CUDA指令和架构

A100显卡上的 tensorcore有自己的私有寄存器吗，微架构探索 https://zhuanlan.zhihu.com/p/620257581

乘影 GPGPU架构文档手册 v2.01.pdf

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

https://www.tinyedi.com/cuda_learning/#pipeline

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

https://zhuanlan.zhihu.com/p/486224812

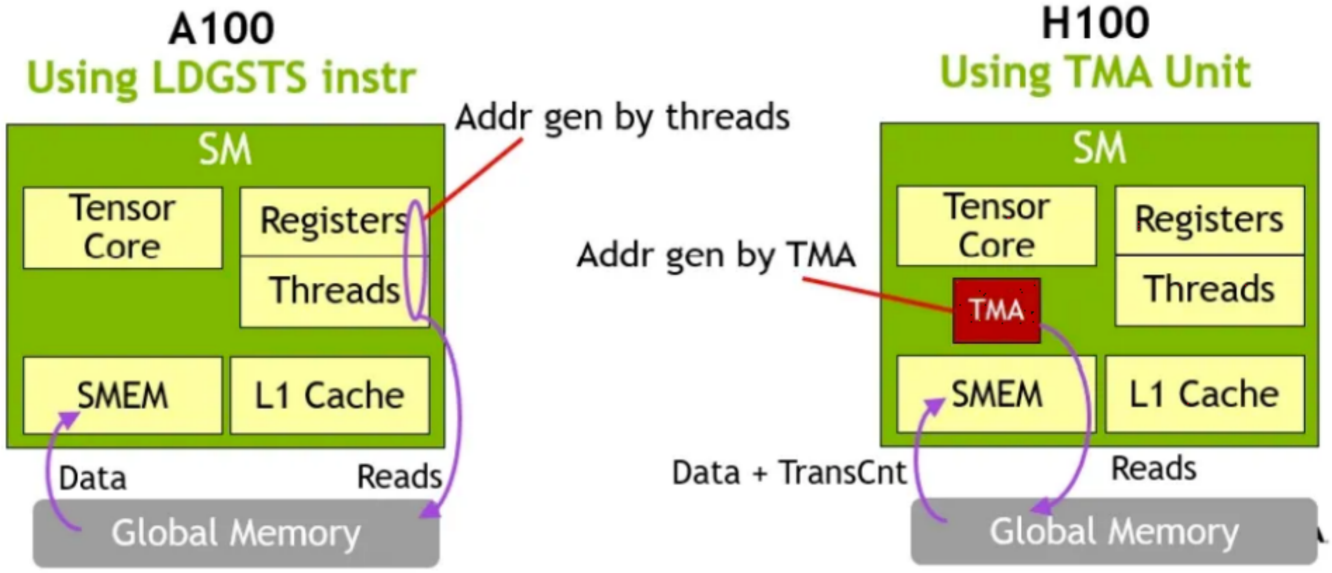
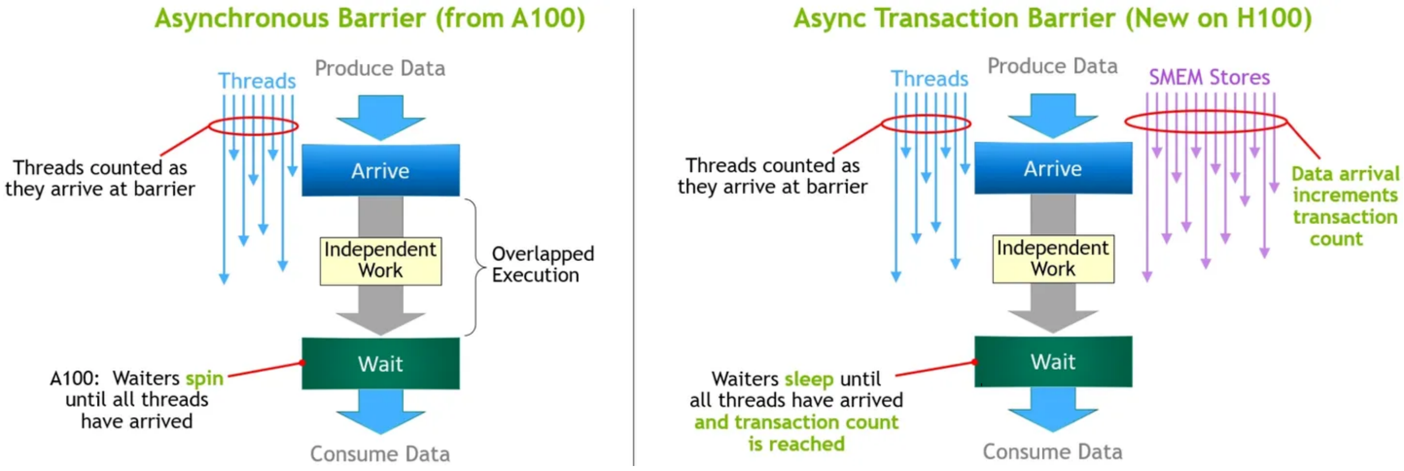


Figure 19. Asynchronous Memory Copy with TMA on H100 vs LDGSTS on A100



虽然异步屏障仍然是 Hopper 编程模型的一部分，但 Hopper 添加了一种新形式的屏障，称为 Asynchronous Transaction Barrier。Asynchronous Transaction Barrier 与异步屏障非常相似。它也是一个拆分屏障，但它不仅计算线程到达，还计算事务。Hopper 包含一个用于写入共享内存的新命令，该命令传递要写入的数据和事务计数。事务计数本质上是一个字节计数。异步事务屏障将阻塞等待命令的线程，直到所有生产者线程都执行了到达，并且所有事务计数的总和达到预期值。异步事务屏障是用于异步内存副本或数据交换的强大新原语。

Predicate 有时候代指指令前加的形如 @P0, @!P3 等的 predication，PTX文档里也把这个叫 guard predicate。有时候也会特指那个 predicate register，如 P0, P6, PT，反正都差不多这个意思，不混淆就行。在之前的文章也提到过，在现有的所有架构中，每个指令都有 4bit的编码来指定每个 predicate，3bit用来指定索引（所以每线程有 2^3=8个 predicate register P0~P7，其中 P7=PT 为恒 True），1bit表示是否取反。如果是 @PT，那就会在反汇编中省去不显示。那如果是 @!PT 呢？嗯，大家自己想……

Predicate是控制某个线程是否执行某个指令的两种方式之一，另一种就是 conditional branch。两者的区别在于用 predicate时，可以让 warp内的所有线程名义上走同一路径而省去跳转的开销，从而也避免了可能的 divergence。因为 branch的 latency比较长，还涉及到 instruction cache的问题，一般很短的分支是不太愿意跳转的。当然 branch在出现 divergence的时候，内部也有一个 mask，表明当前这个 thread是否 active，但是用户不能直接修改这个 mask。PTX中可以通过 warp vote或是 load特殊寄存器 %lanemask_* 之类的方法获得当前 warp内的 mask情况。

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。PTX有相关的控制指令，但是我在 Turing上测试过几次，只测到过 TLB的 latency被消除，实际访问 latency好像没变，存疑。

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。

memory类的指令是性能优化的重点中的重点。绝大部分未经优化的程序都会是 memory bound，多数实际应用优化完了还是 memory bound。所以，通过合理的选择相应指令，搭配合适的内存排布，从而更好的隐藏内存访问的 latency，或者是减少相应访问的开销，是性能优化中的主要

课题之一。

SASS里并没有直接的条件跳转指令，所有的条件跳转是用predicate实现的。

每个指令能否发射还要满足相应的依赖关系和资源需求。比如指令 `LDG.E R6, [R2] ;` 首先需要等待之前写入 `R[2:3]` 的指令完成，其次需要当前memory IO的queue还有空位，否则指令也无法下发。还有一些指令可能有conflict的情况，比如shared memory的bank conflict，register的bank conflict，atomic的地址conflict，constant memory在同一warp内地址不统一的conflict等等，这些都有可能导致指令re-issue（甚至cache miss也可能导致指令replay）。这些情况会不会重复占用dispatch port发射带宽

当前CUDA的所有架构都没有乱序执行（Out of order）

Register Reuse Cache：编译器hint，明确一个寄存器的值会被下面的指令使用到，可以放到cache里面去，避免再次read

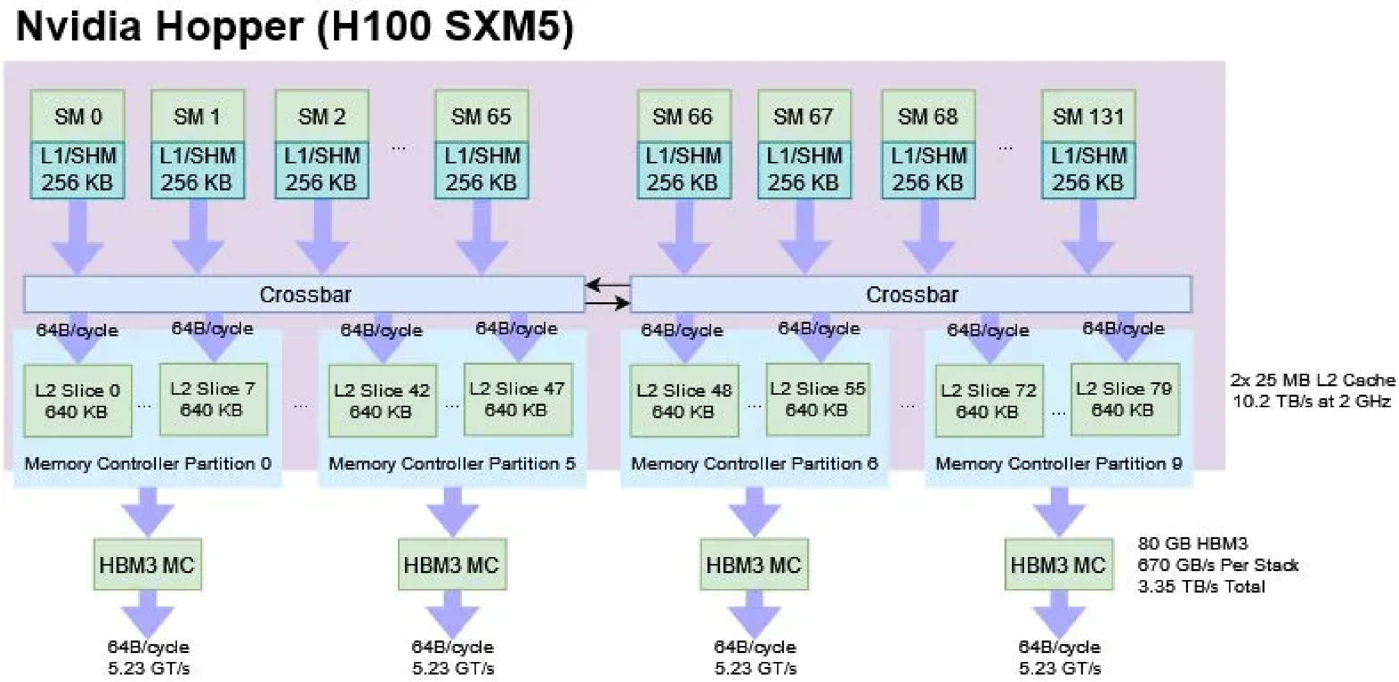
Wait Dependency Barrier：有6bit，每个bit表示是否需要等待对应的dependency barrier。每个线程有6个dependency barrier，每个barrier都可以被后面的Read或Write操作设置上。设置wait dependency barrier是等待依赖的其中一种方式。SASS里面还有一个对应的指令，如 `DEPBAR.LE SB0, 0x0, {2,1} ;`

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

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

Yield hint flag：是1bit。如果Yield，就表示下一个cycle会优先发射其他warp的指令。也许yield这个bit就是stall count的高位。只是假如这个bit不为0，那stall的cycle会>16，相当于warp被切换的概率也会大大增加。

Stall count：有4bit，表示当前指令后需要stall指令发射的cycle数，然后再决定是不是要继续发射。



CUDA性能优化原则

- 寻找并行化顺序代码的方法。
- 最大限度地减少主机和设备之间的数据传输。
- 调整内核启动配置以最大限度地提高设备利用率。 Adjust kernel launch configuration to maximize device utilization.
- 确保全局内存访问被合并。
- 尽可能减少对全局内存的冗余访问。 Minimize redundant accesses to global memory whenever possible.
- 避免同一扭曲内的线程进行长序列的发散执行。 Avoid long sequences of diverged execution by threads within the same warp.

TMA

TMA 允许应用程序在全局内存和共享内存之间以及同一集群中不同 SM 的共享内存区域之间双向传输 1D 和最多 5D 张量（请参阅 **线程块集群**______）。此外，对于从共享内存到全局内存的写入，它允许指定按元素归约操作，例如 add/min/max 以及按位和 / 或最常见的数据类型。

memcpy_async的 API有三个变形，分别是 group,barrier,pipeline.

- group: 搭配 `cooperative_groups::wait(group)` 使用 ,注意不是 `group.sync()`, 从逻辑上说，相当于动态为group插入了一个异步线程. `cooperative_groups::wait` 只是让当前线程等待所有异步线程完成 ,并不会有 `group.sync()` 的效果 .
- barrier: 搭配 `barrier.arrive_and_wait()` 使用 ,从实现角度讲 ,相当于动态增大了 barrier的 count,这个count只有在 async_memcpy完成时才会下降
- pipeline: 约等于线程安全的 queue

Revision #1
Created 11 January 2025 09:46:27 by Colin

Updated 12 January 2025 06:33:19 by Colin