如何设置CUDA Kernel中的grid_size和block_size?

> ***前言：在刚接触 CUDA 编程时，很多人都会疑惑在启动一个 kernel时，三个尖括号里面的参数应该如何设置？这些参数受到哪些因素的制约？以及他们如何影响 kernel 运行的性能？本文参考 CUDA 官方文档，分析了这些参数应该如何设置。***

我们在代码中一般会看到使用以下方式启动一个 CUDA kernel:

```
cuda_kernel<<<grid_size, block_size, 0, stream>>>(...)
```

`cuda_kernel` 是 global function 的标识，`(...)` 中是调用 cuda_kernel 对应的参数，这两者和 C++ 的语法是一样的，而 `<<<grid_size, block_size, 0, stream>>>` 是 CUDA 对 C++ 的扩展，称之为 Execution Configuration，参考 CUDA C++ Programming Guide（后续简称 Guide ）中的介绍

> *The execution configuration is specified by inserting an expression of the form <<< Dg, Db, Ns, S >>> between the function name and the parenthesized argument list, where:*
>
> *Dg is of type dim3 (see dim3) and specifies the dimension and size of the grid, such that Dg.x * Dg.y * Dg.z equals the number of blocks being launched;*
> *Db is of type dim3 (see dim3) and specifies the dimension and size of each block, such that Db.x * Db.y * Db.z equals the number of threads per block;*
> *Ns is of type size_t and specifies the number of*

> *bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array as mentioned in **shared**; Ns is an optional argument which defaults to 0;*
> *S is of type cudaStream_t and specifies the associated stream; S is an optional argument which defaults to 0.*

Dg 代表的是 grid 的维度，Db 代表 block 的维度，类型为 dim3，如果是简单的一维结构，也就是除了x以外，yz两个维度对应的值都是1，Dg 和 Db 也可以直接用 x 维度对应的数字代替，也就是文章一开始的表示方式，对 grid dim 与 block dim 两者更具体的说明可以参考 Programming Model，接下来我们讨论一下这两个值一般应该取什么值。

grid_size 和 block_size 分别代表了本次 kernel 启动对应的 block 数量和每个 block 中 thread 的数量，所以显然两者都要大于 0。

Guide 中 K.1. Features and Technical Specifications 指出，Maximum number of threads per block 以及 Maximum x- or y-dimension of a block 都是 1024，所以 block_size 最大可以取 1024。

同一个 block 中，连续的 32 个线程组成一个 warp，这 32 个线程每次执行同一条指令，也就是所谓的 SIMT，即使最后一个 warp 中有效的线程数量不足 32，也要使用相同的硬件资源，所以 block_size 最好是 32 的整数倍。

block 有时也会被称之为 Cooperative Thread Arrays，参考

> *The Parallel Thread Execution (PTX) programming model is explicitly parallel: a PTX program specifies the execution of a given thread of a*

> *parallel thread array. A cooperative thread array, or CTA, is an array of threads that execute a kernel concurrently or in parallel. Threads within a CTA can communicate with each other. To coordinate the communication of the threads within the CTA, one can specify synchronization points where threads wait until all threads in the CTA have arrived.*

与 block 对应的硬件级别为 SM，SM 为同一个 block 中的线程提供通信和同步等所需的硬件资源，跨 SM 不支持对应的通信，所以一个 block 中的所有线程都是执行在同一个 SM 上的，而且因为线程之间可能同步，所以一旦 block 开始在 SM 上执行，block 中的所有线程同时在同一个 SM 中执行（并发，不是并行），也就是说 block 调度到 SM 的过程是原子的。SM 允许多于一个 block 在其上并发执行，如果一个 SM 空闲的资源满足一个 block 的执行，那么这个 block 就可以被立即调度到该 SM 上执行，具体的硬件资源一般包括寄存器、shared memory、以及各种调度相关的资源，这里的调度相关的资源一般会表现为两个具体的限制，`Maximum number of resident blocks per SM` 和 `Maximum number of resident threads per SM` ，也就是 SM 上最大同时执行的 block 数量和线程数量。因为 GPU 的特点是高吞吐高延迟，就像一个自动扶梯一分钟可以运送六十个人到另一层楼，但是一个人一秒钟无法通过自动扶梯到另一层楼，要达到自动扶梯可以运送足够多的人的目标，就要保证扶梯上同一时间有足够多的人，对应到 GPU，就是要尽量保证同一时间流水线上有足够多的指令。

要到达这个目的有多种方法，其中一个最简单的方法是让尽量多的线程同时在 SM 上执行，SM 上并发执行的线程数和SM 上最大支持的线程数的比值，被称为 Occupancy，更高的 Occupancy 代表潜在更高的性能。显然，一个 kernel 的 block_size 应大于 SM 上最大线程数和最大 block 数量的比值，否则就无法达到 100% 的 Occupancy，对应不同的架构，这个比值不相同，对于 V100 、 A100、 GTX 1080 Ti 是 2048 / 32 = 64，对于 RTX 3090 是 1536 / 16 = 96，所以为了适配主流架构，如果静态设置 block_size 不应小于 96。考虑到 block 调度的原子性，那么 block_size 应为 SM 最大线程数的约数，否则也无法达到 100% 的 Occupancy，主流架构的 GPU 的

SM 最大线程数的公约是 512，96 以上的约数还包括 128 和 256，也就是到目前为止，block_size 的可选值仅剩下 128 / 256 / 512 三个值。

还是因为 block 调度到 SM 是原子性的，所以 SM 必须满足至少一个 block 运行所需的资源，资源包括 shared memory 和寄存器，shared memory 一般都是开发者显式控制的，而如果 block 中线程的数量 * 每个线程所需的寄存器数量大于 SM 支持的每 block 寄存器最大数量，kernel 就会启动失败。目前主流架构上，SM 支持的每 block 寄存器最大数量为 32K 或 64K 个 32bit 寄存器，每个线程最大可使用 255 个 32bit 寄存器，编译器也不会为线程分配更多的寄存器，所以从寄存器的角度来说，每个 SM 至少可以支持 128 或者 256 个线程，block_size 为 128 可以杜绝因寄存器数量导致的启动失败，但是很少的 kernel 可以用到这么多的寄存器，同时 SM 上只同时执行 128 或者 256 个线程，也可能会有潜在的性能问题。但把 block_size 设置为 128，相对于 256 和 512 也没有什么损失，128 作为 block_size 的一个通用值是非常合适的。

确定了 block_size 之后便可以进一步确定 grid_size，也就是确定总的线程数量，对于一般的 elementwise kernel 来说，总的线程数量应不大于总的 element 数量，也就是一个线程至少处理一个 element，同时 grid_size 也有上限，为 `Maximum x-dimension of a grid of thread blocks` ，目前在主流架构上都是 2^31 - 1，对于很多情况都是足够大的值。

有时为每个 element 创建一个线程是可行的，因为线程的创建在 GPU 上是一个开销足够低的操作，但是如果每个线程都包含一个公共的操作，那么线程数的增多，也代表着这部分的开销变大，比如

```
__global__
void kernel(const float* x, const float* v, float* y) {
   const float sqrt_v = sqrt(*v);
   const int idx = blockIdx* gridDim.x + threadIdx.x;
   y[idx] = x[idx] * sqrt_v;
}
```

这个 kernel 中对 v 的处理是公共的，如果我们减少线程的数量并循环处理 y 和 x，那么 `sqrt(*v)` 的开销就会相应降低，但是

grid_size 的数值不应低于 GPU 上 SM 的数量，否则会有 SM 处于空闲状态。

我们可以想象，GPU 一次可以调度 SM 数量 * 每个 SM 最大 block 数个 block，因为每个 block 的计算量相等，所以所有 SM 应几乎同时完成这些 block 的计算，然后处理下一批，这其中的每一批被称之为一个 wave。想象如果 grid_size 恰好比一个 wave 多出一个 block，因为 stream 上的下个 kernel 要等这个 kernel 完全执行完成后才能开始执行，所以第一个 wave 完成后，GPU 上将只有一个 block 在执行，GPU 的实际利用率会很低，这种情况被称之为 tail effect，我们应尽量避免这种情况。将 grid_size 设置为精确的一个 wave 可能也无法避免 tail effect，因为 GPU 可能不是被当前 stream 独占的，常见的如 NCCL 执行时会占用一些 SM。所以无特殊情况，可以将 grid_size 设置为数量足够多的整数个 wave，往往会取得比较理想的结果，如果数量足够多，不是整数个 wave 往往影响也不大。

综上所述，普通的 elementwise kernel 或者近似的情形中，block_size 设置为 128，grid_size 设置为可以满足足够多的 wave 就可以得到一个比较好的结果了。但更复杂的情况还要具体问题具体分析，比如如果因为 shared_memory 的限制导致一个 SM 只能同时执行很少的 block，那么增加 block_size 有机会提高性能，如果 kernel 中有线程间同步，那么过大的 block_size 会导致实际的 SM 利用率降低，这些我们有机会单独讨论。

其他人都在看