

Deep Into Graphics Rendering

wegatron

2021.11.2

摘要

本文介绍图形渲染相关的一些重要的基础知识。在“拥抱现代图形 API”，首先通过现代和传统图形 API 的对比，阐述了现代图形 API 的优势。然后，分析了使用现代图形 API 开发的一些挑战，并给出了基于现代图形 API 的渲染管线设计的一些常规的思路。为了更快速的理解现代图形 API 中的基本概念，进行了常用图形 API 的核心类/函数比较。其后，在“深入理解 GPU 硬件运行机制”一章中，以 Nvidia 显卡为例，介绍了 GPU 的硬件设计，以及渲染工作在 GPU 中的执行过程。在该章末尾，着重介绍了 GPU 中的几个重要技术。最后一章简单介绍了一些渲染优化的简单建议。

1. 拥抱现代图形 API

参考《游戏引擎随笔 0x05: 现代图形 API 讲义》[7]

1.1. 现代图形 API VS 传统图形 API

传统图形 API: OpenGL(ES), D3D

现代图形 API: Vulkan, Metal, D3D12

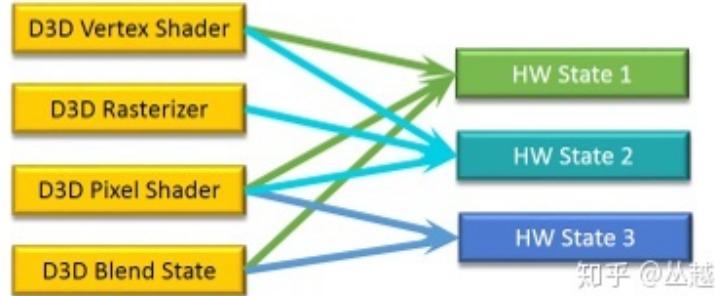
比喻:

- 公交车 (传统图形 API-固定管线)，满足大众常用的出行需求，无法定制且出行效率低。使用代价最低 (廉价)。
- 打车 (传统图形 API-可编程管线)，可以自定义出行起点 + 终点 (甚至可以提出一些路线或速度上的要求)，但仍然需要一定的等待，但最终得依靠司机，无法满足一些特殊的需求 (如在偏僻地方打不上车)。使用代价增高。
- 自驾车 (现代图形 API)，完全掌控自己的出行路线、时间、速度等。使用代价最高 (需要自己会开)。
 - 自动档 (Metal)，做了部分自动化，使用方便且不失性能。
 - 手动档 (Vulkan)，暴露了很多细节，使用难度更大，可操控性更强。
- 辅助驾驶 (UE 等渲染引擎)，结构复杂，自动化程度高。

现代图形 API 与传统图形 API 要点比较:

- 渲染管线状态维护

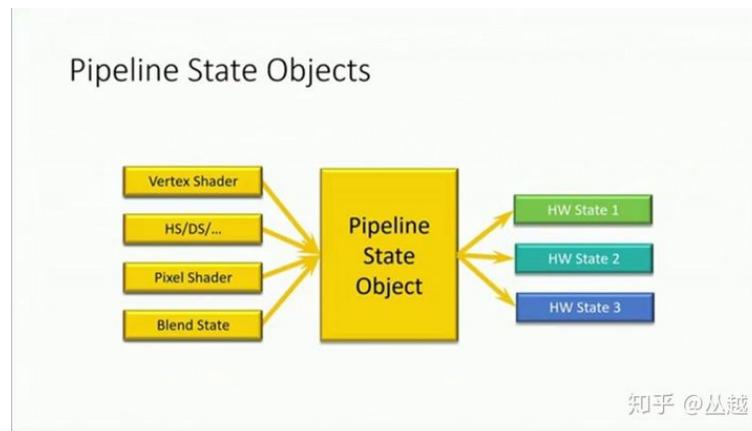
传统的图形 API 使用隐式、全局的状态管理机制。缺陷: 1. 无法精准的控制管线当前状态；2. 应用程序必须频繁的调用 API 来设置管线状态和恢复；3. API 还会对每个渲染状态的设置进行校验和二次处理，比如合并状态统一提交 (新型 GPU 会将这些固定管线状态合并成单一硬件状态，因此需要 Driver 在运行时为不同的状态组合创建、查找缓存内部的管线状态集合，这很可能会导致管线卡顿，同时额外增大了开销)。



知乎 @从越

图 1: D3D11 分散的状态对象设置导致图形硬件切换的额外开销

现代图形 API 使用 PipelineState 将管道状态提前创建并绑定，在渲染时，通过设置不同的 PipelineState，Driver 只需要少量的切换开销即可将预先创建的状态绑定到 GPU 中，而无需像传统 API 那样校验每种状态有效性以及动态合并状态，从而降低了绘制调用开销，并且可以大幅增加每帧的绘制调用次数。



知乎 @从越

图 2: PipelineState 将管道状态提前创建并绑定

- 资源绑定

与渲染管线状态类似，传统图形 API 逐个 API 调用来实现资源的绑定。而现代图形 API 通过 RootSignature(D3D12) 或者 PipelineLayout(Vulkan) 通过 Descriptor 预先设置好 Shader 所需要使用的资源布局信息，包括：Constant、Texture(SRV)、Buffer(UAV)、Sampler 等，在渲染时 Shader 通过 DescriptorTable 或者 DescriptorSet 用间接寻址方式获取资源，无需渲染时绑定，大大降低了资源绑定和校验开销，从而提高了渲染性能。

Metal 虽然也有 PipelineState，但并没有提供指定 Shader Resource Layout 的机制。

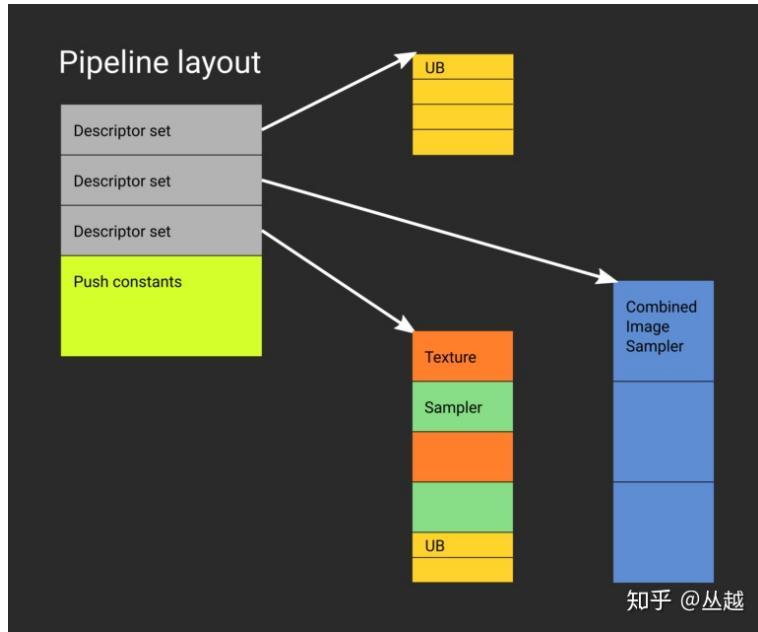


图 3: Vulkan 的 Pipeline Layout 使用 DescriptorSet 间接引用 Shader 资源

- 显式内存管理

传统的 API 的内存管理是隐式的, 当创建资源时 Runtime/Driver 内部也同时创建用于这个资源的内存, 这个过程对开发者是透明的, 尽管接口简单, 但带来的问题是很难优化内存分配, 并且容易造成 GPU 内存碎片.

现代 API 都提供了 CPU/GPU 堆内存管理接口, 可以通过创建堆, 在堆上分配空间的方式来创建资源, 应用程序可以精准的控制资源堆内存的分配. 比如可以通过资源别名 (在同一堆空间中创建不同的资源), 在不同的时段可重复利用同一的设备内存来完成渲染逻辑, 从而更有效的使用有限的 GPU 内存资源.

Aliasing Placed Resource

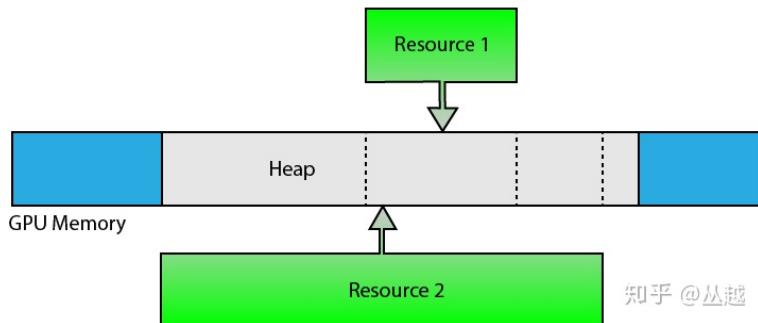


图 4: 在重叠的 GPU 堆空间中创建不同的资源

- 状态的跟踪和同步

传统 API 在 Driver 内部维护跟踪状态, 自动管理资源及调度, 以及进行运行时的校验. Driver 还要负责 CPU 与 GPU 的同步. 典型的例子: 动态更新 GPU Buffer 数据, 如果此时 GPU 正在使用这个 Buffer, 则 Map/Lock 之后 API 返回的是 Runtime/Driver 内部动态创建的新内存地址, 以更新新数据, 当 GPU 使用完旧内存后, 再使用新数据. 另外, API 的提示标记并不能保证 Driver 一定按照预想的方式执行,

Driver 会根据自身当前的状态来决定, 比如上述情况, 即使在 Map 时指定了 Discard, 也不能完全保证运行时 CPU 和 GPU 是完全异步的.

现代 API 通过使用资源屏障 (Resource Barrier) 来要求应用层明确控制资源的状态迁移, 通过 Fence 对象和 WaitFence 函数完成 CPU 和 GPU 的同步. 整个过程完全由应用程序来控制, 这样应用程序可以根据需要, 更加精准的控制同步时机.

- 并行渲染

现代 API 都增加了 Command List 或 Command Buffer 记录渲染指令, 再通过 Queue 提交到 GPU 中, 而每个 Command List/Buffer 都可以在不同的线程中单独填充, 这意味着可以并行录制渲染指令, 充分发挥了现代 CPU 多核的并行能力. 甚至还可以创建多个异步计算或者上传数据的 Command Queue, 利用 GPU 的并行机制实现渲染和计算、上传数据的并行.

- shader code

Shader Code 预编译机制

- 附加模块支持

- RayTracing

- ML: Vulkan ML, Metal Performance Shader

1.2. 基于现代图形 API 开发的挑战

要在现代 API 基础上实现更好的性能, 需要更多更复杂的图形管线管理开发工作. 由于是显式的, 更接近图形硬件的设计, 现代 API 的驱动不再负责传统 API Driver 复杂的内部逻辑, 但这些工作并不是自动消失, 而是转移到应用层, 由开发者负责. 传统 API 上和现代 API 上图形管线管理开发工作量的比较:



图 5: 基于传统 API 和现代 API 上图形管线管理代码量的比较

这些工作量包括:

- 重新设计图形 API 抽象接口.

通过 PipelineState 对象来维护渲染管线状态, Command List 录制图形指令, 实现并行渲染. 资源屏障等, 来进行同步.

- 堆内存管理.

包括 GPU 和 CPU 内存, 精细控制内存预算和跟踪内存分配和使用, 甚至还要根据不同使用场景来定制内存管理策略. 比如根据 GPU 是否只读、CPU 是否只读、CPU/GPU 的写入频率等, 不同的情况需要对应不同的内存管理策略, 充分利用 GPU 的 Copy/Upload/Transfer 硬件引擎完成数据传输, 才能实现最佳的性能.

- 描述符 (Descriptor) 管理

在传统 API 上绑定 Shader 资源只需要简单调用形如 SetTexture/SetConstant/SetSampler 之类的接口即可, 而在现代 API 中, 通过 Shader 所需要使用的资源布局信息是预置的 (前文所述), 渲染时需要通过 Descriptor 来间接寻址资源, 由于 Descriptor 也是一种 API(GPU 硬件) 资源, 一般来说 GPU 可见

的 Descriptor 是有限的 (硬件相关限制), 渲染时需要对有限的 GPU Descriptor 使用有效的管理方式加以重用, 才能完成复杂的渲染逻辑. 另外由于并行渲染的存在, Descriptor 的分配和释放还要考虑到线程同步, 如何在并行中减少线程同步所带来的开销也是需要仔细考虑的问题, 这进一步增加了管理复杂度.

- 渲染帧管理

为了最大化并行 CPU/GPU, Swapchain 通常需要创建多个 back Buffer, 这样 GPU 绘制当前帧 (或者上一帧), CPU 可以并行填充绘制下一帧的命令, 在开始录制每个 Frame 的渲染指令时, 可以通过这一帧上一次绘制的 GPU Fence 查询 GPU 是否完成上一次渲染, 如果完成则开始录制逻辑, 否则等待. 要达到这样的结果, 每个 Frame 需要有自己的 CommandList 和相关的 GPU 资源, 这就需要实现渲染帧逻辑, 还需要在提交到 GPU 渲染时对 Frame 进行调度管理.

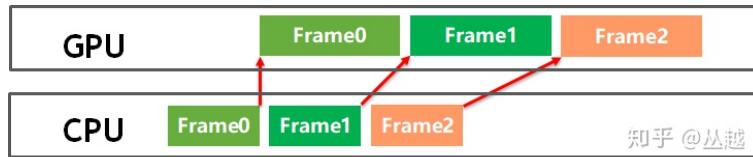


图 6: 并行渲染

- API 对象的生命周期控制

要保证 API 渲染、计算管线在使用对象的过程中不能释放对象. 比如上层逻辑在 Frame2 中释放某个对象, 但这个对象还在 Frame1 中被使用, 则此时不能执行真正的释放, 需要通过 GPU Fence 事件通知或者轮询方式等到 Frame1 执行完成才能释放.

- 并行提交绘制指令

现代 API 的 Command List 都是 Thread Free 的, 所以可以实现多个线程并行填充绘制、计算指令, 以达到并行提交渲染工作的目的. 可利用 Task/Job System 来实现.

1.3. 基于现代图形 API 的渲染管线设计

GPU 并行架构利用并行提交特性, 渲染管线可设计为多线程结构, 可根据当前 CPU 硬件线程数量动态决定 CommandList/Buffer 的数量, 这样在架构上也是可缩放的。

另外, 现代 GPU 其内部都会有多个专用于不同功能的 GPU 硬件, 一般可以抽象为图形 (3D)、计算 (Compute)、Copy(Transfer) 三种, 这三者在 GPU 内部可并行执行, 如下图所示:

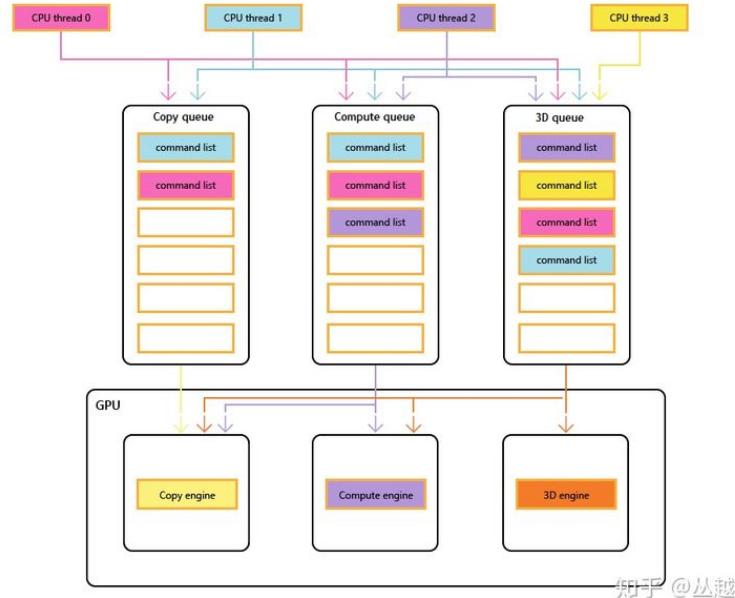


图 7: GPU 硬件功能抽象

利用这个机制可以将管线设计为这样:

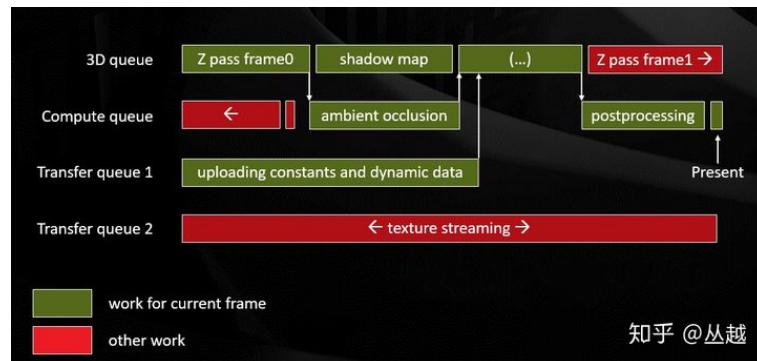


图 8: GPU 渲染管线的初步设计

如图所示，图形 (3D) 队列负责 PreZ、GBuffer、Shadow 等场景渲染，同时计算队列负责计算后处理 (SSAO、Bloom、ToneMapping、AA 等等)，Copy 队列同时执行纹理 Streaming 或者 Virtual texture 等操作。这些都可以在 GPU 时间线上同时进行，大大提高了渲染管线的性能。

进一步优化 Render Graph: 通过检查渲染帧内的资源依赖关系，利用别名资源重复利用内存堆，优化 GPU 内存资源使用。如下所示：

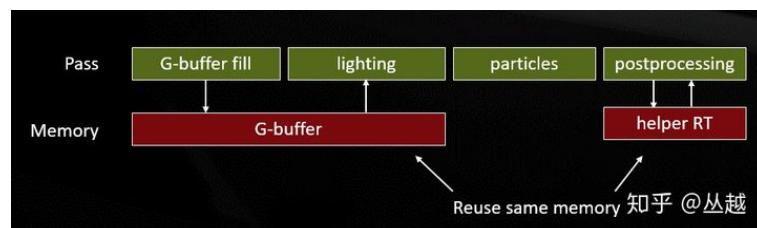


图 9: GPU 渲染管线进一步优化

GPU 驱动的渲染管线利用间接绘制/计算 (IndirectDraw/IndirectCompute), 可实现渲染时更少的 CPU/GPU 之间的切换, 在 CPU 中准备场景数据, 一次性提交到 GPU 中, 使用 GPU 进行可见性剔除, 并填充 CommandList/Buffer, 这样可以将传统的 CPU 负担的工作交由 GPU 执行, 降低 CPU 的负载, 甚至可以进行更高精度级别的可见性剔除.

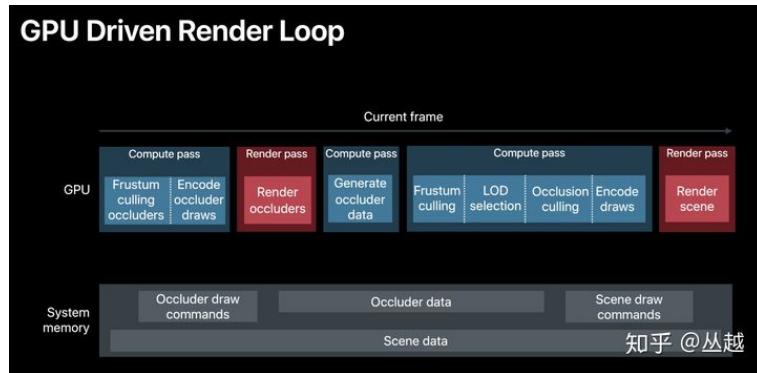


图 10: 基于 Metal API 的 GPU 驱动渲染管线设计

1.4. 常用图形 API 核心类/函数的比较

TODO

2. 深入 GPU 硬件运行机制

参考《深入 GPU 硬件架构及运行机制》[1]

2.1. 了解 GPU 的硬件

2.1.1 GPU 中的一些基本概念

这里以 Nvidia Turing 架构为例, Turing 架构是 Nvidia 在 2018 年发布的 GPU 架构, 其下 2060、2080 系列显卡均使用该种架构.



图 11: Nvidia Turing 架构



图 12: Nvidia Turing 架构-Stream Multiprocessor

- Giga Thread Engine 管理所有正在进行的工作
- GPC(Graphics Processing Cluster)

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).

- Raster Engine
- TPC(Texture Processing Cluster)
 - 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
 - Register File

- SM(Stream Multiprocessor)

多个 sp 加上其他的一些资源组成一个 SM, 其他资源也就是存储资源, 共享内存, 寄储器等.
 - Warp(Warp Schedular+Dispatch)

多个 SP(一般是 32 个) 组成一个 Warp, Warp 是最小调度单位. 同一个 Warp 内的 SP 执行指令相同只是数据不同. 只有全部线程执行完毕才会进行下一个 Warp 的工作.
 - SP(Stream Processor)/Core/Thread

SP 是最基本的处理/计算单元.

 - ALU
 - FPU
 - Tensor Core
 - SFU

特殊数学函数, 如 sin, cos, 与普通 sin, cos 实现 (一般会查表) 不同.
 - L1 Cache
 - Texture Cache
 - LD/ST
 - RT(RayTracing) Core
- Grid
编程中的一个概念. 指在 GPU 上由多个 Thread block(SM) 执行的一套代码.

2.1.2 GPU 的内存分级

GPU 内存等级 (参考《Computer Architecture》[3])

More descriptive name	Closest old term outside of GPUs	Official Nvidia GPU term	Description
GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
Local Memory	Local Memory	Shared Memory	fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors
SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

访存速度:

存储类型	寄存器	共享内存	L1 缓存	L2 缓存	纹理、常量缓存	全局内存
访存周期	1	1 – 32	1 – 32	32 – 64	400 – 600	400 – 600

2.2. 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。这里以 Nvidia-Fermi 架构为例：

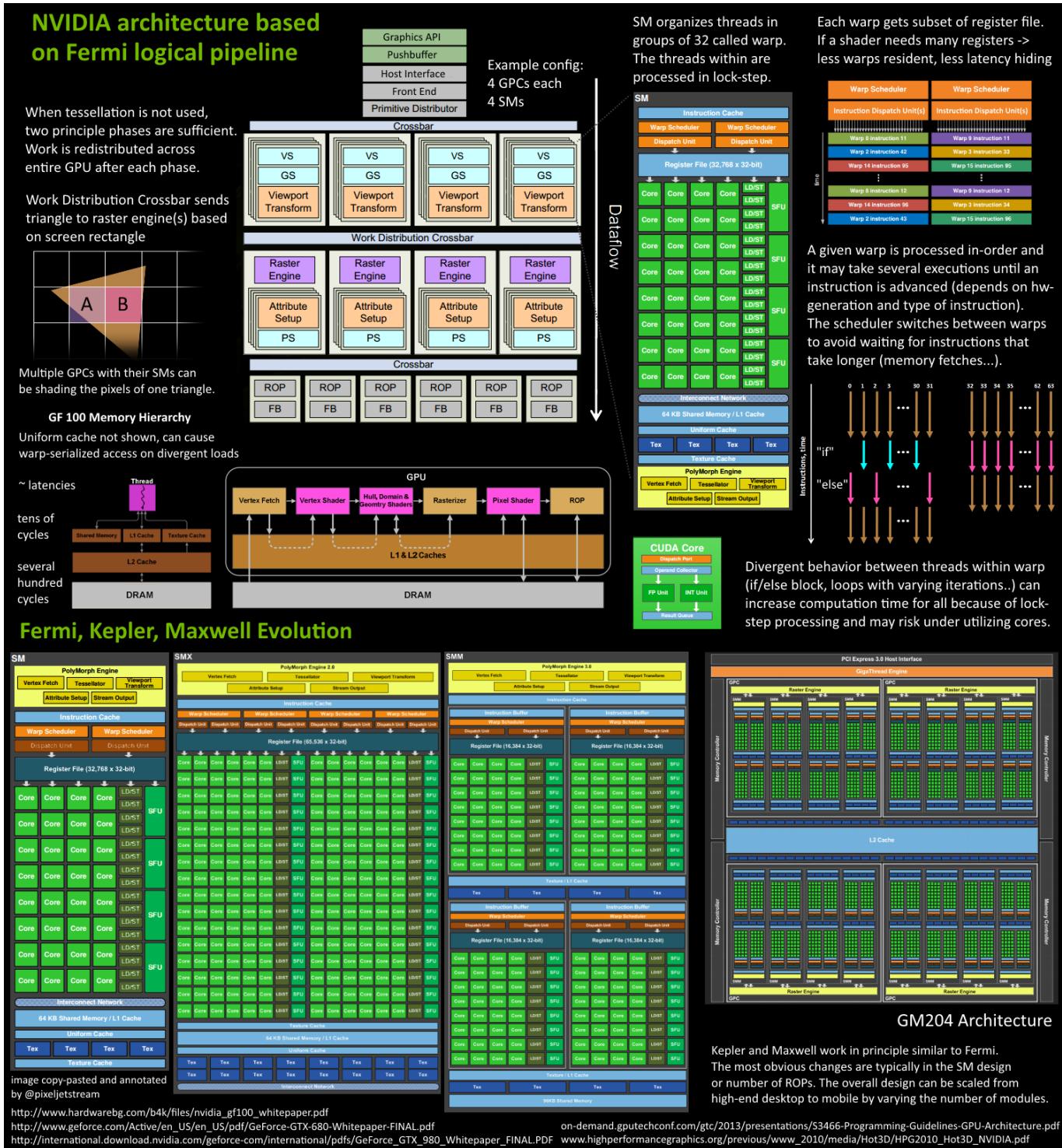


图 13: GPU 渲染管线的执行细节

从 Fermi 开始 NVIDIA 使用类似的原理架构, GPU 中使用 Giga Thread Engine 来管理所有正在进行的工作。GPU 被划分为多个 GPC, GPC 通过 Crossbar 与其他功能模块相连。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。

对于相同架构的 GPU, 具体有多少个 GPC, GPC 有多少 SM... 则根据不同的显卡型号有所不同.

2.2.1 GPU 渲染管线的执行

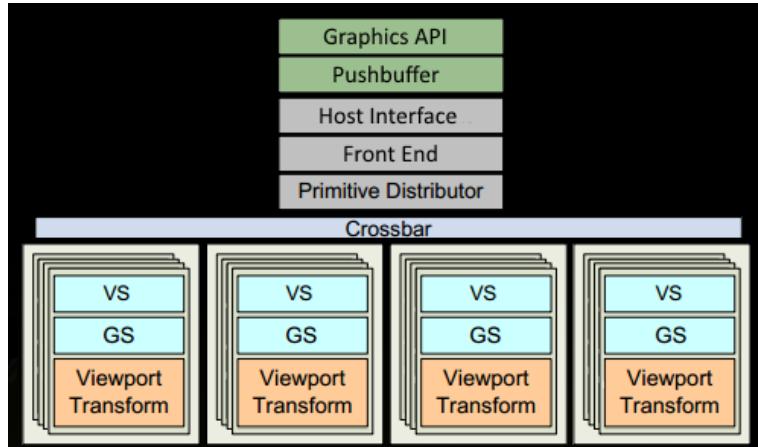


图 14: 渲染管线执行: 图元数据的输入

1. Drawcall → 驱动程序 (合法性检查) → Pushbuffer(flush) → Host Interface(并通过 Front End 处理).
2. 在图元分配器 (Primitive Distributor), 将图元分成批次发送给多个 PGCS.
3. 在 GPC 的 SM 中的 Poly Morph Engine 负责取出三角形的数据 (vertex data).
4. 在获取数据之后, 在 SM 中以线程束 (Warp) 来调度, 来开始处理顶点数据. 这里 SM 的 warp 调度器会按照顺序分发指令给整个 Warp, Warp 中的线程锁步执行, 不激活状态下会被遮掩 (be masked out). Shader 中若有分支等价于所有分支全走一遍 (除非 Warp 中所有线程均走同一分支). 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, 此时 Warp 调度器可能会简单地切换到另一个没有等待的 Warp, 这也是 GPU 克服内存读取延迟的关键. 这里会产生一个矛盾, 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.

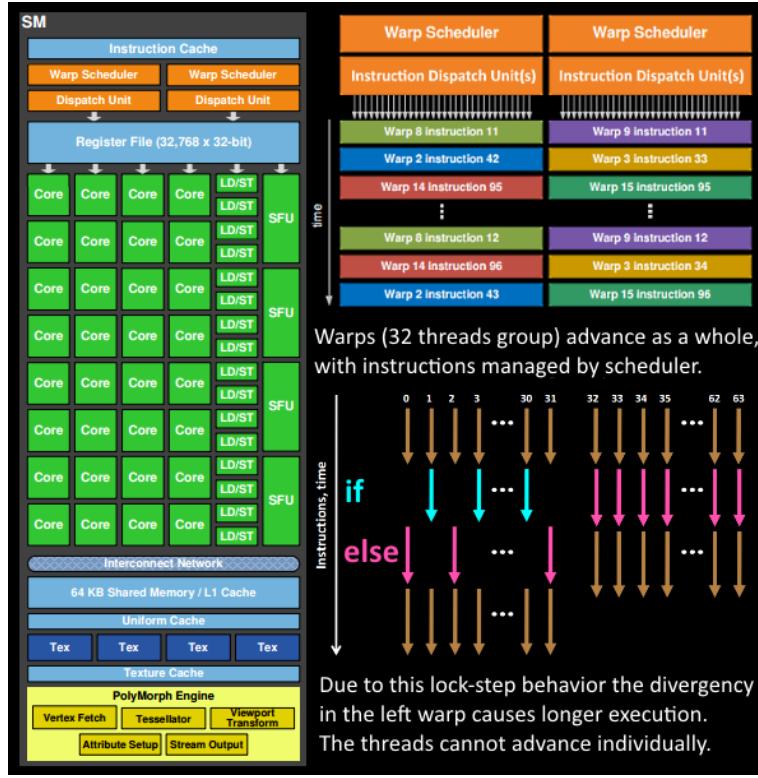


图 15: 渲染管线执行: Execution

5. Vertex Shader 完成后, 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.

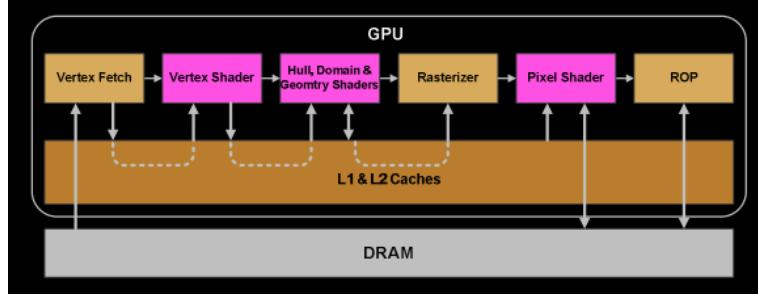


图 16: 渲染管线执行: 各阶段完成后数据存到 Cache

6. 接下来这些三角形将被分配给多个 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.

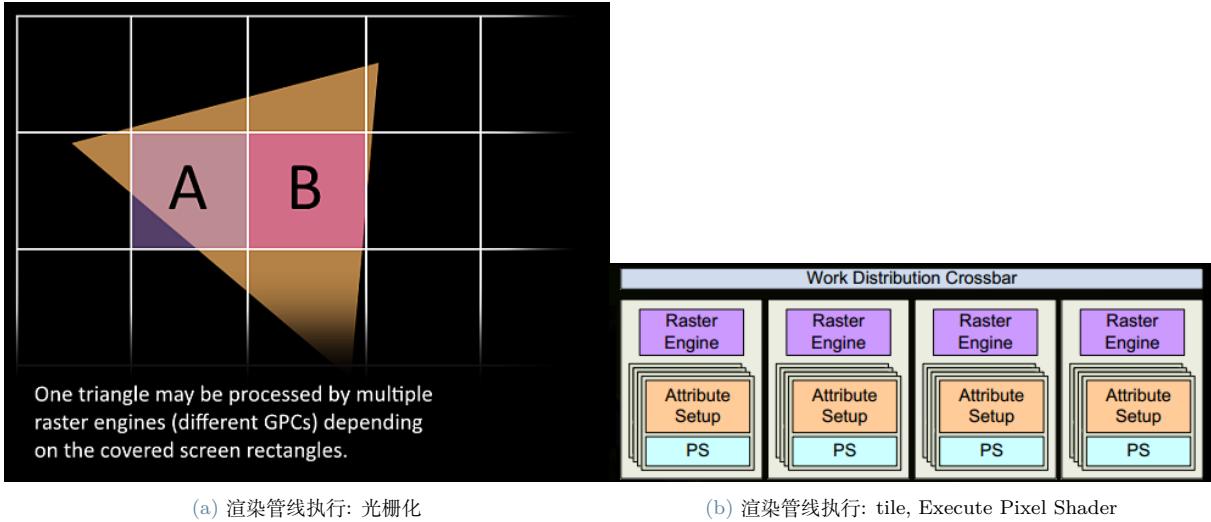


图 17: 渲染管线执行: 光栅化 +pixel shader

7. 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.
8. 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).
9. 32 个像素线程将被分成一组, 或者说 8 个 2×2 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.
10. 在 SM 中执行 Fragment/Pixel Shader.
11. 最后一步, 在 ROP 单元中计算深度值以及与 FrameBuffer 混合.

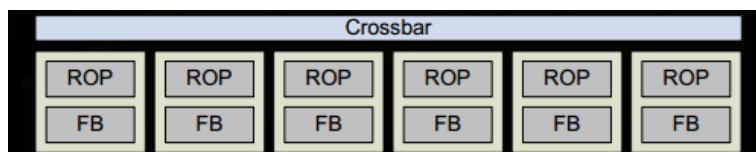


图 18: 渲染管线执行: ROP 进行像素点的深度值计算

2.3. GPU 技术要点

2.3.1 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

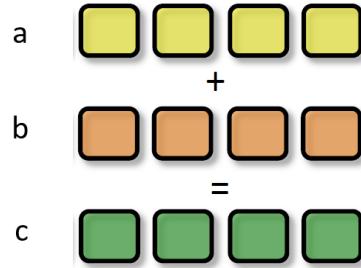


图 19: SIMD

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:

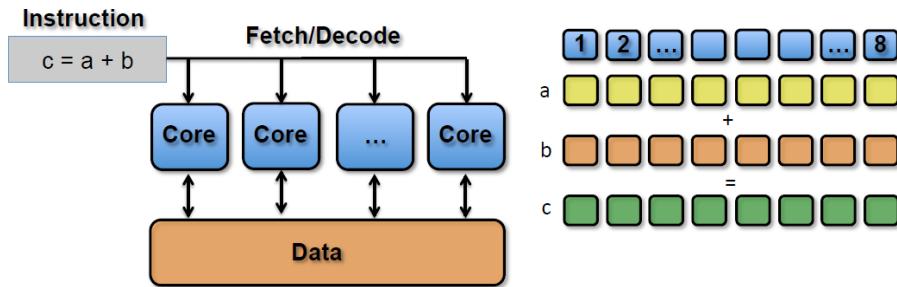


图 20: SIMD

2.3.2 co-issue

co-issue 是为了解决 SIMD 运算单元无法充分利用的问题. 为了解决着色器在低维向量的利用率低的问题, 可以通过合并 1D 与 3D 或 2D 与 2D 的指令, 提升利用率/效率.

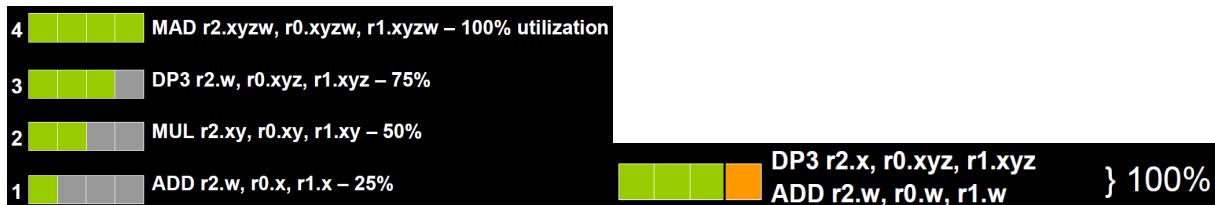


图 21: co-issue, 指令合并

但是, 对于向量运算单元 (Vector ALU), 如果其中一个变量既是操作数又是存储数的情况, 无法启用 co-issue 技术:

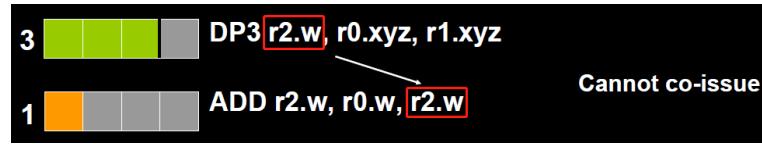


图 22: 无法合并的情况

于是标量指令着色器 (Scalar Instruction Shader) 应运而生 (register 空间小, 低能耗), 它可以有效地组合任何向量, 开启 co-issue 技术, 充分发挥 SIMD 的优势.

2.3.3 Early-Z

早期 GPU 的渲染管线的深度测试是在像素着色器之后才执行 (下图), 这样会造成很多本不可见的像素执行了耗性能的像素着色器计算。

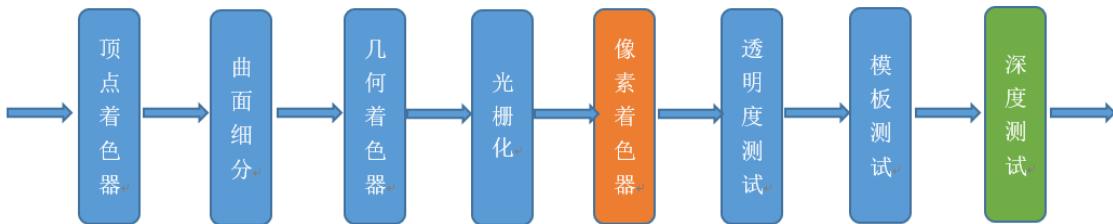


图 23: 早期没有 Early-Z 的渲染管线

后来, 为了减少像素着色器的额外消耗, 将深度测试提至像素着色器之前 (下图), 这就是 Early-Z 技术的由来.

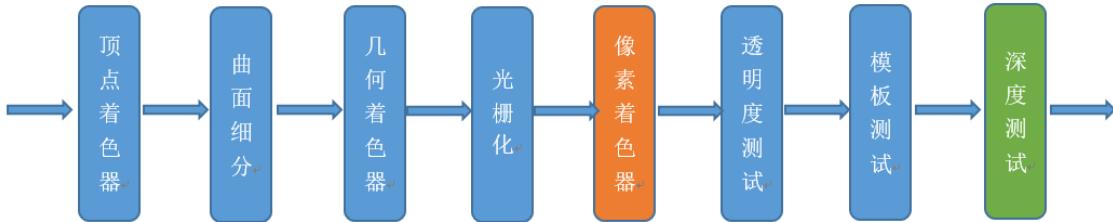


图 24: 添加 Early-Z 的渲染管线

Early-Z 技术可以将很多无效的像素提前剔除, 避免它们进入耗时严重的像素着色器。Early-Z 剔除的最小单位不是 1 像素, 而是像素块 (pixel quad, 2x2 个像素).

但是, 以下情况会导致 Early-Z 失效:

- 开启 Alpha Test: 由于 Alpha Test 需要在像素着色器后面的 Alpha Test 阶段比较, 所以无法在像素着色器之前就决定该像素是否被剔除.
- 开启 Alpha Blend: 启用了 Alpha 混合的像素很多需要与 frame buffer 做混合, 无法执行深度测试, 也就无法利用 Early-Z 技术.
- 开启 Tex Kill: 即在 shader 代码中有像素摒弃指令 (DX 的 discard, OpenGL 的 clip)。关闭深度测试。Early-Z 是建立在深度测试看开启的条件下, 如果关闭了深度测试, 也就无法启用 Early-Z 技术。
- 开启 Multi-Sampling: 多采样会影响周边像素, 而 Early-Z 阶段无法得知周边像素是否被裁剪, 故无法提前剔除.

2.3.4 统一着色器架构 (Unified shader Architecture)

在早期的 GPU, 顶点着色器和像素着色器的硬件结构是独立的, 它们各有各的寄存器、运算单元等部件. 这样很多时候, 会造成顶点着色器与像素着色器之间任务的不平衡. 于是, 为了解决 VS 和 PS 之间的不平衡, 引入了统一着色器架构 (Unified shader Architecture). 用了此架构的 GPU, VS 和 PS 用的都是相同的 Core. 也就是说, 同一个 Core 既可以是 VS 又可以是 PS、GS、CS.

2.3.5 渲染模式

不同的渲染模对比参考 [6]:

IMR: 一种设计比较简单的流水线作业多架构方式, PC 端主流. 图元数据获取后 (draw call) 直接渲染 (走一整个流程). 阶段之间, 通过显存进行数据交互, 一帧之中会多次读写 Color/Depth Buffer, 高带宽.

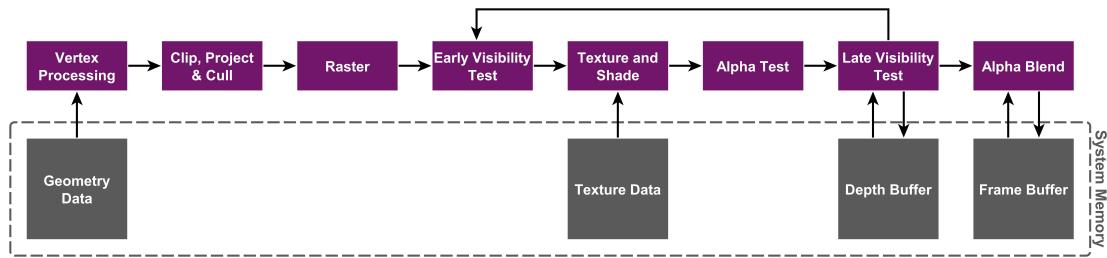


图 25: Immediate Mode Rendering pipeline

TBR: 传统移动端架构渲染模式. 通过延时渲染, 减少过度绘制. 基于 Tile 的渲染, 可将 Color/Depth Buffer 直接放在 On-Chip Memory 中, 减少了主存读写. 满足移动端功耗/发热的要求.

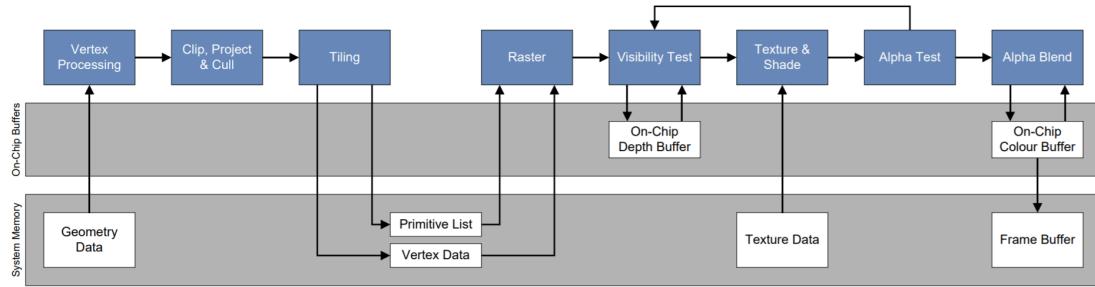


图 26: Tile-Based Rendering (TBR) pipeline

理想的并行渲染, 希望 GPU 内部资源 (SM、Raster) 被充分利用, 每个硬件之间不需要通信. 但渲染过程是有顺序的, 而且在最后阶段需要统筹排序 (无论深度测试还是 Blend 混合). IMR 和 TBR 的不同在于, 前者在最后阶段进行排序, 简单高效. 而后者在中间阶段将 Pipeline 打断, 进行排序, 复杂, 但可以优化使减少无效绘制.

IMR 的 pipeline 畅通无干扰, sorting 简单, TBR 的 sorting 较复杂, 但也给低功耗优化提供了灵活的选择. 另外 TBR/TBDR pipeline 的分割让 pipeline 中断了, 各种 defer, 跟 IMR 比起来, 速度也可能会进一步被影响而变慢.

TBR 用增大 memory resource, 以及 (有可能) 降低 render rate 的代价, 获得降低 bandwidth, power 的效益.

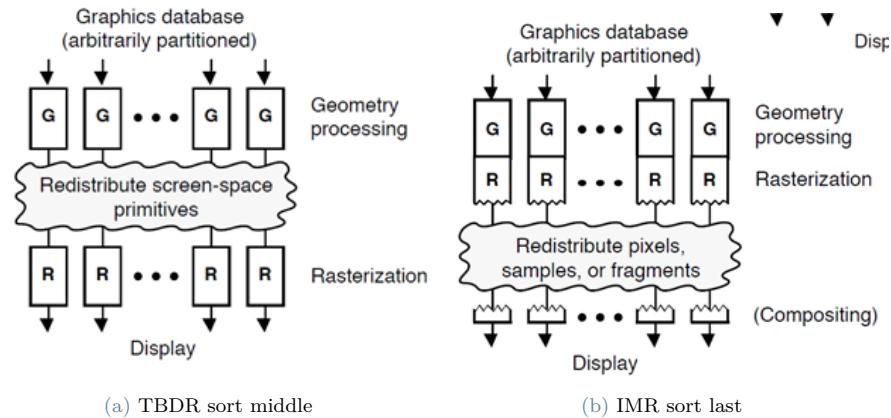


图 27: TBR 与 IMR 渲染过程上的区别 [6]

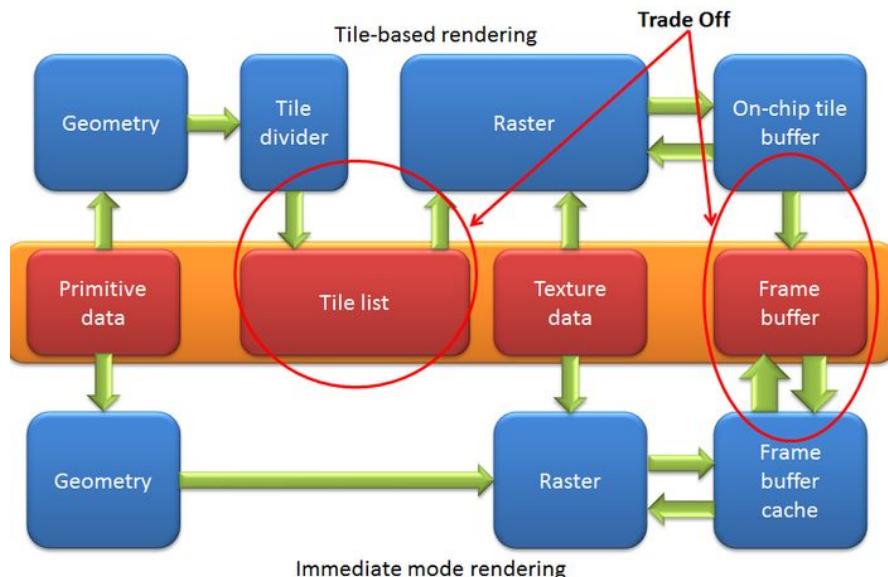


图 28: TBDR、IMR 访存上的区别

TBDR: 现代移动端渲染模式. 加入 HSR, 进一步降低了过度绘制. PowerVR 的专利, Apple 最新的 GPU 也用了 HSR, 应该向 PowerVR 交钱了.

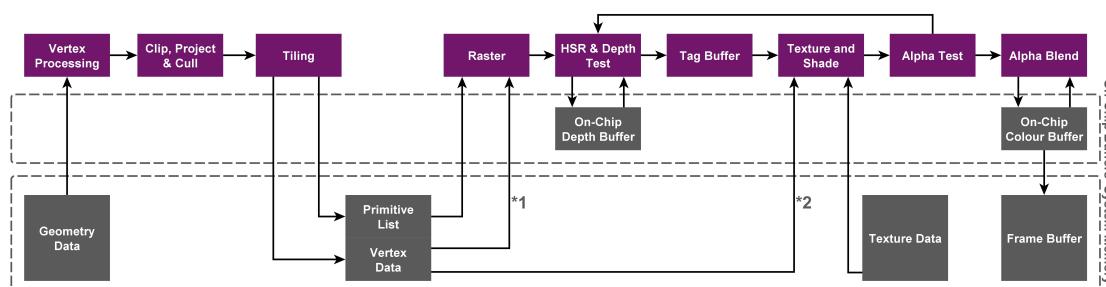


图 29: Tile-Based Deferred Rendering (TBDR) pipeline[4]

虽然，现代 GPU 有 Early-Z 可以剔除遮挡，但是对与深度上的遮挡仍然取决于绘制的顺序。TBDR 通过再一次延迟，让 Early-Z 不再依赖于绘制顺序，最大程度上剔除无效片元。当一个像素通过了 Early-Z 准备执行 PS 进行绘制前，先不画，只记录标记这个像素归哪个图元来画。等到这个 Tile 上所有的图元都处理完了，最后再真正的开始绘制每个图元中被标记上能绘制的像素点。

TBDR 的渲染流程：

- 1. 获取图元数据，进行 Vertex Processing(Vertex Shader)。
- 2. 转换到屏幕坐标系并进行裁剪，Tiling，划分为 32×32 或 16×16 的 tile，并将数据写入主存。
- 3. 当一帧的数据集齐或强制 commit/flush 时，从主存中读取数据，进行光栅化。然后 HSR 和深度测试，丢弃不需要绘制的片元。
- 4. 片元着色。
- 5. Alpha 测试。
- 6. AlphaBlend。

2.4. CPU VS GPU

比喻：

- CPU 敌后特工组织。具有强大/复杂的组织网络 (CPU 控制单元)。特工 (核心) 很强大 (大 Cache, 复杂指令集...), 能完成特别复杂的任务。以任务为单位, 特工 (CPU 核心) 数量不多, 大多时候一个特工 (CPU 核心) 需要并发处理很多任务。
- GPU 正面兵团。组织结构比较简单 (GPU 控制单元), 具有大量的士兵 (核心), 作战以小队为单位 (Warp)。单个士兵能力比较简单, 适合大规模的正面作战 (大数据运算)。

CPUs are optimised to execute large, heavily branched tasks on a few pieces of data at a time. [4]

GPUs are optimised to work on the principle that the same piece of code will be executed in multiple threads, often numbering into the millions, to handle the large screen resolutions of today's devices. [4]

CPU - 多任务并发处理 (具有强大的控制单元, Cache)。GPU - 大数据并行处理 (多核心, 多寄存器)。

	CPU	GPU
并行目标	任务 (Task)	数据 (Data)
核心架构	多线程核心	SIMT 核心
线程数量级别	10	10^4
吞吐量	低	高
缓存需求量	高	低
线程独立性	低	高

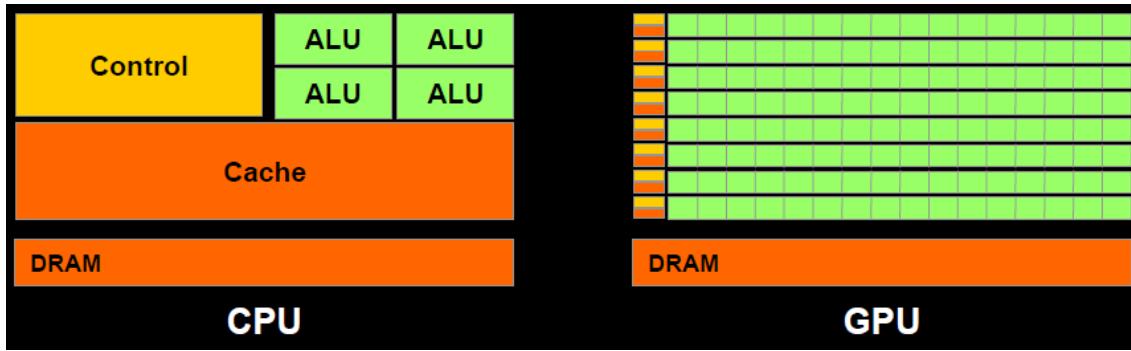


图 30: CPU 和 GPU 硬件上的比较

3. 渲染优化经验/建议

参考《深入 GPU 硬件架构及运行机制》[1]

- 减少 CPU-GPU 数据交换: 合批 (Batch); 减少顶点数、三角形数; 视锥裁剪 (BVH, Portal, BSP, OSP); 避免每帧提交数据 (CPU 版的粒子, 动画每帧修改、提交数据 [可移至 GPU 端]); 减少渲染状态的查询; 启用 GPU Instance; 开启 LOD; 避免从 GPU 读取数据.
- 避免过度绘制: 尽可能使 Early-Z 有效; 开启裁剪 (背面裁剪、视口裁剪、遮挡裁剪); 控制物体数量.
- Shader 代码优化: 避免 if-else, switch, 可变 for 循环; 减少纹理采样次数; 减少复杂数学函数调用.

更多优化经验, 可参考:

- 《移动游戏性能优化通用技法》[2]
- 《PowerVR Performance Recommendations The Golden Rules》[4] 或中文版本 [5]

参考文献

- [1] 0 向往 0. 深入 gpu 硬件架构及运行机制, 2019. URL.
- [2] 0 向往 0. 移动游戏性能优化通用技法, 2019. URL.
- [3] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. 5 edition, 2011. URL.
- [4] PowerVR. Introduction to powervr for developers. URL.
- [5] topameng. Powervr 性能建议-黄金法则, 2018. URL.
- [6] xiaocai. Tile-based 和 full-screen 方式的 rasterization 相比有什么优劣, 2019. URL.
- [7] 丛越. 游戏引擎随笔 0x05: 现代图形 api 讲义, 2019. URL.