# Tesla GPU 架构分析

## 一、GPU 概述

GPU 缩写为 Graphics Processing Unit 的，一般称为视觉处理单元。

GPU 被广泛用于嵌入式系统、移动电话、个人电脑、工作站和电子游戏解决方案当中。

现代的 GPU 对图像和图形处理是十分高效率的，这是因为 GPU 被设计为很高的并行架构这样使得比通用处理器 CPU 在大的数据块并行处理算法上更具有优势。

NVIDIA 公司在 1999 年发布 GeForce 256 图形处理芯片时首先提出 GPU 的概念。从此 NVIDIA 显卡的芯片就用这个新名字 GPU 来称呼。

GPU 使显卡削减了对 CPU 的依赖，并执行部分原本 CPU 的工作，尤其是在 3D 图形处理时。GPU 所采用的核心技术有钢体 T&L、立方环境材质贴图与顶点混合、纹理压缩及凹凸映射贴图、双重纹理四像素 256 位渲染引擎等，而硬体 T&L 技术能够说是 GPU 的标志。

## 二、GPU 与 CPU 的区别

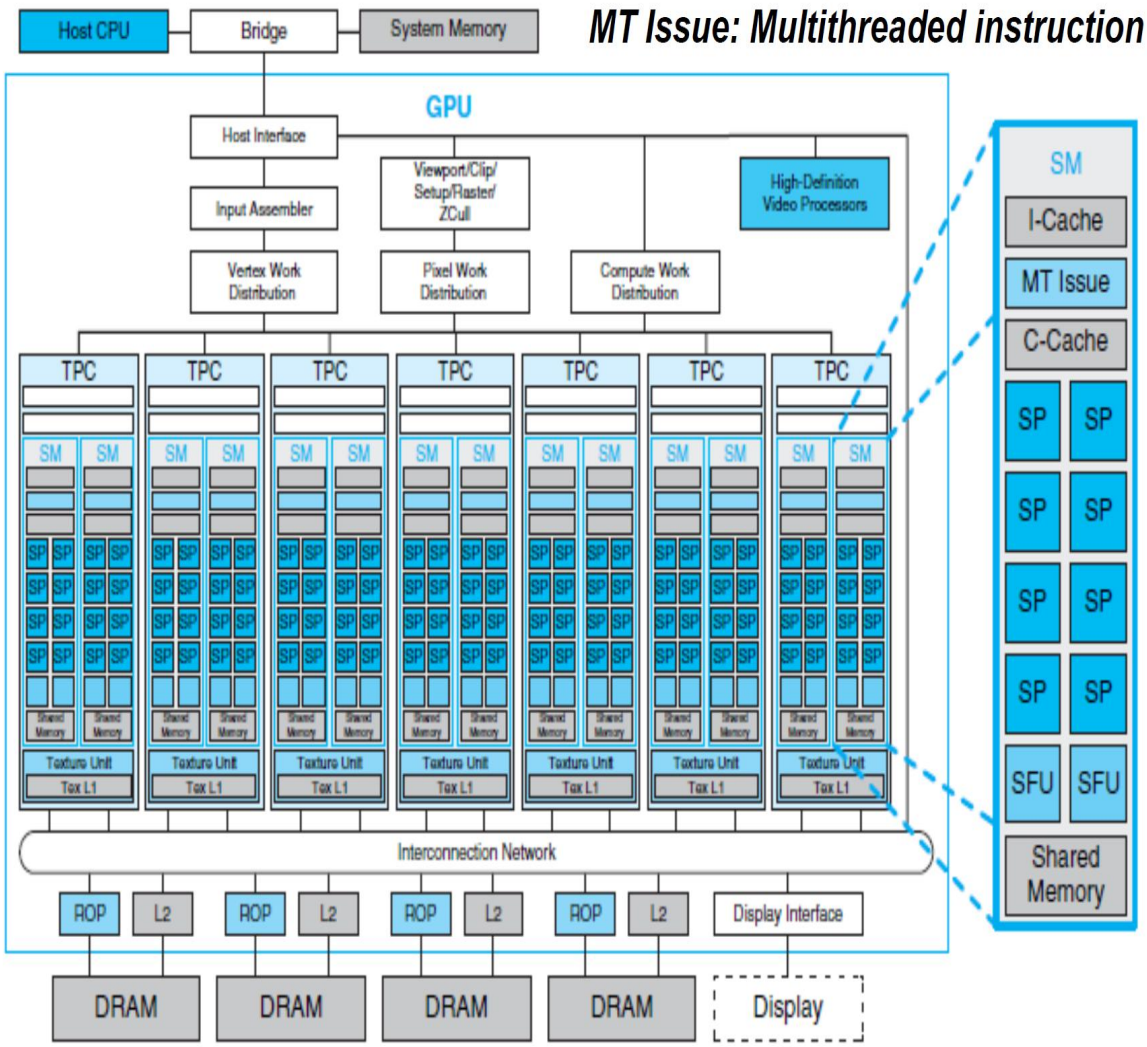
- CPU 是电脑的中央处理器。
- GPU 是电脑的图形处理器。
- CPU 是一块超大规模的集成电路，其中包含 ALU 算术逻辑运算单元、Cache 高速缓冲存储器以及 Bus 总线。
- CPU 是一台计算机的控制和运算核心，它的主要功能便是解释计算机发出的指令以及处理电脑软件中的大数据。
- GPU 是图像处理器的缩写，它是一种专门为 PC 或者嵌入式设备进行图像运算工作的微处理器
- GPU 的工作与上面说过的 CPU 类似，但又不完全像是，它是专为执行复杂的数学和几何计算而生的，而这游戏对这方面的要求很高。

## 三、GPU 架构一览

- 2008-Tesla

Tesla 最初是给计算处理单元使用的，应用于早期的 CUDA 系列显卡芯片中，并不是真正意义上的普通图形处理芯片。

# NVIDIA Tesla Architecture



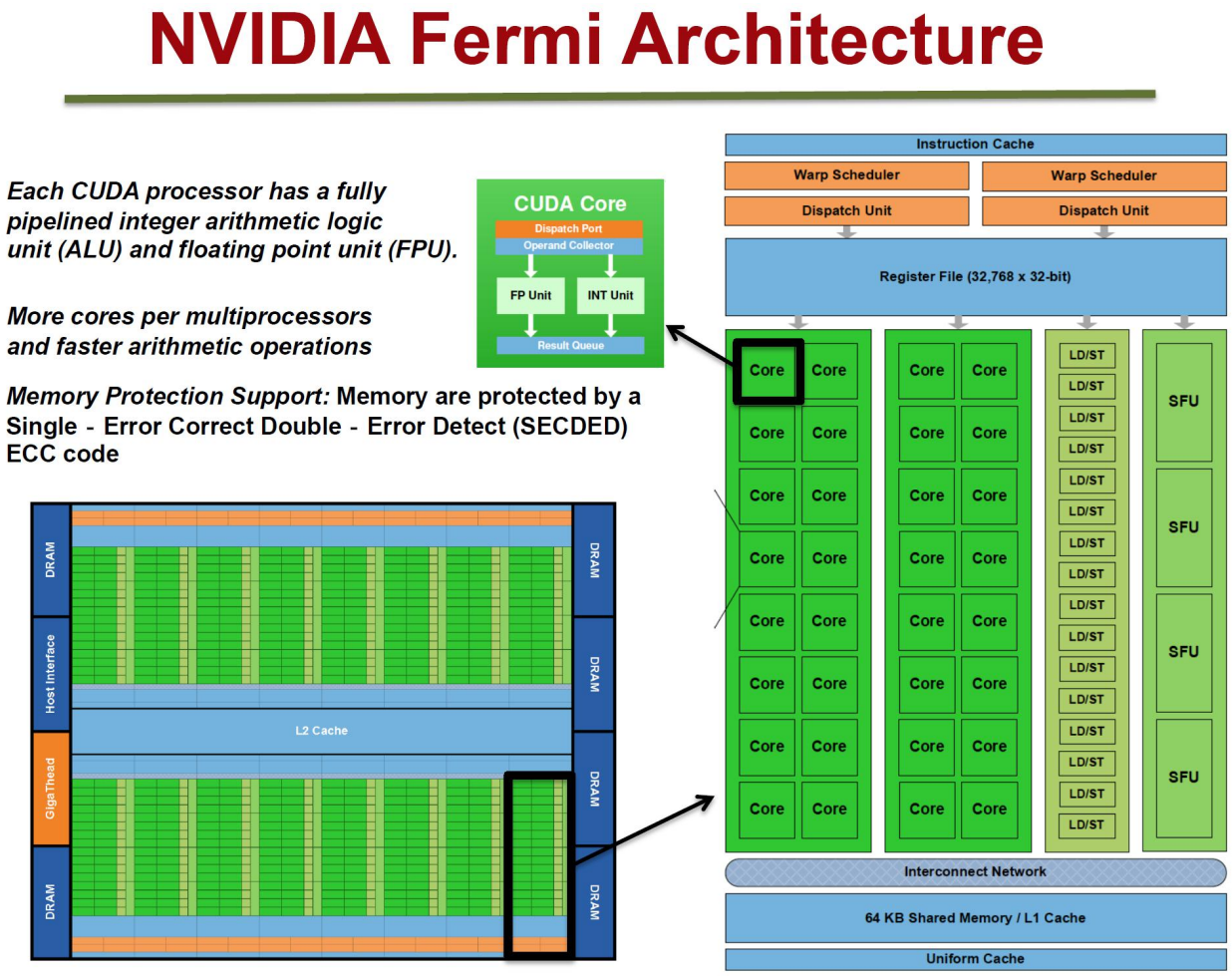

Tesla 微观架构总览图如上。下面将阐述它的特性和概念：

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。

- 2010-Fermi

Fermi 是第一个完整的 GPU 计算架构。首款可支持与共享存储结合纯 cache 层次的 GPU 架构，支持 ECC 的 GPU 架构。




Fermi 架构如上图，它的特性如下：

拥有 16 个 SM

每个 SM：

- 2 个 Warp（线程束）
- 两组共 32 个 Core
- 16 组加载存储单元（LD/ST）
- 4 个特殊函数单元（SFU）

每个 Warp：

- 16 个 Core
- Warp 编排器（Warp Scheduler）
- 分发单元（Dispatch Unit）

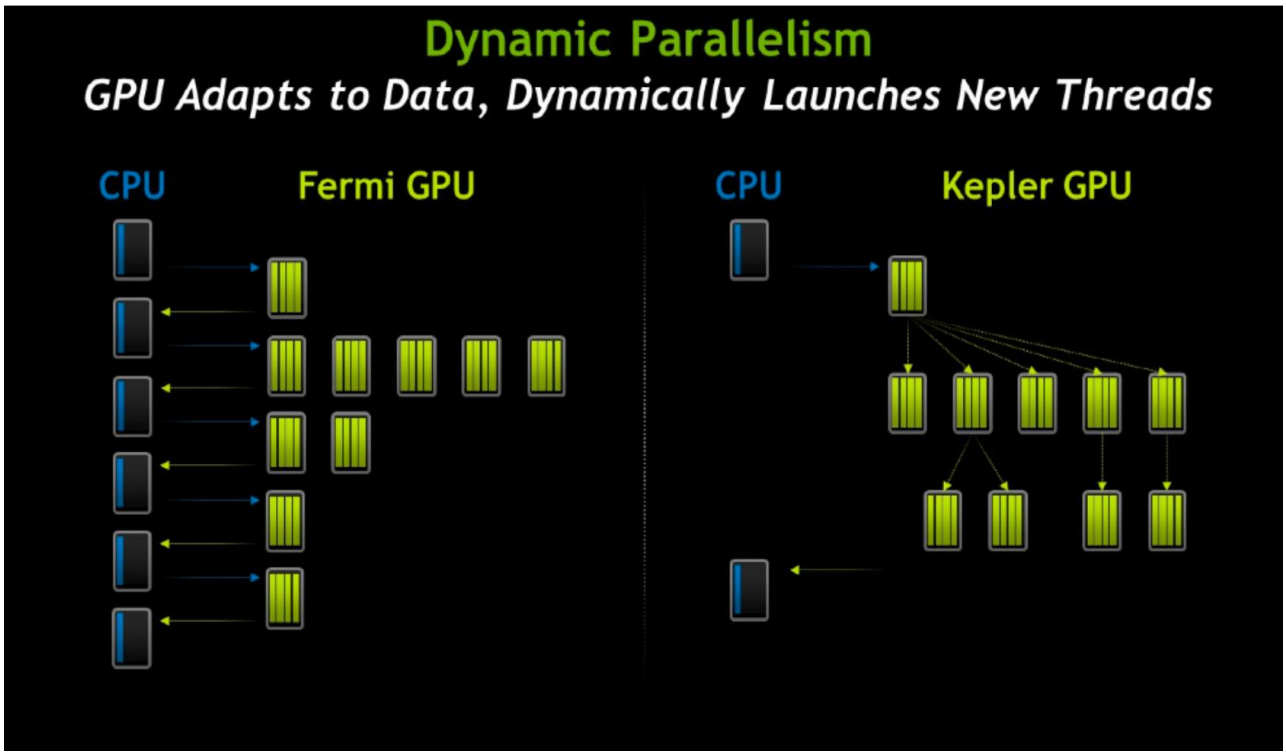
每个 Core：

- 1 个 FPU（浮点数单元）
- 1 个 ALU（逻辑运算单元）

- 2012-Kepler

Kepler 相较于 Fermi 更快，效率更高，性能更好。

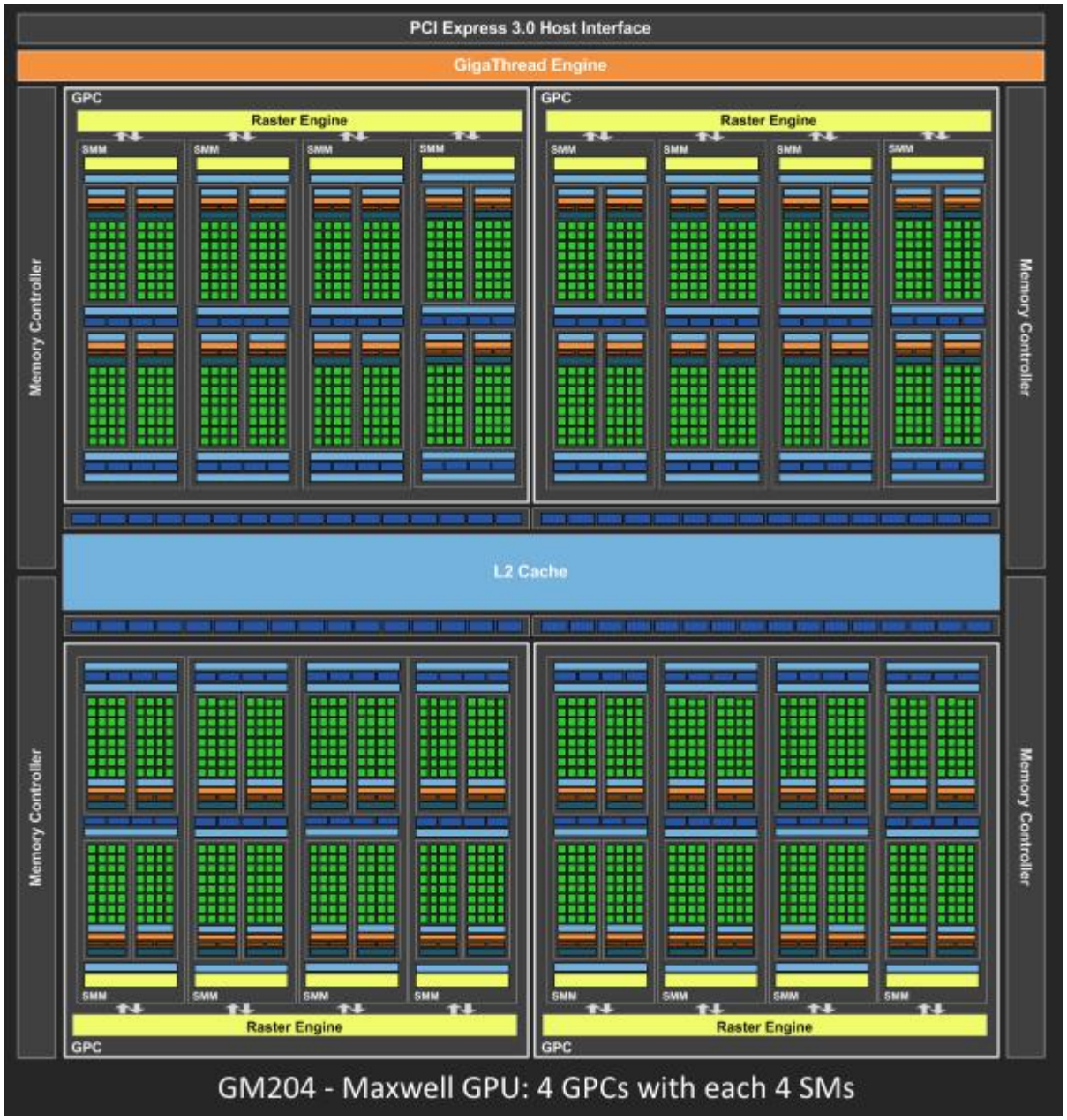

架构方面：Kepler 除了在硬件有了提升，有了更多处理单元之外，还将 SM 升级到了 SMX。SMX 是改进的架构，支持动态创建渲染线程（下图），以降低延迟。




- 2014-Maxwell

其全新的立体像素全局光照（VXGI）技术首次让游戏 GPU 能够提供实时的动态全局光照效果。

基于 Maxwell 架构的 GTX 980 和 970 GPU 采用了包括多帧采样抗锯齿(MFAA)、动态超级分辨率（DSR)、VR Direct 以及超节能设计在内的一系列新技术。




架构方面：采用了 Maxwell 的 GM204，拥有 4 个 GPC，每个 GPC 有 4 个 SM，对比 Tesla 架构来说，在处理单元上有了很大的提升。

- 2016-Pascal

Pascal 架构将处理器和数据集成在同一个程序包内，以实现更高的计算效率。

例如：1080 系列、1060 系列是基于 Pascal 架构的

- 2017-Volta

Volta 配备 640 个 Tensor 核心，每秒可提供超过 100 兆次浮点运算 (TFLOPS) 的深度学习效能，比前一代的 Pascal 架构快 5 倍以上。

注：因设计原因，直接跳过 Volta 结构而到 Turing 结构

- 2018-Turing

Turing 架构配备了名为 RT Core 的专用光线追踪处理器，能够以高达每秒 10 Giga Rays 的速度对光线和声音在 3D 环境中的传播进行加速计算。

Turing 架构将实时光线追踪运算加速至上一代 NVIDIA Pascal™ 架构的 25 倍，并能以高出 CPU 30 多倍的速度进行电影效果的最终帧渲染。

2060 系列、2080 系列显卡也是跳过了 Volta 直接选择了 Turing 架构。


上图是采纳了 Turing 架构的 TU102 GPU，它的特点如下：

6 GPC（图形处理簇）

36 TPC（纹理处理簇）

72 SM（流多处理器）

每个 GPC 有 6 个 TPC，每个 TPC 有 2 个 SM

4,608 CUDA 核

72 RT 核

576 Tensor 核

288 纹理单元

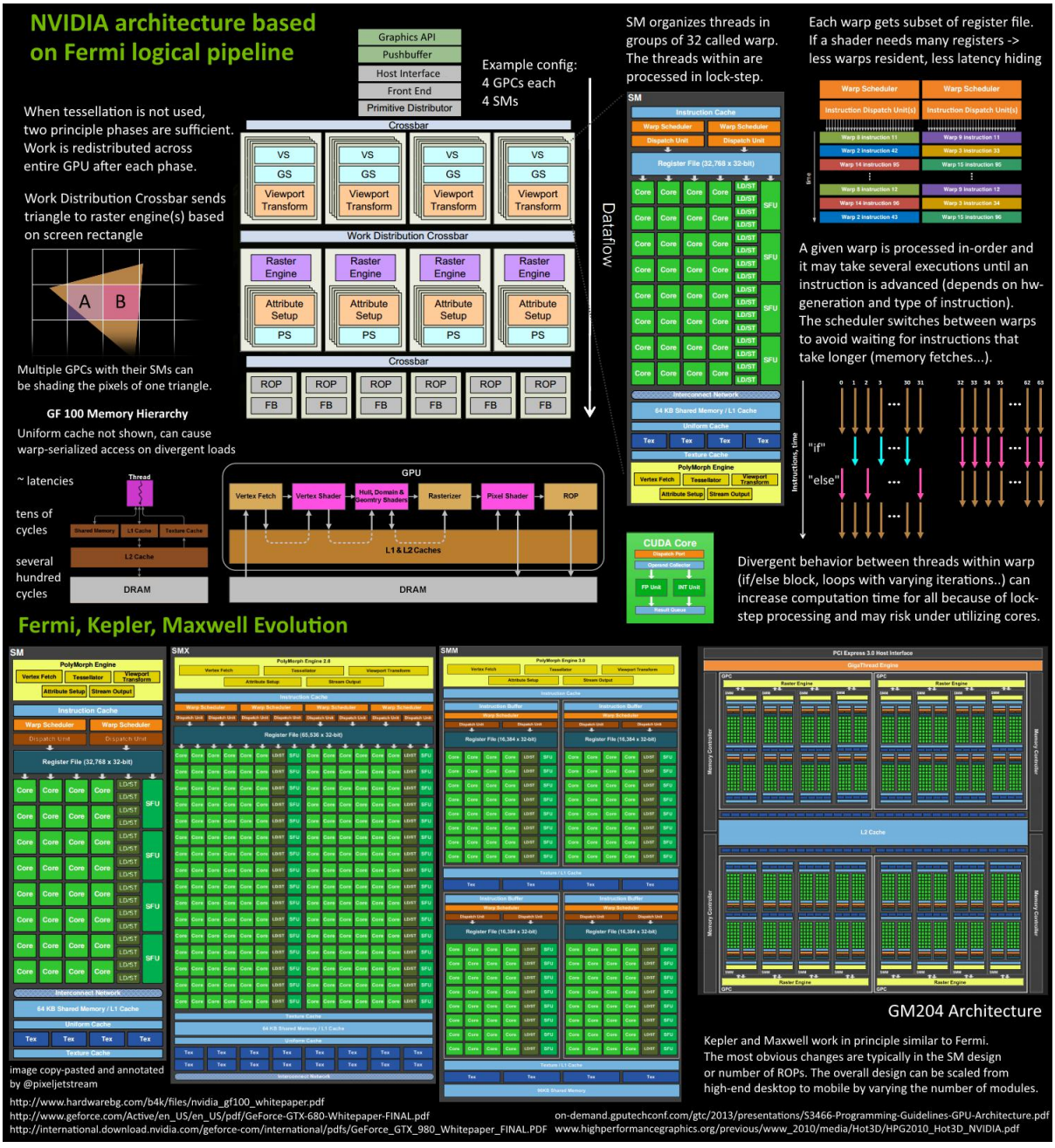
12x32 位 GDDR6 内存控制器 (共 384 位)

单个 SM 的结构图如下：

# 四、GPU 部分架构共性

## • GPU 渲染总览：

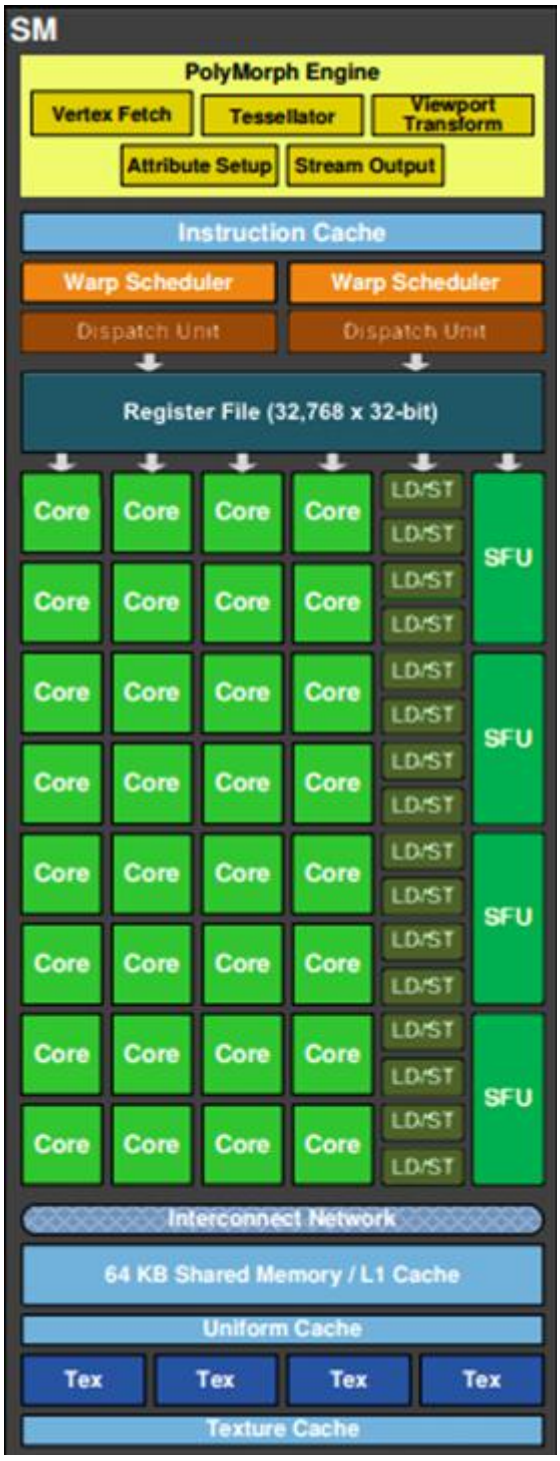
现代 GPU 有着相似的结构，有很多相同的部件，在运行机制上，也有很多共同点。下面是 Fermi 架构的运行机制总览图：




从 Fermi 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。

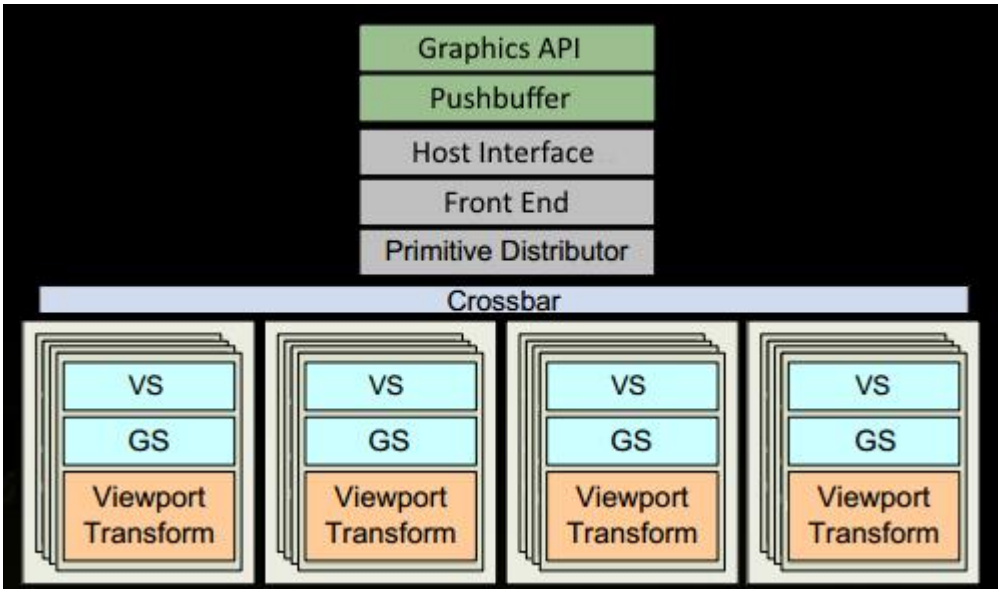


如上图，对于某些 GPU（如 Fermi 部分型号）的单个 SM，包含：

- 32 个运算核心 （Core，也叫流处理器 Stream Processor）
- 16 个 LD/ST（load/store）模块来加载和存储数据
- 4 个 SFU（Special function units）执行特殊数学运算（sin、cos、log 等）
- 128KB 寄存器（Register File）
- 64KB L1 缓存
- 全局内存缓存（Uniform Cache）
- 纹理读取单元
- 纹理缓存（Texture Cache）
- PolyMorph Engine：多边形引擎负责属性装配（attribute Setup）、顶点拉取(VertexFetch)、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（Instruction Cache）
- 内部链接网络（Interconnect Network）

## • GPU 逻辑管线：

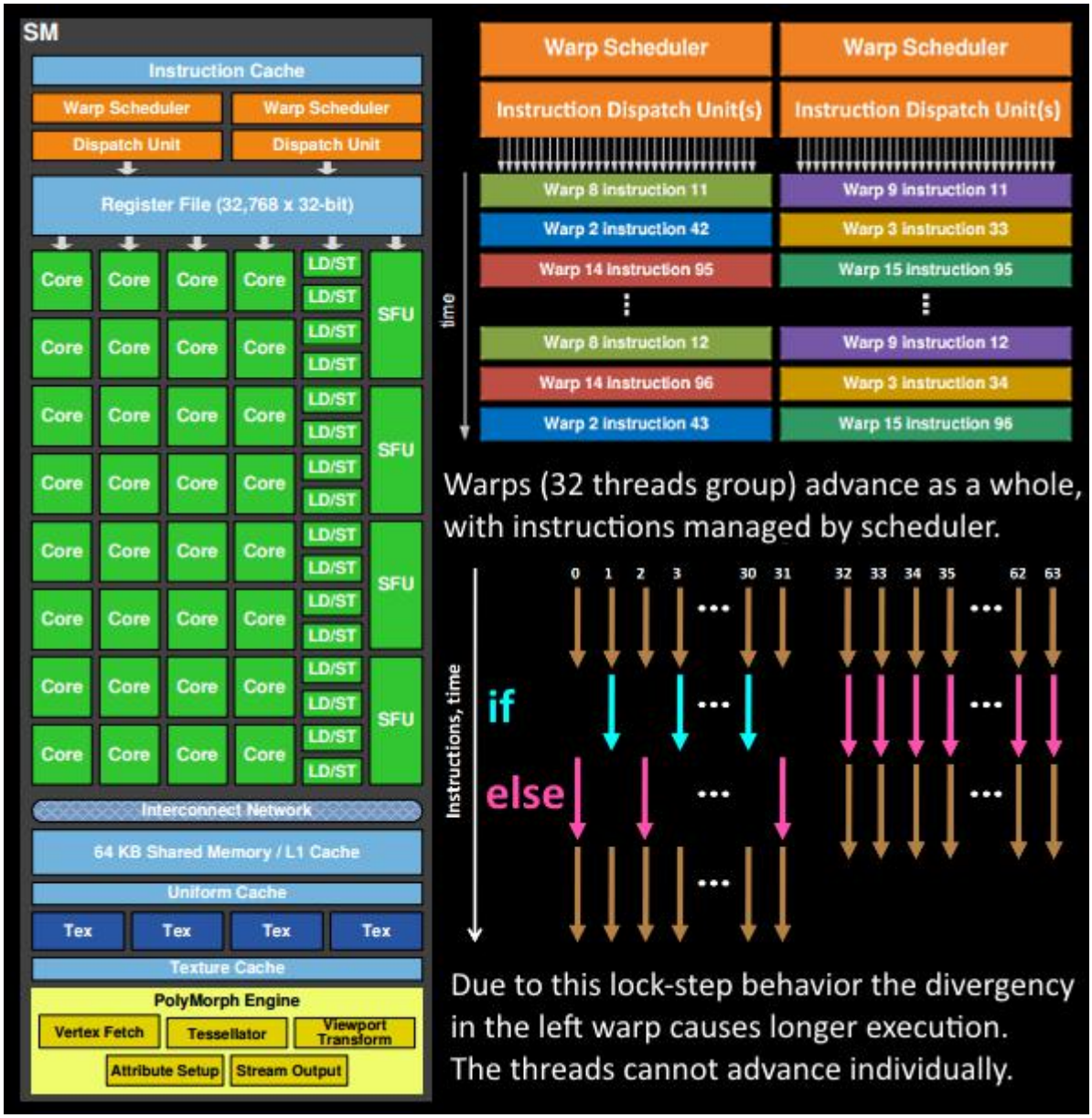
以 Fermi 家族的 SM 为例：




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。

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（Host Interface）接受这些命令，并通过前端（Front End）处理这些命令。

在图元分配器(Primitive Distributor)中开始工作分配，处理indexbuffer中的顶点产生三角形分成批次(batches)，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。




在GPC中,每个SM中的Poly Morph Engine负责通过三角形索引(triangle indices)取出三角形的数据(vertex data)，即图中的Vertex Fetch模块。

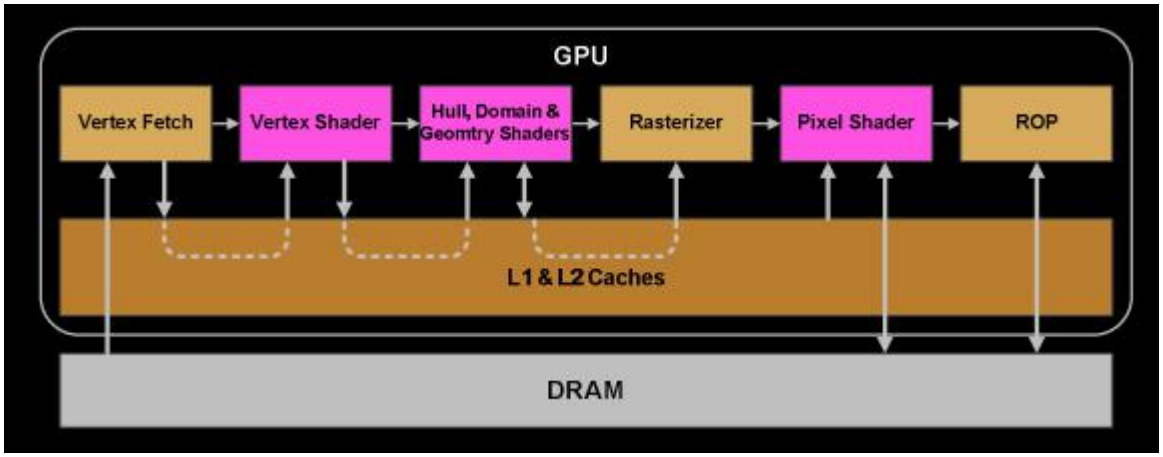
在获取数据之后，在SM中以32个线程为一组的线程束(Warp)来调度，来开始处理顶点数据。Warp是典型的单指令多线程（SIMT，SIMD单指令多数据的升级）的实现，也就是32个线程同时执行的指令是一模一样的，只是线程数据不一样，这样的好处就是一个warp只需要一个套逻辑对指令进行解码和执行就可以了，芯片可以做的更小更快，之所以可以这么做是由于GPU需要处理的任务是天然并行的。

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(lock-step)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

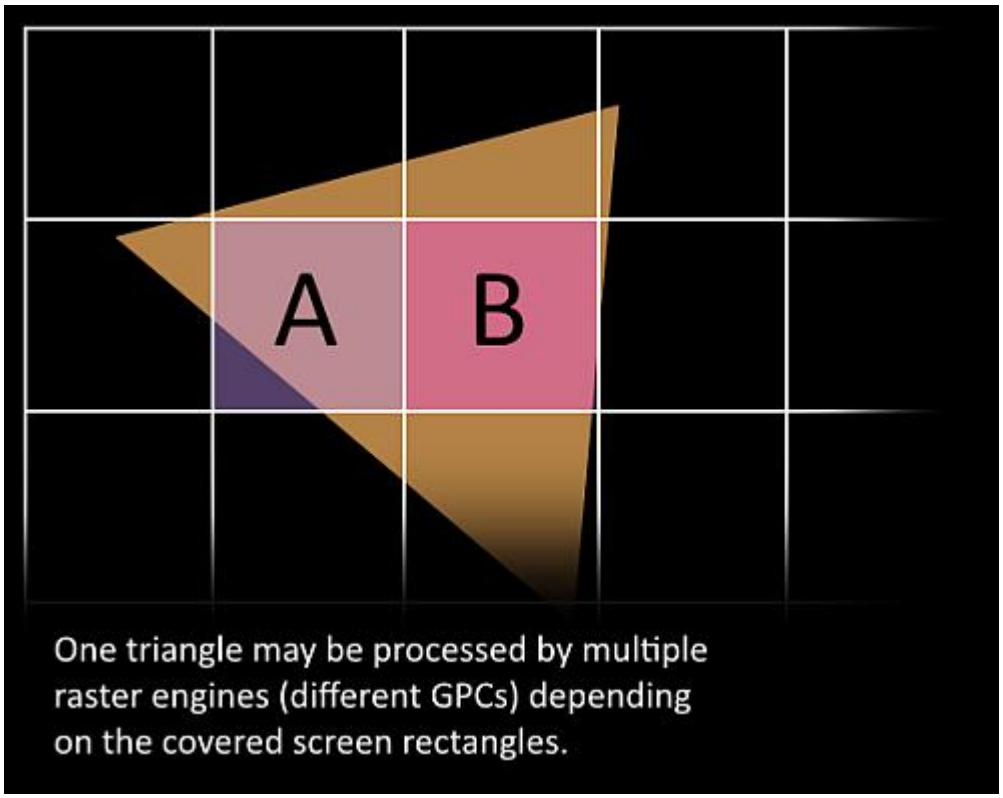
掩(be masked out)。被遮掩的原因有很多，例如当前的指令是 if(true)的分支，但是当前线程的数据的条件是 false，或者循环的次数不一样（比如 for 循环次数 n 不是常量，或被 break 提前终止了但是别的还在走），因此在 shader 中的分支会显著增加时间消耗，在一个 warp 中的分支除非 32 个线程都走到 if 或者 else 里面，否则相当于所有的分支都走了一遍，线程不能独立执行指令而是以 warp 为单位，而这些 warp 之间才是独立的。

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。

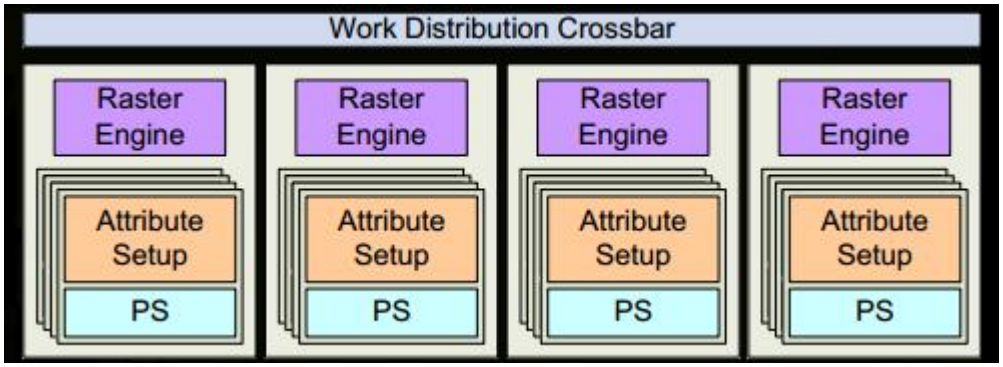
由于某些指令比其他指令需要更长的时间才能完成，特别是内存加载，warp 调度器可能会简单地切换到另一个没有内存等待的 warp，这是 GPU 如何克服内存读取延迟的关键，只是简单地切换活动线程组。为了使这种切换非常快，调度器管理的所有 warp 在寄存器文件中都有自己的寄存器。这里就会有个矛盾产生，shader 需要越多的寄存器，就会给 warp 留下越少的空间，就会产生越少的 warp，这时候在碰到内存延迟的时候就会只是等待，而没有可以运行的 warp 可以切换。




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。




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(raster engines)，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。

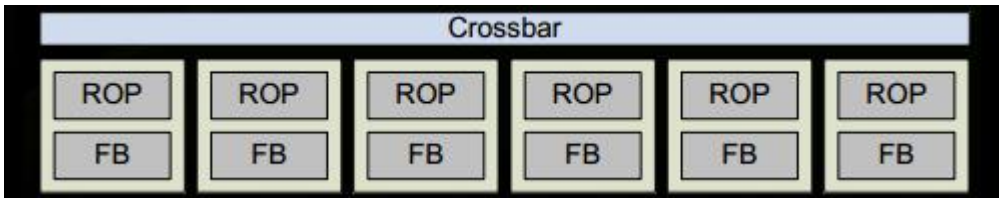



SM 上的 Attribute Setup 保证了从 vertex-shader 来的数据经过插值后是 pixel-shade 是可读的。

GPC 上的光栅引擎(raster engines)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（同时会处理裁剪 Clipping、背面剔除和 Early-Z 剔除）。

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。

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。




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(render output unit，渲染输入单元)，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。