NVIDIA GPU性能优化基础

1 introduction

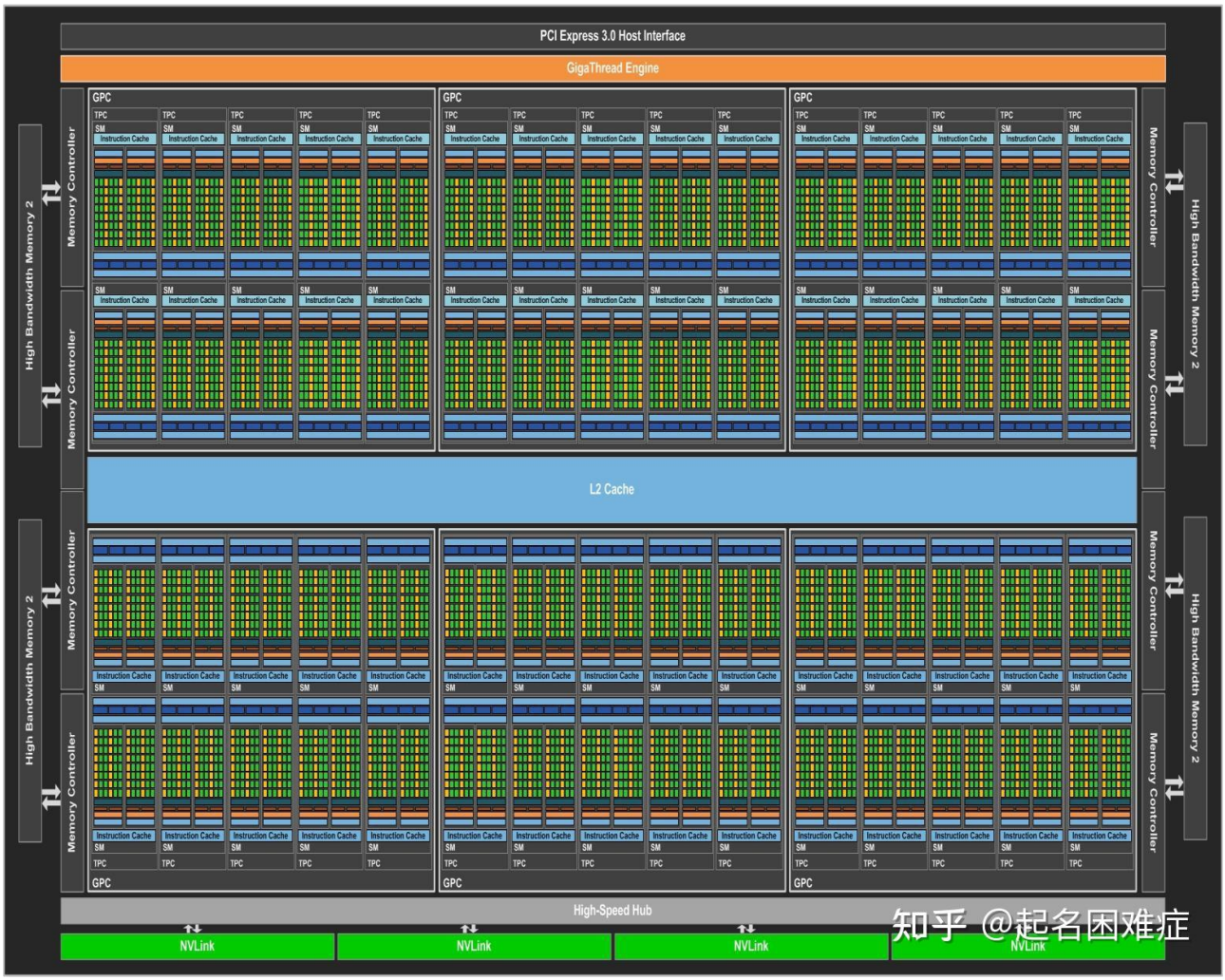
本文介绍NVIDIA GPU上做性能优化的一些基础知识，包括SM structure, memory hierarchy, execution model等体系结构方面的知识，此外也简单介绍了nsight compute profiling工具的使用。

文章的内容大部分都可以在网络上找到相关资料，本文更多地是对这些纷繁、离散的资料做了一些整合、梳理，然后加入了自己的思考、理解，最终希望能形成一个初步的知识体系。由于个人水平和文章篇幅的限制，难免有不深入、不准确的地方，欢迎大家指正、交流。

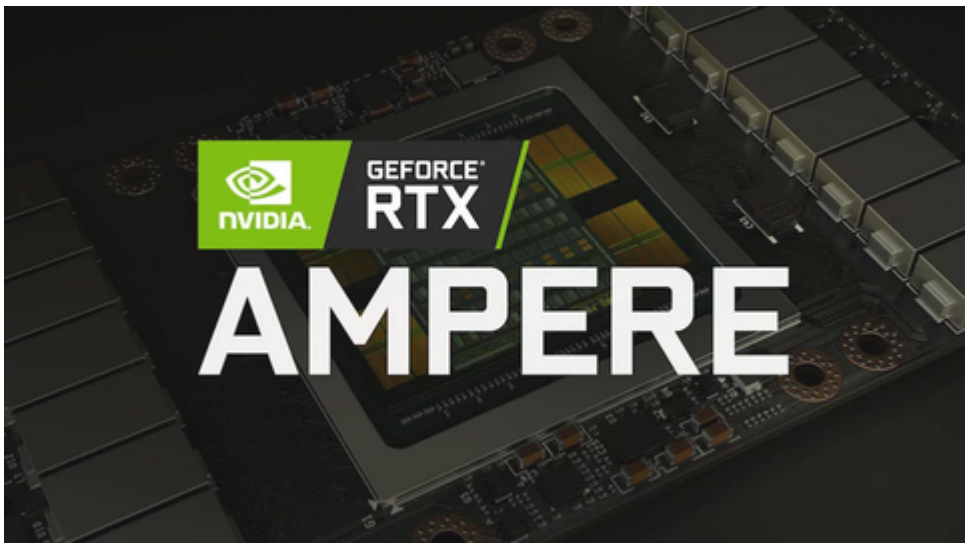
2 stream multiprocessor

2.1 overview

在本章节中，我们以pascal架构为例，介绍一下SM的典型结构。下图是pascal架构GP100的结构示意图。GP100主要由Graphics Processing Cluster (GPC)和Memory Controller等主要部件构成。GPC是GPU计算的核心部件，GP100共有6个GPC，每个GPC的结构是分层的，其中包含了5个Texture Processing Cluster (TPC)，每个TPC中又包含了两个Streaming Multiprocessor (SM)。所以GP100一共包含60个SM。memory controller是GPU访存的核心部件，GP100共有8个memory controller，每个memory controller通过512bit的接口与HBM2交互，并且绑定了512KB的L2 Cache，所以完整的GP100有4096bit的memory interface，并且有4MB的L2 Cache。L2 Cache是所有的计算单元共享的。



nvidia GPU的架构是不断演进的，不同generation的架构之间往往存在不小的差异，但上述这种分层的组织架构一直是延续的。对于nvidia gpu架构的发展史，可以参考这篇文章：



2.2 typical SM structure

下图是一个pascal SM的内部结构示意图，其包含两个sub-partition。每个sub-partition有独立的instruction buffer、warp scheduler、register file，计算单元和访存单元。同一个SM上的多个sub-partition会共享统一的Texture/L1 cache、Texture Unit和shared memory。

GP100 SM结构图

下面简单介绍一下各个单元的功能[1]：

- CUDA Core：负责执行单精度操作，比如FMUL、FADD和FFMA等。为了提高指令的throughput，CUDA Core是pipelined。
- DP Unit：负责执行双精度（duoble precision）操作。
- LD/ST：负责memory load，store等操作。
- SFU：特殊函数单元（Special Function Unit）负责计算特殊的函数，比如cuda中的__cosf()、__expf() intrinsics就调用了SFU单元。需要注意的是，SFU实现的是快速近似操作，可能会影响精度。

- register file：寄存器组，用来存储数据
- dispatch unit：负责将不同类型的指令要被分配到不同的单元上去执行
- warp scheduler：负责管理一系列的warp，并从中选取符合条件的warp发射指令

整个GPU芯片的算力等于单个SM的算力乘以SM的数量。而在一个SM中，不同的计算单元的数量以及单个时钟周期内能做的计算次数是不同的。所以单个SM的算力通过如下公式计算：

> *单个SM算力 = GPU主频 * 计算单元数量 * 单个时钟周期内能做的计算次数*

以Tesla P100为例，GPU主频是1.48GHz，单精度CUDA Core的数量是64，每个CUDA core单个时钟周期能执行一条FFMA指令，做两个浮点计算。所以单个SM和整个GPU的算力计算如下：

> *单SM FP32 算力 = 1.48 GHz * 64 CUDA Cores * 2 Operations/cycle = 189.44 GFLOPs*
> *P100 FP32 算力 = 189.44 GFLOPs * 56 SMs = 10.6 TFLOPs*
> *P100 FP16 算力 = 10.6 * 2 = 21 TFLOPs*

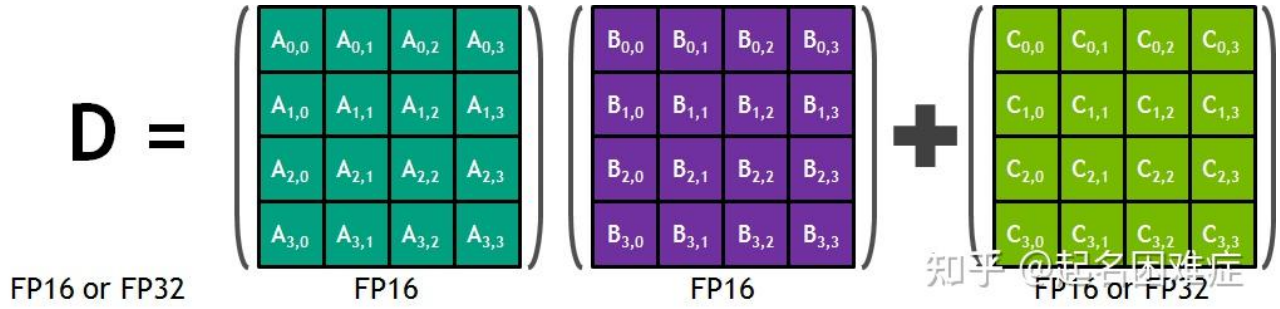
在逻辑上，一个cuda core做的是scalar运算，一个sub-partition中32个cuda core，可以执行一个warp实现vector运算。

2.3 tensor core

在深度学习等应用中，矩阵乘法运算很常用。在vector架构下，要做一个矩阵乘法，需要迭代多次分别计算不同行列的dot product。为了加速矩阵乘法运算，从volta架构开始，nvidia gpu上引入了一种新的计算单元，tensor core。如下图所示，volta的tensor core一个cycle可以计算两个4x4矩阵的乘加操作，即64个FMA。一个volta的SM中有8个tensor core，峰值计算能力是8x64=512 FMA。而一个

pascal的SM种有64个cuda core，一个cycle可以计算64个FMA。。整个Tesla V100 GPU有80个SM，主频是1.53GHz，其算力计算如下：

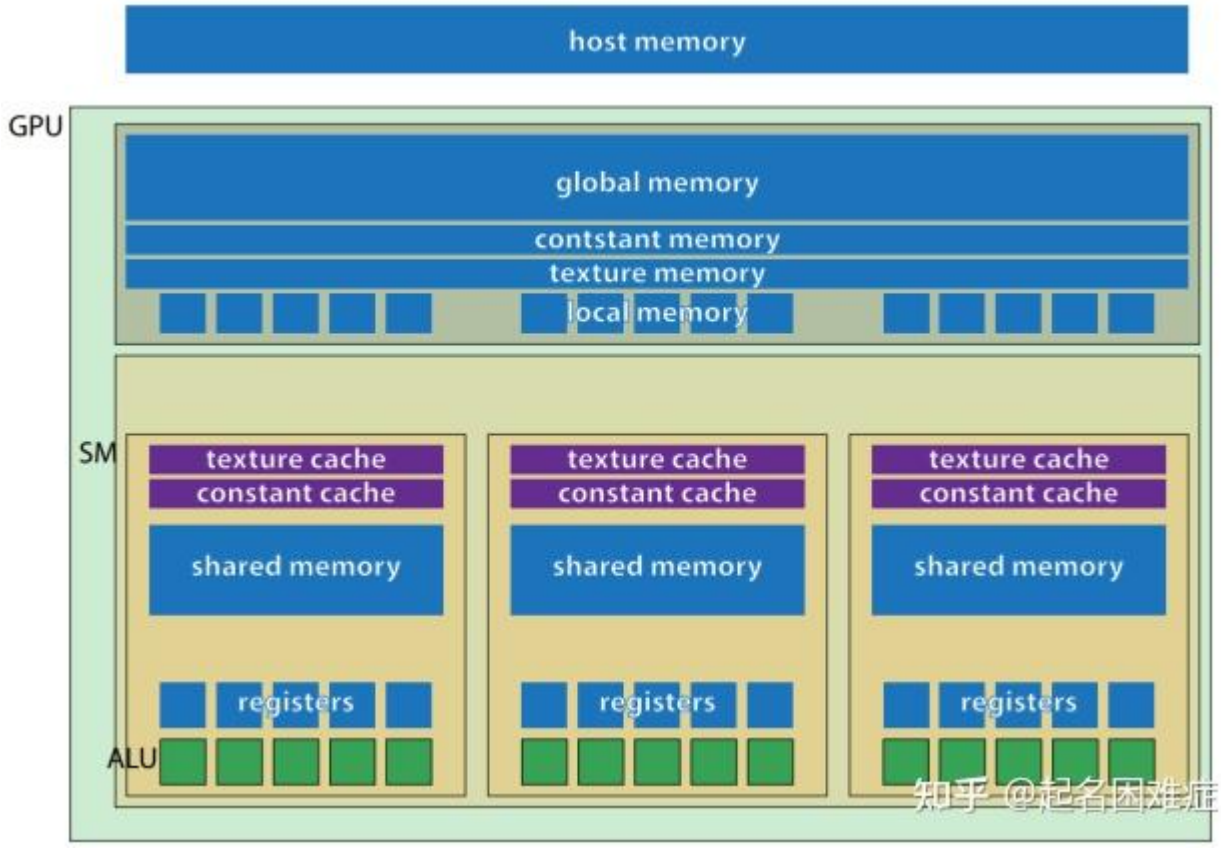
> *单SM FP16 算力 = 1.53 GHz * 8 Tensor Cores * 128 Operations/cycle = 1556.72 GFLOPs*
> *V100 FP16 算力 = 1556.72 GFLOPs * 80 SMs = 124.5 TFLOPs*



拥有了tensor core的加持，V100的FP16算力是P100的FP16算力的6倍。tensor core的引入极大提升了GPU的算力，但是memory bandwidth的进步却相对较为缓慢。这也使得越来越多的workload越来越倾向于memory bound。因此，在现代gpu开发中，如何优化memory访问变得越来越重要。

3 memory hierarchy

如下图所示，nvidia GPU上的memory有多种类型，如global memory，shared memory等，不同类型的memory有不同的特性，需要不同的优化方法。



在介绍具体类型的memory特性和优化方法之前，我们先了解几个跟memory相关的概念[2]：

- request：一个request指的是一条memory access instruction
- transaction：一个transaction指的是两个memory区域之间的一次单元数据移动。

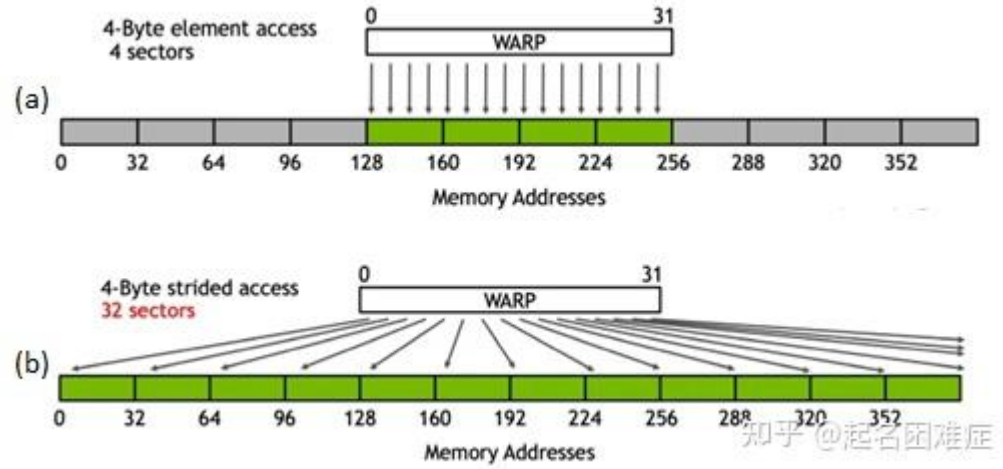
高效的访存模式通常就是指最小化每次request的平均transaction数量。而低效的访存模式往往需要大量的transactions，但只利用了被传输的数据中的一小部分，浪费了memory bandwidth。实现高效的访存是memory优化的关键。

3.1 global memory

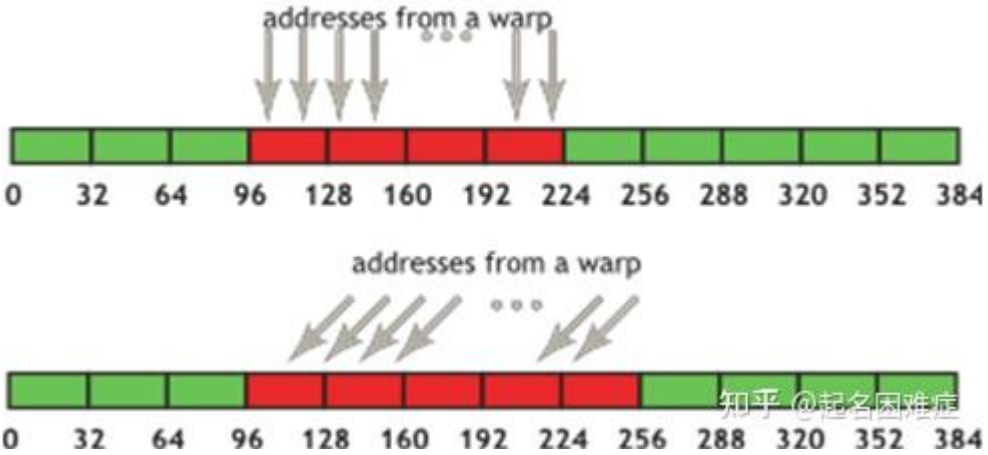
Global memory是以sector为单元来组织的[3]，每个sector为32 Bytes，每次访存的粒度是一个sector。如下图(a)，我们假设每个thread访问一个4 bytes的float数据，并且一个warp中的thread访问的地址是连续的，那么每四个thread的内存访问可以合并成一个

sector的访问，总共只需要访问4个sector。这就是常说的合并访存（coalesced memory access）。

如果warp内每个thread访问的地址是不连续的，如下图(b)所示的strided访存模式，每个thread依然读取一个float数据，但两个thread的访存地址间隔32Bytes，则一共要访问32个sectors，总共读取了32x32=1024Bytes的数据，但实际用到的只有4x32=128Bytes，带宽利用率只有128/1024=12.5%。

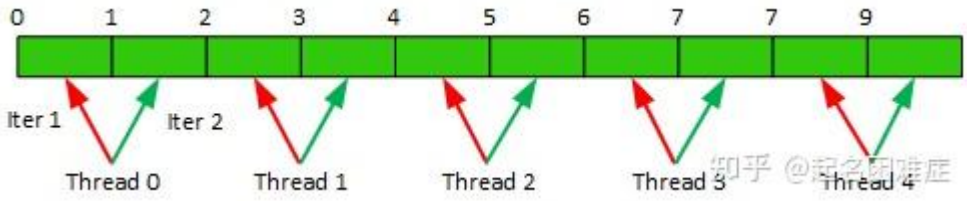


通过上面的对比我们可以看到，合并访存能减少memory access的次数，更加高效的利用memory的带宽。此外，如果内存首地址不对齐，则读取同样的数据可能需要更多的访存操作。如下图所示。同样读取128个字节，首地址不对齐的情况要比对齐的情况多读取一个sector。



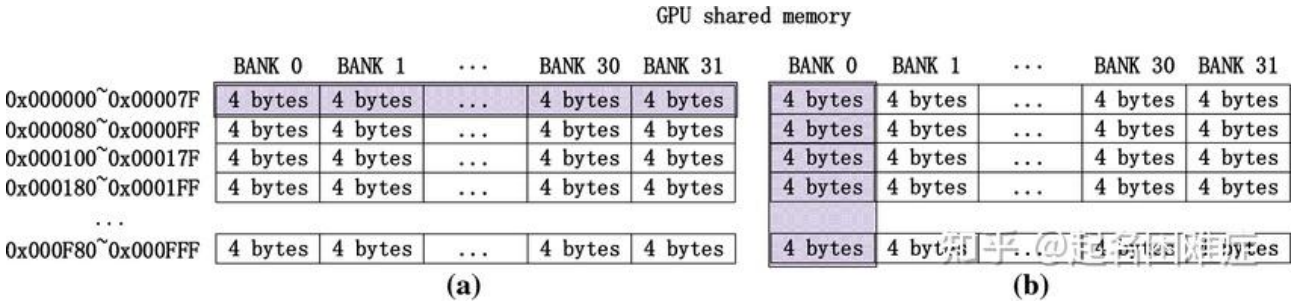
如下图所示，在有的情况下，一个thread可能需要访问相邻的多个float数据。如果每个thread按照循环的方式逐个读取元素，则会遇到上面提到的strided访存的问题。此时，我们可以用cuda提供的

float2类型，一次性的读取两个float，就能满足合并访存的要求了。对每个thread，访存粒度可以是1B,2B,4B,8B,16B。

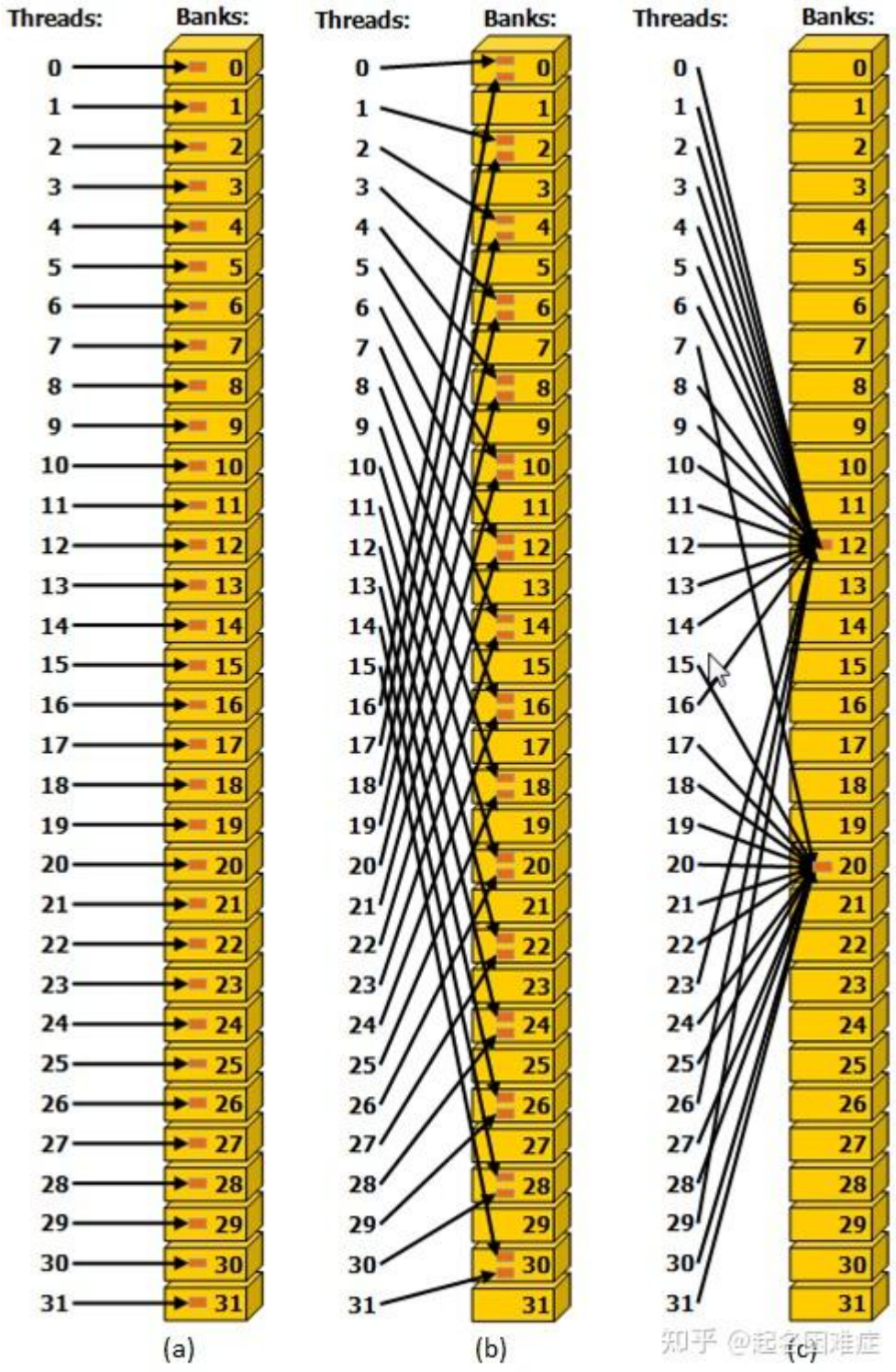


3.2 shared memory

shared memory是每个SM独有的高速片上memory。为了提高bandwidth[4]，shared memory是按照banking的方式组织的。如下图所示，整个shared memory一共分成32个bank，每相邻的4Bytes被映射到相邻的bank中。



如果一个warp中的多个thread访问的地址处于不同的bank，则多个bank可以同时去准备不同的数据，在一个cycle之内将所有数据准备完毕。但如果同一个warp中多个thread访问同一个bank中的不同地址(不同的地址不在32bit之内)，就会发生bank conflict。在发生bank conflict的时候，硬件会将一个request分成多个没有conflict的requests，一个bank要花费多个cycle分别响应多个requests，访存的latency变长。如果多个thread访问的是同一个bank中的同一个地址，那么只会发生一次访存，然后broadcast给不同的线程，也不会有bank conflict。如下图所示，(a)和(c)所示的访存模式没有bank conflict，而(b)的访存模式存在bank conflict。



3.3 local memory

local memory实际应该被称为thread local global memory，是位于global memory中一片内存区域。local memory的访问速度和global memory一样，非常的慢。对于在kernel code中定义的一些

局部变量或者数组，在特定情况下会被nvcc自动放到在local memory中[5]：

- 动态索引的数组：比如在访问数组的时候，下标是runtime动态计算出来的，那么这种数组会被放在local memory中
- 寄存器数量不够：当register数量不够用的时候，compiler会将一部分的寄存器数据暂存到local memory中，需要用的时候再load回register。这个过程被称为register spilling[6]。
- 可能会占用大量寄存的大结构体或者数组，也会直接放在local memory里面。

local memory会被L1 cache handle。尽管如此，使用local memory还是可能会hurt performance，我们还是需要注意的。

- 增加了memory的访问量
- local memory的load/store操作增加了instruction的数量

优化方法通常有：

- 降低thread的register使用数量，或者提高register上限
- 增大L1 cache来提高hit rate
- 使用non-caching的global memory load，从而减少L1 cache的convention

3.4 texture memory 和 constant memory

texture和constant memory是和global memory一样，都是位于片外的device memory上，访问速度和global memory一样。区别在于：但constant memory和texture memory都是ready-only的，并且有专用的constant cache和texture cache。

3.5 cache

nvidia gpu上有下面这几种不同的cache：

- L2 cache：L2 cache是所有SM共享的，容量通常在MB级别，比如P100 L2 cache size是4MB。
- L1 cache：L1 cache是每个SM独享的，容量通常在几十KB级别，比如P100的L1 cache size是24KB per SM。在有些架构中，L1 cache和shared memory是unified，可以配置大小。
- texture cache：texture cache是texture memory对应的cache，针对2D locality做了专门的优化。在有些架构中，texture cache和L1 cache是在一起的。
- constant cache：constant cache是constant memory专用的cache。

一般来说，global memory和local memory会被L1/L2 cache缓存。但在不同的compute capability中，cache behaviors 有细微差异。如下图所示[7]，在CC 6.0和CC 7.X中，global memory是默认被L1和L2 cache缓存的，但在其他更低的CC中，global memory默认只被L2缓存，用户需要通过设置特定的compilation flag来enable L1 cache，或者需要在代码中将read-only的指针用const __restrict__修饰，从而enable L1 cache[8]。

Table 1. Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

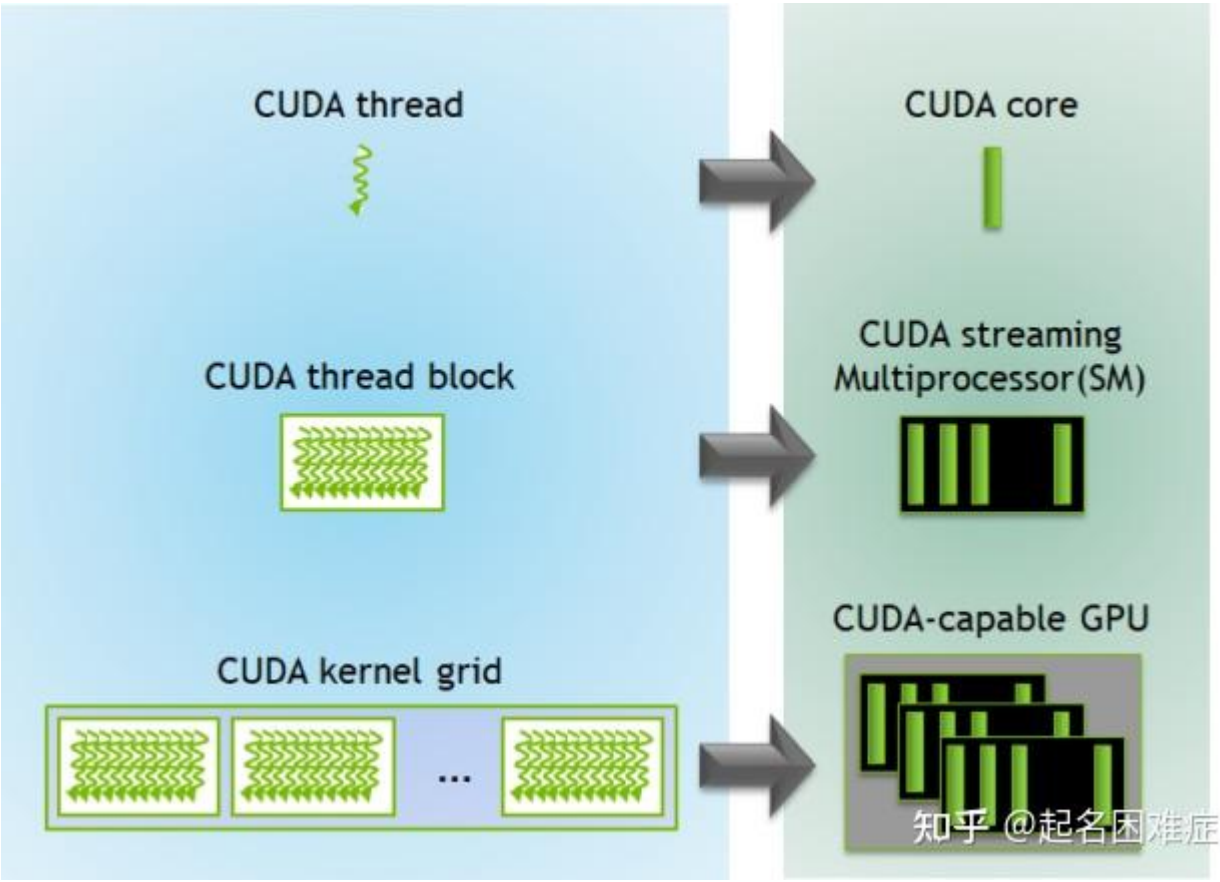
† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

4 execution model

这里的execution model是指CUDA kernel在GPU硬件上的运行方式。在CUDA编程模型中，计算任务是以thread和thread block的形式进行组织的。我们通常会将计算任务切分成多个可并行的子块，交给多个thread block计算。在thread block内部，我们再将任务进一步划分成多块，由每个thread计算。GPU硬件也是采用了分层次的组织方

式，被划分成多个SM，每个SM内部又有多个CUDA Core。CUDA thread和thread block最终是运行在CUDA Core和SM上面，如下图所示[9]。



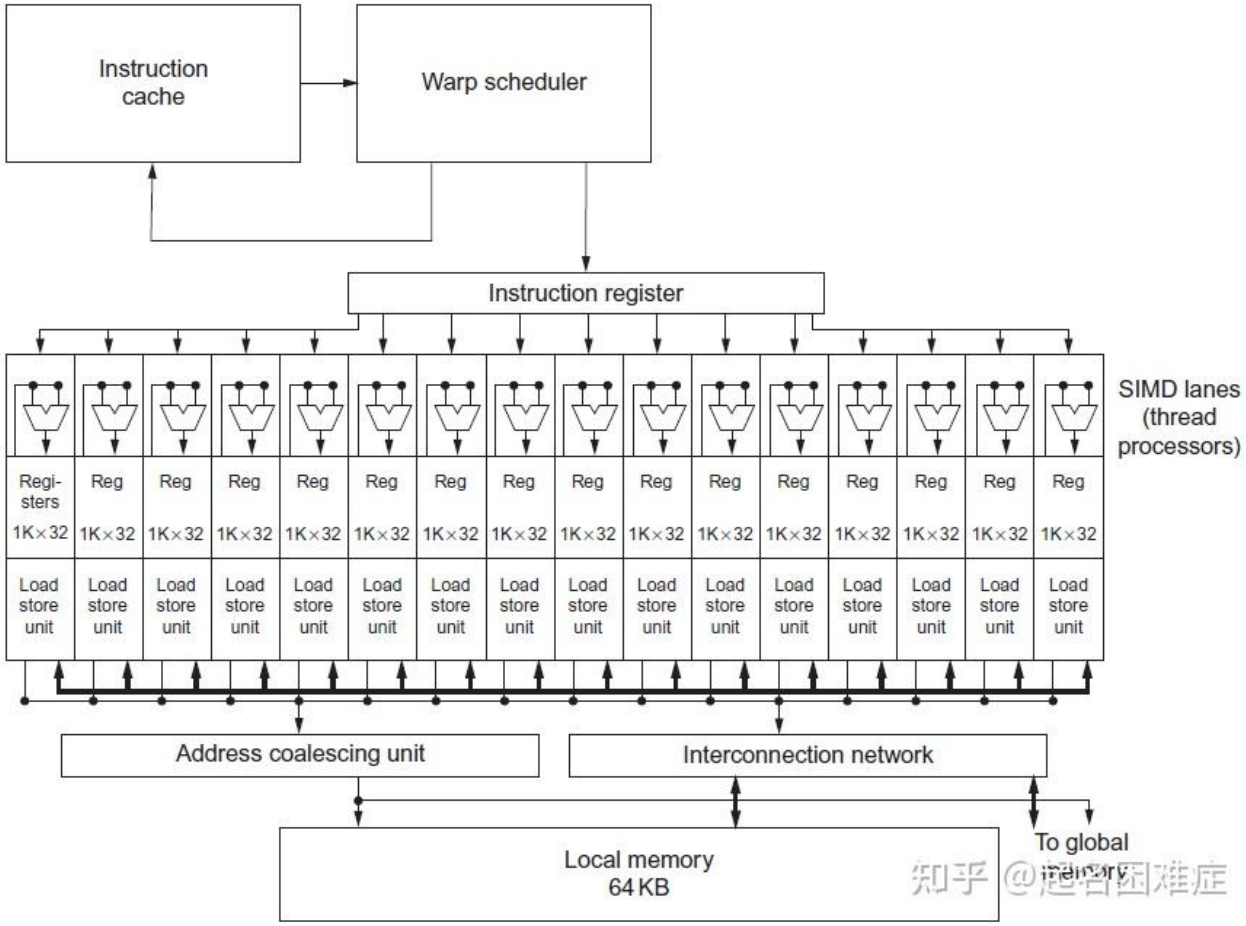
上文只是定性的描述。CUDA Core和SM的数量因GPU架构而异，thread和thread block的数量也取决于workload，不可能存在固定的一一映射关系。下面我们就更加深入的研究一下GPU的execution model。

4.1 thread execution

(1) SIMT execution

CUDA thread不像CPU thread那么灵活，它并不能完全独立的运行，而是将32个thread id连续的threads分组成一个warp，一个warp中的所有threads在任意时刻必须执行相同的指令。这是因为，GPU硬件实际上是采用的如下图所示的SIMD架构[10](借用《Computer Architecture: A Quantitative Approach》中的图)。一个SIMD处理单元有N个lane，这些lane在相同指令的控制下，同时对N个数据做相同的操作。不同的指令类型对应着不同的执行单元，单精度运算

执行单元的lane就是所谓的CUDA core，双精度执行单元的lane，就是所谓的DP Unit。



GPU的指令都是以warp-wide（即一条指令的作用范围是一个warp），但SM中的执行单元可能并没有足够多的lane在一个cycle之内同时处理一个warp中所有thread的运算，这时执行单元会分成多个cycle多个批次执行该指令[11]。比如，pascal SM的每个sub-partition只有8个DP Unit，如果warp中的每个thread都要做双精度运算，则一共需要32次双精度运算，至少需要分4次才能完成。如果DP Unit每次运算的latency是1个cycle，那么至少需要4个cycles才能完成一个warp-wide的双精度运算。同理，在有的架构中，一个sub-partition只有16个CUDA Core，则需要2个cycles才能完成一个warp-wide的单精度运算。虽然因为资源的限制，一个warp-wide的操作可能要分多个周期才能执行完，但从宏观的层面来看，一个warp中所有thread的指令还是同步执行的。

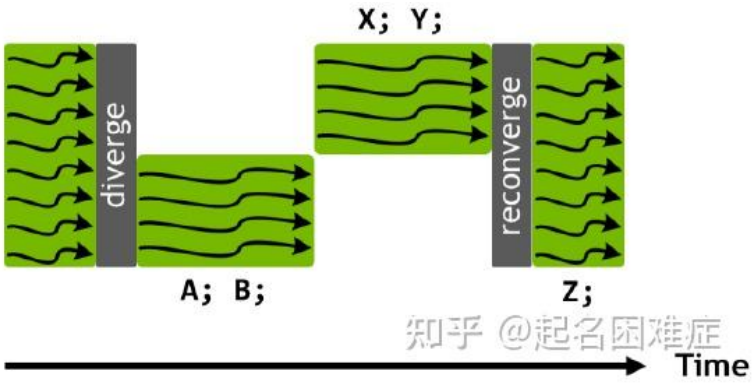
既然GPU硬件是按照SIMD的方式运行，那我们很自然的会联想到两个问题：如果同一个warp中不同的threads运行到不同的条件分支怎么办？

如果不同的thread要访问不同的memory地址怎么办？这里就涉及到两个概念：divergence和instruction replay。

(2) divergence

在Volta之前的gpu架构中，一个warp中的所有的threads共用同一个PC指针，然后用active mask来指定哪些threads是active的，只有active的threads才会执行PC执行的指令。所以，程序中的分支操作会被串行化，如下图所示。当分支部分的代码执行完之后，分支之前的active mask会被restore，warp中的threads再次一起运行。

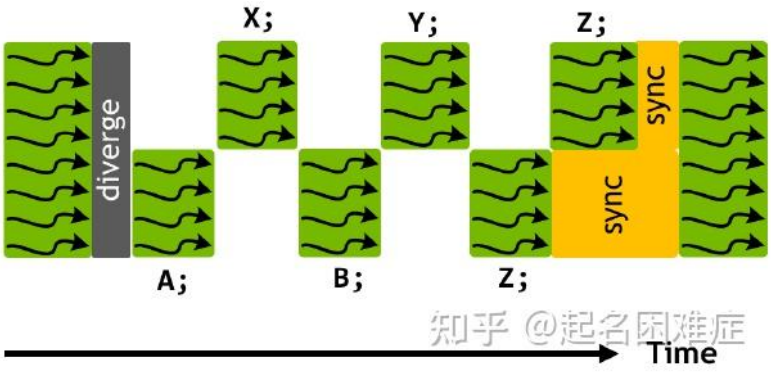
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



上述对divergence的处理方式不需要对单独的thread进行状态追踪，实现上相对简单，需要的资源也比较少。但是，为对divergent path进行串行化，执行A、B和执行X、Y的两部分threads有了必然的前后顺序，在reconvergence之前都失去了concurrency。失去concurrency导致这两部分threads不能互相交互数据，在极端情况下，可能会发生deadlock。

Volta之后的gpu架构引入了一种叫做independent thread scheduling的机制，一个warp中的每个thread都有自己的PC指针，当一部分threads因为A、B的执行而发生stall时，另外一部分threads可以去执行X、Y。independent thread scheduling使得threads的执行方式更加灵活，并且能提高执行效率。而且，active threads在任何一个时刻执行的都是相同的instruction，并没有改变SIMD的执行模式。

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



更加详细和深入的介绍可以参考下面这篇文章：
cloudcore: CUDA微架构与指令集（5）-Independent Thread Scheduling97 赞同 · 12 评论文章

(3) instruction replay

除了上述的branching divergence之外，还存在memory divergence。比如各个threads访问的memory地址不同，且不满足合并访存要求，则无法仅执行一条load指令就将一个warp中所有threads需要的数据准备好。这是，warp scheduler会重复发射访存指令，并且设置相应的active mask，使得每次发射的指令只对某一部分threads生效[12]。这种处理方法被称为instruction replay。只有当同个warp中所有threads的操作执行结束，replay结束，相应的instruction才会算是执行完。所以说，为了执行完一条指令，可能要重复发射多次。导致instruction replay的原因不仅仅有非合并访存，还包括shared memory的bank conflict等[13][14]。

4.2 warp execution

一个thread block会包含多个warp，当一个thread block准备在SM上运行时，首先会为分配shared memory，register等资源。被分配了资源的thread block被称为activate block。activate block中所有的warp共享block的资源，被称为activate warp[15]。activate warp中可以被发射的被称为eligible warp，由于各种原因不能被发射的被称为stall warp。stall的原因主要包括：

- 等待instruction fetch
- memory dependency（等待memory instruction的结果）

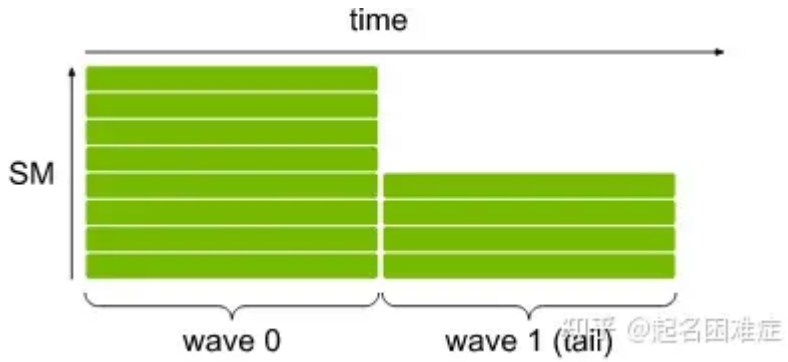
- execution dependency （等待之前instruction的结果）
- synchronization barrier

SM中的warp scheduler每个cycle会从eligible warp中选择一个warp发射指令，被选中的就称为selected warp。selected warp的数量受到SM中的warp scheduler数量以及issue slot的数量的限制。

由于一个block中各个warp所需要的资源都提前分配好了，互相不会冲突，所以，从一个warp到另一个warp的切换并不需要像CPU线程切换那样保存上下文，而是非常迅速的。所以，GPU编程中很重要的一点是要有足够多的eligible warp，当一个warp stall之后，可以无缝切换到另一个eligible warp继续做计算，从而使得GPU的计算单元被占满，也就是所谓的用计算隐藏延迟。

4.3 block execution

一个thread block会被分配到一个SM上执行，但一个SM允许同时执行多个block。当一个SM上有足够的资源时，block scheduler就会调度新的block运行。我们将同时在运行的所有block称为一个wave。如下图所示，如果最后一个wave中的thread block数量有限，不足以填满所有的SM，算力就被浪费，这种情况叫做tail effect[16]，在实际的编程中，也需要注意最小化tail effect[17]。



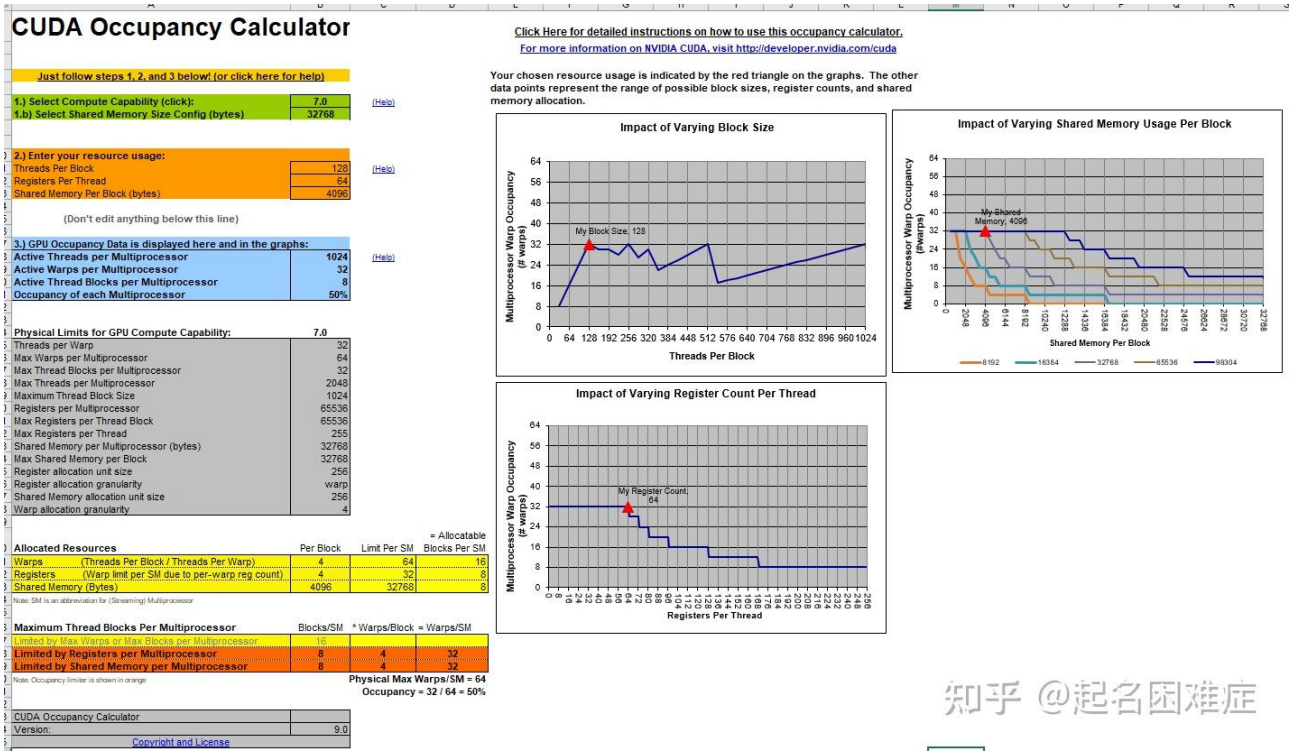
4.4 occupancy

对于特定的GPU架构，一个SM所能支持warp数量是有上限的，我们可以称之为Device Limit warps。每个SM的寄存器，shared memory等资源也是有限的。如果每个thread 占用的资源太多，那么能在一个SM上同时保持active的warp数量机会越少。比如，一个thread需要占用128个寄存器，那么对于GP100架构，一个SM上有64K个register，则最多能供给64K/128=512个thread，即16个warp。如上，根据理论计算得到的active warp数量比上Device Limit就得到了理论占用率Theoretical Occupancy。

影响Theoretical Occupancy的因素不止有register数量，还包括一个thread block占用的shared memory size和thread block的size。下面分别做解释[18]：

- shared memory size per block: shared memory是以thread block为单位分配的，如果一个thread block占用的shared memory size越大，那能在一个SM上面同时保持active的thread block的数量就越少，如果单个thread block中的thread数量固定，那active warp的数量就越少。
- thread block size：一个SM所能支持的thread block数量也有上限，GP100架构是32。所以，即使thread block占用的资源很少，它的数量也不可能无限增加。此时，每个理论active warp的上限就是thread block上限乘以thread block size。

NVIDIA专门提供了一个excel工具CUDA_Occupancy_Calculator[19]用以计算Theoretical Occupancy。



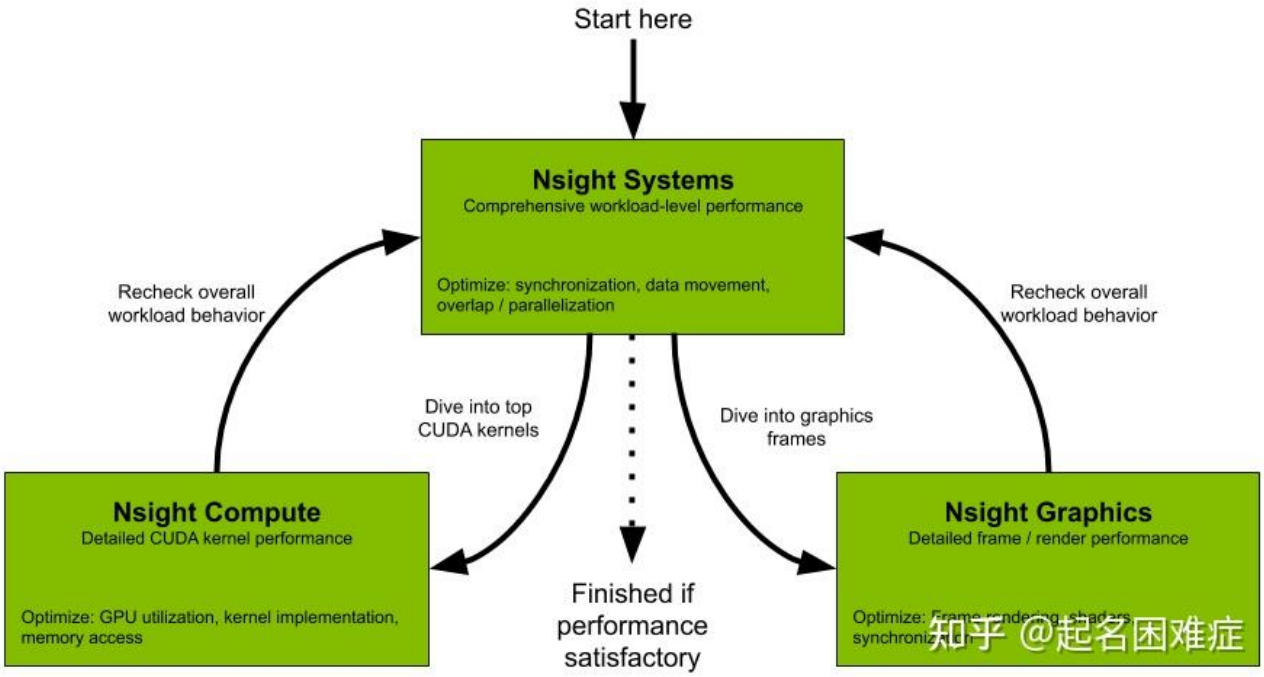
Theoretical Occupancy是理论计算得到的，此外还有Achieved Occupancy是kernel运行期间统计得到的。GPU performance counter累计每个warp scheduler每个cycle中实际active的warp总数，然后除以一个SM active的总cycle数量，就能得到该SM的achieved occupancy。我们希望achieved occupancy能接近theoretical occupancy。导致achieved occupancy较低的原因有[20][17]：

- block内部的warp之间负载不平衡
- block之间负载不平衡
- luanch的block数量太少

5 profiling tools

nvidia目前推荐使用的profiling工具是nsight，包括nsight system，nsight compute和nsight graphics，三者的关系如下图所示。nsight system是在系统层面做profiling，nsight

compute是从cuda kernel 层面做profiling，而nsight graphics用于图形应用的profiling[21]。



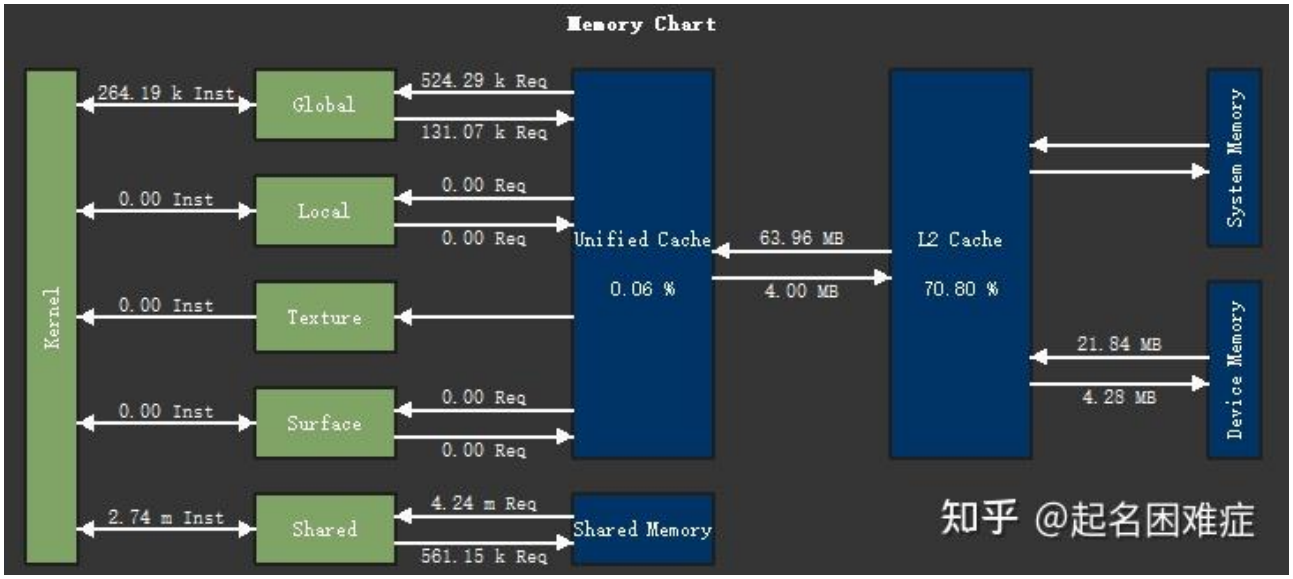
本文简单介绍一下nsight compute。nsight compute的最主要的feature就是收集GPU的一系列performance metrics，比如执行的指令数量，内存访问的次数等[22]。metrics的数量非常多，为了方便使用，nsight compute将metrics归类成不同的sections。目前支持的sections[23]如下图所示，本文简单介绍其中的几个section。

Table 1. Available Sections

Identifier and Filename	Description
ComputeWorkloadAnalysis (Compute Workload Analysis)	Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.
InstructionStats (Instruction Statistics)	Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution.
LaunchStats (Launch Statistics)	Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.
MemoryWorkloadAnalysis (Memory Workload Analysis)	Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Depending on the limiting factor, the memory chart and tables allow to identify the exact bottleneck in the memory system.
Nvlink (Nvlink)	High-level summary of NVLink utilization. It shows the total received and transmitted (sent) memory, as well as the overall link peak utilization.
Nvlink_Tables (Nvlink_Tables)	Detailed tables with properties for each NVLink.
Nvlink_Topology (Nvlink_Topology)	NVLink Topology diagram shows logical NVLink connections with transmit/receive throughput.
Occupancy (Occupancy)	Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.
SchedulerStats (Scheduler Statistics)	Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps, the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.
SourceCounters (Source Counters)	Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.
SpeedOfLight (GPU Speed Of Light Throughput)	High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor.
WarpStateStats (Warp State Statistics)	Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle.

5.1 memory workload analysis

如下图所示，memory workload analysis可以帮助分析kernel访存相关的参数，包括访问各种memory类型的request数量，transactions数量，cache命中率等指标。

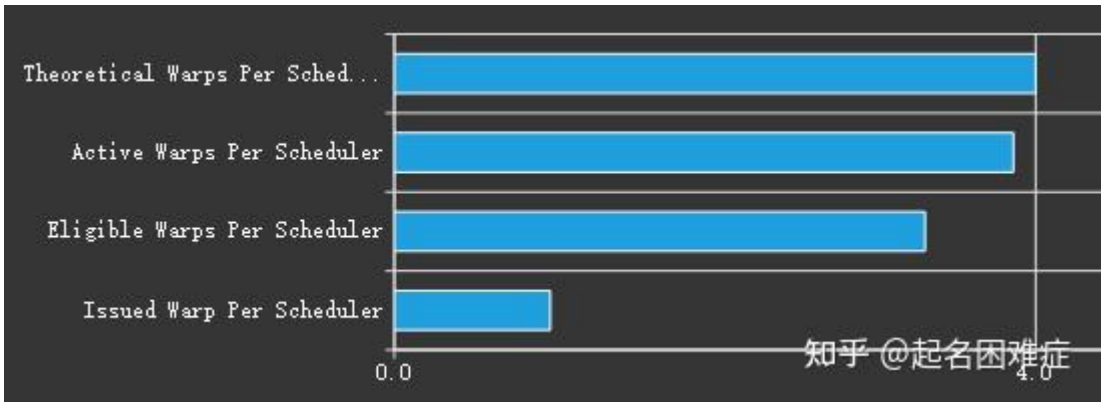


前文提到的global memory的合并访存，shared memory的bank conflict，以及kernel是否使用了local memory（比如做register spilling）等，都能通过这个section的指标分析出来。比如下图中对shared memory访存的统计：

	Instructions	Requests	% Peak	Bank Conflicts
Shared Load	2,097,152	4,194,304	34.83	0
Shared Store	327,680	786,432	6.53	262,144
Shared Atomic	0	-	-	-
Total	2,424,832	4,980,736	41.37	262,144

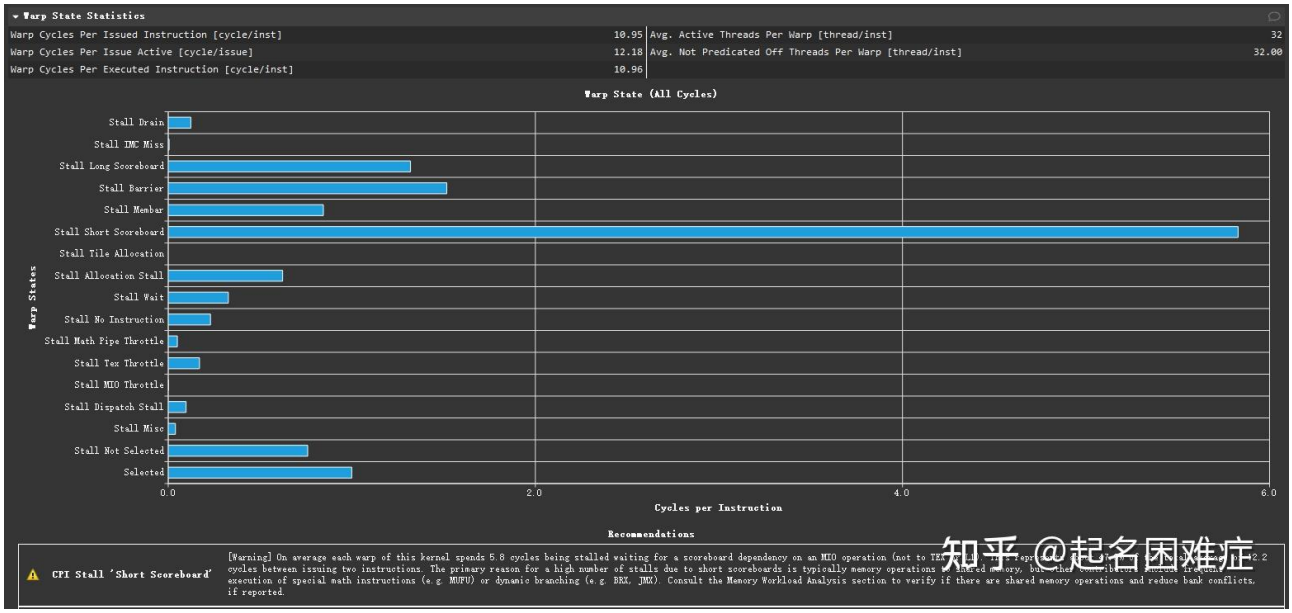
5.2 scheduler statistics

前文提到，一个warp只有在eligible的情况下才可能被warp scheduler选中发射指令。如果没有足够多的eligible warp，就难以让SM中的计算单元处于忙碌状态，从而无法充分的发挥性能。scheduler statistics section就统计了各个warp scheduler的active warp，eligible warp和issued warp的数量信息，如下图所示：



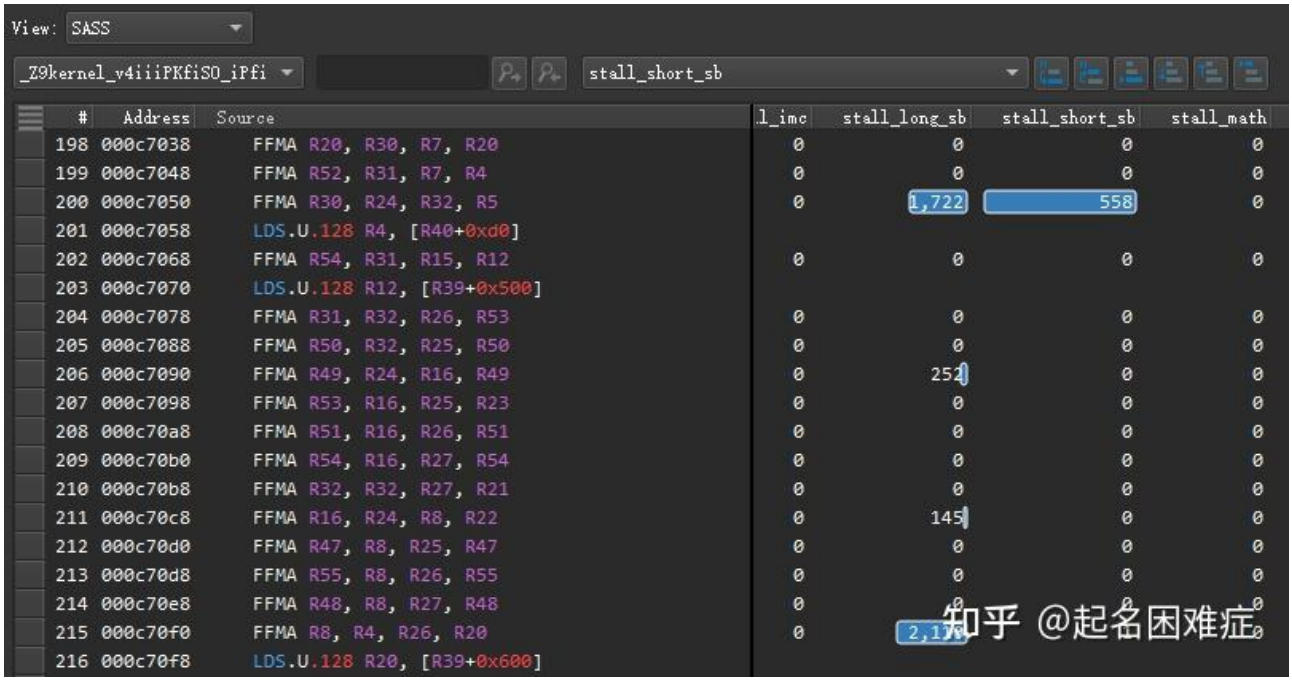
5.3 warp state statistics

前文提到，active的warp如果因为某些原因不具备执行条件就会stall。warp state statistics能够帮助分析warp stall的各种原因。如下图所示，可以看到Short Scoreboard Stall占据主导地位。根据nvidia的doc，short_sb_stall通常是由对shared memory的访问引起的[24]，这为我们指明了优化方向。



5.4 source page

除了各个section中的metrics之外，nsight compute还可以查看kernel的SASS，并且将metrics与kernel中具体的指令联系起来，对汇编级别的优化很有帮助。



关于nsight compute的使用就不过多介绍了，nvidia有相关的文档[25][26]。需要注意的是，在使用nsight的时候，需要开启管理员权限或者在nvidia控制面板中打开gpu性能计数器的使用权限[27]，否则profile report里面都是空的内容（一堆n/a和感叹号）。

6 conclusion

本文系统地介绍了在NVIDIA GPU上做性能优化所需要的基础知识，包括体系结构，影响性能的因素，常见优化方法，性能分析工具等等。但因为个人水平有限，并且篇幅限制，难免有疏漏。所以，本文后续可能还会更新、完善，并且可能会考虑拆分成多篇文章，欢迎关注！

参考