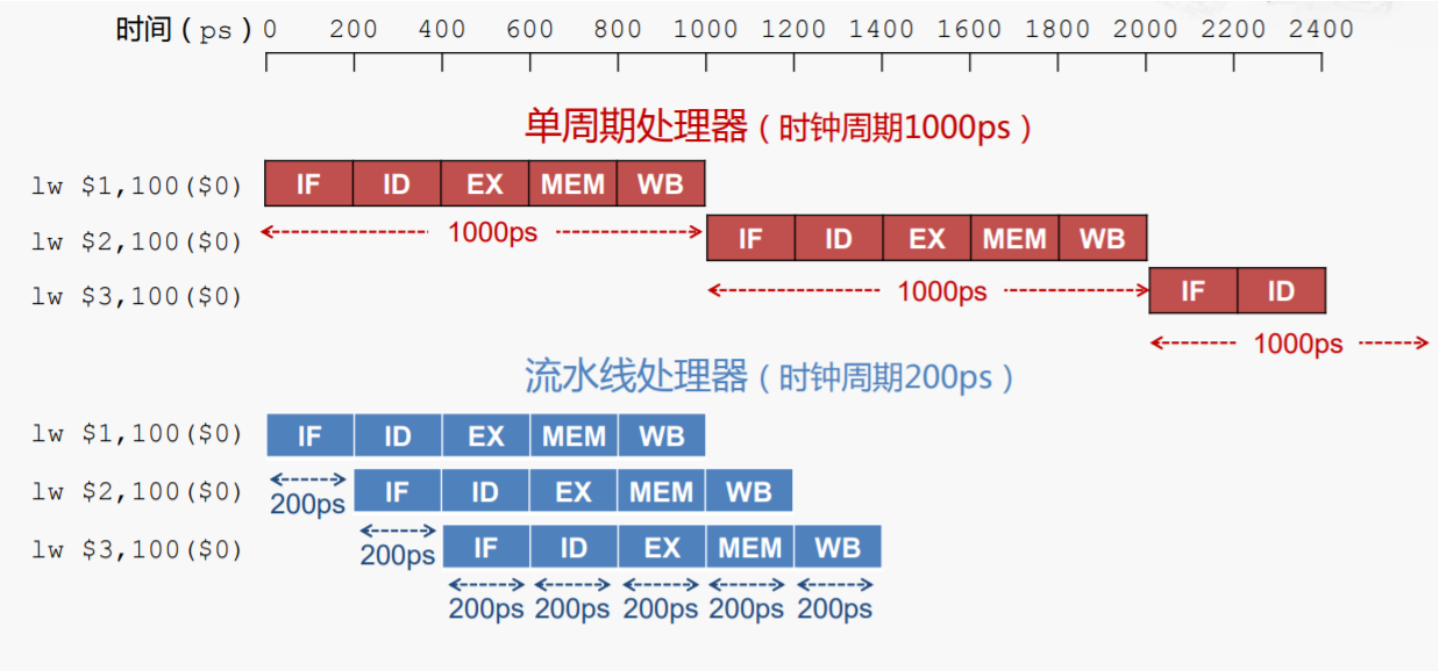
CPU

（1）流水线前传机制

在 CPU 设计中, 可以采用类似工厂中的流水线方式以提高性能。CPU 流水线技术是一种将指令分解为多步，并让不同指令的各步骤操作重叠， 从而实现几条指令并行处理，以加速程序运行过程的技术。指令的每步有各自独立的电 路来处理，每完成一步，就切换执行到下一步。在流水线中一条指令的生命周期分为五个过程：

1. 取指：指令取指（Instruction Fetch）是指将指令从存储器中读取出来的过程。
2. 译码：指令译码（Instruction Decode）是指将存储器中取出的指令进行翻译的过程。经过译码之 后得到指令需要的操作数寄存器索引，可以使用此索引从通用寄存器组（Register File）中将操作 数读出。
3. 执行：指令译码之后所需要进行的计算类型都已得知，并且已经从通用寄存器组中读取出了所需的 操作数，那么接下来便进行指令执行（Instruction Execute）。指令执行是指对指令进行真正运算 的过程。譬如，如果指令是一条加法运算指令，则对操作数进行加法操作；如果是减法运算指令， 则进行减法操作。在“执行”阶段的最常见部件为算术逻辑部件运算器（Arithmetic Logical Unit， ALU），作为实施具体运算的硬件功能单元。
4. 访存：存储器访问指令往往是指令集中最重要的指令类型之一，访存（Memory Access）是指存 储器访问指令将数据从存储器中读出，或者写入存储器的过程。
5. 写回：写回（Write-Back）是指将指令执行的结果写回通用寄存器组的过程。如果是普通运算指 令，该结果值来自于“执行”阶段计算的结果；如果是存储器读指令，该结果来自于“访存”阶段从存 储器中读取出来的数据



采用流水线技术后，并没有加速单条指令的执行，每条指令的操作步骤一个也不能 少，利用流水线的并行原理，多条指令的不同操作步骤同时执行，因而从总体上看加快了指令流速度，缩短 了程序执行时间。它增加了 四组寄存器，每一个流水线级数内部都有各自的组合逻辑数据通路，彼此之间没有复

用资源，因此，其 面积开销是比较大的，但是由于可以让不同的流水线级数同时做不同的事情，而达到流水的效果，提高 了性能，优化了时序，增加了吞吐率。

流水线使得原先有先后顺序的指令同时处理，当出现某些指令组合时，一个指令在执行的时候, 如果需要等待流水线上前一个指令先执行完毕的话, 那么这两个指令相互之间 彼此有依赖关系，可能会导致使用 了错误的数据。 流水线前传用来解决一种数据冲突， 即在进行后面的操作时需要前面操作的运算结果， 前面操作尚未完全完成， 运算结果并未写入内存中， 然而其实运算已经完成， 这时候就 可以通过数据前传让后面的操作可以直接得到前面操作的运算结果，而不需等待前面操 作完全完成。

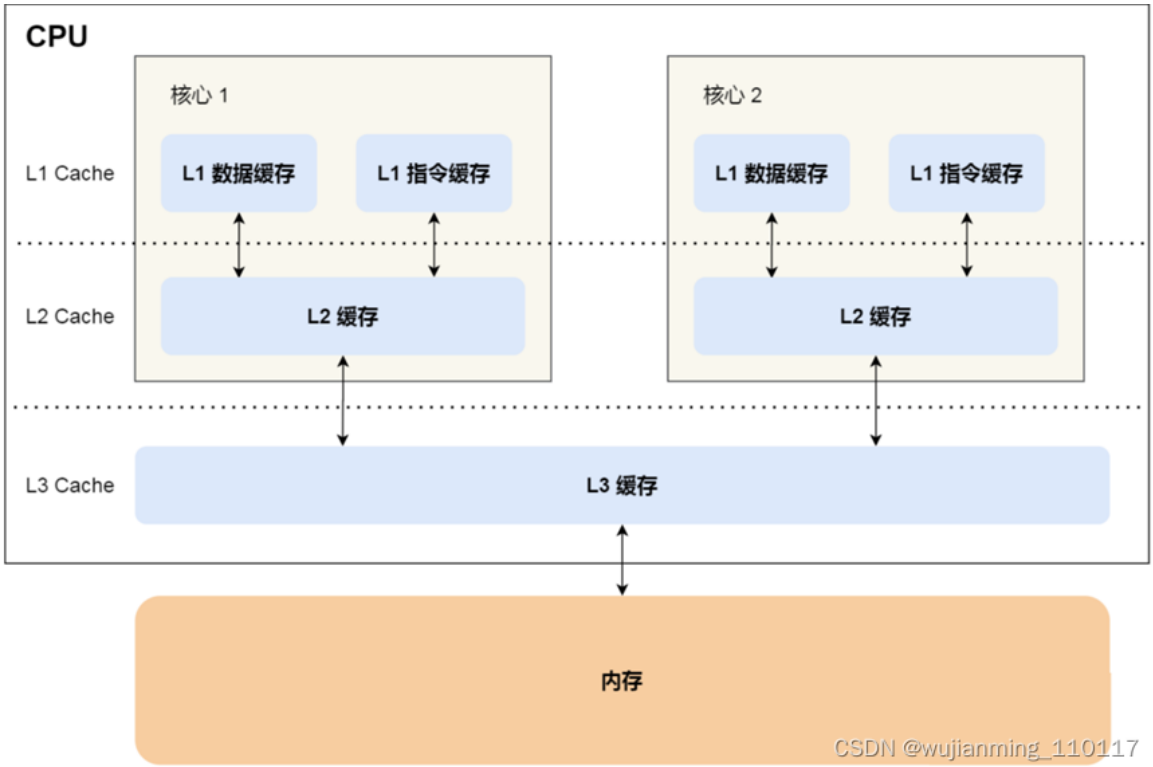
（2） CPU 的三级缓存

CPU缓存，英文叫Cache Memory，它是位于CPU和内存之间的临时存储器。CPU缓存的作用主要 是为了解决CPU运算速度与内存读写速度不匹配的矛盾，而缓存的容量要比内存要小的太多，但是其速度要比内存快的多。当CPU需要读取数据并进行计算时，首先需要将CPU缓存中查到所需的数据，并在最短的时间下交 付给CPU。如果没有查到所需的数据，CPU就会提出“要求”经过缓存从内存中读取，再原路返回至CPU进 行计算。而同时，把这个数据所在的数据也调入缓存，可以使得以后对整块数据的读取都从缓存中进 行，不必再调用内存。

根据数据读取顺序和与CPU结合的紧密程度，CPU缓存可以分为一级缓存，二级缓存，三级缓存， 每一级缓存中所储存的全部数据都是下一级缓存的一部分，这三种缓存的技术难度和制造成本是相对递减的，所以其容量也是相对递增的。

当CPU要读取一个数据时，首先从一级缓存中查找，如果没有找到 再从二级缓存中查找，如果还是没有就从三级缓存或内存中查找。如果找到了必要的数据，则称为缓存命中；如果缓存中不存在数据，则 CPU 必须请求将其从主内存或存储设备加载到缓存中。这需要时间，并且会对性能产生 不利影响。 一般来说，每级缓存的命中率大概都在80%左右，也就是说全部数据量的80%都可以在一级缓存中 找到，只剩下20%的总数据量才需要从二级缓存、三级缓存或内存中读取，由此可见一级缓存是整个 CPU缓存架构中最为重要的部分。

- 一级缓存（L1 Cache）：CPU一级缓存，就是指CPU的第一层级的高速缓存，主要的工作是缓存指令和缓存数据（ L1 指令缓存包含需要由 CPU 执行的 指令， 而且还保留预解码数据和分支信息； L1 数据缓存用于保存将被写回到主存储器的数据）。 一级缓存 的容量与结构对CPU性能影响十分大，但是由于它的结构比较复杂，又考虑到成本等因素，一般来说， CPU的一级缓存较小，通常CPU的一级缓存也就能做到256KB左右的水平。
- 二级缓存（L2 Cache）：CPU二级缓存，就是指CPU的第二层级的高速缓存，二级缓存用于存储 CPU 最先 读取的数据且 L1 缓存空间存储不下的数据。二级缓存的容量会直接影响到CPU的性能， 二级缓存的容量越大越好。 L2 缓存比 L1 大得多，但同时也慢一些。
- 三级缓存（L3 Cache）：CPU三级缓存，就是指CPU的第三层级的高速缓存，其作用是进一步降低内存的延迟，同时提升海 量数据量计算时的性能。和一级缓存、二级缓存不同的是，三级缓存是核心共享的，能够将容量做的很 大。L3 缓存是最低 级别的缓存。



越靠近 CPU 核心的缓存其访问速度越快

部件	CPU 访问所需时间
L1 Cache	2~4 个时钟周期
L2 Cache	10~20 个时钟周期
L3 Cache	20~60 个时钟周期
内存	200~300 个时钟周期

时钟周期是 CPU 主频的倒数，
比如 2GHZ 主频的 CPU，一个时钟周期是 0.5ns

（3） 什么样的问题适合 GPU

（a）大量的轻量级运算 ： 即用大量数据或者用同一数据多次调用同一公式或者计算过程，公式本身并不复杂， 只是执行的次数较多，这是 GPU 先天的优势。

（b）高度并行 的任务：高度并行指的就是各个数据之间运算互不影响，即耦合度较低。由于 GPU 本身硬 件基础决定，各个 workgroup 之间并不相互通信，只有同一 workgroup 内的 work-item 之 间才相互通信，所以 GPU 本身并不支持迭代等数据耦合度较高的计算，这是 GPU 本身要求。

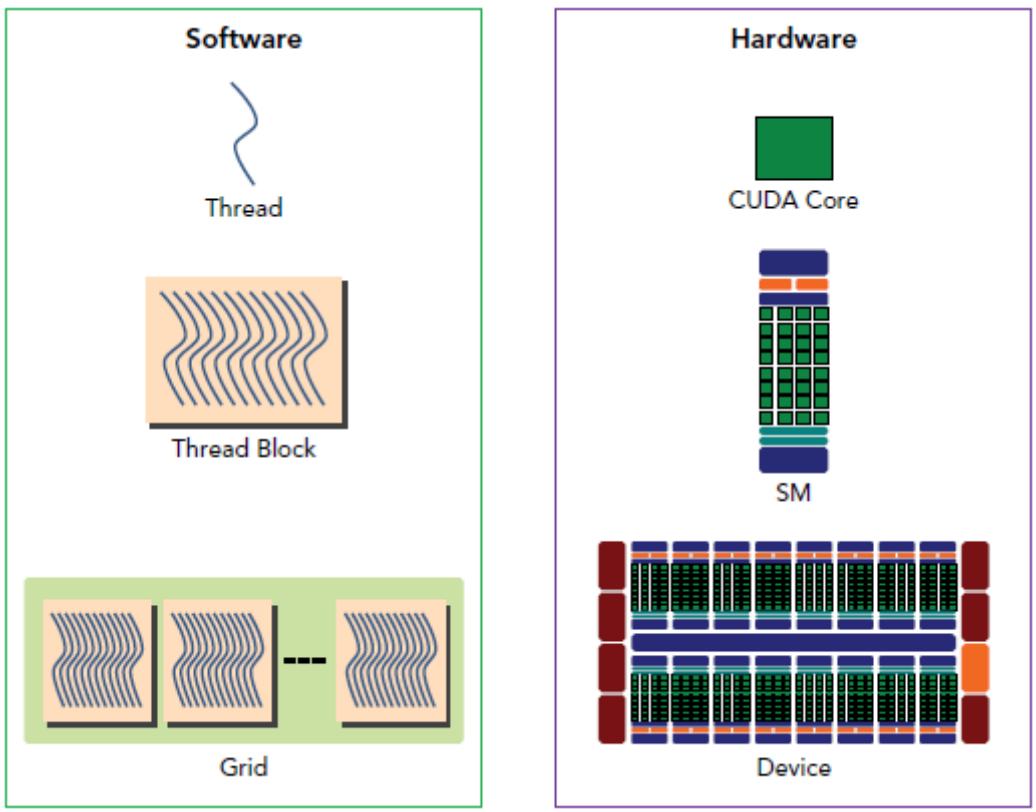
（c）计算密集型任务 ： 任务可以分为计算密集型和 IO 密集型。计算密集型，即少量的 IO 读取+大量的计 算，消耗 CPU 资源较多；而 IO 密集型，是指多次使用 IO 读取+少量计算，这种情况涉 及到寄存器与内存之间以及与设备内存之间的通信问题，主要限制原因是显存带宽问题。

（d）控制简单 的任务：对比 GPU 来说， CPU 更擅长判断、逻辑控制、分支等，有通用计算能力，并含有 强大的 ALU（算术运算单元）；而 GPU 更适合于逻辑简单的运算。
（e）多个阶段执行：运算程序可分解为多个小程序或者同一程序可分多个阶段执行，这就类似于使用集 群处理同一任务，将其分解为多个任务碎片分发到各节点执行，以提高运算速率。
（f）浮点型运算：GPU 擅长浮点型运算。

常见的具有这种性质的问题： 计算机图形学（图形学中的光线跟踪算法，非常适合GPU并行处理大量光线的 跟踪和绘制，使用光路追踪计算每个像素的颜色，并行性 高），计算机视觉（传统视觉中对每个像素、每个区域的处理比较容易并行），深度学习（大 量矩阵运算，适合并行），或者数据传输量相对较小， 计算相对简单（加减乘除更多， 三角函数、指数对数更少），计算量 大， 可并行性高的问题适合使用GPU计算。

GPU

（1）线程束 warp



硬件概念：
● CUDA core：也就是stream processor(SP)，是GPU最基本的处理单元。具体指令和任务都是在SP上处理的，GPU并行计算也就是很多SP同时处理。一个SP可以执行一个thread，但是实际上并不是所有的thread能够在同一时刻执行。

● SM：是stream multiprocessor，SM包含SP和一些其他资源，一个SM可以包含多个 SP。SM可以看做GPU的 核心。GPU中每个SM都设计成支持数以百计的线程并行执行，并且每个GPU都包含了很多的 SM，所以GPU支持成百上千的线程并行执行。

软件概念：（CUDA编程上的概念，以方便程序员软件设计，组 织线程）
● Thread：一个线程 grid
● Block：多个线程组成一个block
● Grid：多个block组成yi'ge
● Warp：线程束，通常一个warp包含32个线程
从软件角度来看， 启动内核时， 内核中似乎所有的线程都是并行的运行的，block是线 程的集合， 它们可以被组织为一维、 二维或三维布局。
从硬件角度看， 不是所有线程在物理上都可以同时并行的执行， 线程块是一堆线程的集 合， 在线程块中的线程被布局成 一维， 每 32 个连续线程组成一个线程束（ 32 这个数字是硬件端已经设计 好的）。
所以线程在软件端可以被布局成一维， 二维和三维。 通过计算线程 ID 映 射到硬件上的物理单元上。硬件端是 32 个线程组成的线程束是一个基本 执行单元。 所以线程束是执行核函数的最基本的单元。任务分配到 Block 之后， 理想状态下其所有的线程都要并行执行，在逻辑上这是正确的,但从硬件的角度来看,不是所有线程在物理硬件上都可以同时并行地执行，不可能对 一个任意大小的 Block 都给出一个同等大小的 CUDA 核心阵列去推动它的并行计算。因而有 了 warp 这个概念。 Block 被划分成一块块的 warp 分别映射到 CUDA 核心阵列上执行， 为的 是线程数量固定可以给他分配统一的硬件资源。 并且 warp 中的 32 个 SP 是一起工作的，执 行相同的指令。
虽然 sp 为硬件最小计算单元，但是由于并行和硬件结构的关系使得 GPU 采取了 warp 线程束作为调度和运行的最小执行单元，目前Nvidia把32个threads组成一个warp（Nvidia 保留修改数量的权利，程序员是透明的，无法修改）。warp 取 32 的原因：SM 最主要的执行资源是 8 个 32bit ALU 和 MAD（multiplyadd units，乘加器）。它们能够对符合 IEEE 标准的单精度浮点数（对应 float 型）和 32-bit 整数（对应 int 型，或者 unsigned int 型）进行运算。每次运算需要 4 个 时钟周期（SP 周期，并非核心周期）。因为使用了四级流水线，因此在每个时钟 周期，ALU 或 MAD 都能取出一个 warp 的 32 个线程中的 8 个操作数，在随后 的 3 个时钟周期内进行运算并写回结果。也就是 SM 每发射一条指令，8 个 SP 将各执行 4 遍。因此由 32 个线程组成的线程束（warp）是 Tesla 架构的最小执行 单位。

当一个 kernel 被执行时， grid 中的block被分配到 SM 上执行， 一旦block被调度到一个 SM 上，线程块中的线程会被进一步划分为线程束， 一 个线程束由 32 个连续的线程组成。同一个block的 thread 只能在同一个 SM 中并行执行， SM一般可以调度多个block。 在一个线程束中， 所有的线程按照单指令多线程(Single Instruction Multiple Thread，SIMT) 方式执行。 也就是说，所有线程都执行相同的指令， 每个线程在私有数据上进行操作（每个 thread 拥有它自己的程序计数器和状态寄存器， 并且用该线程自己的数据执行指令）。

GPU 在线程调度的时候， 一个warp需要占用一个SM运行，多个warps会由SM的硬件warp scheduler负责调度，每次 选择一个线程束分配到 SM 上。1 个 SM 中的 Warp Scheduler 的个数和 SP 的个数决定了到底 1 个 SM 可以同 时运行几个 warp。 尽管warp中的线程从同一 程序地址，但可能具有不同的行为，比如分支结构，因为GPU规定warp中所有线程在同 一周期执行相同的指令，warp发散会导致性能下降。一个SM同时并发的warp是有限 的，因为资源限制，SM要为每个线程块分配共享内存，而也要为每个线程束中的线程 分配独立的寄存器，所以SM的配置会影响其所支持的线程块和warp并发数量。。

当我们 定义的一个 block 中含有的线程数量不是 32 的整数倍时（当软件层面的线程块大小不是硬件层面线程束大小的整数倍时），那么该 block 会被分 配给 n+1 个线程束， 其中 n 表示的是 线程数量与 32 的整数倍，多出来的线 程束会有部分线程不活跃，但因为线程束是基本执行单元，所以这个额外的线程束 依旧会消耗和其他线程束相同的资源。如果没有这么多 thread 需要工作，那么这个 warp 中的一些 thread(sp)是不工作的。也就是当进行并行计算时，线程数尽量为 32 的倍数，如 果线程数不上 32 的倍数的话；假如是 1，则 warp 会生成一个掩码，当一个指令 控制器对一个 warp 单位的线程发送指令时，32 个线程中只有一个线程在真正执 行，其他 31 个进程会进入静默状态。故 真正决定核函数执行速度的不是线程块， 而是线程束。一个线程块的线程束的数量可以根据下式确定:

$$\text{一个线程块中线程束的数量} = \text{向正无穷取整}\left(\frac{\text{一个线程块中线程的数量}}{\text{线程束大小}}\right)$$

GPU在执行时，会将threads以block为单位进行划分，把一个个block分配给SM。SM负责处理分 配到其上的所有block，但是一个时刻SM只会处理一个block。
GPU 运算调度过程是：
1）首先，CUDA 运行时系统将该核函数任务指定到 当前的 GPU 设备上，即将 Grid 分配到一个 Device 上；
2）然后，根据<<<>>>内 的第一个参数，确定要调度多少个 Block，将各个 Block 分配到各个 SM 上；
3）当 SM 收到一个 Block 任务后，会根据 <<<>>>内的第二个参数，告诉 Warp Scheduler 要调度多少个 Thread。为了提高 执行效率，Warp Scheduler 会将这些 Thread 分组，每 32 个 Thread 为一组称为一 个 Warp（线程束），不足 32 的情况会补全，只是补充的线程不会产生真实影响。 最后，每个 Warp 会被分配到 32 个 Core 上运行。

Warp的运行机制是：首先让所有的线程提取线程标号，计算得到数组地址，然后发出一条内存获取的指令，接着下一条指令是做执行，但这必须是在从内存读取完所有的数据之后。由于读取内存的时间很长，因此线各会挂起。当这组中的32个线程全部挂起，硬件就会切换到另一个线程束。在中我 们可以看到，当线程束0由于内存读取操作而挂起时，线程束就成为了正在执行的线程束。GPU上直 以此种方式运行直到所有的线程束到成为挂起状态。例如一个核函数定义了 128 个线程， 那么就会划分

为 4 个 warps。 只有被选择到的线程有资格访问运算单元， 其余的线程都将处于挂起状态， 等待访存延迟或者在就绪队列中等待。 线程束是执行核函数的最基本单元是因为在代码运行的时候， 同一个 warp 内的 32 个线程都会执行相同的指令。

线程束warp在软硬件端的执行过程：

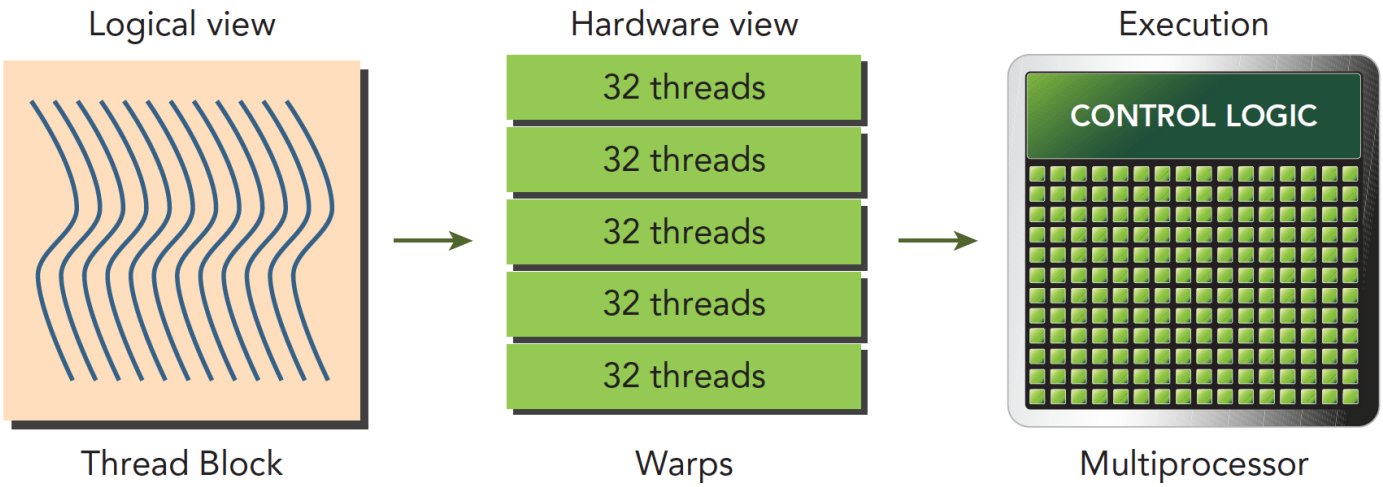
a) 一个网格被启动（网格被启动，等价于一个内核被启动，每个内核对应于自己 的网格），网格中包含线程块；

b) 线程块被分配到某一个 SM 上；

c) SM 上的线程块将分为多个线程束，每个线程束一般是 32 个线程；

d) 在一个线程束中，所有线程按照单指令多线程 SIMT 的方式执行，每一步执行 相同的指令，但是处理的数据是私有数据。

下图展示了线程块的逻辑视图和硬件视图之间的关系：



（2）线程 ID计算

CUDA的软件架构由网格（Grid）、线程块（Block）和线程（Thread）组成，相当于把GPU上的计算单元分为若干（2~3）个网格，每个网格内包含若干（65535）个线程块，每个线程块包含若干（512）个线程。

Thread，block，grid是CUDA编程上的概念，为了方便程序员软件设计，组织线程。

- thread：一个CUDA的并行程序会被以许多个threads来执行。
- block：数个threads会被群组成一个block，同一个block中的threads可以同步，也可以通过shared memory通信。
- grid：多个blocks则会再构成grid。

一个Grid可以包含多个Blocks，Blocks的组织方式可以是一维的，二维或者三维的。block包含多个Threads，这些Threads的组织方式也可以是一维，二维或者三维的。CUDA中每一个线程都有一个唯一的标识ID—ThreadIdx，这个ID随着Grid和Block的划分方式的不同而变化，这里给出Grid和Block不同划分方式下线程索引ID的计算公式。

1、 grid划分成1维，block划分为1维

```
int threadId = blockIdx.x *blockDim.x + threadIdx.x;
```

2、 grid划分成1维，block划分2维

```
int threadId = blockIdx.x * blockDim.x * blockDim.y+ threadIdx.y * blockDim.x +
threadIdx.x;
```

3、 grid划分成1维，block划分为3维

```
int threadId = blockIdx.x * blockDim.x * blockDim.y * blockDim.z
          + threadIdx.z * blockDim.y * blockDim.x
          + threadIdx.y * blockDim.x + threadIdx.x;
```

4、 grid划分成2维，block划分为1维

```
int blockId = blockIdx.y * gridDim.x + blockIdx.x;
int threadId = blockId * blockDim.x + threadIdx.x;
```

5、 grid划分成2维，block划分为2维

```
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y)
           + (threadIdx.y * blockDim.x) + threadIdx.x;
```

6、 grid划分成2维，block划分为3维

```
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
         + (threadIdx.z * (blockDim.x * blockDim.y))
         + (threadIdx.y * blockDim.x) + threadIdx.x;
```

7、 grid划分成3维，block划分为1维

```
int blockId = blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;

int threadId = blockId * blockDim.x + threadIdx.x;
```

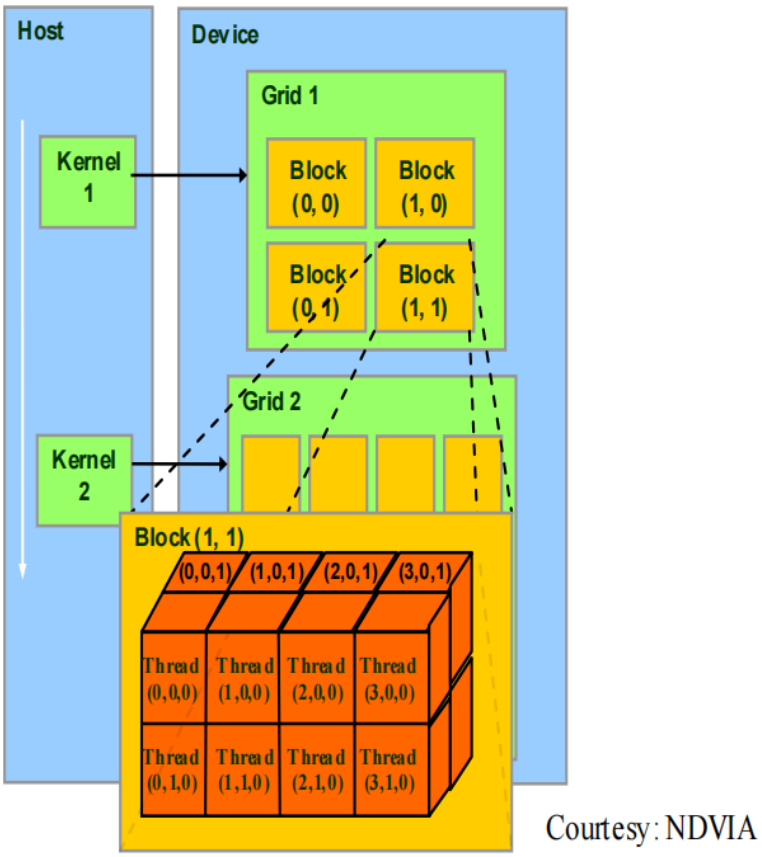
8、 grid划分成3维，block划分为2维

```
int blockId = blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;

int threadId = blockId*(blockDim.x*blockDim.y) + (threadIdx.y*blockDim.x) +
threadIdx.x;
```

9、 grid划分成3维，block划分为3维

```
int blockId = blockIdx.x + blockIdx.y * gridDim.x
          + gridDim.x * gridDim.y * blockIdx.z;
int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
          + (threadIdx.z * (blockDim.x * blockDim.y))
          + (threadIdx.y * blockDim.x) + threadIdx.x;
```



图中的 grid 划分成 2 维， block 划分为 3 维。图中 thread(3,0,0)的 threadId 为：

```
Blockid = blockIdx.x + blockIdx.y * gridDim.x = 3
Threadid = BlockId * (blockDim.x * blockDim.y * blockDim.z)
    + threadIdx.z * (blockDim.x * blockDim.y)
    + threadIdx.y * blockDim.x + threadIdx.x
    = 3*(4*2*2)+0+0+3 = 51
```