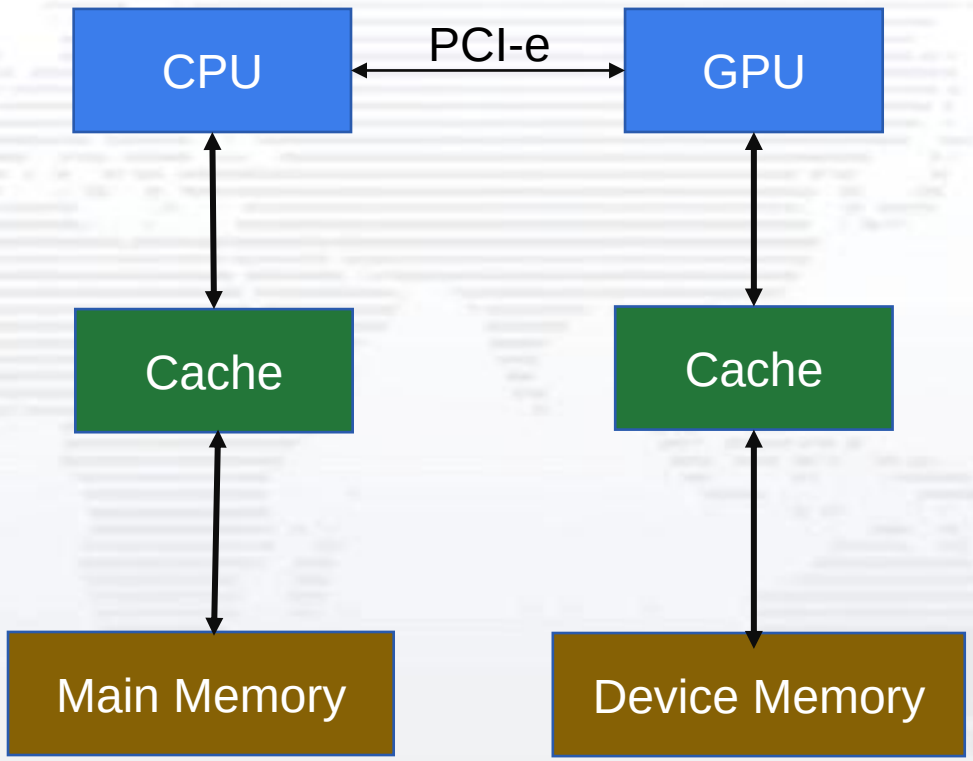
GPU架构与性能优化

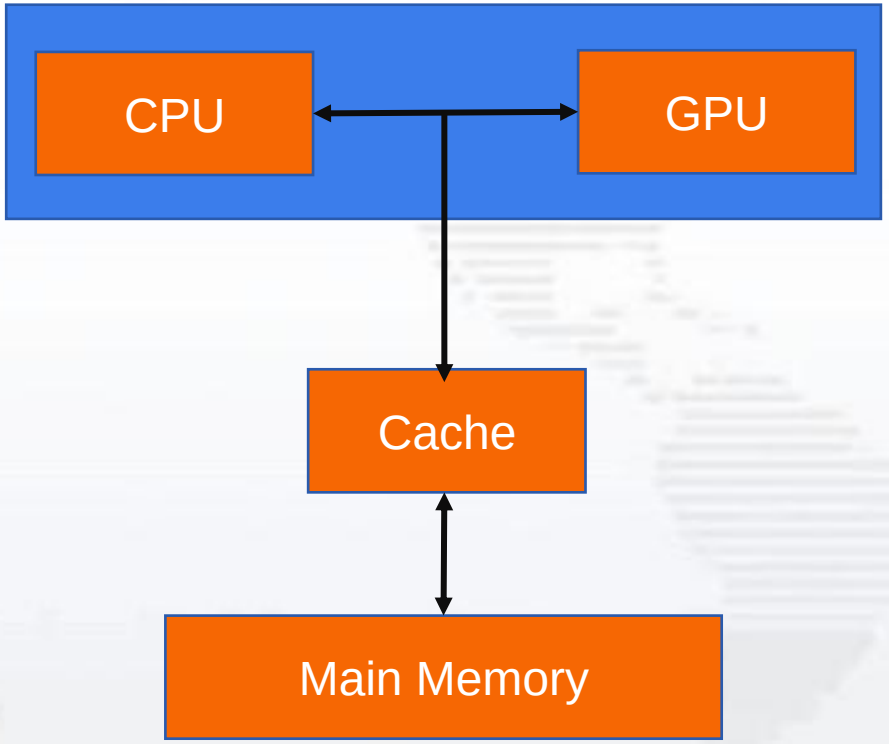
毕胜旺

CONTENTS

基本概念

分离式架构与耦合式架构





基本概念 --- 分离式架构

分离式架构：现代主流设计架构，CPU和GPU各自有独立的缓存和内存，它们通过PCI-E等总线通讯

缺点：

1. PCI-E 相对于两者具有低带宽和高延迟，数据的传输成了其中的性能瓶颈
2. 一般来说分离式结构中 CPU 和 GPU 都各自拥有独立的内存，两者共享一套虚拟地址空间，必要时会进行内存拷贝
3. 浪费带宽，发热严重
4. 功耗高

优点：

1. 设计简单直接
2. CPU和GPU各自有独立的缓存和内存
3. 扩展性好

基本概念 --- 耦合式架构

耦合式架构：游戏主机上使用，GPU 没有独立的内存，与 CPU 共享系统内存，由 MMU 进行存储管理(AMD APU)

缺点：

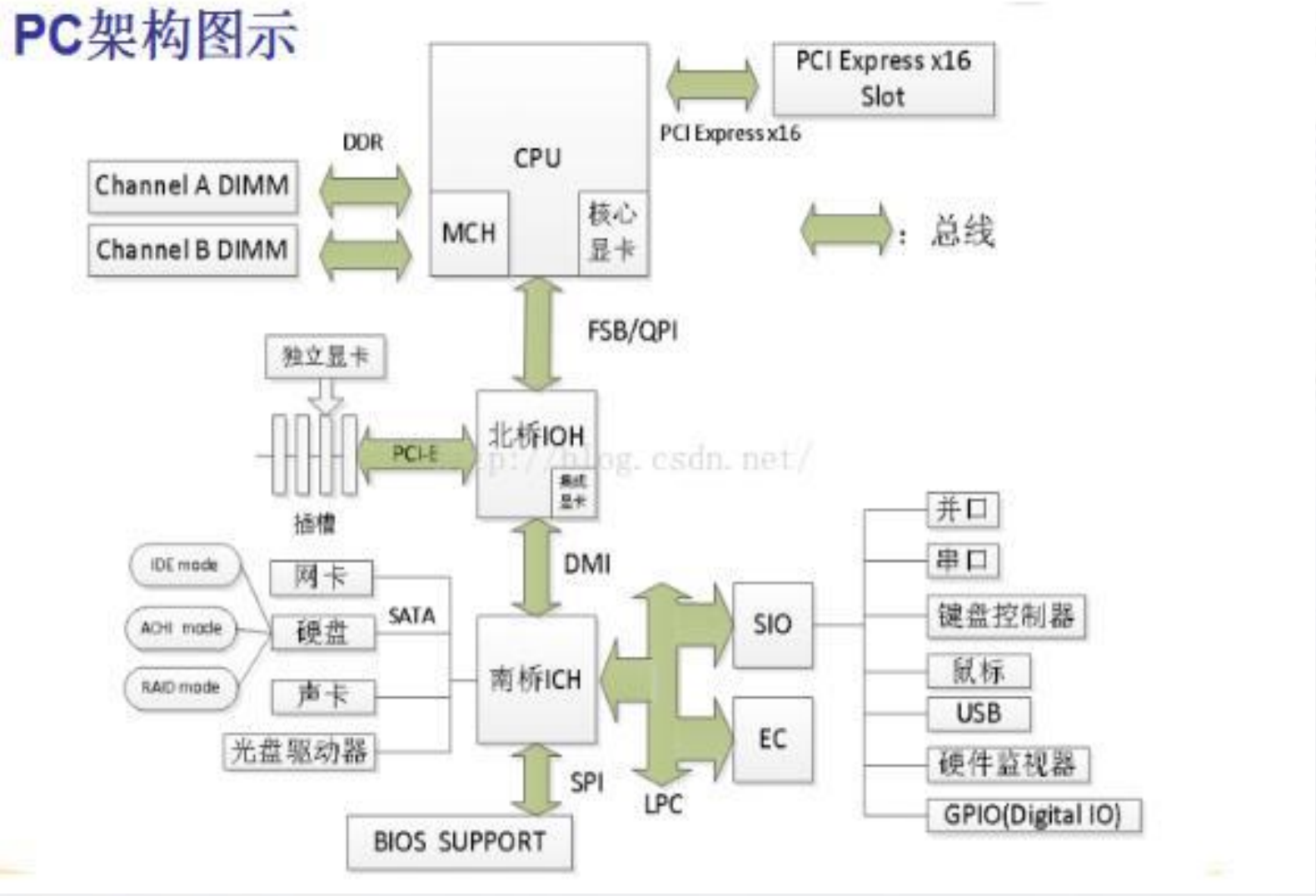
1. 设计复杂
2. 满载会导致CPU降频
3. 内存有限，运算量受阻，做不了大型的渲染

优点：

1. 在小型渲染场景情况下，功耗低
2. 双+通道及高频，性能优秀

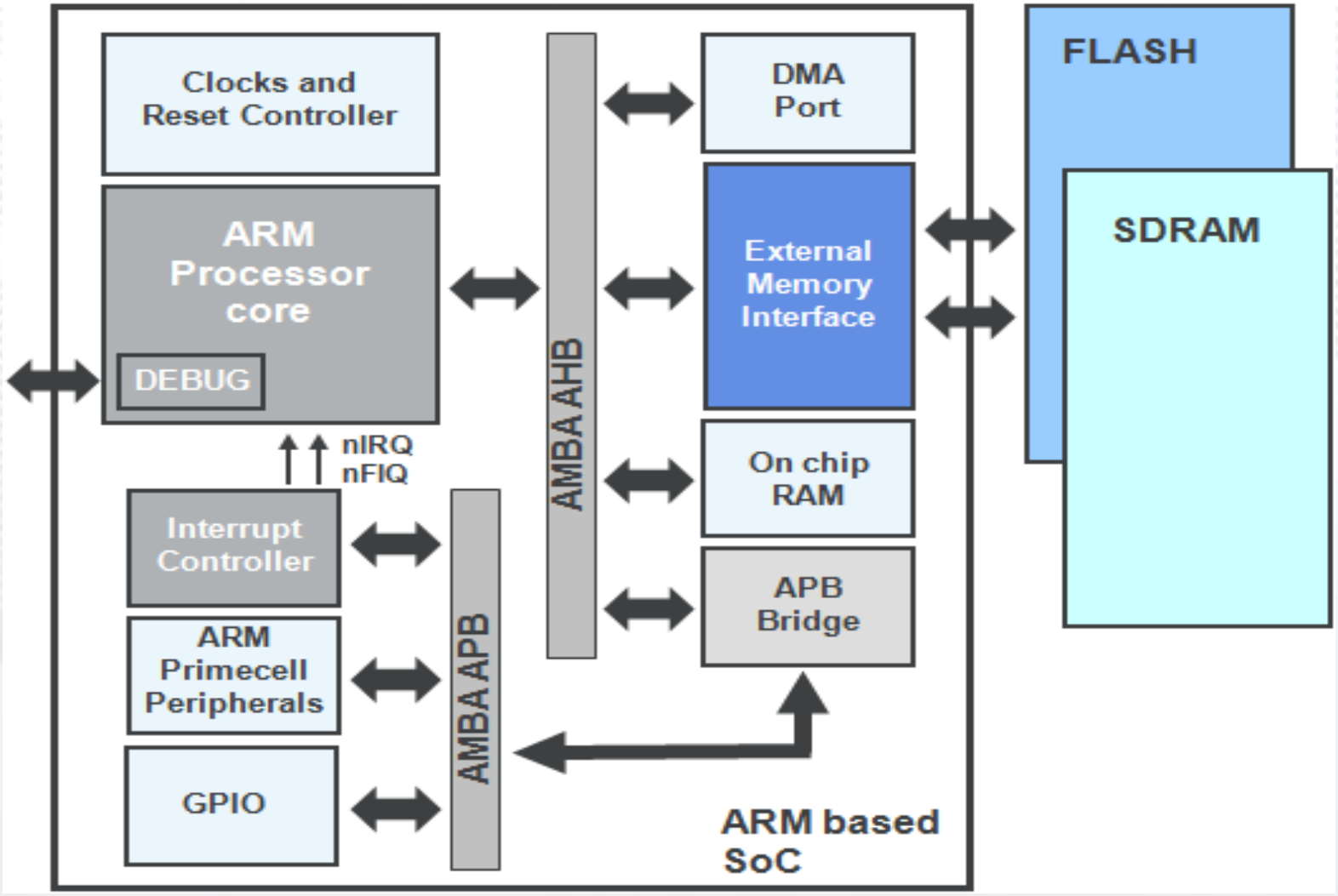
基本概念 --- x86架构

x86架构： x86架构采用分离式架构，采用PCI-e总线进行数据交换传输

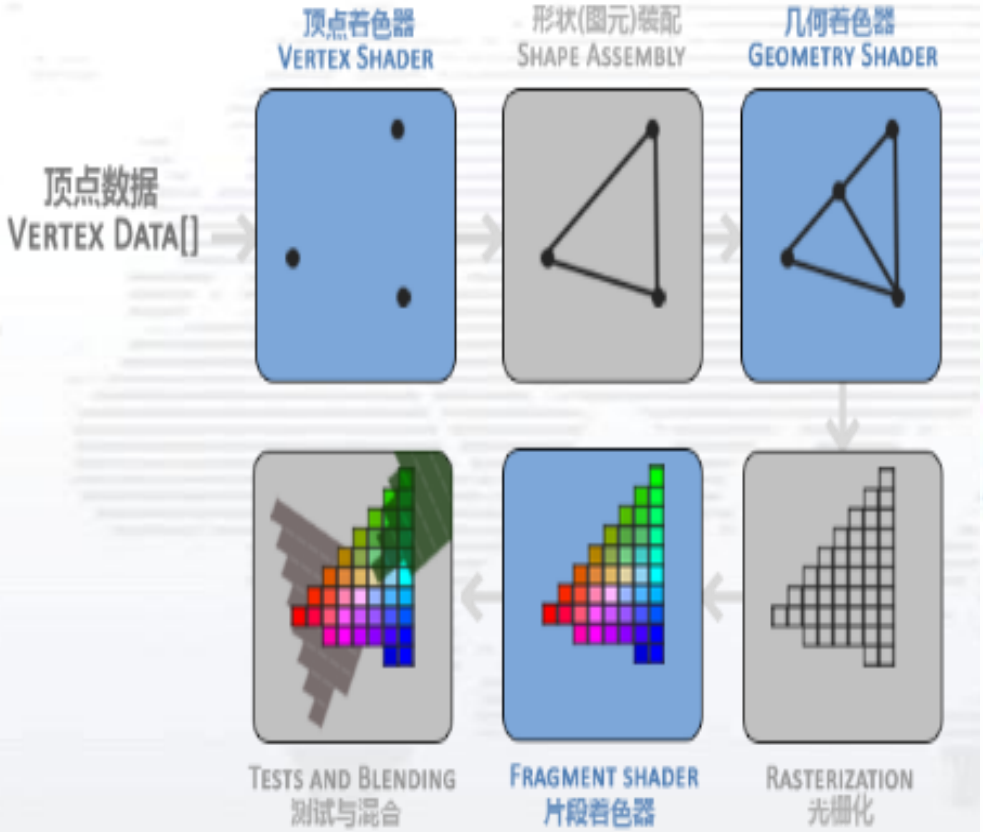


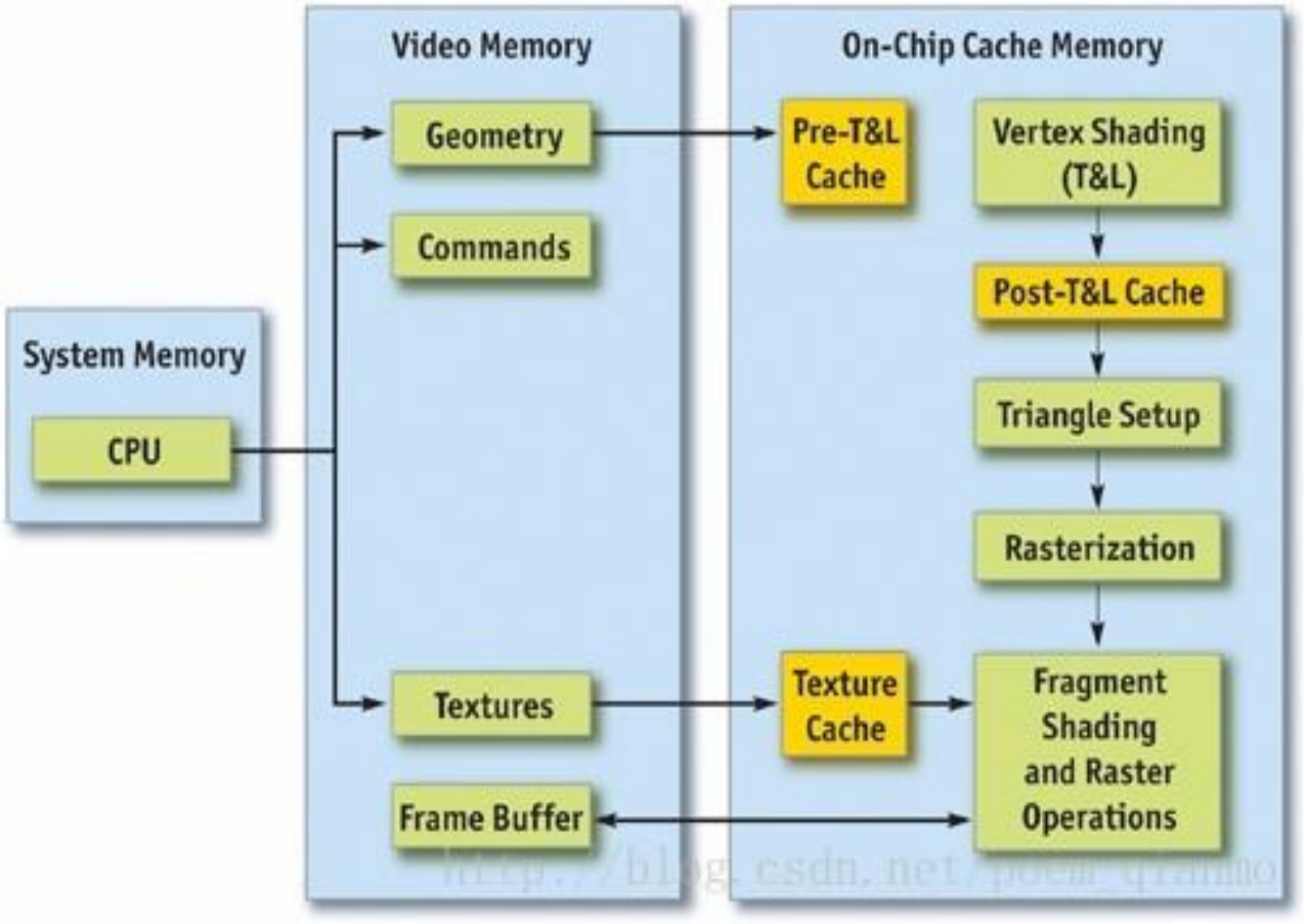
基本概念 --- ARM架构

ARM架构： arm架构也是采用分离式架构，只是数据交换和传输的总线等都做到了SOC上



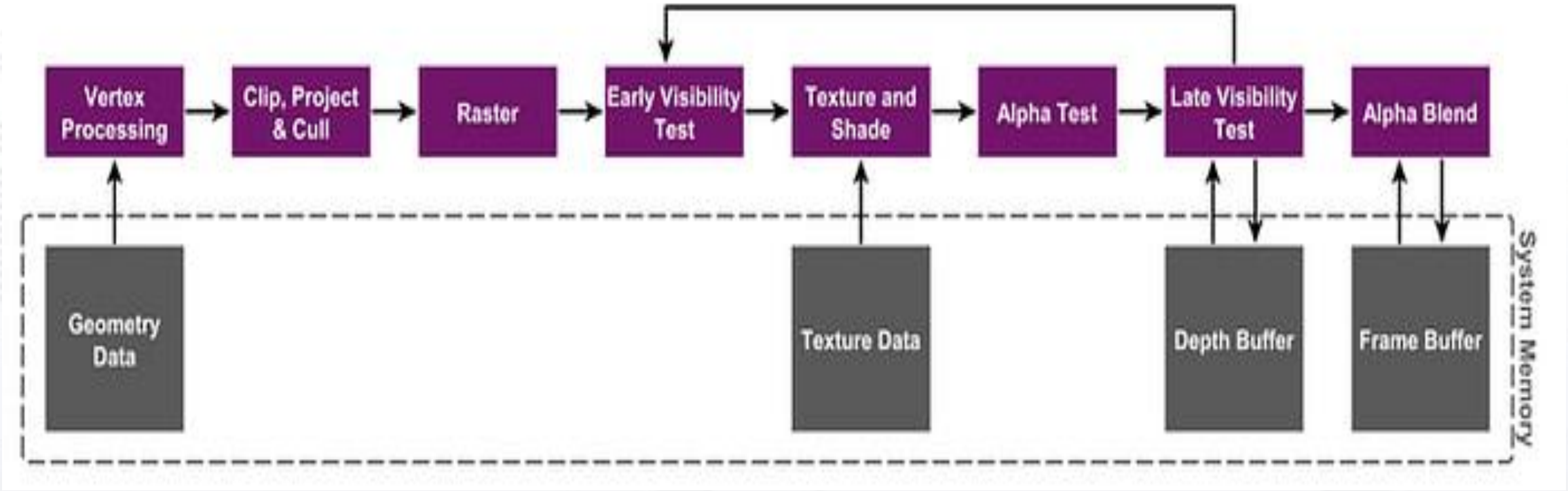
图形渲染管线


顶点着色器
VERTEX SHADER
形状(图元)装配
SHAPE ASSEMBLY
几何着色器
GEOMETRY SHADER
顶点数据
VERTEX DATA[]
TESTS AND BLENDING
测试与混合
FRAGMENT SHADER
片段着色器
RASTERIZATION
光栅化

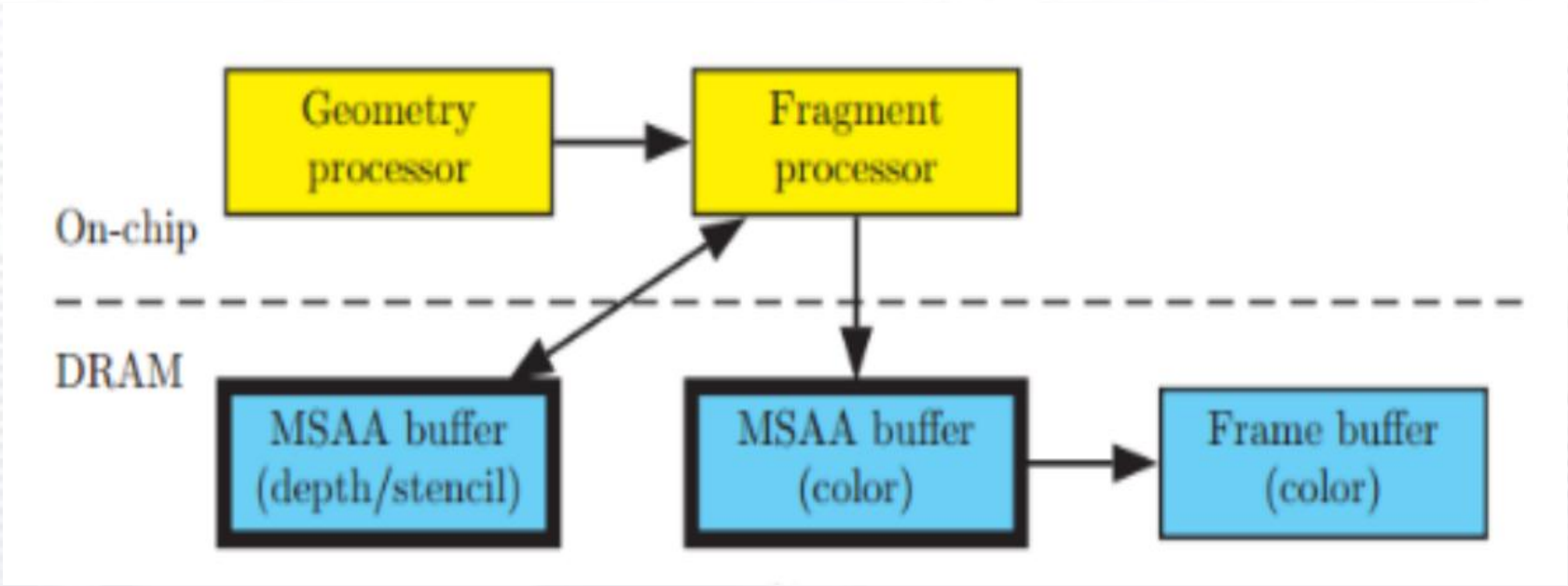

Video Memory
On-Chip Cache Memory
Geometry
Commands
Textures
Frame Buffer
System Memory
CPU
Pre-T&L Cache
Vertex Shading (T&L)
Post-T&L Cache
Triangle Setup
Rasterization
Texture Cache
Fragment Shading and Raster Operations

IMR架构

GPU IMR架构： Immediate Mode Rendering，立即渲染模式



IMR架构


Geometry processor
Fragment processor
On-chip
DRAM
MSAA buffer (depth/stencil)
MSAA buffer (color)
Frame buffer (color)

IMR架构

特点：

1.每一个绘图的指令来到显卡，显卡便立即执行，从头到尾跑完整个管线，渲染管线里的读写操作都是直接在显存和GPU中传输数据的，最终将结果输出到Frame Buffer中

2.每一次渲染完的Color和Depth以及Stencil模板数据写回到Frame Buffer和 Depth Buffer以及Stencil Buffer都会产生很大的带宽消耗

3.开启深度测试或模板后，每个fragment的输出都要和Depth Buffer或Stencil Buffer中的深度值或模板值进行测试，如果通过测试则需要更新Depth Buffer或Stencil Buffer和Frame Buffer

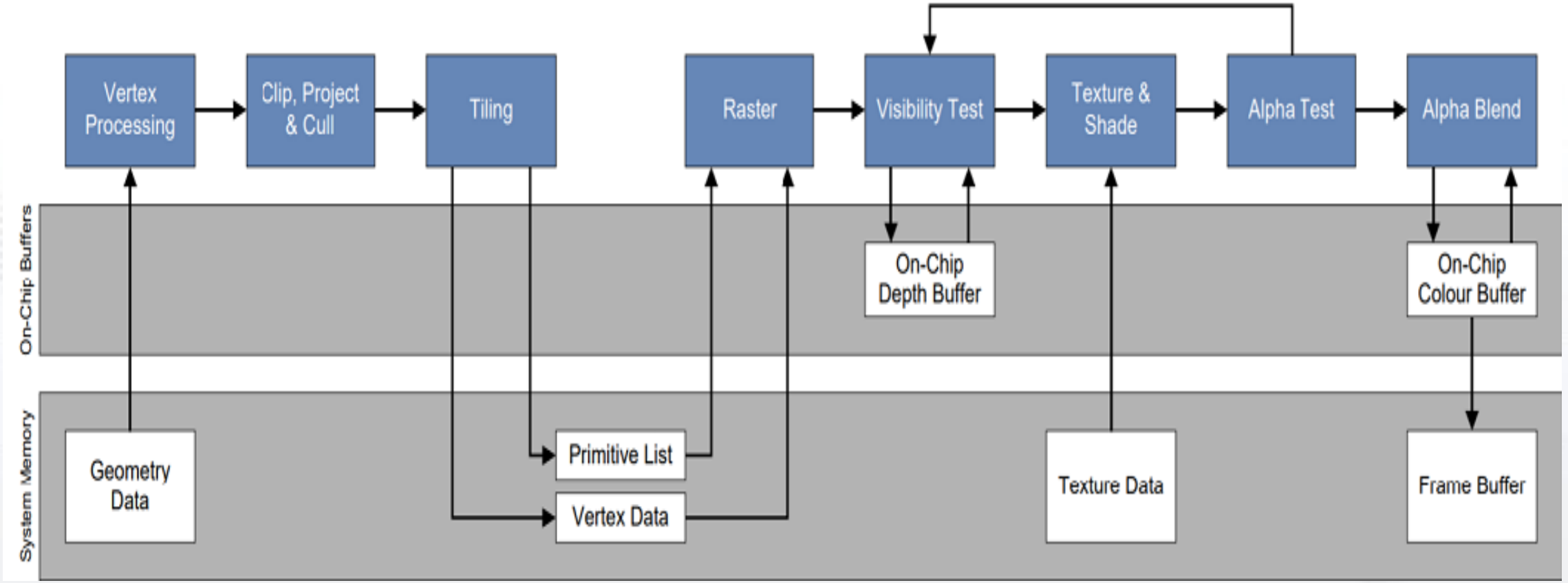
4.对System Memory的一次读取和两次写入，而fragment的数量巨大，这样就带来了很大的访问System Memory的压力

5. 执行空间DRAM空间足和带宽消费大

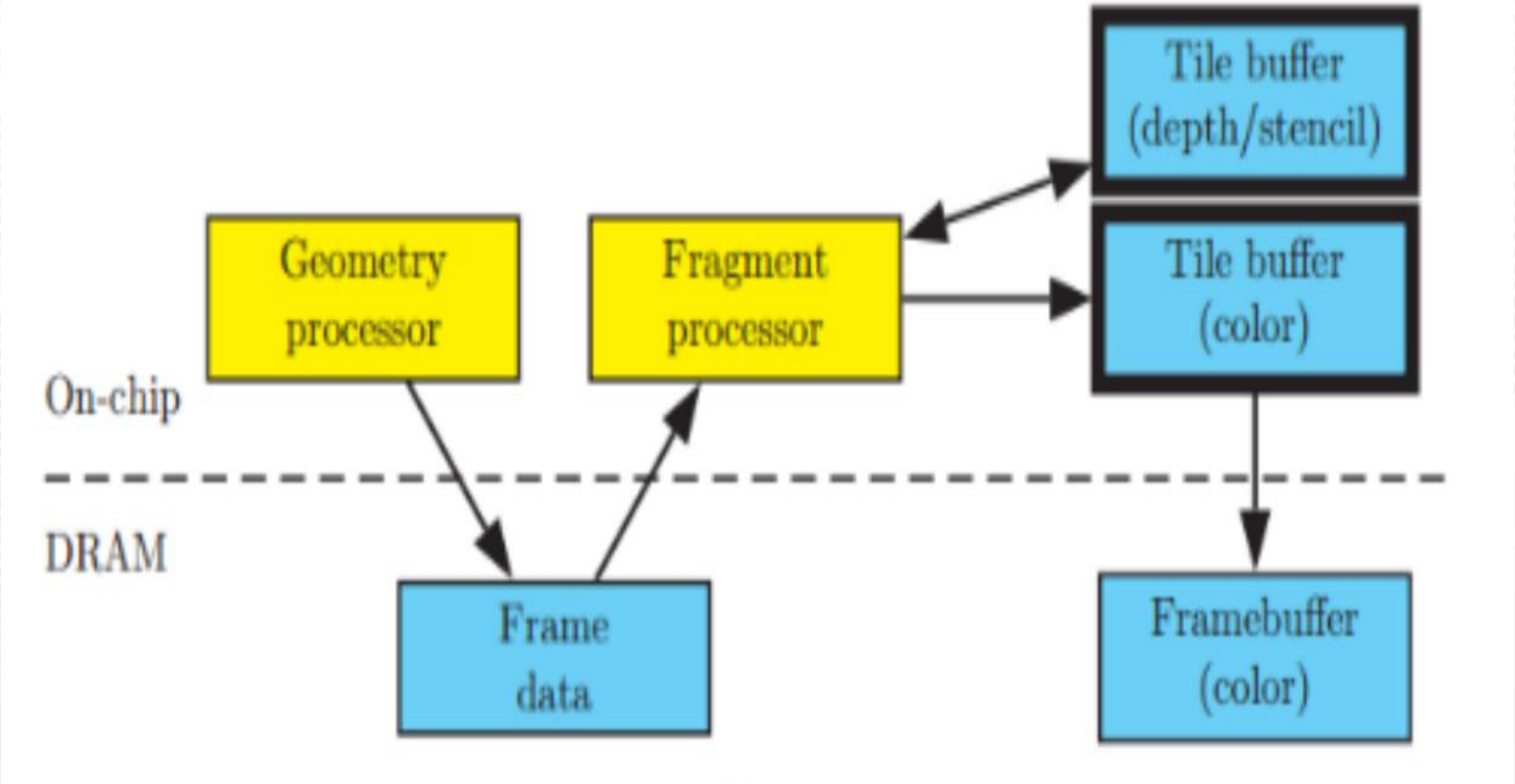
6. 功耗高，发热严重

TBR架构

GPU TBR架构： Tile Based Rendering，基于瓦片的渲染模式



TBR架构


Tile buffer
(depth/stencil)
Geometry
processor
Fragment
processor
Tile buffer
(color)
On-chip
DRAM
Frame
data
Framebuffer
(color)

TBR架构

特点：

1.高速缓存Tile Memory 存储计算的FrameData

2. Tile大小一般 16*16 、32*32 多个tile片段并行处理

3.一个绘图指令抵达显卡时，将通过vertex shader和裁剪后的顶点数据，根据所在tile分组，将分组数据写入 DDR(System memory)中，即PB(Primitives list & vertex data)

4.并行处理所有tile分组，在每个tile上执行叠加的所有绘制命令，执行结果写入PB

5.当所有绘制指令的顶点数据都做好处理存进PB或是PB达到一定容量之后才开始进行管线的下一步，显卡会以tile为单位从PB中取回相应的顶点数据，进行光栅化、fragment shader以及逐片元处理

6. 无需频繁访问system memory，全部访问在On-Chip memory(L1/L2 cache)上

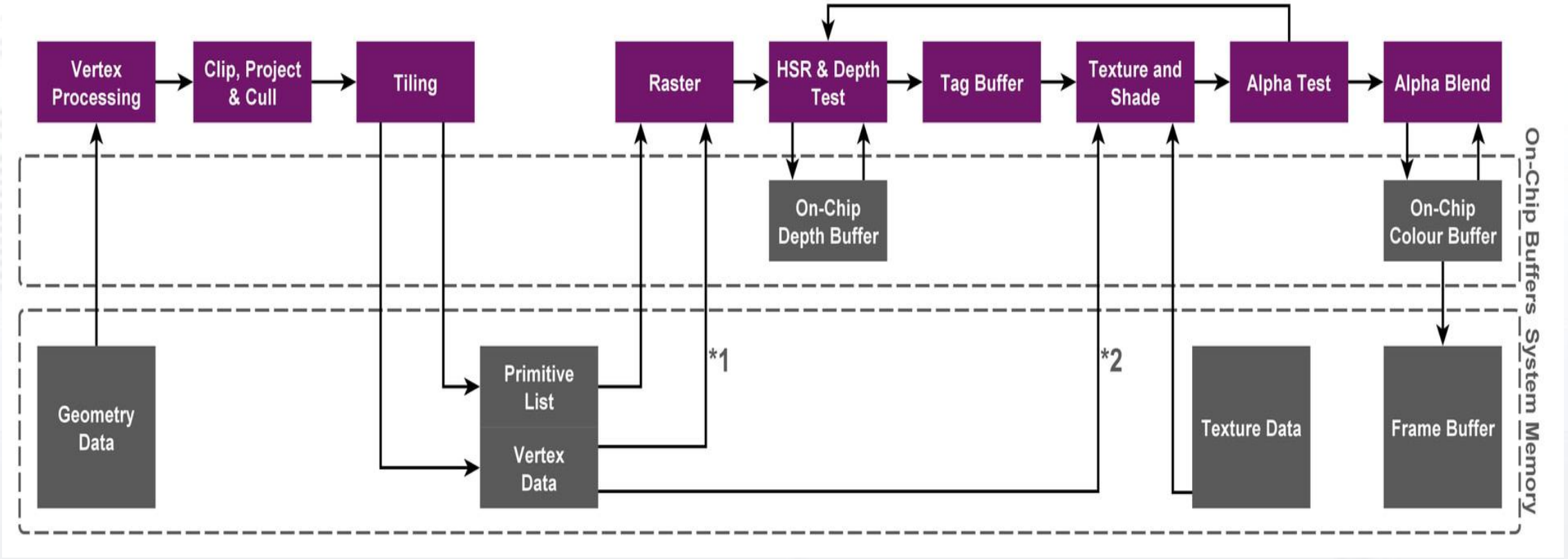
7.渲染管线中断时，处理麻烦

8.节省带宽， 功耗低

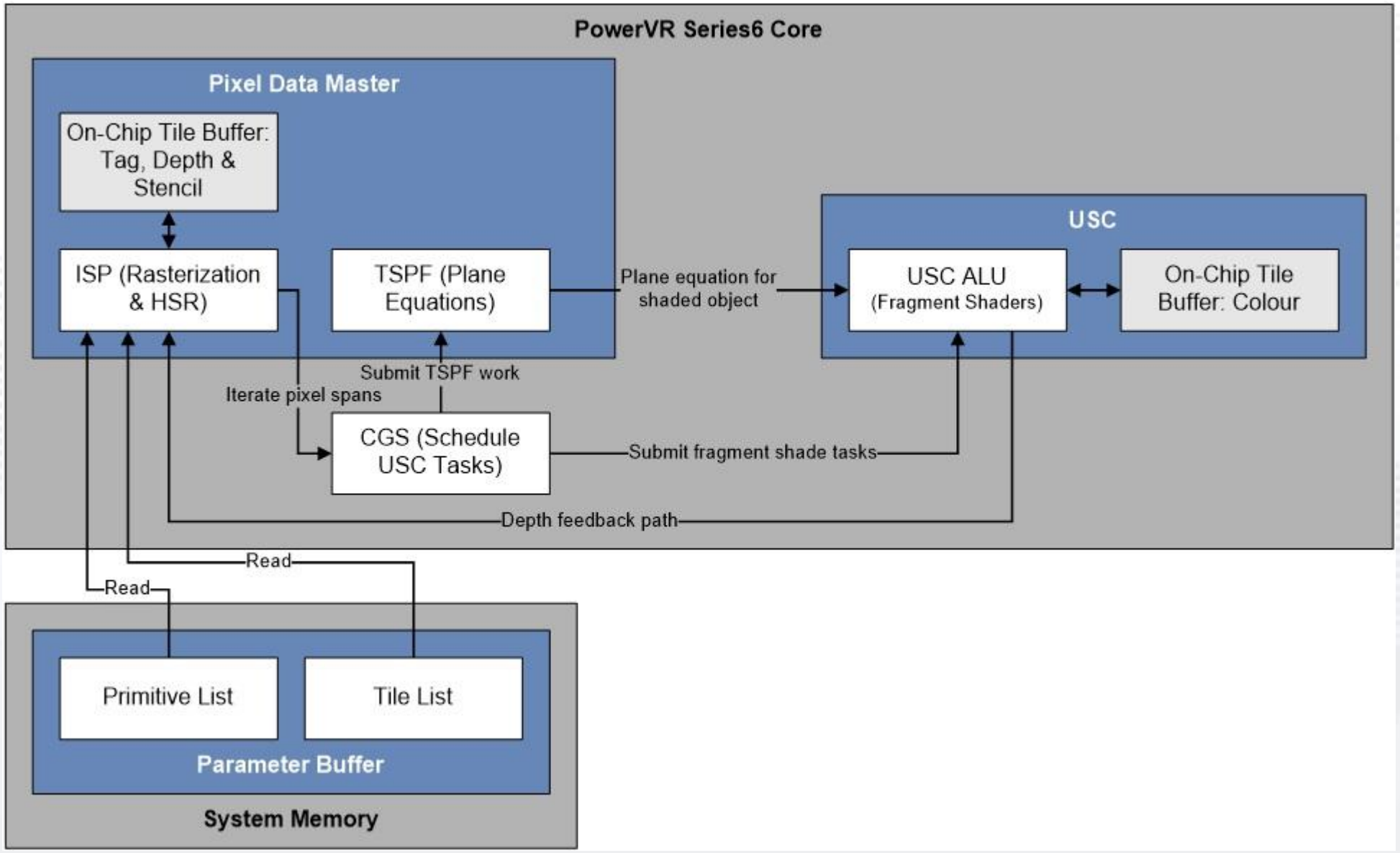
9. 整体上，渲染没有及时执行，而是缓存FrameData，优化空间大

TBDR架构

GPU TBDR架构： Tile Based Deferred Rendering，基于瓦片的延迟渲染模式



TBDR架构


PowerVR Series6 Core
Pixel Data Master
On-Chip Tile Buffer: Tag, Depth & Stencil
ISP (Rasterization & HSR)
TSPF (Plane Equations)
Plane equation for shaded object
USC
USC ALU (Fragment Shaders)
On-Chip Tile Buffer: Colour
Submit TSPF work
Iterate pixel spans
CGS (Schedule USC Tasks)
Submit fragment shade tasks
Depth feedback path
Read
Read
Primitive List
Tile List
Parameter Buffer
System Memory

TBDR架构

特点：

1.具有TBR的所有特点

2. 避免过度渲染(overdraw)

3. ISP对PB顶点数据进行差值并对差值得到的片元数据计算深度，并进行深度和模板测试

4. 存储可见图元到Tag buffer， 取可见图元的其他数据(uv/depth/stencil)，进行fragment shader

5. 对fragment shader中的discard片元无法得知是否绘制，会进行提前fragment shader反馈结果到ISP，阻塞其他片元计算

6. Alpha混合运算，会强制绘制缓存，退化为IMR

IMR/TBR/TBDR原理分析

一、IMR/TBR/TBDR都是分离式架构

IMR：每一个primitive(点、线、三角面等)都会直接提交渲染，渲染管线并行执行，速度非常快。并且读写是直接对DRAM（显存）上进行操作，DRAM上读写的速度是最快的。其缺点是很多需要对当前帧进行读取比较的（比如blending, depth testing 或者stencil testing）都需要频繁从framebuffer中读取数据，这样就会产生过高的带宽压力，而带宽压力也意味着功耗上升

PC端考虑以性能效率为主，功耗为次，执行效率高，每帧绘制渲染指令立即执行，GPU直接访问显存，不会缓存，GPU与DRAM的数据交换传输频率高，带宽消耗大

IMR/TBR/TBDR原理分析

二、移动端采用TBR/TBDR架构，以空间换时间，减少大量频繁的GPU访问显存DRAM，节省带宽，降低功耗，小缓存On-Chip memory(L1/L2 cache)在GPU内部，支持Tile的计算，其原理还是以空间换时间的思维

三、TBR/TBDR架构整体渲染流程

1. 缓存所有绘制指令直到最后才开始进行真正绘制

2. 首先对所有几何顶点执行VS，把相关的所有绘制数据保存起来

3. 计算每个tile都包含哪些图元

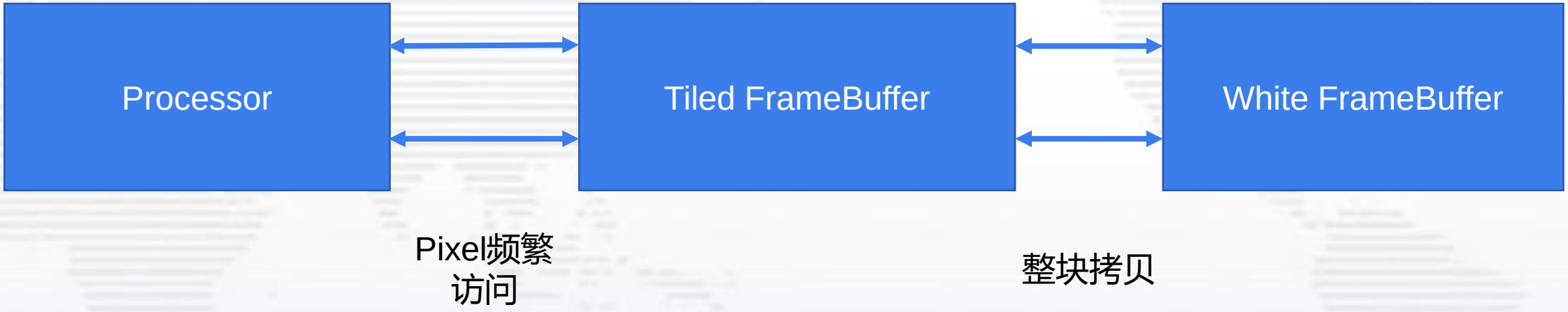
4. 开始绘制一个tile

5. 依次绘制tile中的每个图元

6. 把绘制好的tile拷贝到framebuffer对应的位置上

IMR/TBR/TBDR原理分析

TBR/TBDR渲染原理图



IMR/TBR/TBDR原理分析

TBR/TBDR架构，对于一个FBO（帧缓冲），先把FBO拆成许多小的tile（通常16x16和32x32,由SRAM决定），再执行所有的vertex shading，组成图元，生成Triangle List 并保存在FrameData(powerVR叫parameter buffer,arm叫plolygon lists)，再把FrameData储存在物理内存。最后需要刷新整个FrameBuffer的时候（如API：Swap Back and Front Buffer，glflush，glfinish，glreadpixels，glcopyTexImage，glBitFrameBuffer，queryingocclusion，unbind the framebuffer），从物理内存读取每个tile然后进行光栅化和fragment shading。全部tile完成之后把整个FBO写入到显存中

IMR/TBR/TBDR原理分析

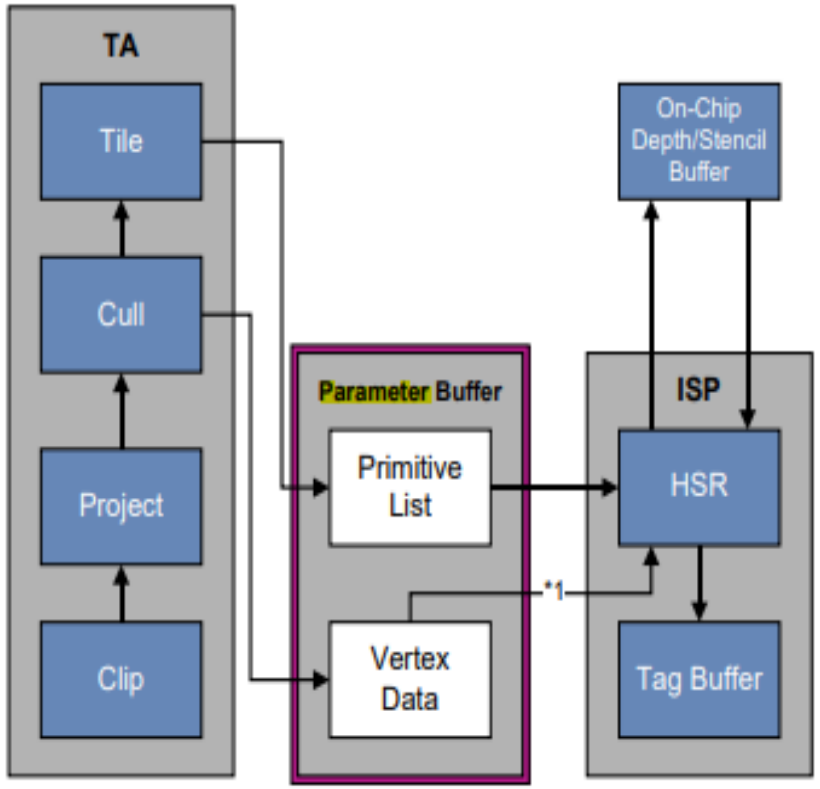


Figure 5. Parameter Buffer

TBDR架构的HSR(Hidden Surface Removal)

IMR/TBR/TBDR原理分析

TBR没有解决Overdraw的问题，TBR的设计主要是减少IMR的带宽开销

EarlyZ，提前深度测试，其实就是当不透明的图元从光栅化阶段开始逐像素进行处理时，首先进行Depth Read & Test，通过后直接写入深度，后续再执行该像素上的着色，否则就可以停下来休息等待下一个要处理的像素

基于EarlyZ是无法完全的避免overdraw

IMR/TBR/TBDR原理分析

TBDR是PowerVR在TBR的基础上加入了HSR&tag Buffer，其目的是为了在硬件层面彻底解决EarlyZ，避免overdraw, 且它不关心软件层面的物体排序

当一个像素通过了EarlyZ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。这样每个像素上实际只执行了最后通过EarlyZ的那个PS，而且由于TBR的机制，Tile块中所有图元的相关信息都在片上，可以极小代价去获得。最终零overdraw

架构不同，渲染管线执行的机制不同，优化策略不同

性能优化 --- IMR

IMR的渲染流程特点是每帧都会立即执行，每个像素的渲染都对DRAM进行高速访问，不存在延迟渲染，每帧都会对framebuffer进行全部更新，所以针对这种渲染架构，优化的方向主要是减少每帧的计算量

1. 在效果评估满足的情况下，降低模型精度
2. 在效果评估满足的情况下，降低纹理分辨率
3. 尽量采用压缩纹理，开启mipmap
4. Fragment shader精度
5. 无需执行glClear，在某些GPU上可提高性能

性能优化 --- TBR

1. 尽量减少vertex shader计算量，因为每个tile的triangle list需要存到物理内存，fragment shading又要从物理内存里面读回

2. 减少一帧内的渲染次数， 避免一帧内的前后渲染结果依赖。因为一帧数内有>=2的渲染，且后面的渲染需要前面的渲染的结果。TBR需要所有的图元执行完毕之后再执行frag

3. 渲染前对渲染物体进行排序，避免overdraw

4. 不适用framebuffer的时候，要clear或discard，每个tile在初始化的要从DRAM对framebuffer对应的数据拷贝过来，减少不必需要的读操作，节省带宽

性能优化 --- TBR

5. EarlyZ/ZPrePass斟酌开启关闭，一些低端安卓机器drawcall敏感，开启ZPrePass会更加加重负担

6.同一帧内减少提交给GPU资源的改动。如VBO提交给GPU被先保存到framedata。这时候这个VBO被改变了，又提交给GPU进行渲染。GPU会对这个VBO生成新的数据保存到framedata

7.不要在一帧里面频繁的切换framebuffer

8.尽量不要使用LUT纹理

9. 尽量使用压缩纹理&开启mipmap

性能优化 --- TBDR

1. 避免大量的drawcall和顶点量，因为每个tile的triangle list需要存到物理内存，fragment shading又要从物理内存里面读回

2. 减少一帧内的渲染次数， 避免一阵内的前后渲染结果依赖

3.记得不使用Framebuffer的时候clear或者discard,节省很多不必要的绘制

4.在每帧渲染之前尽量clear

5.不要在一帧里面频繁的切换framebuffer，即Swap Back and Front Buffer，glflush，glfinish，glreadpixels，glcopytexiamge，glbitframebuffer，queryingocclusion，unbind the framebuffer慎用

性能优化 --- TBDR

6.同一帧内减少提交给GPU资源的改动。如VBO提交给GPU被先保存到framedata。这时候这个VBO被改变了，又提交给GPU进行渲染。GPU会对这个VBO生成新的数据保存到framedata

7. EarlyZ/ZPrePass斟酌开启关闭，可能会增加CPU的负担

8. 尽量以Alpha blend代替alpha test， blending和MSAA的效率其实很高，alpha-test效率很低

9. 避免gpu上的copy-on-write，会增加framedata的数据量，加大计算

10. 尽量不要使用LUT纹理

11.尽量使用压缩纹理&开启mipmap

性能优化 --- 总结

1. 以上IMR/TBR/TBDR的很多优化策略都是可以通用的，比如降低模型精度、降低纹理分辨率、压缩纹理&mipmap以及计算精度，尽量避免使用LUT等

2. TBR/TBDR大部分优化策略都是通用的，不同点主要在渲染排序与alpha-blending与alpha-test，PowerVR的TBDR架构在硬件上解决了EarlyZ，解决避免overdraw的问题，但alpha-test会使其退化成IMR，尽量使用alpha-blend

3. 无论是TBR架构还是TBDR架构，渲染顺序推荐：opaque----→alpha-test---→blending, 这样alpha-test不会阻塞不透明物体的zpass剔除.

4. 将Opaque, AlphaTest与AlphaBlend打乱顺序渲染会极大的降低渲染性能，任何情况下都不应该这么做

5. 不要尝试使用AlphaTest替代AlphaBlend，这并不会产生太多优化

6. TBR/TBDR的Tile Memory使得以前的多个渲染pass变成一个pass就可以执行

7. 随机纹理寻址相对于相邻纹理寻址会增加开销

性能优化 --- 总结

8. 3DTexture Sampling会显著增加开销

9. Trilinear/Anisotropic相对于Bilinear有显著的开销，纹理尽量采用临近采样

10. 通道图能合并就合并，减少Shader中贴图采样次数

11. 控制Framebuffer大小

12. 不要尝试使用AlphaTest替代Opaque，这会产生负优化

13. 不要尝试使用AlphaBlend替代AlphaTest，这会造成错误的渲染结果

14. 在保证正确渲染顺序情况下，AlphaTest与AlphaBlend开销相似，不存在任何替代优化关系

15. 增加少量顶点以减少AlphaTest图元的绘制面积是可以提升一些渲染性能的

16. 首先统一绘制AlphaTest图元的DepthPrepass，再以ZTest Equal和不含discard指令的Shader统一绘制AlphaTest图元，大多数情况下是可以显著提升总体渲染性能的

扩展 --- PFK & LRZ

1. PFK是ARM的Mail TBDR架构，功能对标PowerVR的HSR，也是减少overdraw的硬件优化

2. 它是和Early-Z共存互补的，不过Early-Z是从前到后绘制，而FPK是从后往前绘制。原理是通过Early-Z测试的quad(2x2像素，Fragment Shader以quad为单位)会进入FIFO buffer Queue（FPKQ，可以容纳256个）。当Quad的4个像素全部被覆盖时(Fully Coverage)，后进入FPKQ的quad被标记为to kill的，它会kill掉相同位置的先进入FPKQ的quad。被kill掉的quad就不会产生Fragment Thread进行Fragment Shader

3. 限制：在渲染Opaque的时候不能有Alpha Blend和Alpha Test。如果要在渲染Opaque的时候非要有Alpha Test的话，则要做好排序

4. 高通（Qualcomm）的Adreno在5系列之后也添加外置LRZ模块进行优化。在Adreno5系列之前的GPU从前往后绘制是可以提升性能，但是Adreno5系列之后前后顺序不会造成影响。原理是在正常渲染前，先多执行一次Vertex Shader生成低精度depth texture，从而提前对不可见的进行剔除，即ZPrePass。做LRZ时执行VS只需用到position信息，所以单独抽出position stream，能带来bandwidth和cache的优化

扩展 --- 性能优化资料

1. 花屏没有glclear：https://blog.csdn.net/ZCMUCZX/article/details/79247691
2. https://www.nvidia.com/docs/IO/8230/GDC2003_OGL_Performance.pdf
3. https://www.khronos.org/opengl/wiki/Performance
4. https://www.inf.pucrs.br/flash/tcg/aulas/opt/opengl_perf_opt.html
5. https://www.opengl.org/pipeline/article/vol003_8
6. https://developer.arm.com/documentation/dui0555/b
7. https://blog.csdn.net/MyArrow/article/details/17583559
8. https://cloud.tencent.com/developer/article/1370101