gpu 硬件架构

里程

英伟达不同时代产品的芯片设计不同，每代产品背后有一个微架构代号，微架构均以著名的物理学家为名，以向先贤致敬。当前比较火热的架构有：

Ampere 安培

2020 年 5 月发布 专业显卡：Telsa A100

Telsa 微架构：Tesla V100 共有 84 个 Streaming Multiprocessor 上图为 Tesla V100 的设计，它共有 84 个 SM。图中密密麻麻的绿色小格子就是 GPU 计算核心，多个计算核心一起组成了一个 SM。将 SM 放大，**单个 SM 如下图所示**

运算核心 （Core，也叫流处理器 Stream Processor）

LD/ST（load/store ）模块来加载和存储数据

SFU（Special function units ）执行特殊数学运算（ sin、cos、log 等 ）

寄存器（Register File）

L1 缓存

全局内存缓存

纹理读取单元

Warp Schedulers：这个模块负责 warp 调度，一个 warp 由 32 个线程组成，warp 调度器的指令通过 Dispatch Units 送到 Core 执行。

可以看到一个 SM 中包含了计算核心、存储等部分：

针对不同类型计算的小核心，包括 64 位浮点核心（FP64），整型核心 (INT)，32 位浮点核心 (FP32)，优化深度学习的 Tensor Core。

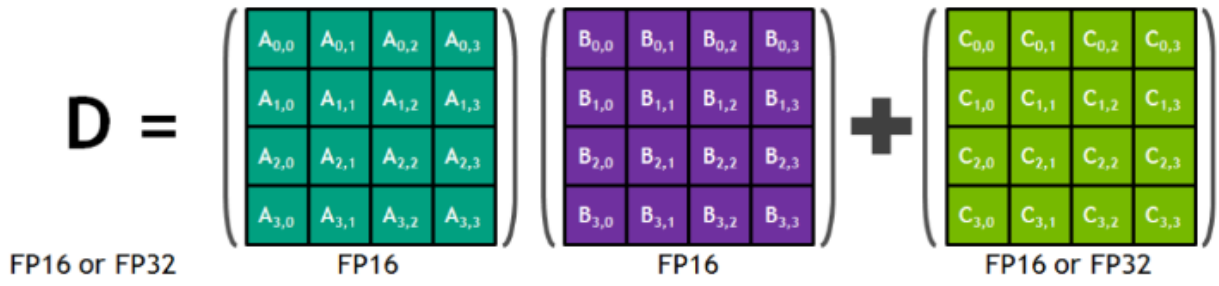
计算核心直接从寄存器（Register）中读写数据。

调度和分发器（Scheduler 和 Dispatch Unit）。

L0 和 L1 级缓存。

具体而言，SM 中的 FP32 进行 32 位浮点加乘运算，INT 进行整型加乘运算，SFU（Special Functional Unit）执行一些倒数和三角函数等运算。

这里对 Tensor Core 做一些简单解释。Tensor Core 是英伟达新的微架构中提出的一种混合精度的计算核心。我们知道，当前深度神经网络中使用到最频繁的矩阵运算是：$D = A \times B + C$。Tensor Core 可以对 4*4 的矩阵做上述运算。其中，涉及乘法的A 和B 使用 FP16 的 16 位浮点运算，精度较低；涉及加法的C 和D 使用 FP16 或 FP32 精度。Tensor Core 是在 Volta 架构开始提出的，使用 Volta 架构的 V100 在深度学习上的性能远超 Pascal 架构的 P100。



Tensor Core 是一种为优化深度学习计算核心

前面提到的以物理学家命名的名称是英伟达各代 GPU 的微架构代号，微架构表示英伟达不同时代的芯片设计。不同微架构里各类计算核心和显卡存储的设计不同。2020 年，比较流行的微架构为 Volta 和 Turing。

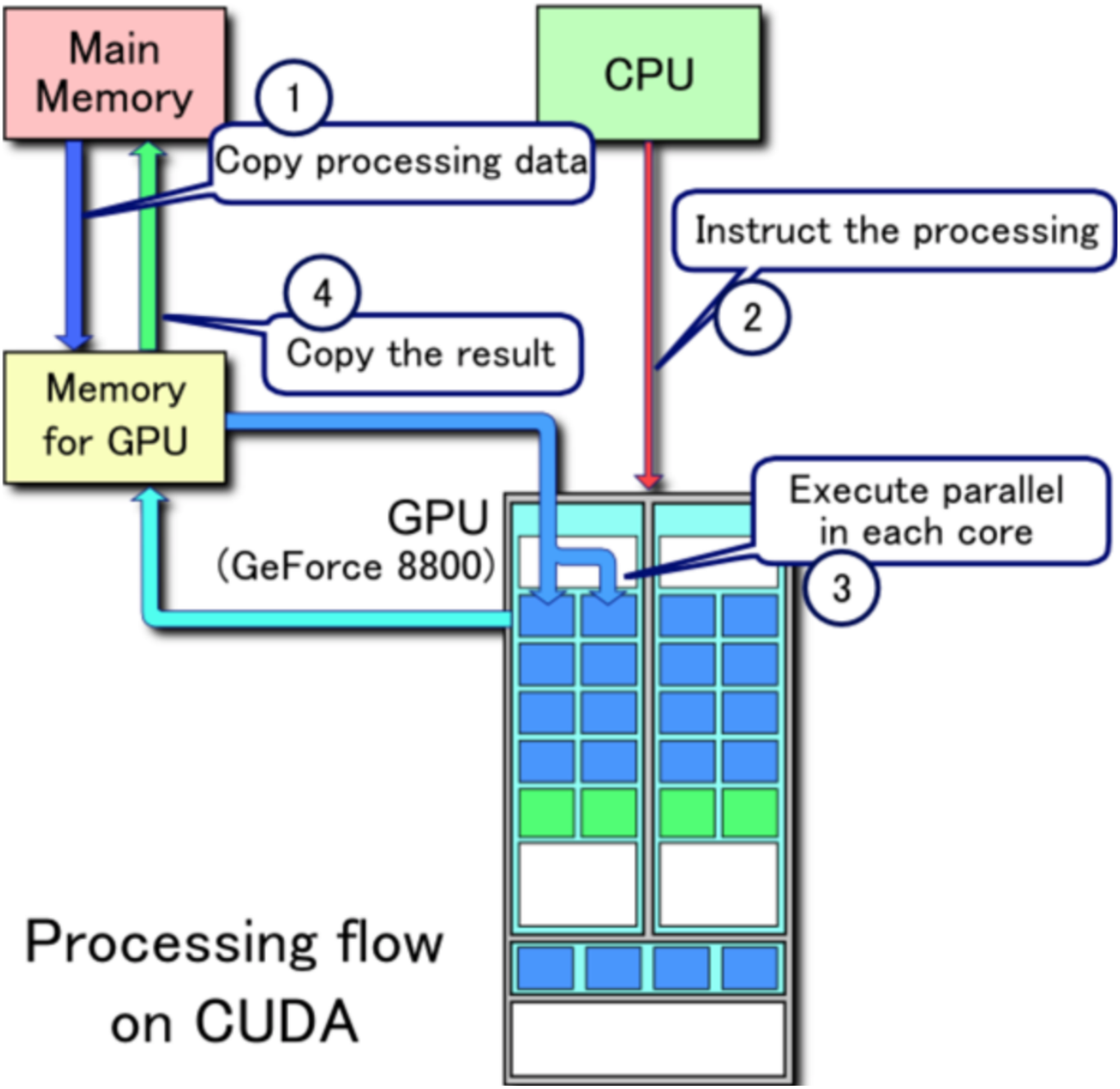
Turing 图灵

2018 年发布 消费显卡：GeForce RTX 2080 Ti、Titan RTX Volta 伏特

2017 年末发布 专业显卡：Telsa V100 (16 或 32GB 显存 5120 个 CUDA 核心) Pascal 帕斯卡

2016 年发布 专业显卡：Telsa P100 (12 或 16GB 显存 3584 个 CUDA 核心)

GPU 是如何与 CPU 协调工作的？



1.将主存的处理数据复制到显存中。

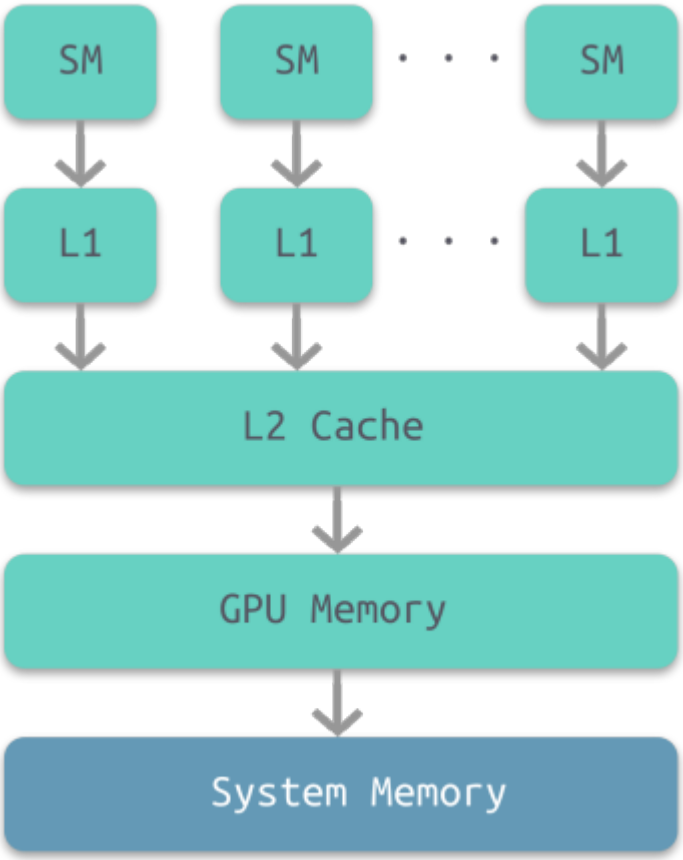
2.CPU 指令驱动 GPU。

3.GPU 中的每个运算单元并行处理。此步会从显存存取数据。

4.GPU 将显存结果传回主存。

GPU 也有缓存机制吗？有几层？它们的速度差异多少？

和 CPU 类似，也有多级缓存机构，同理也有 Cache Miss 的问题

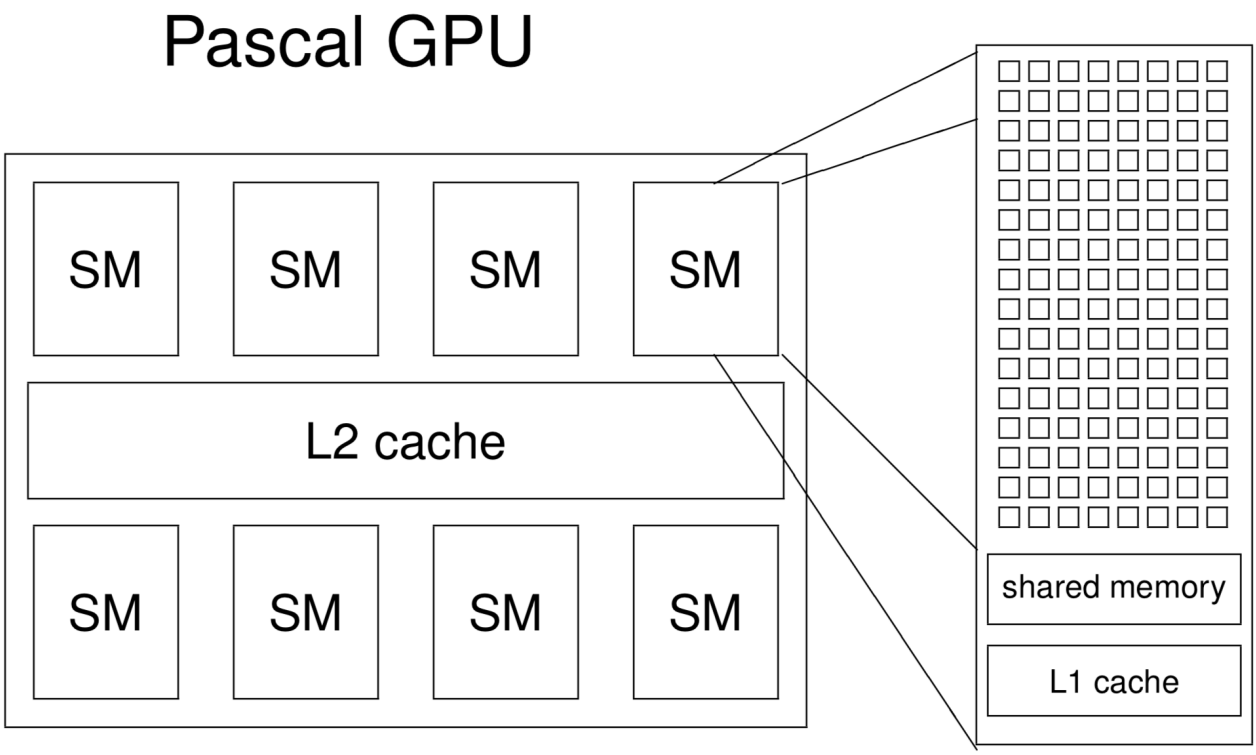


一般为这几层缓存：寄存器、L1 缓存、L2 缓存、GPU 显存、系统显存。

储存类型	寄存器	共享内存	L1 缓存	L2 缓存	纹理、常量缓存	全局内存
访问周期	1	1-32	1-32	32-64	400-600	400-600

英伟达 GPU 硬件架构

在英伟达的设计里，多个核心组成一个 Streaming Multiprocessor（SM），一张 GPU 卡有多个 SM。从 “Multiprocessor” 这个名字上也可以看出 SM 包含了多个处理器。实际上，英伟达主要以 SM 为运算和调度的基本单元。



Pascal 微架构示意图：一个 GPU 中有多个 SM，每个 SM 里有计算核心、Shared Memory 和 L1 Cache

GPU 技术要点

SIMD 和 SIMT

SIMD（Single Instruction Multiple Data）是单指令多数据，在 GPU 的 ALU 单元内，一条指令可以处理多维向量（一般是 4D）的数据。比如，有以下 shader 指令：

float4 c = a + b; // a, b都是float4类型 对于没有 SIMD 的处理单元，需要 4 条指令将 4 个 float 数值相加，汇编伪代码如下：

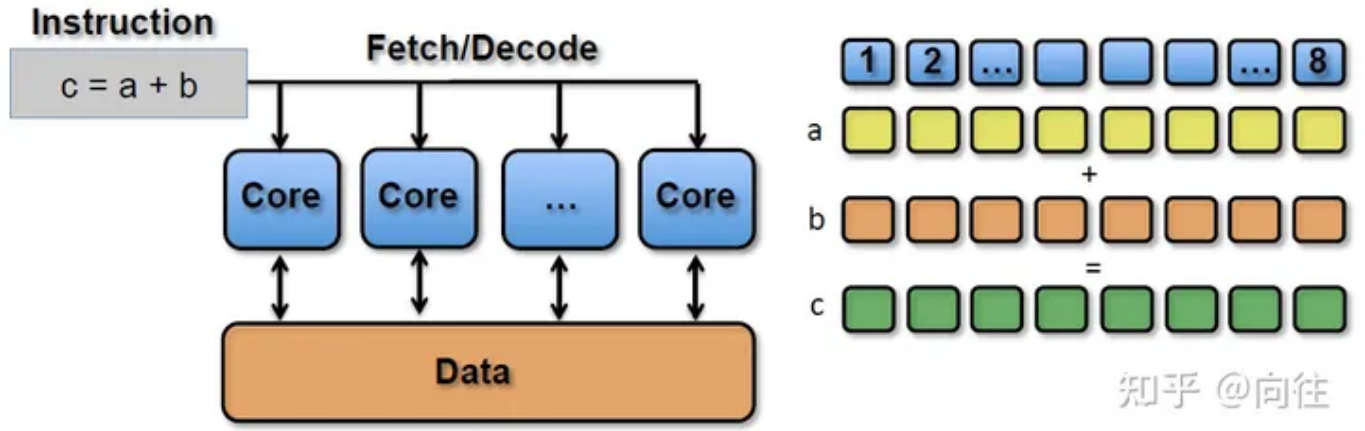
```
ADD c.x, a.x, b.x
ADD c.y, a.y, b.y
ADD c.z, a.z, b.z
ADD c.w, a.w, b.w
```

但有了 SIMD 技术，只需一条指令即可处理完：

```
SIMD_ADD c, a, b
```

SIMT（Single Instruction Multiple Threads，单指令多线程）是 SIMD 的升级版，可对 GPU 中单个 SM 中的多个 Core 同时处理同一指令，并且每个 Core 存取的数据可以是不同的

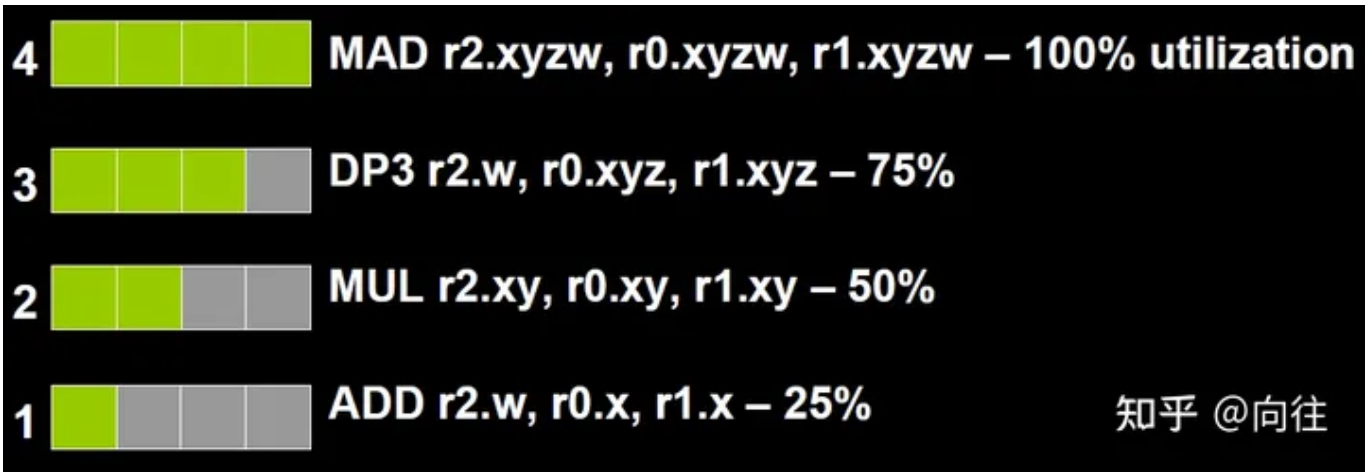
```
SIMT_ADD c, a, b
```

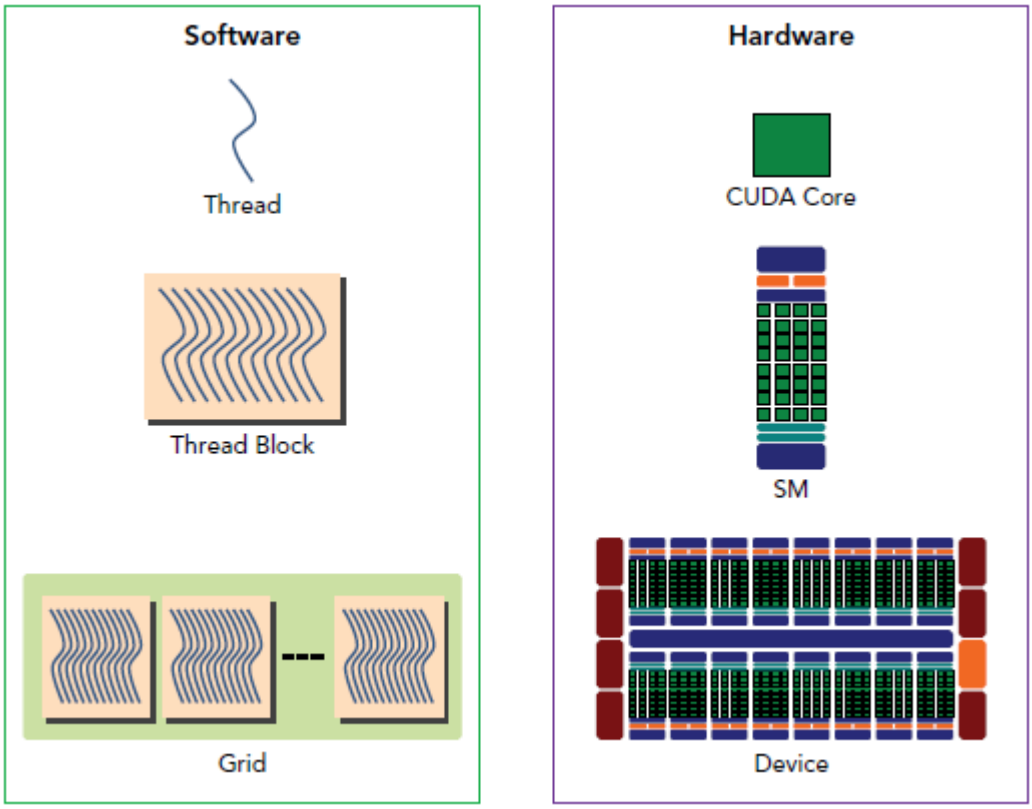


上述指令会被同时送入在单个 SM 中被编组的所有 Core 中，同时执行运算，但 a、b 、c 的值可以不一样：

co-issue

co-issue 是为了解决 SIMD 运算单元无法充分利用的问题。例如下图，由于 float 数量的不同，ALU 利用率从 100% 依次下降为 75%、50%、25%。





block 是软件概念，一个 block 只会由一个 sm 调度，程序员在开发时，通过设定 block 的属性，**告诉**GPU 硬件，我有多少个线程，线程怎么组织。而具体怎么调度由 sm 的 warps scheduler 负责，block 一旦被分配好 SM，该 block 就会一直驻留在该 SM 中，直到执行结束。一个 SM 可以同时拥有多个 blocks，但需要序列执行。下图显示了软件硬件方面的术语对应关系：

部分 threads 只是逻辑上并行，并不是所有的 thread 可以在物理上同时执行。例如，遇到分支语句（if else, while, for 等）时，各个 thread 的执行条件不一样必然产生分支执行，这就导致同一个 block 中的线程可能会有不同步调。另外，并行 thread 之间的共享数据会导致竞态：多个线程请求同一个数据会导致未定义行为。CUDA 提供了 cudaThreadSynchronize () 来同步同一个 block 的 thread 以保证在进行下一步处理之前，所有 thread 都到达某个时间点。　　同一个 warp 中的 thread 可以以任意顺序执行，active warps 被 sm 资源限制。当一个 warp 空闲时，SM 就可以调度驻留在该 SM 中另一个可用 warp。在并发的 warp 之间切换是没什么消耗的，因为硬件资源早就被分配到所有 thread 和 block，所以该新调度的 warp 的状态已经存储在 SM 中了。不同于 CPU，CPU 切换线程需要保存 / 读取线程上下文（register 内容），这是非常耗时的，而 GPU 为每个 threads 提供物理 register，无需保存 / 读取上下文。

CUDA 线程模型

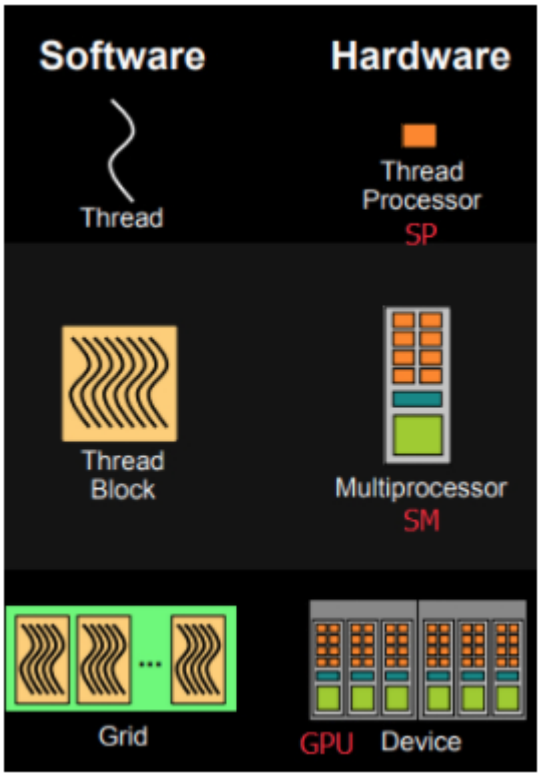
线程模型（软件角度） kernel 在 device 上执行时实际上是启动很多线程，一个 kernel 所启动的所有线程称为一个网格（grid），同一个网格上的线程共享相同的全局内存空间，grid是线程结构的第一层次，而 grid 又可以分为很多线程块（block），一个线程块里面包含很多线程，这是第二层次。

- Thread：线程，并行的基本单位；
- Thread Block：线程块，互相合作的线程组，线程块有如下几个特点：
  - 允许彼此同步；
  - 可以通过共享内存快速交换数据；
  - 可以以 1 维、2 维或 3 维组织（一般为 3 维）；
  - 每一个 block 和每个 thread 都有自己的 ID（blockIdx 和 threadIdx），我们通过相应的索引找到相应的线程块和线程。

- Grid：一组线程块 可以以 1 维、2 维或 3 维组织（一般为 2 维）；
- 共享全局内存； **grid 和 block 都是定义为 dim3 类型的变量，dim3 可以看成是包含三个无符号整数 (x, y, z) 成员的结构体变量，在定义时，缺省值初始化为 1；** dim3 仅为 host 端可见，其对应的 device 端类型为 uint3。**Warp：GPU 执行程序时调度和运行的单位（是 SM 的基本执行单元），目前 cuda 的 warp 的大小为 32；同在一个 warp 的线程，以不同数据资源执行相同的指令，这就是所谓 SIMT（Single Instruction Multiple Threads，单指令多线程）。** Kernel：在 GPU 上执行的核心程序，这个 kernel 函数是运行在某个 Grid 上的。 One kernel <==> One Grid

流处理器（硬件角度）

SP（streaming processor）：也称 CUDA core。最基本的处理单元，具体的指令和任务都是在 SP 上处理的。一个 SP 可以执行一个 thread



- SM（streaming multiprocessor）：也叫 GPU 大核，由**多个 SP** 加上其他资源（如：warp scheduler, register，shared memory 等）组成。block 在 SM 上执行。 SM 可以看做 GPU 的心脏（对比 CPU 核心），register 和 shared memory 是 SM 的稀缺资源。CUDA 将这些资源分配给所有驻留在 SM 中的 threads。因此，这些有限的资源就使每个 SM 中 active warps 有非常严格的限制，也就限制了并行能力； 每个 SM 包含的 SP 数量依据 GPU 架构而不同，Fermi 架构 GF100 是 32 个，GF10X 是 48 个，Kepler 架构都是 192 个，Maxwell 架构都是 128 个； SM 上并不是所有的 thread 能够在同一时刻执行。Nvidia 把 32 个 threads 组成一个 warp，warp 是调度和运行的基本单元。warp 中所有 threads 并行的执行相同的指令。一个 warp 需要占用一个 SM 运行，多个 warps 需要轮流进入 SM。由 SM 的硬件 warp scheduler 负责调度。目前每个 warp 包含 32 个 threads（Nvidia 保留修改数量的权利）。所以，一个 GPU 上 resident thread 最多只有 SM * warp 个。 block 一旦被分配好 SM，该 block 就会一直驻留在该 SM 中，直到执行结束。一个 SM 可以同时拥有多个 blocks，但需要序列执行。