驻留在另一组中。宽64位寄存器组使得寄存器组冲突不太可能发生。
-

其他

- CUDA很多错误是不造成CPU中断或抛出异常的，需要手动check返回值。Kernel运行没有返回值，需要cudaDeviceSynchronize()之后，用cudaGetLastError()来检查。
- Tesla、Fermi、Kepler、Maxwell、Pascal、Volta、Turing、Ampere

	80	**81**	**82**	**83**	**88**	**89**	**90**	**91**
76	92-							
77	93							
78	94-							
79	24							
84	28-							
85	32							
86	36-							
87	40							

乘法

- 内积：又叫点积。
  - 向量对应元素积的和。
  - 矩阵对应元素积的和。
- 叉积：又叫向量积。
  - 向量a、b叉积的结果是与a、b都垂直的向量c，模长为|a||b|sin。

- 外积：两个向量外积的结果是矩阵。
- 元素积：又叫逐项积、Hadamard积。element-wise
    - 结果矩阵与原矩阵形状相同，新元素是旧矩阵对应元素的积。

在cublas中使用TensorCore

```
// First, create a cuBLAS handle:
cublasStatus_t cublasStat = cublasCreate(&handle);

// Set the math mode to allow cuBLAS to use Tensor Cores:
cublasStat = cublasSetMathMode(handle, CUBLAS_TENSOR_OP_MATH);

// Allocate and initialize your matrices (only the A matrix is shown):
size_t matrixSizeA = (size_t)rowsA * colsA;
T_ELEM_IN **devPtrA = 0;

cudaMalloc((void**)&devPtrA[0], matrixSizeA * sizeof(devPtrA[0][0]));
T_ELEM_IN A  = (T_ELEM_IN *)malloc(matrixSizeA * sizeof(A[0]));

memset( A, 0xFF, matrixSizeA* sizeof(A[0]));
status1 = cublasSetMatrix(rowsA, colsA, sizeof(A[0]), A, rowsA, devPtrA[i],
rowsA);

// ... allocate and initialize B and C matrices (not shown) ...

// Invoke the GEMM, ensuring k, lda, ldb, and ldc are all multiples of 8,
// and m is a multiple of 4:
cublasStat = cublasGemmEx(handle, transa, transb, m, n, k, alpha,
                          A, CUDA_R_16F, lda,       // CUDA_R_32F也行
                          B, CUDA_R_16F, ldb,
                          beta, C, CUDA_R_16F, ldc, CUDA_R_32F, algo);
```

理论计算

- 延迟是从发出指令到最终返回结果间的时间间隔。
- 吞吐（带宽）是单位时间内处理的指令条数。
- 算力峰值：FP32 core * SM频率 * 2，单位FLOPS
    - 3060：128 * 30 * 1702M * 2 = 13071360M 大概13TFlops
- 主频是每秒钟产生的时钟周期数，3060SM的主频是1702MHz, 则每个cycle是1/1702M 秒。

- 内存频率而言，DDR4的运行速度与GDDR5X和GDDR6大致相同（1750至1800MHz），但是图形内存的工作方式意味着有效带宽是原来的4倍（1750 * 4 = 7,000MHz）。

- DDR4只能在一个周期内执行一项操作（读或写）。GDDR5和GDDR6可以在同一周期内处理输入（读取）和输出（写入），实质上使总线宽度加倍。

FP16

- nv的fp16是1位符号位、5位指数位、10位小数位。
- fp32是1位符号位、8位指数位、23位小数位。
- 直接从fp32截取高16位就是1位符号位、8位指数位、7位小数位。这是TensorFlow的fp16规格。