总体流程设计（1）-CUDA程序的等级结构

https://zhuanlan.zhihu.com/p/129375374

None Mon May, 24 03:29

上次聊了聊CUDA程序的性能优化思路，发现写起来真是停不下来……因为换了工作，后续就拖了挺久。现代人真的挺忙的，长文章意义有限，我懒得写，大家也懒得看。想了想，先把一些系统的东西讲讲清楚，然后讲些琐碎的感想，每次就讲一两个要点，理论少点，随性些，不拘于格式，多举点具体例子，也许效果更好些。不过这一次，好吧，长文预警……

顺便还是强调一下，这些知识还是有些门槛的，对刚入门的初学者可能不太友好。但只要你有些基础知识，看完后如果有不清楚的地方，可以自己找找资料深入研究，也许可以增加一些可能并不怎么有用的知识。我也会尽量多给出一些参考材料。有些地方我可能也会有一些想当然的地方，也希望大家能批评指正，共同学习。

今天开始，准备花几篇文章聊聊CUDA的总体流程设计。这是第一部分，主要讲CUDA程序的等级结构。

我这里说的总体流程设计，指的是怎么把一个具体任务转化为Kernel，Kernel每个thread和block之间应该怎么组合。如果一个kernel做不成，那多个Kernel怎么分工。这里的流程也包括Kernel之间，Kernel和其他API（比如`cudaMemcpy`），或者是和其他CPU任务之间，怎么配合等等。想对CUDA程序做一个很好的规划，就要先深入了解CUDA程序的各级组成单元。只有了解每层单元具体是怎么工作的，清楚它们相互同步、通信的机制，才能更好的让它们协作配合。所以这篇先讲CUDA程序的等级结构。

Kernel 上层的等级结构

首先简单阐述一下CUDA kernel以上的等级结构。每个CUDA程序都必须是一个CPU进程，其中可能包括一个到多个CPU线程，CPU线程间的共享关系与普通多线程程序是一样的。每个CUDA程序都包含（或隐含）一个到多个**CUDA context**（driver api里的`CUcontext`）。CUDA context可以看成是类似CPU进程的GPU进程，大部分的runtime api，包括memcpy，启动kernel，bind texture等等，都会运行在这个context上。每个context都有自己的资源空间，相互隔离，互不影响，所以有一个出错也不会影响到其他context。每个CUDA程序至少会有一个context，不显式初始化的话程序会自动分配一个，这也是runtime api大部分时候的用法。也可以申请多个，但每线程当前只能用一个。CUDA会维护一个context的栈，可以通过类似push/pop的方式切换当前context。每个context会包含一些模块(`CUmodule`，有的是以cubin文件形式保存)，类似dll。模块内会有一些kernel function（`CUfunction`），也会有一些模块内资源的描述（比如constant memory，texture reference等），load模块时这些资源就会得到分配。值得一提的是，kernel function不仅可以访问本模块内的资源，其实整个context的内容都是可见的。更具体的描述可以参考CUDA C++ Programming Guide中driver api的部分。具体的driver api的内容也可以看CUDA Driver API.

这里有几个微妙的地方：

- 一是多线程的CPU程序能共享context吗？答案是可以。其实本身同一进程的资源就是相互可见的，所以多线程可以共享context。每个context执行时有一个或多个stream，同一个stream内的api是序列化的，前一个返回后才能运行后一个。所以其实多线程并不适合同时操作同一个stream，但不同stream其实是可以同时操作的。一些更大型的应用也可以让每个线程维护自己的context，这样相互干扰更少，万一一个出错也不会导致整个进程崩溃。当然，CPU同时执行api是一回事，到了GPU设备端肯定还是有一个queue来维护顺序执行关系（比如申请资源这种肯定要排队，不可能完全并行）。
- 二是多GPU的情况怎么办？其实每个卡（或者更精确的说是每个GPU芯片，对应driver api里的`CUdevice`）都有相互独立的context。资源也是相互独立的。CUDA里有managed memory，可以让它看起来是每个device都可访问（也包括host），但其实只是把数据搬运自动化后隐藏起来而已，本质上每个卡的资源还是分开的。CUDA程序暂时还不能做到自动把多块卡当成一块来用，一个kernel也不能自动分在两个GPU上运行，有NVlink也不行。所以当前情况下多GPU还是需要用户自己来手动分割任务。

Kernel的等级结构

一个CUDA Kernel大概可以分为这么几层（从底层到顶层）：thread，warp，block，grid。下面就分别讲讲这几层的特征和运行模式。

1. Thread

Thread是CUDA程序的底层任务单元，与CPU的thread具有比较高的相似性（其实独立性上更像进程一些）。每个Thread都有一些私有资源：

- General Perpose Register: **通用寄存器**，简称**GPR**，有时也直接叫**Register**（**寄存器**）。GPR通常按个算，一个是32bit，在CUDA的SASS汇编里一般写成 `R0` ， `R123` 这种格式。每个线程使用的具体GPR数目是编译器根据需要进行配置的，每个kernel的所有线程都保有相同数目的GPR。最近几代架构单线程最大可用数目是255，因为指令集里GPR编码有8bit（2^8=256），而且 `R255=RZ` 被用来做恒零的输入，或者当输出时表示抛弃输出。

Tips：GPU的GPR实际来自于SRAM组成的一大块Register File，每个线程可以分得其中的一部分。一旦线程创建后，物理上的单个GPR和线程里的R0、R1等就建立了一一对应关系，不再改变，直到退出。这与X86 CPU的寄存器运行机制上有些差别。因为X86的ISA中支持的通用寄存器比较少，为了减少指令间的依赖，有时会用**寄存器重命名**（Register Renaming）的方式用多个逻辑上的寄存器去对应ISA的同一个寄存器。所以X86 CPU在物理上的寄存器与ISA中的寄存器不是完全一一对应的，物理寄存器数目是更多的。GPU并不存在寄存器重命名这种方式。

- Predicate：1bit的bool谓词，每线程有7个, SASS里用 `P0` ~ `P6` 表示。Predicate通常是一些bool运算（比如大于，不等于）的输出。CUDA的每个汇编指令都可以用Predicate来控制是否真正运行。形式如 `@!P0 BRA 0xc40` 。这里前面的 `@!P0` 表示P0为假时这行才运行（更准确的说法是才起作用），前面带 `!` 表示取反。Predicate与直接branch跳转的方式是可以避免warp divergence，而且与branch指令相比开销更小。因为一般branch的延迟比较高，还涉及到指令Cache和内部pipeline连续性的问题，所以开销大一些。不过，即使Predicate为否，本身这个指令的运行开销也不会有变化，只是不写回结果而已。（PS：如果是load指令，是什么情况呢？相当于cache hit吗？我还没有仔细研究过……）

当然，Predicate不仅可以用作指令的谓词，也可以做操作数。比如整数加法的carry输入输出，`SEL` 、 `FMNMX` 、 `IMNMX` 等这种选择指令的操作数等等。

CUDA的SASS指令集里Predicate是4bit编码，3bit用来索引（2^3=8），1bit表示是否取反。前面说了，每线程可用的Predicate只有7个(`P0` ~ `P6`)，还有一个哪去了？其实 `P7` 也有的，就相当于 `PT` (true)，表示一直是真值。这个predicate在控制指令运行中是不用显示表示的。如果前面没写，就是 `@PT` 。另外 `PT` 有时候也会作为一些指令的操作数。

- Local Memory：Local Memory是每个线程私有的一段内存空间。它在物理上与Global memory并没有区别，只是访问方式和可见性不一样。Local memory主要有两个使用场景，一是一些线程内的私有数组如果索引在编译期不能确定，就需要放在Local memory里。另一个是每个线程的GPR是有限的，但有些程序的逻辑比较复杂，需要的GPR超过了

一定限制，导致有些GPR被暂存到内存里（称为register spill）。这虽然会影响性能，但有时为了减少GPR的使用，也是不得已的手段。

注意：local memory并不是线程自己动态申请的资源，而是整个kernel启动时为每个线程分配好的固定大小的资源。这个size是编译器决定的，运行过程中并不能改变。用户可以用nvcc的`--resource-usage`选项查看具体每个kernel的资源使用情况。

- 其他资源：前面说的GPR、Predicate和Local Memory是线程内比较明显可见的资源。但也有一些不那么明显的资源。比如每个线程有自己在block内的索引地址寄存器`SR_TID.X`，`SR_TID.Y`，`SR_TID.Z`（对应threadIdx的x、y、z分量）。如果需要获得这些值，则可以用`S2R R0, SR_TID.X`这种指令。另外每个线程有几个做dependency check的barrier（有兴趣的可以参考Scott Gray的关于Control Codes的相关介绍）。对于最新的Volta和Turing架构，每个线程还有自己独立的PC，等等。每个线程一般都有很多配套的硬件寄存器（是真的register，不是GPR），比如为了支持debug有debug的寄存器，profiling常用到的performance counter等，其他还有用于实现跳转和返回的PC堆栈等等。这些资源对大多数终端用户虽然不可见，但却会限制每个SM上能同时驻留的线程数（参考CUDA Programming Guide: Features and Technical Specifications的Table 15）。

2. Warp

Warp本来是个纺织学上的术语，叫做经线或经纱。Thread本意是线，翻译成线程。Warp就是一排很长的thread，一般翻译成线程束。Warp固定好后，梭子带着纬线（weft）在warp里来回穿梭，就织成了布。NV起名字也是蛮有意思的。

Warp是CUDA里指令同步运行的最小单元，包含**32**个线程（理论上说，这个数虽然一直没变，但未来也许可能改变，安全起见可以用内置变量`warpSize`来代替）。同一个warp的线程同时只能发射同样的指令。假如这个32个线程走了不同的分支（比如一部分进if，一部分走else），则有一部分的线程只能处于等待的状态。这就是所谓的Warp divergence。加Predicate与分支类似，都是有些指令没有真正起作用，但是少了跳转的开销。另外，如果block的总线程数不是warp的整数倍，那就存在不满的warp，相当于有些线程一直是空转的。

注意：这里warp的同步性针对的是warp内的32个线程，而不是指令。根据微架构的不同，同一个warp可能同一周期发射多个指令（比如ALU和Load/Store指令，只要是dispatch port不同就行），也可以不同warp的两条指令同时发射。这个与CPU常见的dual issue或是multi issue的方式是比较类似的。但不管是什么样的多发射形式，warp内的32个线程都是以同样的方式执行多发射。具体多发射的形式取决于SM里warp scheduler的工作方式，可以参考各个架构的Tuning guide介绍SM的部分（比如Maxwell，Pascal，Turing）。

很多内存指令的访问模式都是以warp为单位（**注意**：这里的访问模式都是针对warp内的**同一个指令**而言，warp内的不同指令或不同warp的指令，是独立的，并不存在这个关系）。比如：

```
LDG.E.64.SYS R6, [R0] ;
LDG.E.64.SYS R8, [R2] ;
```

则只有当前warp运行第一个 `LDG` 时，32个线程的 `R0` 的分布情况才是要关注的，下一条 `LDG` 指令的 `R2` 的pattern与之无关，另一个warp运行时的 `R0` 的pattern也与当前warp无关。

下面列举几个典型场景：

- Global Memory：同一个warp内如果访问的全局内存在128B（32*4B）的对齐范围内，则只需要一次transaction。如果是散列或是不对齐的，则需要多个transaction。具体的模式与微架构有关，甚至早期的架构里每个warp里线程的访问顺序都有要求。
- Shared Memory：Shared memory是分bank的（当前都是32个bank），每个bank一次只能输出一个值。warp如果要访问同一个bank的不同地址（同一地址会做broadcast），则会导致bank conflict。Shared memory的bank conflict的第一个影响是增加了访问延迟，增加量与每个bank的不同地址数有关，但是这个延迟是可能被其他指令隐藏的，不一定关系很大。第二个，也是很多时候更重要的影响，是减小了访问带宽的使用效率。举个例子，没有bank conflict的情况下，shared memory每次传输可以最多输出4B*32bank=128B。有了bank conflict后这可能需要多次传输，相当于传了几次才传完128B的数据，也就是带宽使用率下降了。这个问题在Shared memory的访问带宽是瓶颈时会显得比较突出。
- Constant Memory：CUDA里的constant memory可以用内存指令载入，也可以直接作为指令的操作数。虽然每个SM有相应的Constant cache，它的访问模式却是最单一的：所有warp的线程同一周期只能访问同一个地址，如果访问了不同的地址，那会被serialize成多个操作，直接影响就是指令延迟增大。
- Atomic Operation：当前Turing架构global memory和shared memory都支持原生的atomic操作。如果atomic操作的对象是同一个地址，那也会被serialize。所以它也有类似conflict的情况，正确性不受影响，但效率降低是无法避免的。

需要指出的是，在大部分情况下，这些场景都只是导致指令**延迟**增加和访问的**带宽利用率**下降，一般不会影响到后续非依赖指令的继续发射。所以如果瓶颈不在这些带宽上，而延迟又能被有效隐藏的话，对程序性能的影响未必很大。

从Turing架构开始，NV引入了一套uniform data path的功能，用于处理一些整个warp内的公共路径（AMD的GCN架构就有scalar单元，功能基本类似）。比如warp内有个变量`a=Array[blockIdx.x]`，这个`a`就是个warp内的公共标量（相当于32个线程用的是同一个值），可以用公共路径来处理。相应的，有独立的uniform的ALU单元，也有一套uniform的Register和Predicate配合。不过，这个功能暂时无法在CUDA C直接使用，主要还是靠编译器判断某个指令是否对于整个warp是公共的。有兴趣的读者可以研究一下Turing White Paper和Turing ISA中**Uniform Datapath Instructions**中的内容。

Warp内的各个线程交换数据可以用**warp shuffle**，是直接基于寄存器的数据交换，并不需要额外的存储空间。模式可以一个lane广播到所有的lane，也可以有比较复杂的交换pattern。warp shuffle与基于shared memory的数据交换各有优劣，将来聊CUDA指令集的时候也许会仔细讲讲。

3. Block

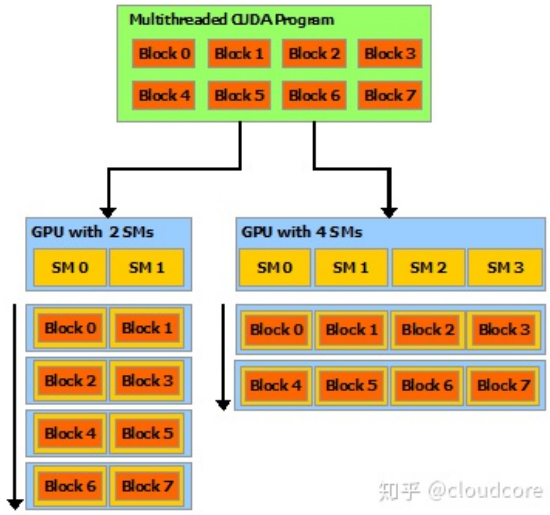
Block由用户指定的若干个warp组成，当前最大是**1024**个线程，相当于**32**个warp。除了上面说的thread和warp所拥有的资源以外，block还有一些block层的资源：

- **Shared Memory**：一般译作共享内存，是block层最显著的资源。它的容量可以由编译器决定也可以运行时动态指定，但是一般有个上限（参考CUDA Programming Guide: Features and Technical Specifications的Table 15）。Shared Memory对于整个block内的每个thread都是可读可写的，block外则无法访问。同时比较新的架构里Shared Memory还支持atomic操作，比用global memory做atomic操作延迟还是有很大的提升。
- **Synchronize Barrier**：block有一个重要操作就是同步，比如`__syncthreads()`，有兴趣的可以参考Synchronization Functions，或者是PTX中bar和barrier指令的介绍。注意同步不仅仅是保证warp都运行到某个指令的位置，还要求之前的一些操作如memory load的dependency要先完成。每个block有16个barrier，每个可以支持独立的同步操作（规定到达线程数，是arrive还是sync模式等等，具体参考上面链接的PTX的bar指令。自从CUDA引入Cooperative Groups后，同步的方式更丰富了，也提供了一些上层接口。
- **其他资源**：比如一些特殊寄存器（`blockIdx.x/y/z`）等。还有一些内部状态变量，比如GPR在整个register file里的起始地址，local memory的起始地址，一些debug用的资源等等。这些对用户不完全开放，所以一般不用太过关注。

Block是kernel运行时进行**资源分配**的最小完整单元：block的所有资源限制在同一个SM内，启动前必须全部分配到位。如果部分warp提前退出，它的资源应该可以先被释放，但shared memory只能在当前block的所有线程完成后才释放。也就是说，block运行时的所有资源（包括每个thread的私有资源，warp私有资源，block一级的shared memory等等），都必须在block启动前就绪。如果一个SM无法提供足够的资源，则该block无法在这个SM上启动。这种情况可能是SM总资源不够，也可能是有其他block在运行导致占用了一部分。如果一个SM的所有资源都无法满足单个block的某个需求（比如单纯GPR超了），那这个kernel将无法运行。

Block由于这个资源使用上的特性，成为CUDA程序实现scalablity的基础。block之间基本相互独立，没有数据交换，理论上能以任意顺序发送到任何有空闲的SM上。两个相邻的block可能在生命周期上没有交集，各个block的运行时间上也可以各有长短，后发先至也很正常。这个灵活度对CUDA程序的适应性非常重要：它既能适配不同SM数目的GPU，也能在与其他kernel同时运行分享资源时保持正确性。也正是由于这个特性，block的资源需求成为occupancy的关键因素。一个SM只能容纳整数个block，一个block只能容纳整数个warp，而warp是最小的并行运行单元，每个SM需要同时运行足够多的warp才能有效隐藏指令的延迟。因此，调整block的资源占用（每个线程的GPR数目，warp数，shared memory大小等），就可以影响同一个SM上能容纳的block数目，从而调整occupancy。

【注】：如果CUDA支持block部分warp先退出则资源先回收，那一个SM可能容纳分数个block。相当于后进的完整block能与之前block残留的warp同时运行。举个例子，假如一个SM最多放32个warp，一个block有24个warp。这样，一个SM放不下两个完整的block。但是如果一个block有16个warp先退出了，资源也回收了。那就留下32-（24-16）=24个warp的可用资源，正好可以放进一个新的完整的block。 这样上一个block残留的8个warp和新block的24个warp就可以同时共存。有shared memory的情况与此类似，关键是warp的GPR资源。这个模式我还没有确认过，待测试。



每个grid的所有Block都具有相同的维度。尽管 `blockDim` 是三维向量，实际上仍然按 `x` 、 `y` 、 `z` 的顺序摊平成一维，然后组成一组组的warp。 `blockDim` 分三个维度的好处是可以减少一些计算索引时的除法或取余运算，毕竟整数除法和取余的开销比较大。

4. Grid

每个Kernel都有且仅有一个grid。 `gridDim` / `blockIdx` 也有三个维度，与 `blockDim` 的三个维度其实也没有什么对应关系。因为block并没有什么空间相邻关系，所以grid的维度其实都没有摊平成一维的意义（当然，和发射的次序也许是有关系的，但其实block的执行也不依赖这个次序）。分成三个维度的好处与 `blockDim` 一样，也是减少不必要的除法或取余运算。有的时候为了方便对任务进行多维分割， `blockIdx` 与 `threadIdx` 的三个维度会联合起来，共同表示任务的三个维度。比如这种：

```
int tix = threadIdx.x + blockIdx.x*blockDim.x;
int tiy = threadIdx.y + blockIdx.y*blockDim.y;
int tiz = threadIdx.z + blockIdx.z*blockDim.z;
A[tix + tiy*Nx + tiz*Nx*Ny] = ...
```

这个只是处理上的关联，与线程本身的空间结构并不对应。因为block的所有线程是一维的。grid的所有block是散列的，没有空间相邻性。

前面也讲到了，用户可见的grid资源，比如global memory，constant memory，texture/surface reference或object等等，都是可以在同一个context下的所有kernel间继承和共享的。每个Kernel在grid这一层其实并没有太多的私有资源。大多数用于维护grid本身运行的资源对于用户都是不可见的。比如函数的输入参数需要存到constant memory里， `printf` 输出等这种需要额外分配内存资源用来保存临时结果。再比如说我们前面说过的debug信息和profile用的performance counter其实是最后要汇总的，也需要相应的内存资源。然后硬件上肯定有一些状态量要用来记录kernel当前的状态，比如block发送到多少了，kernel指令放在内存的什么地方，运行kernel所需的资源配置等等。这部分一般是硬件和驱动完成的，用户不可见。但这些东西与用户并非毫无关系，比如会限制每个设备可同时运行的kernel数等等。

Kernel等级结构的功能和意义

从Thread到Grid，构成了CUDA kernel的整体结构。每一层都有它特定的功能和意义：

1. **thread**是任务的基本单元，每个线程完成一个任务，类似于CPU线程的特化版。

2. **warp**是实现高效SIMT的基础。通过把warp中的thread同步运行，分摊指令fetch、decode等各种开销，降低运行调度的复杂度，从而避免功耗爆炸。由于线程还是具有比较强的独立性，warp的运行灵活度比CPU常见的SIMD指令（比如SSE，AVX）要更高，调度上更方便。

3. **block**是能完整独立运行的最小单元，它保证了warp和thread在运行前能分配到所有需要的资源。只有保证这些资源尽可能的随时可用，warp间的运行切换才能高效快速，从而摆脱类似CPU线程切换那些overhead（保存、恢复寄存器和栈操作等等）。而快速切换的能力使得GPU可以通过这种方式更高效的隐藏指令的延迟，省去了许多为了减少延迟而做的大量准备工作（比如分支预测），从而获得更好的性能功耗比。而block的shared memory和synchronize提供了warp间数据交换和同步的功能，提高了局部性能和灵活性。block内的完备性和block间的无关性使得block间的调度具有极大的灵活性，提高了scalablity和程序的适应性。

4. **grid**或者说是一个kernel，就相当于一个CPU的函数，接收输入，处理后输出结果。其他CUDA api会帮kernel完成相应的资源申请和各种准备工作，而kernel的任务就是计算。但是，block的无序性，或者说是block的高自由度，也限制了kernel的功能。单个的kernel一般不能完成需要block间数据交换、同步或者是带锁的操作，一般说来也不太适合完成需要block级别内存一致性的任务（比如多个block按指定次序读写一段内存）。当然，通过global memory的一些atomic的操作，有些功能是可以实现的，但通常来讲，除了一些简单的reduction操作，实现较为麻烦，性能也往往不佳。

一些特例和变化

需要指出的是，随着CUDA编程模型的不断发展和丰富，特定条件下这些等级结构也会出现一些变化：

1. 自CUDA 5.0开始，CUDA引入了**Dynamic Parallelism**功能。此前的kernel都只能从host端启动，grid和block的dimension都是在host端就确定好的。这对于形状规则易于均匀划分的计算任务是合适的。但有些应用，一些区域任务多，一些区域任务少，有时候任务大小都需要经过复杂计算，并不能一开始就得到。Dynamic Parallelism允许kernel函数内再启动kernel，由父kernel负责计算子kernel所需的grid、block的dimension，也包括分配子kernel所需要的一些内存资源等等。与此配套还有一些device端的runtime api，主要就是内存申请释放方面的工作。 Dynamic Parallelism相当于在kernel内部（其实是单个thread）又嵌套一层kernel的等级结构。有兴趣的读者可以参考：

- Nvidia developer blog: Adaptive Parallel Computation with CUDA Dynamic Parallelism
- Nvidia developer blog: CUDA Dynamic Parallelism API and Principles
- 官方文档：CUDA C++ Programming Guide: CUDA Dynamic Parallelism

2. 前面我们说过CUDA程序只有block内才能同步，block间是独立运行和调度的，其生命周期也互不关联，所以block间无法同步。另外，除block和warp外的其他颗粒度的同步一般也是不支持的。但CUDA 9.0引入了cooperative group的功能。它支持更多更细致的同步模式，可以block间同步，可以block内的部分线程同步，甚至是warp内的部分线程同步。实际上，为了引入这些

feature，它牺牲了一部分自由度。比如为了保证block能够同时运行（生命周期如果没有重叠，显然就无法同步），它需要在启动前确认设备是否可以同时容纳所请求的所有block。更具体的细节，可以参考：

- Nvidia developer blog: Cooperative Groups: Flexible CUDA Thread Programming
- 官方文档：CUDA C++ Programming Guide: Cooperative Groups

CUDA的feature在不断更新变化，也许将来还会有更多的新内容。所以这些东西也不是一成不变的。

聊完了CUDA程序的等级结构，下次就需要讲讲怎么根据这些等级结构去分配每个线程的任务，设定block或者grid的尺寸，多个kernel怎么配合等等。先等我再构思一下吧！下次绝对不能再写这么长了……

有些地方我研究也不深，也许有一些理解不到位或是没跟上时代的地方，还请各位多多指正~