GPGPU中一些问题的理解与思考（3）- 指令执行吞吐与指令集设计

https://zhuanlan.zhihu.com/p/391238629

None Tue Jul, 20 00:16

今天这次来聊一聊GPGPU指令吞吐和指令集设计的一些问题。NV GPU的机器码指令集叫SASS，之前专栏有几篇文章专门介绍过，对CUDA微架构和指令集不太熟悉的同学可以先参考下：

cloudcore：CUDA微架构与指令集（2）-SASS指令集概述zhuanlan.zhihu.com

cloudcore：CUDA微架构与指令集（3）-SASS指令集分类zhuanlan.zhihu.com

cloudcore：CUDA微架构与指令集（4）-指令发射与warp调度zhuanlan.zhihu.com

指令集是微架构与用户对接的途径。通俗点说，指令集相当于硬件提供给软件的API（或者也可以认为指令集是输入前端，微架构是执行后端）。所以指令集的很多内容都是与微架构深度绑定的。但是，微架构提供的功能是一方面，指令集的接口设计上也有很多值得思考的问题。指令集设计的系统思想有很多内容，很多讲计算机体系结构的书都会有所涉及。这里我不过多的介绍一般思路，主要会聊GPU指令中与常见CPU思路不一致的地方，或者说是有其特殊性的地方。

这里仍然主要介绍NV GPU。其他GPU或是NV不同代的微架构之间，也会有一些或大或小的差异，因此一些表述既不严谨，也不全面，仅供参考。

按照惯例，还是先简单列一下大纲：

- GPGPU指令执行简介
    - GPGPU指令执行流水线
    - GPGPU指令执行吞吐的影响因素
    - GPGPU指令执行的特点
- 指令设计中的一些原则与思路
    - 指令长度的问题
    - 指令集设计与ILP的一些相关性
    - 复合操作与附加操作
    - 立即数操作数和Constant Memory操作数
    - 格式的通用性与信息具体化
- 简单聊聊一些具体的指令
    - FMA, MUL, ADD 系列
    - IMAD, LEA, IADD3
    - LOP3
    - MUFU
    - FSETP
    - LDG/STG, LDS/STS, LDL/STL
    - BAR
- 结语

GPGPU指令执行流水线

首先，先简单介绍一下通用处理器的指令执行逻辑。这里参考《Computer Architecture：A Quantitative Approach》第6版中对RISC V一种简单的五级流水线实现的描述(参考该书页面C-4，这里的内容基本为原文复述加个人补充)：

1. Instruction fetch cycle (IF)：主要是获得Program Counter对应的指令内容。
2. 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。
3. 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。

4. 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。
5. 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。

当然，这只是RISC V流水线的其中一种实现方式，实际实现肯定会有很多调整。

与复杂的CISC指令集相比，多数GPGPU的指令集还是比较接近load-store machine，总体来说与RISC更相似一些。GPGPU典型微架构可以简单表示为下图（简单示意图，引自《General-Purpose Graphics Processor Architecture》, Tor M. Aamodt et al., P22）：

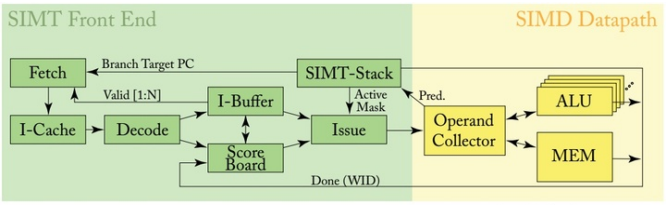

Figure 3.1: Microarchitecture of a generic GPGPU core.

GPGPU指令执行流程示意图

这个图看起来好像和上面介绍的经典五级流水线还是区别很大，因为这里画的是微架构的模块示意图，而非流水线示意图。实际执行流程中Fetch、Decode、Execution这三步是必须的，而Memory access显然只针对memory指令，write-back则只针对需要写回的指令（比如memory load，带GPR输出的指令等）。

注：流水线的配置与ALU的latency有很大的关系。比如Volta前FFMA的延迟是6cycle，Volta及之后FFMA的延迟是4cycle，这绝对与流水线的改进有关。不过，这里的Latency并不是所有流水线的级数。因为Latency在程序中的表现形式是：一个指令发射后，其结果需要多少周期才能就绪（也就是能被其他指令使用）。两个back-to-back dependent的ALU指令（比如`FFMA R0, R1, R2, R0; FFMA R0, R3, R4, R0;`），前一个FFMA只要在第二个FFMA读取操作数之前把结果写回GRF，那后一个FFMA就可以得到正确值。对应到上面的5级流水线形式，就是前一个指令的WB要在后一个指令的ID前执行完就行（相当于4cycle延迟），最开始的IF那一级是不影响的。CPU对于这种形式的依赖还有更激进的旁路逻辑（forwarding），可以直接在前一个ALU的EX后把结果直接送给后一个ALU的EX当输入，从而减少流水线的bubble，提高性能。NV的GPU应该是没有这么紧凑的forwarding，但是NV的operand collector可以作为一个公共的操作数中转站，理论上前一个ALU的结果写回到operand collector就可以被下一个ALU看到了，不一定要回到GRF。当然，这个具体流水线的实现我也不是很清楚，有兴趣的同学可以尝试查查NV的专利。

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。

当然，步骤虽然多而琐碎，但未必都会新增单独的一级流水。GPU应该是为了简化设计和节约功耗，不愿意把流水线拉得太细长，因而很多操作都是塞在同一级流水线里，各种组合逻辑非常复杂。这样也导致它的主频往往就不能太高。比如最近几代NV GPU旗舰和次旗舰的主频：

GPU	Base Clock	Boost Clock
RTX 3090	1395	1695
RTX 3080	1440	1710
RTX 3070	1500	1725
RTX 2080 Ti	1350	1545/1635
RTX 2080 Super	1650	1815
GTX 1080 Ti	1480	1582
GTX 1080	1607	1733
GTX 980 Ti	1000	1075
GTX 980	1126	1226

可以看到主频基本都在1~2 GHz之间，次旗舰的频率往往比旗舰要稍高一些（这里选的公版频率，但非公也大致是这个趋势），有些低端芯片频率可能还会更高一点。而如今（2021年）常见的桌面端x86 CPU，基准频率3~4 GHz，最大睿频4~5 GHz是很寻常的事。很多Arm CPU的大核，主频也能接近甚至超过3 GHz。当然，这么比也许不是特别公平。因为多数独立GPU的功耗很大，会极大的限制频率提升。一些众核CPU的频率也会比少核版的降一些，不过差别不会太大（类似上面GPU的旗舰与次旗舰的关系）。但即使算上这些，GPU的主频比常见CPU的主频还是显著低一些（实际上带核芯显卡的CPU里，GPU频率往往也是显著小于CPU频率）。这里面具体的因果关系我也不是特别明白，感觉肯定还是有些故事的~

GPGPU指令执行吞吐的影响因素

指令执行吞吐一般指的是每个时钟周期内可以执行的指令数目，不同指令的吞吐会有所不同。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。而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。当然，有些微架构下也会有1/3、1/6之类非2的幂次的比率。NV GPU近几代微架构的常见指令吞吐如下（参考）：

Table 3. Throughput of Native Arithmetic Instructions. (Number of Results per Clock Cycle per Multiprocessor)

	Compute Capability								
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128	256[3]	
32-bit floating-point add, multiply, multiply-add	192	128		64	128		64		128
64-bit floating-point add, multiply, multiply-add	64[4]	4		32	4		32[5]	32	2
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (__log2f), base 2 exponential (exp2f), sine (__sinf), cosine (__cosf)	32			16	32		16		
32-bit integer add, extended-precision add, subtract, extended-precision subtract	160	128		64	128		64		
32-bit integer multiply, multiply-add, extended-precision multiply-add	32	Multiple instruct.					64[6]		
24-bit integer multiply (__[u]mul24)	Multiple instruct.								
32-bit integer shift	64[7]	64		32	64				
compare, minimum, maximum	160	64		32	64				
32-bit integer bit reverse	32	64		32	64		16		
Bit field extract/insert	32	64		32	64		Multiple instruct.		
32-bit bitwise AND, OR, XOR	160	128		64	128		64		
count of leading zeros, most significant non-sign bit	32			16	32		16		
population count	32			16	32		16		
warp shuffle	32						32[8]	32	
warp reduce	Multiple instruct.							16	
sum of absolute difference	32	64		32	64				
SIMD video instructions vabsdiff2	160	Multiple instruct.							
SIMD video instructions vabsdiff4	160	Multiple instruct.					64		
All other SIMD video instructions	32	Multiple instruct.							
Type conversions from 8-bit and 16-bit integer to 32-bit integer types	128	32		16	32		64		
Type conversions from and to 64-bit types	32[9]	4		16	4		16[10]	16	2
All other type conversions	32			16	32		16		

CUDA算术逻辑指令吞吐表

从图中可以发现，指令吞吐不仅与指令类型有关，还与微架构具体设计实现有关。它主要会受到以下一些因素的影响：

1. **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。比如浮点的FFMA、FMUL都要用到一个至少24bit的整数乘法器（32bit浮点数有23bit尾数，小数点前还有1bit）。以前一些处理器有24bit的整数乘法指令，两者乘法器就可以共用，从而具有相同的吞吐（不过NV最近几代好像都没有这个指令，ptx以及内置函数的24bit乘法应该是多个指令模拟的）。而FADD虽然用不上那个乘法器，但可以与FFMA共用那个很宽的加法器，以及一些通用的浮点操作（特殊数的处理，subnormal

flush之类）。32bit的整数乘法因为需要更宽的乘法器，有的就不会做成full throughput，甚至可能被拆分成多个指令（比如Maxwell和Pascal用三个16bit乘法指令XMAD完成一次32bit整数乘法）。Turing的IMAD应该是有意识的加宽了，所以32bit的IMAD与FFMA吞吐一样，但印象中带64bit加数的IMAD应该还是一半。再比如一些超越函数指令（MUFU类，比如rcp，rsq，sin，exp之类），由于实际使用量相对不会太频繁，多数是1/4的throughput。

2. 指令**Dispatch Port**和**Dispatch Unit**的吞吐。这个在之前的专栏文章也详细讲过。一个warp的指令要发射，**首先要eligible**，也就是不要因为各种原因stall，比如指令cache miss，constant immediate的miss，scoreboard未就位，主动设置了stall count等等。**其次要被warp scheduler选中**，由Dispatch Unit发送到相应的Dispatch Port上去。Kepler、Maxwell和Pascal是一个Warp Scheduler有两个Dispatch Unit，所以每cycle最多可以发射两个指令，也就是双发射。而Turing、Ampere每个Warp Scheduler只有一个Dispatch Unit，没有双发射，那每个周期就最多只能发一个指令。但是Kepler、Maxwell和Pascal都是一个Scheduler带32个单元（这里指full-throughput的单元），每周期都可以发新的warp。而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。**最后要求Dispatch Port或其他资源不被占用**，port被占的原因可能是前一个指令的执行吞吐小于发射吞吐，导致要Dispatch多次，比如Turing的两个FFMA至少要stall 2cycle，LDG之类的指令至少是4cycle。更详细的介绍大家可以参考之前的专栏文章。
3. **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。
4. 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。

几个需要注意的点：

1. 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。

2. 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。这也是通常计算峰值FLOPS时乘2的由来。但是有些指令，可以计算更多FLOP。比如 `HFMA2 R0, R1, R2, R3;` 可以同时算两组F16的FMA，相当于每个GPR上下两个16bit分开独立计算（类似于CPU的SIMD指令），所以SM86以前的架构HFMA2的指令吞吐与FFMA是一样的，只是每条指令算4个F16的FLOP，而FFMA是2个F32的FLOP。这也就是TensorCore出现前F16的峰值通常是F32两倍的原因。DFMA由于输入宽度比FFMA再翻倍，所以功能单元做成一半就能把GPR吞吐用满（这里说的是满配Tesla卡，消费卡F64常有缩减）。因此，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。当然，这里说的是一般情况，实际上根据产品市场定位的不同，有些功能可能会有所调整。
3. SM86（Ampere的GTX 30系列）的F32比较另类。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。

GPGPU指令执行的特点

与传统的x86 CPU相比，GPGPU在指令执行的逻辑上有很多独特的地方。

静态资源分配：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（Block是GPU进行运行资源分配的单元，也是计算Occupancy的基础）。典型的运行资源有GPR和shared memory。GPU程序运行过程中，一般也不会申请和释放内存（当然，现在有device runtime可以在kernel内malloc和free，供Dynamic Parallelism用，但这个不影响当前kernel能用的资源）。CPU在运行过程中有很多所需的资源是动态调度的。比如，x86由于继承了祖上编码的限制，ISA的GPR数目往往比物理GPR少，导致常常出现资源冲突造成假依赖。实际运行过程中，通常会有register renaming将这些ISA GPR映射到不同的物理GPR，从而减少依赖（有兴趣的同学

可以研究下tomasulo算法）。GPU没有这种动态映射逻辑，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。我暂时没测试到barrier的数目对Occupancy的影响，也许每个block都可以用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。因为每个warp都可以分配到足额，应该对Occupancy也没有影响。另外每个线程有7个predicate，每个warp有7个Uniform predicate，这些也是足额，也不影响Occupancy。

GPU里还有一种半静态的stack资源，通常也可以认为是thread private memory或者叫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（参考CUDA C Programming Guide, Table 15, Maximum amount of local memory per thread=512KB）。

顺序执行：乱序执行是CPU提高CPI的一个重要途径，但乱序执行无论是设计复杂度还是运行控制的开销都很大。CPU的乱序执行可以把一些不相关的任务提前（相关的也可以乱序，但要求顺序提交），从而提高指令并行度，降低延迟。而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。在x86的CPU中，memory结果的可见性是通过缓存的一致性来控制的，这样read-after-write之类的组合可以通过cache的可见性来保证，但多线程的情况也需要通过coherence和memory consistency model来保证。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，具体大家可以参考PTX文档: Memory Consistency Model。

当然这个逻辑虽然是这么设计的，估计偶尔也会有出bug的时候。Maxwell和之前的架构偶尔能看见编译器往程序内插一些NOP。大概就是硬件上有问题，靠编译器来强行修补。Turing上似乎已经比较少见了。

指令设计中的一些原则与思路

指令长度的问题

至少从Kepler开始，SASS就是定长的了。Kepler每条指令64bit，每8条指令含一条control code，之后的Maxwell、Pascal每条指令还是64bit，但是control code变成了每4条指令一条。Volta、Turing、Ampere都是每条指令128bit，每条都自带control code。

定长和变长有什么差别呢？一般讲体系结构的书上会用RISC与CISC来做对比，因为一般CISC指令集是变长的，比如x86。而RISC则通常是定长的。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。而变长则只能顺序解码，不得到前一条指令长度，后一条指令就不知道从哪里开始，当然就无法解码。同时由于长度是变化的，每个操作数的类型和内容都可能会变化，解码也就更繁琐一些。不过变长有一个好处就是可以压缩一些常用指令的长度，从而减少程序大小。而定长就没有这种方式。

不过，现在很多架构和指令集设计都渐渐趋同。比如x86虽然变长（从一个Byte到十几个Byte），但它解码前会先定界，然后可以经过预解码变成一系列的Micro Operation，这样真正执行的时候也类似RISC。而RISC也不都是定长的，比如ARM的Thumb模式可以混用16bit和32bit的指令，RISC-V也可以加长指令实现特定的功能扩展。

对于GPU来讲，我觉得长指令还是有很大的好处的。我的理由如下：

1. 指令Cache是GPU中命中率最高的Cache之一（constant cache也许是真的No.1）。一个warp 32线程已经让指令解码和调度之类的开销大大均摊了。而同时大部分代码都会被成千上万

个warp运行，这个开销还可以被平摊得更小。当然，每个指令变长会加重指令Cache的容量和吞吐负担，但它的格式也可以做得更整齐更有规律，从而大大简化解码过程。像x86这种编码复杂的指令集，连解码都会添加相应的cache（Micro-Op Cache）。而只要指令够长够整齐，解码就可以做得够简单。这高ICache命中率的GPU来说，总体来讲还是赚的。或者你可以认为，x86的Micro-op cache主要利好循环这种高重用代码，而GPU每条代码都与循环类似，那不如直接就用解码后的指令做输入。

2. 前面提过指令执行的颗粒度。由于GPU流水线和发射逻辑的限制，每条指令都有基础开销（至少占用一次Dispatch Unit和相应Port），那在同一条指令中塞进更多信息，就成为减少指令基础开销的重要手段。指令越长，能编码的信息就越多，比如指令内嵌的立即数就可以支持更长。典型的如ALU的立即数操作数，load/store的offset，branch的跳转目标等。同时，多数指令也可以支持更复杂的modifier，从而表述能力和可编程性都更强。同时，更整齐规范的格式也更有利于编译器做优化，性能的可及性更好。
3. 指令够长，就可以加入更多的调度和控制信息，从而简化硬件自主判断，拓宽指令观察视野。Control code就是一个典型的例子。大部分的Control code其实是编译期信息，没有control code，其中很多依赖的判断就需要专门的硬件逻辑去检测和处理。而受限于硬件实现的复杂度，其视野肯定不如编译时的全局视野宽。因此，通过这些控制信息，可以更好的利用编译器在编译时获得的先验知识，减少硬件自主分析和处理，从而简化硬件设计，提高面积利用率的同时也简化设计和验证过程。

Volta开始的128bit指令，我觉得还是一个很有意义的尝试。也许有人觉得这个也许可以做点内存压缩，不过我感觉意义不是太大，压缩和解压其实也算是编码解码过程，寻址还会增加负担，未必便宜。

指令集设计与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，TLP有时在CPU语境下也代指Task Level Parallelism，两者还是有所区别），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。从另一个角度讲，ILP和TLP都可以增加可发射指令的数目，尽量减少功能单元的闲置，从而提高硬件利用效率。

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。典型的显式依赖如：

```
FFMA R3, R1, R2, R0;  // sm_75: stall 4 cycles
FFMA R6, R4, R5, R3;
```

而隐式的依赖比如这种：

```
// sm_61
1:  IADD   RZ.CC, R0, R1 ;   // set condition code as carry
2:  IADD.X R2, RZ, R2 ;      // use condition code as carry
3:  IADD   RZ.CC, R3, R4 ;
4:  IADD.X R5, RZ, R5 ;
```

IADD可以把进位存到专门的CC寄存器（类似x86的carry flag），然后IADD.X可以把这个CC寄存器当成carry读进来再做加和。由于CC寄存器只有一个，虽然2和3两条指令没有数据依赖，仍然不能把2和3互换以隐藏1-2和3-4之间的延迟。而在Turing里这种指令已经可以显式的用Predicate来存储carry，如：

```
// sm_75
1:  IADD3   R4, P0, R0, R2, RZ ;         // set P0 to carry out
2:  IADD3   R10, P1, R6, R8, RZ ;        // set P1 to carry out
3:  IADD3.X R5, R1, R3, RZ, P0, !PT ;    // use P0 as carry in
4:  IADD3.X R11, R7, R9, RZ, P1, !PT ;   // use P1 as carry in
```

这样1、3和2、4两组指令就可以interleave，正确性互不影响，从而相互隐藏延迟，提高ILP。

注：我其实没明白IADD3.X的第二个Predicate（`!PT`）是干什么用的，SASS有很多指令都会带一个predicate输入，多数没看到明显的价值。

这种相对更独立的指令集设计其实有点类似函数式编程：操作专门carry寄存器可以看做是stateful的操作，改成可编程的Predicate后就成为只与输入输出有关的stateless操作，不改变机器状态。也可以认为是所有需要用carry寄存器做输入输出的指令，都需要被序列化。通过与当前机器状态解耦，获得更大的指令调度自由度，编译器的后端优化也会更加方便。

NV GPU在很早就开始有意识的淘汰这种含隐式输入输出的指令。比如早期使用隐式栈的call、ret、break等等（当然其实这和ILP关系已经不太大了）。例如SM61中要return时需要先显式的用PRET设置某个隐式的调用栈，然后直接用RET返回。显然在RET时这个栈必须是对应当初PRET设置的值（中间能不能再进出栈没仔细研究），否则就会出错。而Turing直接使用带GPR地址的指令进行操作，就消除了这种隐式栈的操作过程，减少了指令之间复杂依赖对编译器的干扰。

```
// RET in sm_61
PRET 0x258 ;
...
RET;

// RET.REL in sm_75
MOV R20, 32@lo((_Z7argtestPiS_S_ + .L_9@srel)) ;   // relocation with addend
MOV R21, 32@hi((_Z7argtestPiS_S_ + .L_9@srel)) ;
...
RET.REL.NODEC R20 `(_Z7argtestPiS_S_);

// RET.ABS in sm_75
RET.ABS R32 `(_Z7argtestPiS_S_);
```

再稍微扩展一点。x86中有control register来控制如何做浮点数的rounding，是否做subnormal的flush（x87 FPU control register控制普通的FPU指令，MXCSR控制SSE指令）。但在SASS中，FFMA、FMUL、DFMA等浮点运算指令，每个指令都可以自主控制是否打开flush（使用 `FTZ` modifier），如何做rounding（ `RM, RP, RZ, RN` ）。这就意味着每个指令可以自主决定当前指令的运行方式，而不用改变机器状态。不同模式混用时，就不需要保存和恢复control register了。

那再再扩展一点，x86其实也有控制FP exception的寄存器，NV GPU里是怎么操作的呢？我好像没看见，感觉是被去掉了。这还是一个挺值得思考的问题~

当然，也不是说所有的隐式都应该变成显式。比如每个Warp都有一个隐式的active mask，用来标记当前warp中divergence的情况。active mask与指令predicate的“AND”会共同决定当前指令是否起作用。那把mask交由指令显式操作有意义吗？我觉得没有，因为这没有太多额外的可编程价值。首先divergence造成的mask变化只能被分支或分支同步指令修改，其他指令需要控制效果直接用predicate就可以了，没必要操作mask。其次，这个操作本身是非常固化的，增加相关操作指令并不会带来新功能，反而会增加指令负担（相当于每次可能有divergence的分支时，都要显式的保存和设置mask，遍历不同divergence的分支时就更麻烦了）。因此，在有predicate的情况下，active mask还是做成隐式的比较合理。当然，volta后的Independent Thread Scheduling也要自主控制和依赖内部的mask状态，就更没法做成显式的了。不过，虽然mask是隐式的，但divergence后重新converge一般是显式的（通过BSSY和BSYNC指令），否则程序就不知道应该在哪个点join了。

复合操作与附加操作

前面也提到了每个指令都有基础开销，那尽量在一个指令里塞进更多事情以分摊基础开销，就成为指令改进的一个重要方向。毕竟每个指令都要占用发射机会，但提高频率很困难，也就是发射的总指令数有限，那就只好让每个指令多做事。

比如一般的ALU指令可能有1~3个输入操作数，为了尽量利用GPR的读吞吐和增加每个指令的操作能力，SASS里其实有单指令多操作的情况。最常见的就是FMA（包括FFMA、DFMA、HFMA2和IMAD等），它可以一条指令同时计算乘和加两个操作：`d=a*b+c`。Tensor Core的MMA类的指令也通常是3个输入操作数。这里以Turing为例，常见的多输入或多操作指令有：

```
FFMA R5, R4, R5, R11 ;
DFMA R4, R2, R4, R2 ;
HFMA2 R3, R13, R9, R3 ;
IMAD.WIDE R10, R9, R4, R10 ;
IADD3 R36, -R11, R22, -R34 ;
LOP3.LUT R9, R15, R9, R16, 0xfe, !PT ;

LEA.HI.X R7, R3, R51, R2, 0x3, P0 ;          // Load Effective Address: d = (a<<b) +c
SHF.R.U64 R3, R11, R0, R12 ;                 // Funnel shift
PRMT R61, R60, R58, R61 ;                    // Byte permute

I2IP.S8.S32.SAT R0, R1, R0, R2 ;             // Integer To Integer Conversion and Packing
IDP.4A.S8.S8 R9, R20, R25, R9 ;              // Integer Dot Product and Accumulate
IMMA.8816.S8.S8 R36, R50.ROW, R74.COL, R36 ; // Integer Matrix Multiply and Accumulate
HMMA.1688.F32 R0, R184, R200, R0 ;           // Half Matrix Multiply and Accumulate
```

这个列表并不完整，比如`ISCADD`虽然在Turing指令集里，但是编译器似乎更倾向于用LEA和IMAD，所以好像很少见到了。还有warp shuffle指令`SHFL`，也可以接受4个GPR做操作数，不过这不是通常意义上的ALU指令，其实更接近memory指令一些，所以也没列出来。然后是大部分非tensor类的整数和位操作指令都有Uniform datapath的对应版，这里也没列。

这些多操作指令往往都需要更多的操作数。上一期我们讲吞吐的时候也提到了，GPR的吞吐用不满也就浪费了，所以一般ALU最多是3个输入GPR。其实每cycle最多能读4个，不过多少还是要留点余量给其他异步指令（如内存读写指令），否则抢占GPR的端口和抢占发射机会是一个效果。立即数和Predicate因为不占用GPR的吞吐，所以还可以额外加。比如`LOP3.LUT R9, R15, R9, R16, 0xfe, !PT ;`就是3个GPR+1立即数+1Predicate的输入。于是，NV在设计指令集的时候又经常会在一些ALU操作后加一个免费的Predicate操作。比如*SETP类的指令本来就是通过比较生成一个predicate，但它也可以顺手把生成的predicate与另一个predicate做与、或、异或等，这样就可以把一些链式的bool操作（比如`a < M && b < N && c < K`）附带在比较中。包括前面说的LOP3，以及PLOP3等，都具有这种额外的predicate输入。

```
FSETP.NEU.AND P0, PT, RZ, c[0x0][0x16c], PT ;
ISETP.LT.OR P0, PT, R9, R8, P0 ;
ISETP.EQ.XOR P4, PT, R7, R1, P4 ;
```

还有另外一类额外的附加操作，比如float intrinsic里有一个函数叫`__saturatef(float x)`，它会把输入clamp到`[0,1]`这个区间里。但是它并不占用一个指令，因为float指令自带一个modifier叫`.SAT`，如`FFMA.SAT R6, R2, R7, R0 ;`会自动对结果R6做saturate，并没有额外开销。类似的这种饱和操作在一些整数操作中也会出现（比如`I2I.S16.S32.SAT R11, R11 ;`表示把S32的输入saturate到S16的范围）。

复合操作和附加操作可以大大增强指令的表述能力，用更少的指令做更多的事情。但也要看到，它增加了指令的复杂度，对编译器后端优化也提出了更多的要求。同时，在硬件设计上它也会有一些额外的开销。如果这些操作不是特别常用，对指令运行开销反而是负担，所以多数还是需要一些功耗控制措施。比如IADD3其实多数时候都有一个是RZ，LOP3也通常只有两个有效输入。至于`*SETP`后带Predicate的情况，其实占比也不高。所以这些在具体设计和实施上，应该还是需要做相应的优化的。

立即数操作数和Constant Memory操作数

由于Volta前的SASS指令只有64bit，一般ALU的立即数操作数中只有19bit的编码。这对于一些特定的指令是不太够的，所以早期也有特定的32bit立即数的指令，如FADD32I，FMUL32I，MOV32I等等。不过这些指令最多只有2个输入操作数，FFMA这种就没法弄了。而计算很多函数值（比如sin）一般都是从高次多项式近似开始，需要计算大量给定系数的FMA，这些要用立即数就只能用mov先把32bit立即数写入GPR，然后再FFMA。Volta后每个ALU包括FFMA都能用满32bit立即数，就没有这个问题了。

当然，实际Volta前的多项式逼近不是用立即数实现的，而是用Constant Memory。因为Constant Memory需要的编码更少（几bit的bank编码+16bit的地址编码），而且Constant memory还可以用在64bit的DFMA上，通用性更强。这也是constant memory对指令表达能力的一个重要贡献。

Constant memory与立即数在做操作数时还有不少区别： 1. Constant memory运行有overhead，启动kernel前需要初始化。而立即数是hardcode在编码里，没有额外overhead。 2. 立即数是编译期常数，编译时就必须知道值。Constant memory可以做运行期常数，也就是启动kernel前才需要得到具体值，在运行前是可调的。 3. Constant memory有容量限制，当前应该是每个bank 64KB。立即数的总容量则是程序能写多长就可以多大。 4. Constant memory运行期开销会大一些，因为它毕竟是内存，需要相应的cache和load单元支持。 5. 立即数的编码能力比较有限，当前128bit的指令也就32bit立即数。而Constant memory只要在指令格式中约定好，32bit和64bit都可以，将来扩展到128bit甚至更多也不是不行。

当前在CUDA程序里，编译期常数是放在Constant memory还是做立即数是编译器决定的。一般来讲，32bit能精确表示的是立即数（比如float的所有数，double的1.0，2.5之类），需要64bit才能精确表示的会用constant memory。

注：有的同学可能会问，传运行期常数我直接用kernel的参数不也可以吗？有什么区别呢？

其实很相似，但确实也有区别。因为kernel并没有什么来自host的调用栈，kernel的参数其实也是用constant memory存储的，在kernel启动前会由驱动自动初始化。所以两者从性质上很相似，但又有点差别。kernel参数的scope是当前kernel，每次启动kernel理论上都要重新初始化参数区的constant memory，这其实也是kernel启动的overhead之一（不太确定cuda Graph会不会做优化）。而用户自己设置和copy的constant memory，scope至少是module（可以简单认为是一个cu文件，或者更具体一点是cubin文件）。这就意味着同一个module的kernel都可见。所以如果同一个module内的kernel共用大量参数，且中间不会更改，那就只需要初始化一次（重复启动同一个kernel也是同理）。

这甚至可以是一个极致优化的例子。把所有kernel的输入参数都放在某个struct里，然后把struct复制到constant memory，就可以不用参数启动这所有的kernel了。这应该会节省一点overhead（如果cuda graph不能做这个优化的话）。

PS：其实这个操作的意义主要在于接口的可扩展性。需要增加新参数的时候，只需要在struct里新加一个成员变量就行，不用去改函数参数和调用kernel的地方了。相信我，这个有时候真是很方便！你当然也可以在kernel参数里传一个struct指针，这样代码量也相当。但是你需要自己把struct内容memcpy到device上，读参数时还会增加内存负担。因为constant在SM内有专门的cache，load时又有broadcast机制，hit时几乎没有额外开销。

格式的通用性与信息具体化

很多指令可以支持大量的modifier，把一些输入信息更加具体化。这里我们举一个典型的Turing `IMAD` 的例子：

```
IMAD R4, R5, R0, -R4 ;
IMAD.HI.U32 R49, R5, c[0x0][0x1d8], RZ ;
IMAD.WIDE.U32 R20, R17, c[0x0][0x168], R20 ;

IMAD.MOV.U32 R31, RZ, RZ, c[0x0][0xc] ;
IMAD.SHL.U32 R0, R2, 0x8, RZ ;
IMAD.IADD R2, R7, 0x1, R11 ;
```

前3个其实都没啥特别。`IMAD` 本身是算整数的d=a*b+c，类似浮点的FMA（前面也提到了，实际上Turing的 `IMAD` 和 `FFMA` 共用了功能单元）。单独的 `IMAD` 就是三个32bit的计算，保留低32bit。`IMAD.HI` 就是保留高32bit，`IMAD.WIDE` 表示加数c是64bit（a、b还是32bit），输出d也是64bit。这都算是常规操作。

后三个就有点意思了，`IMAD.MOV`其实就相当于`MOV`，因为它的输入a、b肯定都是RZ（SASS中的恒零寄存器），所以输出d肯定等于c。用`IMAD`做`MOV`主要是`MOV`指令和普通INT32是一个Dispatch Port，而`IMAD`是FMA的port，两者错开有利于提高ILP。但我直接写`IMAD.U32 R0, RZ, RZ, R1;`与`IMAD.MOV.U32 R0, RZ, RZ, R1;`有什么区别呢？功能上两者应该是一样的，但功耗上可能不太一样。因为`IMAD`会用到一个很长的乘法器，功耗会很大。但如果事先知道这个乘法器不用了，那硬件上就可以绕过或是用一些处理不触发它，从而节省功耗。后面两个指令也是类似，`IMAD.SHL`相当于b一定是2的幂次，`IMAD.IADD`相当于b一定是1，其实也是不需要完整的乘法器功能。不写`SHL`或`IADD`不影响结果，但显式的告诉功能单元，可以让功能单元得到更明确具体的信息，从而进行一些优化。

那能不能让硬件自己检测呢？当然可以，但是未必划算。首先，如果大多数`IMAD`指令都是不带特殊性的操作数，那为了检测这种case的功耗优化方案就会给硬件造成额外负担。就像cache一样，hit的时候自然是好，但加了cache往往也会劣化miss时的开销。如果miss比率太高的话，加cache就是个负优化。同样，在这里如果大部分情况执行路径可以被优化，那自然可以省功耗。但如果多数情况其实优化不了，那就白白损失了优化检测的这部分硬件开销，变成了负优化。当然，增加这个modifier本身也是要看使用频率的，如果极少用到，那其实也是降低了面积效率。这都需要对使用场景和规划做准确的判断。

这个逻辑其实也和control code有些相通之处，尽量把编译期的信息融入到程序中，让程序尽量明确运行逻辑。这其实也许体现了NV硬件设计中一个更上层的指导思想：软件尽量具体地告诉硬件运行方式，而硬件则尽量减少自身的判断和复杂监测逻辑，无脑的运行软件的指示即可。这样硬件设计尽量简化，不但能节约面积开销，测试、验证复杂度都能降低。当然，这不是一件简单的事情，想有效的安排和传递各种编译信息，肯定需要软硬件协同设计。这对软硬件的架构规划和协作，包括对应用、编译器、指令集、硬件架构等统筹规划，还是提出了很高的要求，很体现功力。

这里IMAD做这些优化应该是和编译器有确定配合的，编译器在合适的时候会优先选择这些模式。其实这些功能也可以有其他一些实现方式，但是多则惑，也没有必要。开销大致等价的实现中选一个就可以了。

简单聊聊一些具体的指令

指令集内容实在太多了，有很多东西我没有靠谱的输入，瞎猜也意思不大。我就简单列一些SASS中我觉得还比较有意思的点，供大家参考。这里基本都是Turing的指令。

FMA,MUL, ADD 系列

FFMA是衡量GPU算力的标杆之一。从指令吞吐角度讲，FFMA一般都是最高的那一组。FFMA功能上相当于FMUL+FADD（FFMA精度更好，因为只做了一次舍入），那FMUL与FADD与FFMA共用单元就很合理。不过另一个有意思的地方是，FADD的reuse cache用的slot是1、3，是不是暗示它其实是FFMA的套壳，只是把FFMA第二个操作数固定成1？当然，更省的办法应该是有直接的短路逻辑。

DFMA, DMUL, DADD系列具有不定长的延迟，需要靠scoreboard去控制依赖。我觉得一是因为FP64相关单元配置变化很大，有的还是多个SMSP之间share，可能存在竞争，这样延迟不可控。二是它的吞吐可能很小（有的卡是1/16），指令延迟可能会超过stall count能有效表示的范围，所以还是用scoreboard比较靠谱。

这些都支持32bit立即数做操作数。D系列虽然每个操作数是64bit，但如果后32bit都是0，那也可以放进32bit立即数里，用的时候做padding就可以了（指数位还是按FP64来，只是截了尾数后32bit）。HFMA2、HMUL2、HADD2则相当于把它当成两个16bit的立即数。

IMAD, LEA, IADD3

IMAD在前面已经介绍过一些了。这里可以稍微再补充一些相关的。

```
IMAD.MOV.U32 R31, RZ, RZ, c[0x0][0xc] ;
IMAD.SHL.U32 R0, R2, 0x8, RZ ;
IMAD.IADD R2, R7, 0x1, R11 ;

IMAD R5, R5, c[0x0][0xc], RZ ;
IMAD.X R21, RZ, RZ, R7, P0 ;
```

前三个我们已经介绍过，为了控制乘法器的开销，有相应的modifier去明确输入。但是可以发现，加数为RZ时没看到专门的modifier。IADD3其实多数情况下也只有两个操作数，第三个是RZ的概率很大。也没有看到有modifier去优化这个，也许单纯检测RZ开销不大，当然也可能是加法器的开销没那么大，不是很敏感。

另外就是IMAD在接受carry输入时，有的前两个输入也都是RZ，但是这次就没见到相应的modifier。不知道是没设置还是没有，可能是加了carry这个逻辑不适用，也可能是编译器没考虑？也不是很清楚。亦或是这种概率很小，可以忽略？

与IMAD的一些功能有共通之处的还有LEA指令。这是计算地址时常用的指令。IMAD是`d=a*b+c`，LEA则特殊一点是`d=(a<<b)+c`。这样算X[i]这种地址时，相当于`i*sizeof(type) + *X`。如果size是2的幂次，那就可以用LEA。LEA与INT32是一组，IMAD是FMA那组，两个也可以配合使用。

LOP3

LOP3还是一个挺有意思的指令，它可以完成三个32bit数的任意按位逻辑运算。其实逻辑也很简单，把这个映射关系看成一个函数`d=F(a, b, c)`，abcd都是bool值。输入状态有2^3=8种可能，那每个F函数都可以用这8个输入时的输出来完全表示。用8bit的编码（立即数查找表，immeLUT）就可以指定这样一个逻辑函数。更具体的用法和介绍大家可以参考PTX文档。

ta	tb	tc	Oper 0 (False)	Oper 1 (ta & tb & tc)	Oper 2 (ta & tb & ~tc)	...	Oper 254 (ta | tb | tc)	Oper 255 (True)
0	0	0	0	0	0		0	1
0	0	1	0	0	0		1	1
0	1	0	0	0	0		1	1
0	1	1	0	0	0		1	1
1	0	0	0	0	0	...	1	1
1	0	1	0	0	0		1	1
1	1	0	0	0	1		1	1
1	1	1	0	1	0		1	1
immLut			0x0	0x80	0x40	...	0xFE	0xFF

LOP3 ImmeLUT

粗看起来好像挺厉害的，其实多数时候也没有什么大用。我曾经搜过官方库里的所有LOP3指令，看到的情况是：绝大部分情况下第三个操作数是RZ，也就是说3输入的按位逻辑函数其实还是挺少见的。如果真的要找场景，可能密码学或者挖矿之类会有些不错的应用。

不过细品一下，其实还是有一些妙用的。比如我有两个32bit数a，b，我需要把a的某些bit和b的bit拼在一起（对位替换），怎么弄呢？这其实就是一个简单的按位逻辑函数`c[i] ? a[i] : b[i]`。这样lop3就能用上。那什么时候会用到这种功能呢？其实还是有一些。比如copysign，需要把a的符号位替换掉b的符号位。再比如浮点数里有些指数、尾数之类的操作，也是有一些用处的。

更具体一点的例子，比如`y= (x>=0) ? 1 : -1`，这相当于就是把x的符号位复制给整数1。不过…… 这个一条LOP3指令做不到，因为这里c我们已经用了立即数（0x70000000，也就是选中第一个符号位bit），a或b就不能再用立即数了，等于说这里需要先把1或0x70000000移到GPR里，才能用LOP3来操作。

MUFU

MUFU是SASS中计算各种超越函数的指令。数学上，超越函数是相对代数函数（有限次加、减、乘、除、开方等组合）而言。但硬件上不能用多项式表述的好像都归在超越函数里了。多项式求值一般就是乘和加，不需要其他的指令。而超越函数，就需要通过一些其他指令或是软件逼近来实现。有的地方也叫特殊函数，我感觉是不太合理，就这些初等函数怎么也谈不上很特殊吧……

常见的MUFU类的指令有：

```
MUFU.RSQ R5, R10 ;
MUFU.RCP R3, R14 ;
MUFU.EX2 R9, R8 ;
MUFU.LG2 R10, R9 ;
MUFU.COS R9, R19 ;
MUFU.SIN R10, R19 ;

MUFU.RCP64H R3, R7 ;
MUFU.RSQ64H R11, R29 ;
```

其实数学函数求值这个事情还是有很多可聊的点。

首先，这些指令都是给出近似值。对精度要求高的话还是需要调用相应的数学函数库再做软件实现。一般说来，有的数学函数可以迭代更新（比如求倒数reciprocal，RCP），那就可以从近似值开始用不动点迭代得到更精确的值。另一种是可以做argument的range reduction，把全范围的参数缩放到能比较精确计算的范围内。比如对于x具有二进制形式`x=a*2^e`，`log2(x) = log2(a*2^e)=e + log2(a)`。那只需要精确计算`log2(a)`（其中a在1~2之间）就可以得到精度很高的解，而MUFU.LG2在某些特定范围内是可以满足精度要求的。还有一种就是sin和cos这种，它没有迭代形式，也没有特别好的手段做规约，那就先规约到一个不太大的范围（比如正负pi/4），然后用一个高次多项式去近似（不是泰勒展开，大家有兴趣可以去研究一下Remez的minimax方法）。所以MUFU.SIN只是在近似计算中出现，精确计算中可能用不着它。

当然，软件实现上还是有很多自由度，这里只是给出了一种选择方式。NV的数学函数库不直接开源，但是有相应的LLVM的bitcode（NVVM目录下的`libdevice.*.bc`），可读性还比较强，可以参考。

从这些实现可以看到，大量辅助计算仍然是用加、乘这种初级运算完成的，所以即使是使用了大量数学函数，主要的计算量也在FMA、MUL和ADD这种简单指令。MUFU还是占比比较小。如果使用了内置函数，则相对占比高一些，但多数时候还是需要相应的简单指令的配合。而且有些常用函数其实并没有加速实现（比如sqrt，即使是intrinsic的`__fsqrt_rn`这种实现也非常复杂）。这里面肯定是有所考虑的。

MUFU的64bit版本（RCP64H和RSQ64H）其实只用了一个GPR输入，估计觉得反正是近似，多后面32bit意义也不大。

FSETP

FSETP本身也没有什么特殊的，就是根据一些浮点的比较操作设置一个bool变量。比如：

```
FSETP.GT.AND P1, PT, R1, R2, PT ;
FSETP.GEU.AND P1, PT, R1, RZ, PT ;
```

第一个是比较R1是否大于（greater than，GT）R2，然后顺带把结果与另一predicate做了与（AND）。GE是greater or equal，那GEU是什么呢？这其实是浮点比较中的一个特例。PTX把这个叫unordered floating-point comparisons。是专门为了NAN而弄出来的特殊形式。

如果大家熟悉IEEE 754，就知道NAN如果出现在比较浮点运算（`a op b`）中，那结果永远是false，甚至x为nan时`x==x`都是false（甚至能用这个来判断x是不是nan……）。所以PTX或SASS中有两类浮点比较，一类是ordered，有nan就为false，一类是unordered，变成只要有nan就为true。如果两个输入都不是nan，那两者就一样。

这个逻辑有什么用呢？我开始也没太明白，觉得可能就是单纯想提供一个功能吧！不过后来我看到一些编译器在处理比较运算的时候，会选择一个canonical的形式（比如把`a >= b`统一改成`b<a`，这样就不用处理`>=`这种操作了），而对浮点操作这个变化是不正确的，就是因为有nan的存在。但是ordered `a>=b` 与 unordered `b<a` 两者是等价的。看代码的时候也确实能看到一些这种转换，那具体NV是不是这么想的，我也不知道……

浮点的比较操作里还是有不少坑，和整数的比较还是差别很大。除了nan的不等于自己，还有一个就是+0和-0的二进制表示不一样，但却是相等的。denormal的情况我没有仔细研究，要是会被flush的话，这里面也有很多需要注意的问题。

LDG/STG, LDS/STS, LDL/STL

内存相关指令一般都比较复杂，特别是global操作，modifier特别多。这部分其实已经超出指令集设计本身的范畴，更多的是consistency model的问题。这只讨论指令输入输出中的一些点。

比如说LDG的有两个使用UR做base的形式：

```
LDG.E.SYS R16, [R10.64+UR10+0x200] ;
LDG.E.CONSTANT.SYS R54, [R32.U32+UR4] ;
```

第一条中R10.64应该表示它是个64bit的值，如果是32bit会用第二行的R32.U32这种形式。而后面的UR4，据我观察则一般都是64bit。用32bit的好处就是一些简单的地址运算可以用32bit的运算得到，比每次都做64bit操作要更划算。而如果是warp内uniform的base变化，可以直接改UR，UDP的计算与一般的ALU也是独立的。这也算是一种减少计算强度的方式。

LDG/STG也有一些控制cache的逻辑，比如带CONSTANT的modifier表示会在read-only L1中缓存。当然，这个具体操作其实每代架构都会有所不同，但能用constant的话没理由不用。

Shared Memory里有一个有意思的内置left shift。比如下面的`X4, X8, X16`：

```
STS [R2.X4+0x400], R3 ;
STS.64 [R26.X8+UR4+0x10], R28 ;
LDS.U.128 R20, [R57.X16+UR4] ;
```

Shared memory因为地址窗口小，一般32bit地址足够了。而且起始地址永远是0，基址也常常是编译期常数（静态分配）。所以地址计算往往可以表示为`immeBase + index * size`。而元素大小常常是2的幂次，这样就可以省去预先计算乘法或是移位，直接让地址单元进行处理。这也算是减少计算强度的方法。

不过其实LDG/STG/LDL/STL应该也可以用这个逻辑，特别是local memory，不知道为什么没有。

BAR

BAR指令就是barrier，更确切的说是synchronize barrier，就是通常block内用来同步的`__syncthreads()`函数。CUDA中叫barrier的术语挺多的，比如dependency barrier主要指DEPBAR指令，是用来等scoreboard的。memory barrier应该是相当于memory fence，但是现在好像有专门的API操作。还有一种convergence barrier就比较底层了，是用来处理warp divergence的。

CUDA C一般只用一个barrier（就是`BAR.SYNC 0x0`后面的那个0x0），但实际上PTX或SASS里每个block最多可以用16个barrier，每次同步可以选择同步到某个barrier。这样同一个block的线程可以分块同步到不同的barrier上去（BAR有warp计数的参数）。另外，BAR还支持arrive、sync模式，可以搭配出类似生产者-消费者的模型，具体可以参考PTX文档。这个功能没有C API，但是可以用inline PTX其实风险也不大。这其实也算是一个降低瓶颈资源压力提高并行度的方法，与前面提到的用predicate做carry，还有跳转用显式GPR代替隐式栈，应该算是一类思路。

CUDA 11之后有一个新功能叫Asynchronous Barrier。从实现上更偏软件，但功能上没看出它比BAR.arrive与BAR.sync的组合有什么更特别的地方。也许是scope更灵活？或是更适合CPU软件移植和编译器实现？没仔细研究过。

BAR不仅可以实现同步，还可以顺带计数或是做reduction，这个是可以从CUDA C调用的（参考）。从功能上讲，用户自己用shared memory实现应该也可以，不过既然是顺带的，多少还是省点用户的事。具体硬件实现上便不便宜，就不得而知了。

结语

指令集的设计其实是一个挺玄学的问题。好的指令集设计与好的产品之间，其实还隔着非常多其他因素。功能贴合市场需求，性能满足要求，其实就具备了成为好产品的条件。而指令集设计本身依附于产品提供的功能和性能，它更多的是一个接口的角色：连接用户与设备，让功能和性能更容易发挥。但我觉得指令集并不是决定性因素。x86指令集现在常被吐槽是过时设计，

即使它已经不太强势，但生命力仍然顽强。我觉得也不是光靠“兼容性”和“wintel垄断”可以解释得通的，至少它当前具有的功能和性能，一定是多年市场选择和无数技术迭代的结果。当然，好的指令集设计，可以让一个架构更通用，性能可及性更好，硬件效率更高，从而更有生命力和竞争力。

指令集相关的内容真的很多，很杂，最近比较忙，就没有写的太系统和具体，点到为止吧！大家有兴趣也可以自己研究研究~ 文中肯定有很多错误和疏漏，欢迎各位交流、批评、指正~

完结撒花~ 再也不写这么长的文章啦！