1. Tesla GPU架构分析

## ****GPU的功能****

现代GPU除了绘制图形外，还担当了很多额外的功能，综合起来如下几方面：

图形绘制：这是GPU最传统的拿手好戏，也是最基础、最核心的功能。为大多数PC桌面、移动设备、图形工作站提供图形处理和绘制功能。

物理模拟：GPU硬件集成的物理引擎（PhysX、Havok），为游戏、电影、教育、科学模拟等领域提供了成百上千倍性能的物理模拟，使得以前需要长时间计算的物理模拟得以实时呈现。

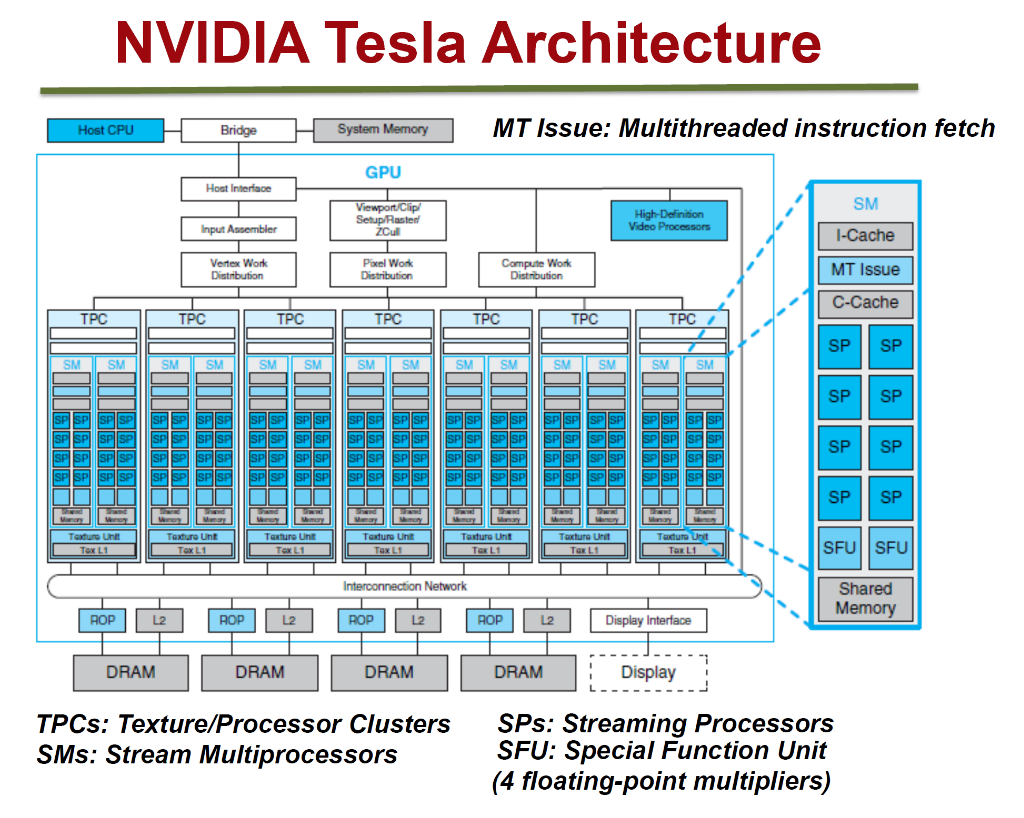
海量计算：计算着色器及流输出的出现，为各种可以并行计算的海量需求得以实现，CUDA就是最好的例证。

AI运算：近年来，人工智能的崛起推动了GPU集成了AI Core运算单元，反哺AI运算能力的提升，给各行各业带来了计算能力的提升。

其它计算：音视频编解码、加解密、科学计算、离线渲染等等都离不开现代GPU的并行计算能力和海量吞吐能力。

GPU的微观结构因不同厂商、不同架构都会有所差异，但核心部件、概念、以及运行机制大同小异。

### ****Tesla GPU****微观架构



Tesla的特性和概念：

拥有7组TPC（Texture/Processor Cluster，纹理处理簇）

每个TPC有两组SM（Stream Multiprocessor，流多处理器）

每个SM包含：

6个SP（Streaming Processor，流处理器）

2个SFU（Special Function Unit，特殊函数单元）

L1缓存、MT Issue（多线程指令获取）、C-Cache（常量缓存）、共享内存

除了TPC核心单元，还有与显存、CPU、系统内存交互的各种部件。

# ****GPU运行机制****

现代GPU有着相似的结构，有很多相同的部件，在运行机制上，也有很多共同点。下面是Tesla架构的运行机制总览图：

从Tesla开始NVIDIA使用类似的原理架构，使用一个Giga Thread Engine来管理所有正在进行的工作，GPU被划分成多个GPCs(Graphics Processing Cluster)，每个GPC拥有多个SM（SMX、SMM）和一个光栅化引擎(Raster Engine)，它们其中有很多的连接，最显著的是Crossbar，它可以连接GPCs和其它功能性模块（例如ROP或其他子系统）。

程序员编写的shader是在SM上完成的。每个SM包含许多为线程执行数学运算的Core（核心）。例如，一个线程可以是顶点或像素着色器调用。这些Core和其它单元由Warp Scheduler驱动，Warp Scheduler管理一组32个线程作为Warp（线程束）并将要执行的指令移交给Dispatch Units。

GPU中实际有多少这些单元（每个GPC有多少个SM，多少个GPC ......）取决于芯片配置本身。例如，GM204有4个GPC，每个GPC有4个SM，但Tegra X1有1个GPC和2个SM，它们均采用Maxwell设计。SM设计本身（内核数量，指令单位，调度程序......）也随着时间的推移而发生变化，并帮助使芯片变得如此高效，可以从高端台式机扩展到笔记本电脑移动。



如图，对于某些GPU的单个SM，包含：

32个运算核心 （Core，也叫流处理器）

16个LD/ST模块来加载和存储数据

4个SFU执行特殊数学运算（sin、cos、log等）

128KB寄存器

64KB L1缓存

全局内存缓存

纹理读取单元

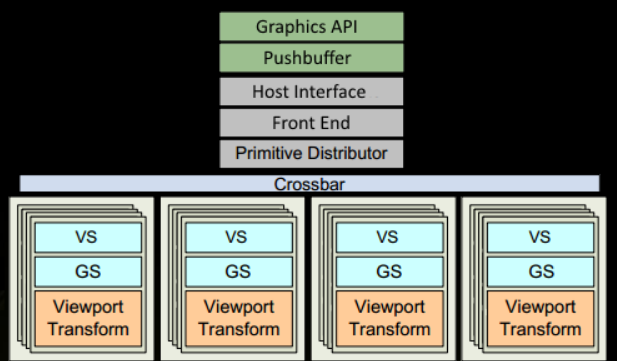
纹理缓存

PolyMorph Engine：多边形引擎负责属性装配、顶点拉取、曲面细分、栅格化（这个模块可以理解专门处理顶点相关的东西）。

2个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。

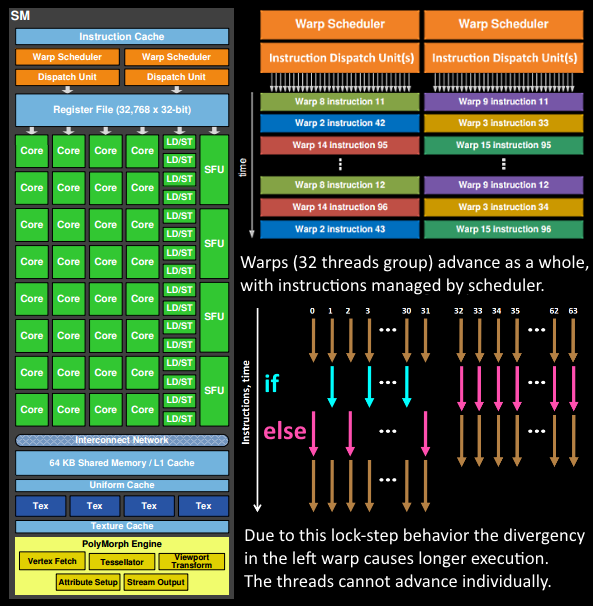
指令缓存，内部链接网络

## ****GPU逻辑管线****



1、程序通过图形API(DX、GL、WEBGL)发出drawcall指令，指令会被推送到驱动程序，驱动会检查指令的合法性，然后会把指令放到GPU可以读取的Pushbuffer中。

2、经过一段时间或者显式调用flush指令后，驱动程序把Pushbuffer的内容发送给GPU，GPU通过主机接口接受这些命令，并通过前端处理这些命令。



3、在图元分配器中开始工作分配，处理indexbuffer中的顶点产生三角形分成批次，然后发送给多个PGCs。这一步的理解就是提交上来n个三角形，分配给这几个PGC同时处理。

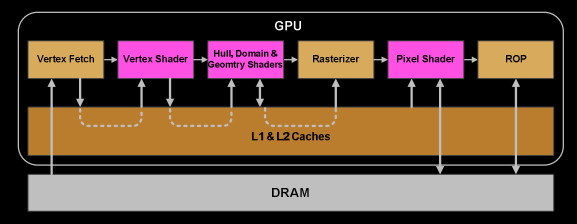
4、在GPC中，每个SM中的Poly Morph Engine负责通过三角形索引取出三角形的数据，即图中的Vertex Fetch模块。

5、在获取数据之后，在SM中以32个线程为一组的线程束来调度，来开始处理顶点数据。Warp是典型的单指令多线程（SIMT，SIMD单指令多数据的升级）的实现，也就是32个线程同时执行的指令是一模一样的，只是线程数据不一样，这样的好处就是一个warp只需要一个套逻辑对指令进行解码和执行就可以了，芯片可以做的更小更快，之所以可以这么做是由于GPU需要处理的任务是天然并行的。

6、SM的warp调度器会按照顺序分发指令给整个warp，单个warp中的线程会锁步执行各自的指令，如果线程碰到不激活执行的情况也会被遮掩。被遮掩的原因有很多，例如当前的指令是if(true)的分支，但是当前线程的数据的条件是false，或者循环的次数不一样（比如for循环次数n不是常量，或被break提前终止了但是别的还在走），因此在shader中的分支会显著增加时间消耗，在一个warp中的分支除非32个线程都走到if或者else里面，否则相当于所有的分支都走了一遍，线程不能独立执行指令而是以warp为单位，而这些warp之间才是独立的。

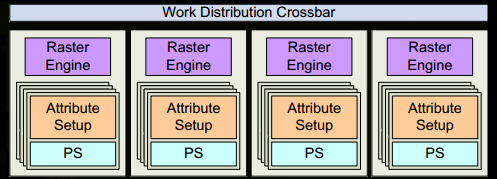
7、warp中的指令可以被一次完成，也可能经过多次调度，例如通常SM中的LD/ST(加载存取)单元数量明显少于基础数学操作单元。

8、由于某些指令比其他指令需要更长的时间才能完成，特别是内存加载，warp调度器可能会简单地切换到另一个没有内存等待的warp，这是GPU如何克服内存读取延迟的关键，只是简单地切换活动线程组。为了使这种切换非常快，调度器管理的所有warp在寄存器文件中都有自己的寄存器。这里就会有个矛盾产生，shader需要越多的寄存器，就会给warp留下越少的空间，就会产生越少的warp，这时候在碰到内存延迟的时候就会只是等待，而没有可以运行的warp可以切换。



9、一旦warp完成了vertex-shader的所有指令，运算结果会被Viewport Transform模块处理，三角形会被裁剪然后准备栅格化，GPU会使用L1和L2缓存来进行vertex-shader和pixel-shader的数据通信。

10、接下来这些三角形将被分割，再分配给多个GPC，三角形的范围决定着它将被分配到哪个光栅引擎，每个raster engines覆盖了多个屏幕上的tile，这等于把三角形的渲染分配到多个tile上面。也就是像素阶段就把按三角形划分变成了按显示的像素划分了。

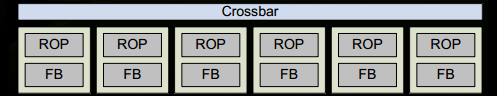


11、SM上的Attribute Setup保证了从vertex-shader来的数据经过插值后是pixel-shade是可读的。

12、GPC上的光栅引擎在它接收到的三角形上工作，来负责这些这些三角形的像素信息的生成。

13、32个像素线程将被分成一组，或者说8个2x2的像素块，这是在像素着色器上面的最小工作单元，在这个像素线程内，如果没有被三角形覆盖就会被遮掩，SM中的warp调度器会管理像素着色器的任务。

1. 接下来的阶段就和vertex-shader中的逻辑步骤完全一样，但是变成了在像素着色器线程中执行。 由于不耗费任何性能可以获取一个像素内的值，导致锁步执行非常便利，所有的线程可以保证所有的指令可以在同一点。



15、最后一步，现在像素着色器已经完成了颜色的计算还有深度值的计算，在这个点上，我们必须考虑三角形的原始api顺序，然后才将数据移交给ROP，一个ROP内部有很多ROP单元，在ROP单元中处理深度测试，和framebuffer的混合，深度和颜色的设置必须是原子操作，否则两个不同的三角形在同一个像素点就会有冲突和错误。

## ****GPU资源机制****

### 内存架构：部分架构的GPU与CPU类似，也有多级缓存结构：寄存器、L1缓存、L2缓存、GPU显存、系统显存。

### 它们的存取速度从寄存器到系统内存依次变慢：

### 

### 由此可见，shader直接访问寄存器、L1、L2缓存还是比较快的，但访问纹理、常量缓存和全局内存非常慢，会造成很高的延迟。