Analysis of Unreal Rendering System (12) - Mobile Special Topic Part 2 (GPU Architecture and Mechanism)

Table of contents

12.4 Key Points of Mobile Rendering Technology

During this period, I have studied the papers on mobile terminals from Siggraph and GDC in recent years, and consulted the development guidelines of mobile GPU manufacturers such as Qualcomm, Arm, PowerVR and some mobile device manufacturers. In this chapter, I will summarize the common dedicated rendering technologies on mobile terminals.

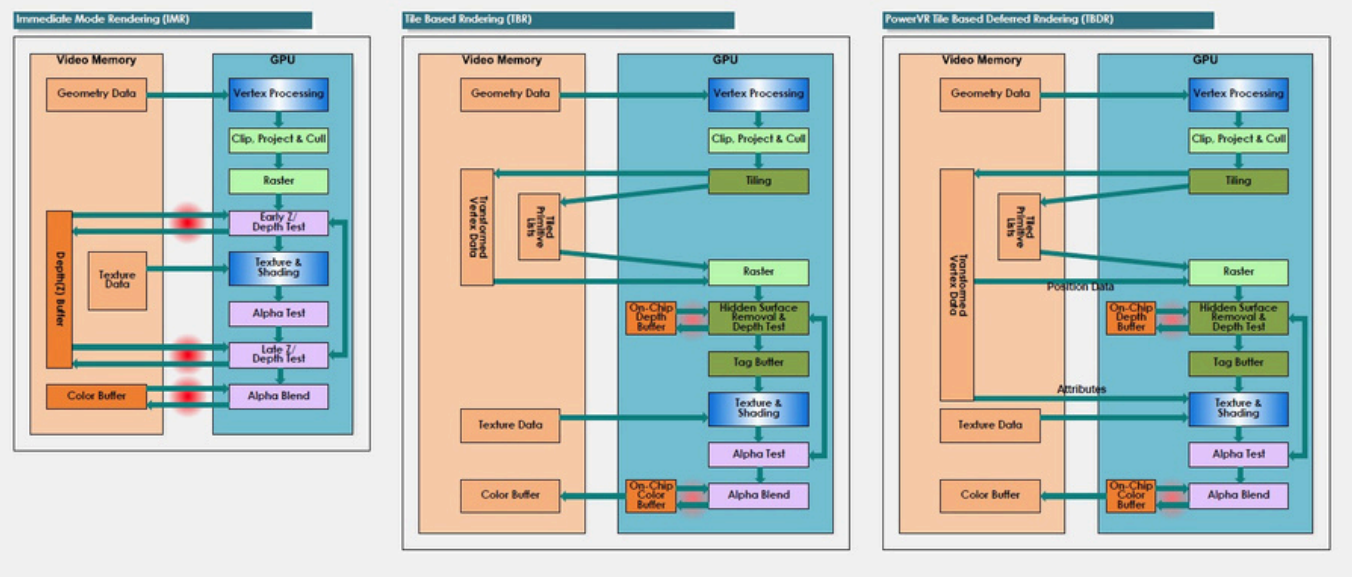
> See the reference list below for details.

12.4.1 Tile-based (Deferred) Rendering

TBRstands for**Tile-based Rendering**,which is a widely used technology in mobile GPU architecture to speed up rendering and reduce bandwidth and energy consumption.

TBDRstands for**Tile-based Deferred Rendering**,an improved version of TBR, which means blockbased deferred rendering. It was first applied to GPU chips by PowerVR. Its most significant difference is that pixels that pass the Early-Z test will not execute the pixel shader immediately, but will first mark which primitive the pixel belongs to. When the Tile has processed all primitives (all objects in the scene), all marked pixels on the Tile will be drawn. TBDR achieves hardware-level occlusion pixel culling, reduces OverDraw, and reduces bandwidth and memory access. PowerVR's TBDR captures the entire scene before rendering begins, so that occluded pixels can be identified and culled before being applied to the pixel shader. Each tile is rasterized and processed individually, and the small size of the render allows all data to be kept in very fast tile memory.

Corresponding to TB(D)R is the**Immediately Rendering (IMR)**mode for PC. The comparison chart of IMR, TBR, and TBDR architectures is as follows:

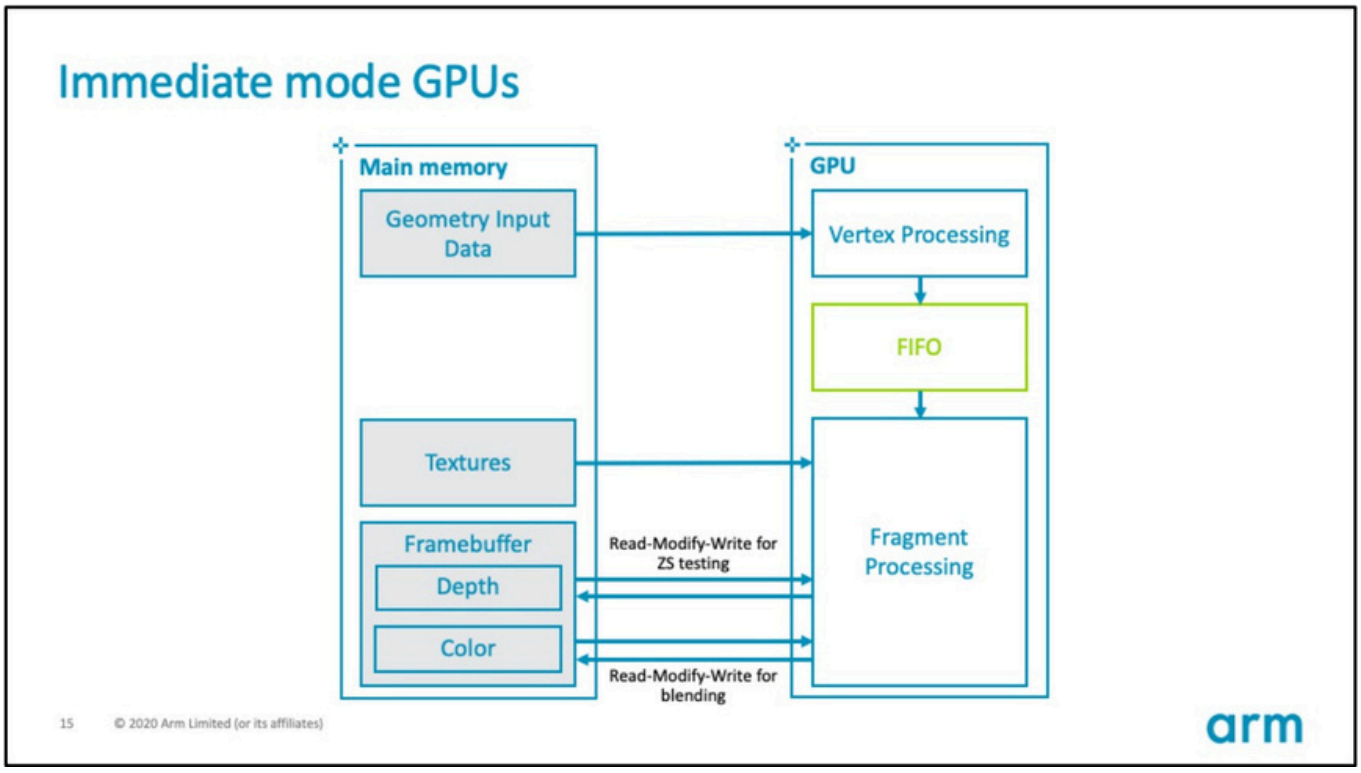


Schematic diagram of IMR, TBR, and TBDR architecture operation. The red ellipse indicates high bandwidth, which will cause performance bottlenecks.

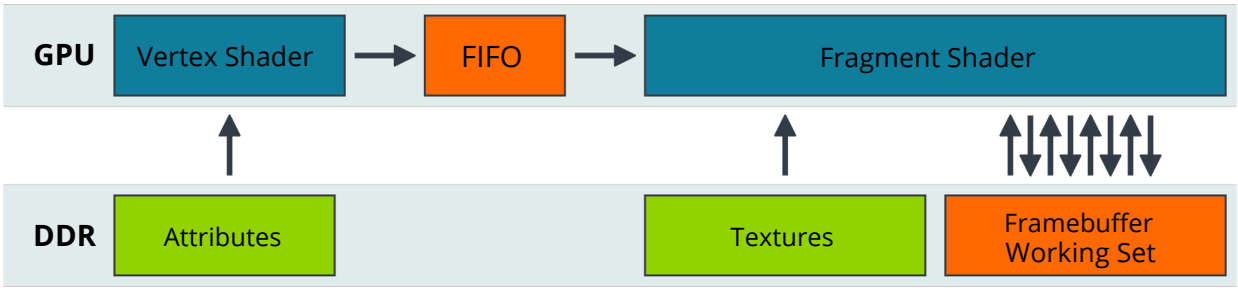
For a GPU in IMR mode, ignoring the parallel processing logic, the pseudo code executed is as follows:

```
fordrawinrenderPass:
    forprimitiveindraw:
        forvertexinprimitive:
            execute_vertex_shader(vertex) if
        primitivenotculled:
            forfragmentinprimitive:
                execute_fragment_shader(fragment)
```

The GPU hardware architecture of IMR is as follows:



The hardware data flow and memory interaction diagram are as follows:



The advantage of a GPU in IMR mode is that the output of vertex shaders and other geometryrelated shaders can remain on-chip within the GPU. The output of these shaders can be stored in FIFO buffers until the next stage in the pipeline is ready to use the data, and the GPU can use very little external bandwidth memory to store and retrieve intermediate geometry results.

The disadvantage of an IMR-mode GPU is that pixel shading jumps around the screen, and because triangles are processed in draw order, any triangle in the data stream could potentially cover any part of the screen (below). This means that the active working set is the size of the entire

framebuffer. For example, consider a 1440p device, using 32 bits per pixel (BPP) for color, and 32 bits per pixel for fill depth/stencil, which would give a total working set of 30MB, which is too large to store all on chip, so it must be stored off chip in DRAM.

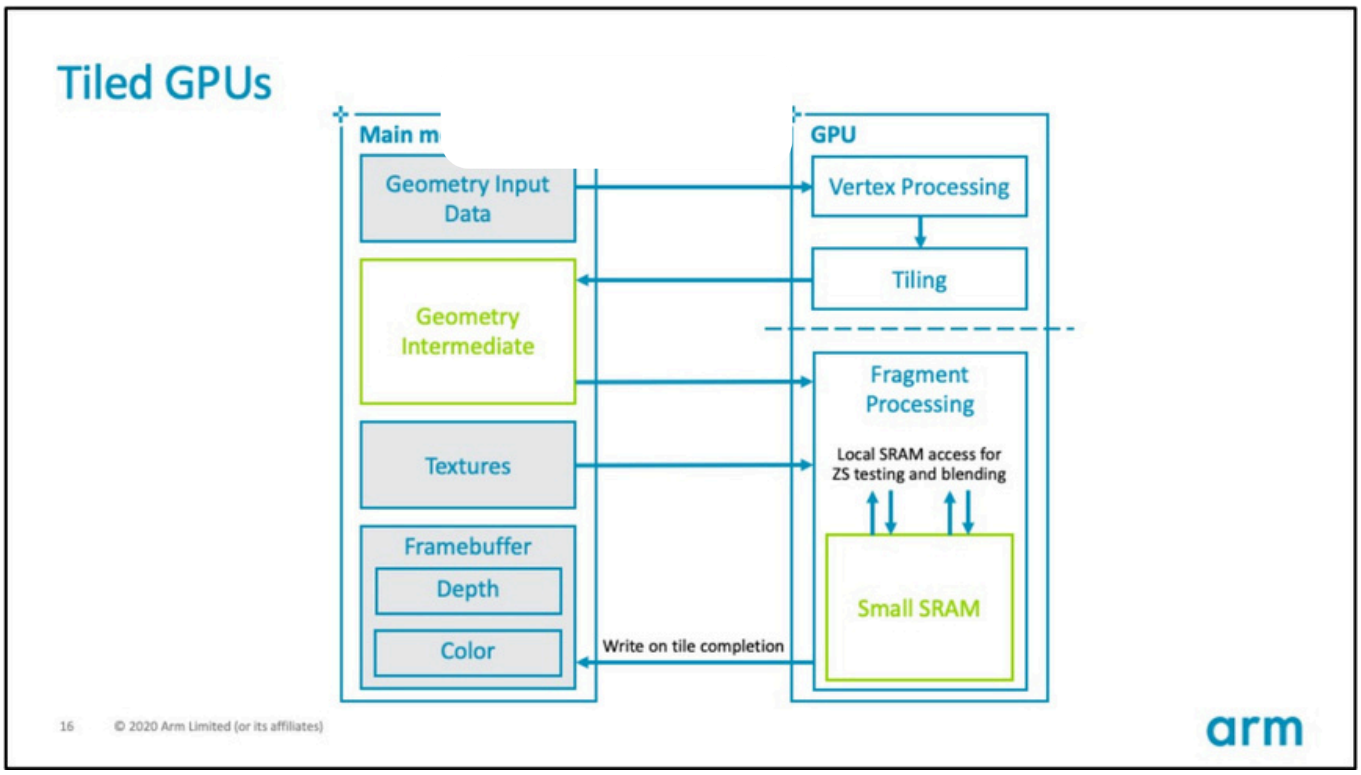
Schematic diagram of IMR's parallel rendering. Random access is spread across the entire screen, resulting in a significant reduction in cache hit rate.

When processing high resolution images, the bandwidth load placed on the memory can be very high because there are multiple read-modify-write operations for each pixel. The high bandwidth load can be alleviated by keeping recently accessed parts of the framebuffer close to the GPU.

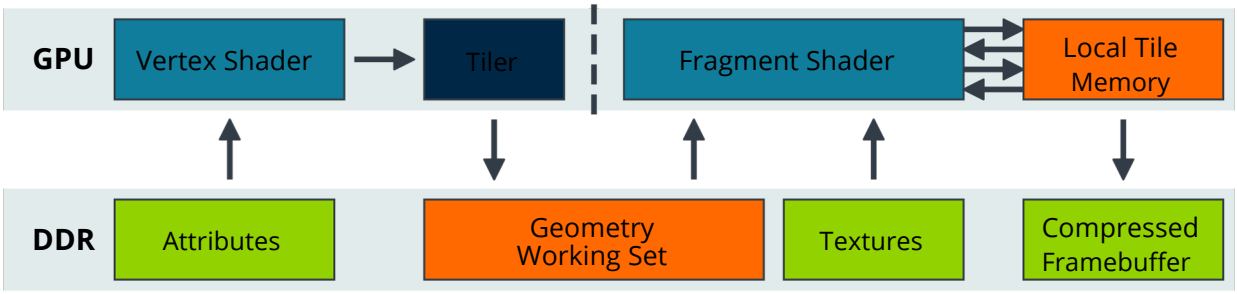
TB(D)R's GPU is different from IMR GPU. It divides the screen into several fixed-size areas before performing shading calculations. The following is the execution pseudo code of TBR:

```
# Pass one
for draw in renderPass:
    for primitive in draw:
        for vertex in primitive:
            execute_vertex_shader(vertex) if
        primitive not culled:
            append_tile_list(primitive)

# Pass two
for tile in renderPass:
    for primitive in tile:
        for fragment in primitive:
            execute_fragment_shader(fragment)
```

The hardware architecture of TB(D)R GPU is as follows:



The hardware data flow and memory interaction diagram are as follows:



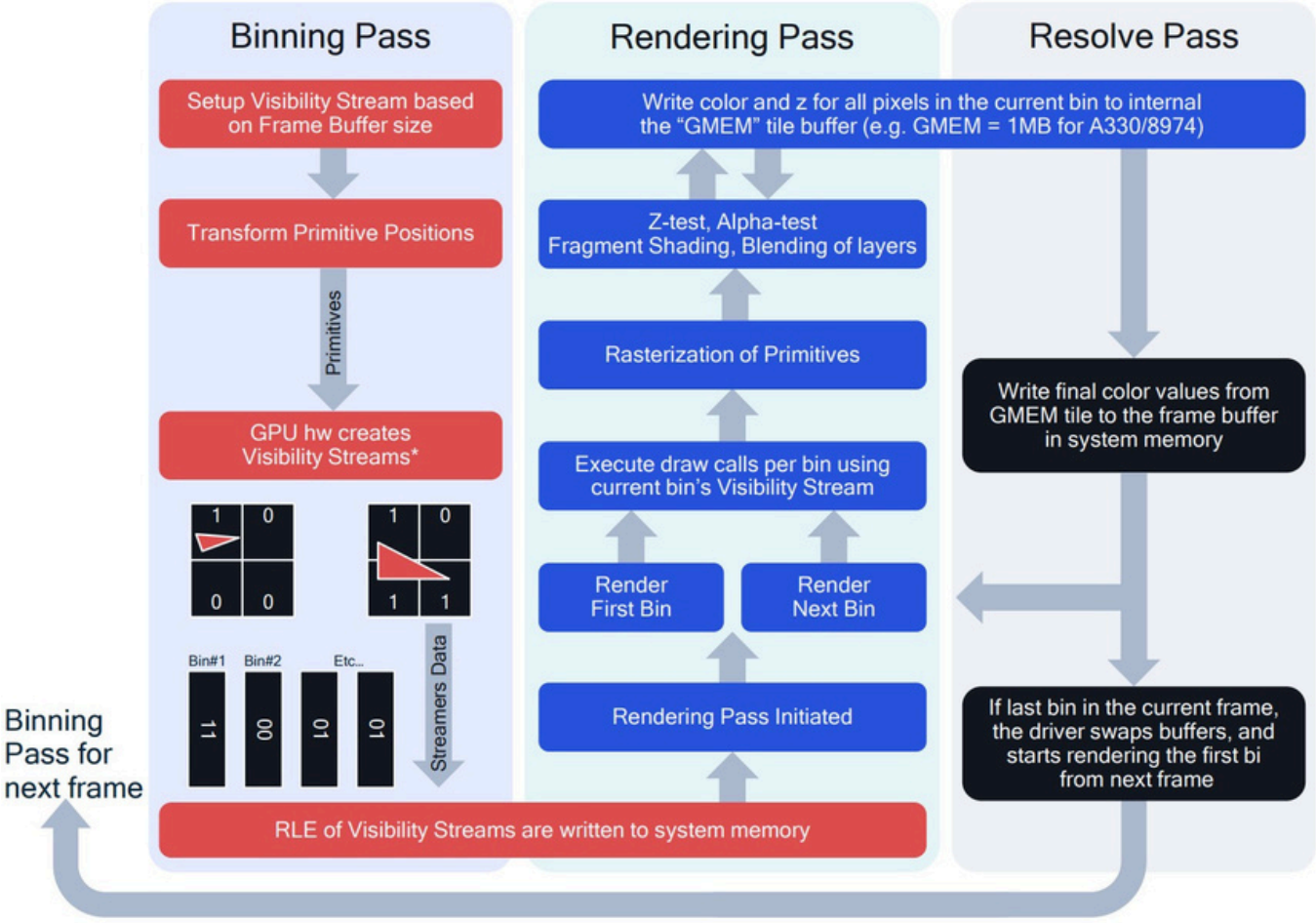
The advantage of TB(D)R is that the tile only occupies a small part of the entire framebuffer. Therefore, the entire working set of color, depth, and stencil can be stored on fast on-chip RAM, tightly coupled with the GPU shader core. The framebuffer data required by the GPU for depth testing and blending transparent pixels can be obtained without accessing external memory, which can significantly improve the energy efficiency of pixel-intensive content by reducing the number of external memory accesses required by the GPU for general framebuffer operations. In addition, in most cases there is a depth and stencil buffer, which are transient and only need to exist during the shading process. If the GPU driver is explicitly told that it does not need to save attachments, then the driver will not write them back to main memory.

> The following graphics APIs can instruct the driver to discard an attachment:
> OpenGL ES 2.0: `glDiscardFramebufferEXT`
> OpenGL ES 3.0: `glInvalidateFramebuffer`
> Vulkan: Proper render pass storeOp

It is worth mentioning that since the size of each Tile is usually not very large, the GPU computing unit has good neighborhood when accessing data within a single Tile, which can improve the cache

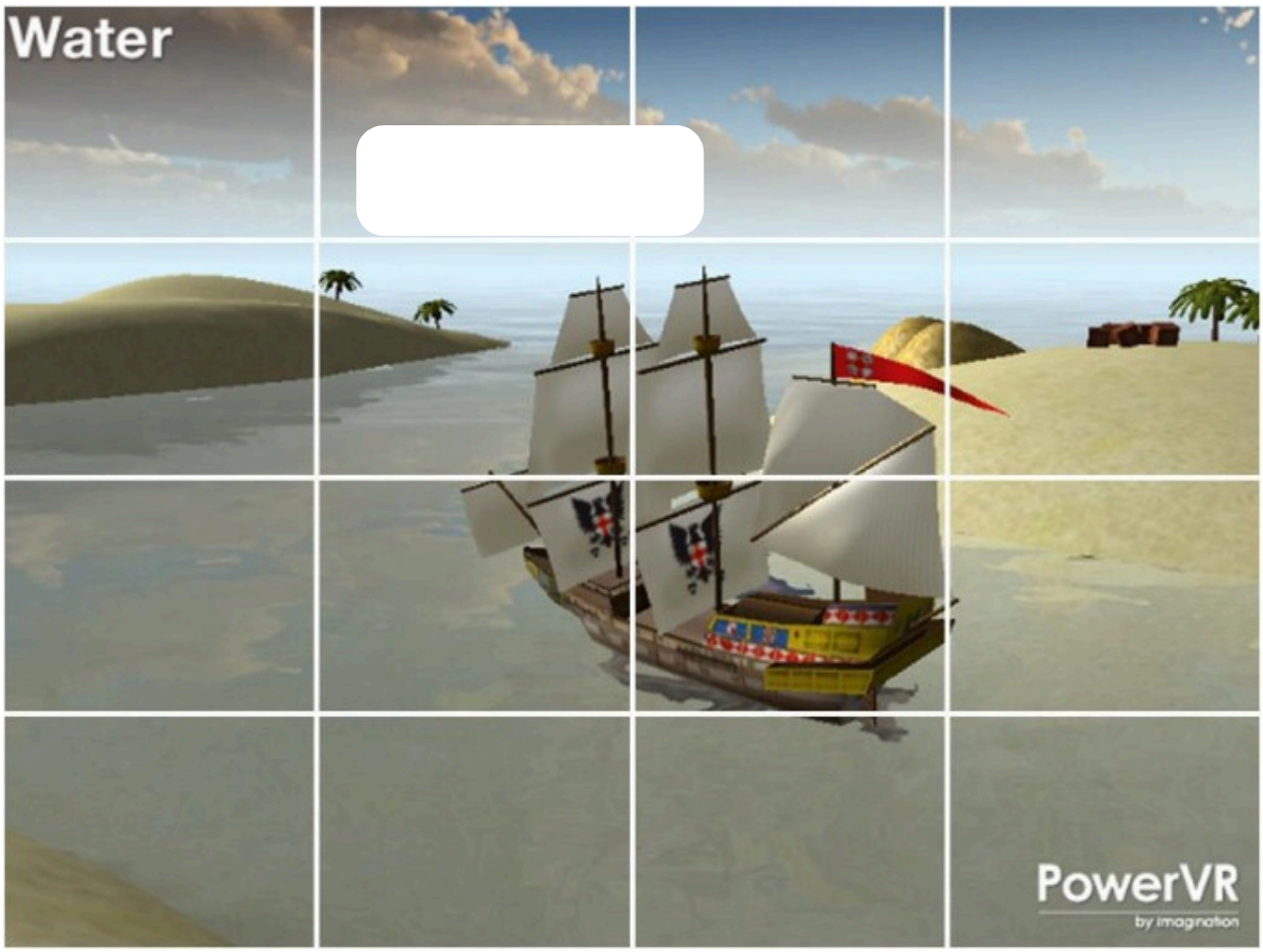
hit rate.

Of course, there is no free lunch, and TB(D)R does have some disadvantages. For example, the GPU must store the output of the geometry pass (the per-vertex variation data and the intermediate state of the tile) to main memory, which is then read by the shading pass. Therefore, a balance needs to be struck between the additional bandwidth cost associated with geometry and the bandwidth saved for framebuffer data. It is also important to consider that some rendering operations, such as tessellation, are disproportionately expensive for TBR. Operations such as tessellation are designed to accommodate the advantages of the IMR mode architecture, as the explosion of geometry data can be buffered within on-chip FIFO buffers instead of being written back to main memory.
The following uses Qualcomm Adreno series GPUs to illustrate TB(D)R's architecture, operation process, rendering technology involved, and optimization techniques.

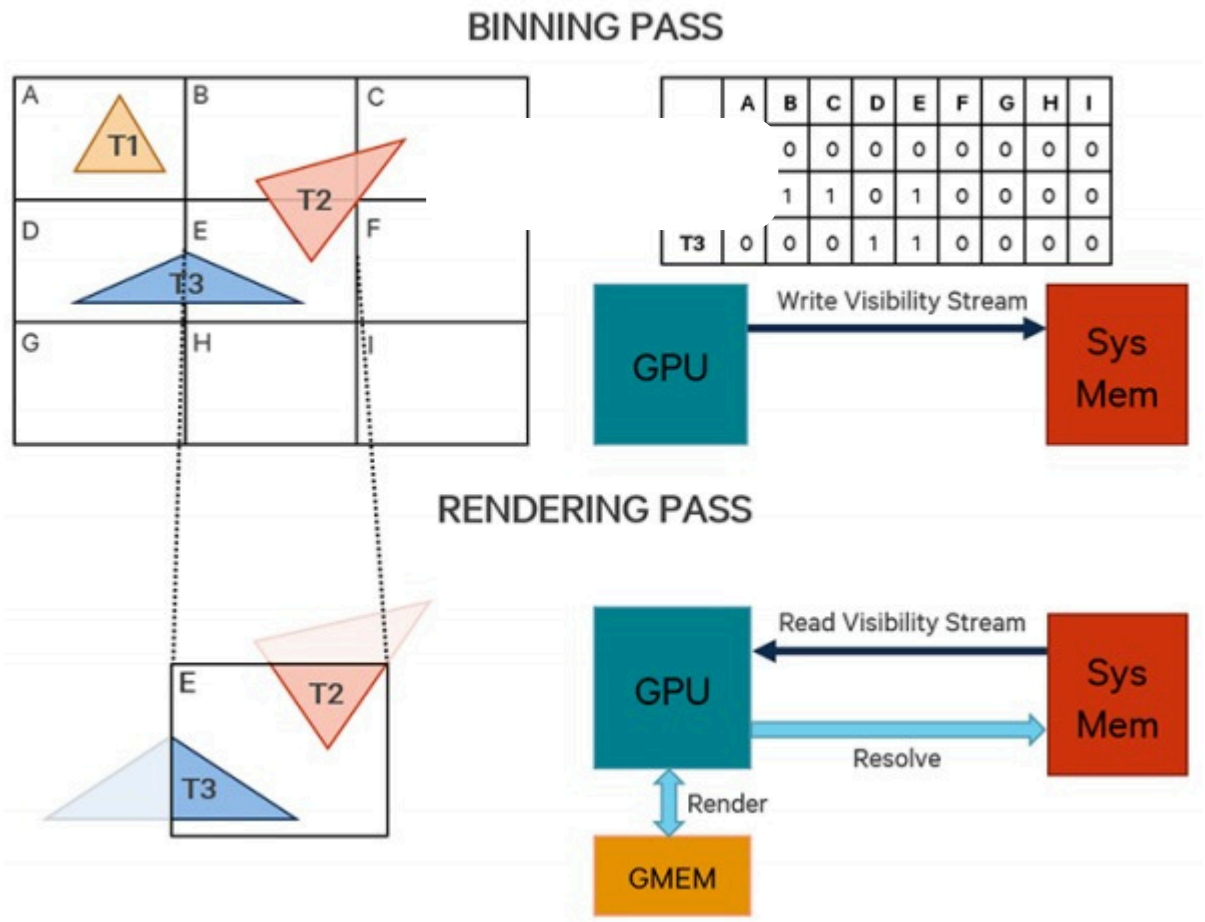


TB(D)R rendering is different from IMR mode. The drawing process is divided into three stages: **Binning Pass**,**Rendering Pass**,and**Resolve Pass .**
The Binning Pass The process is roughly as follows:

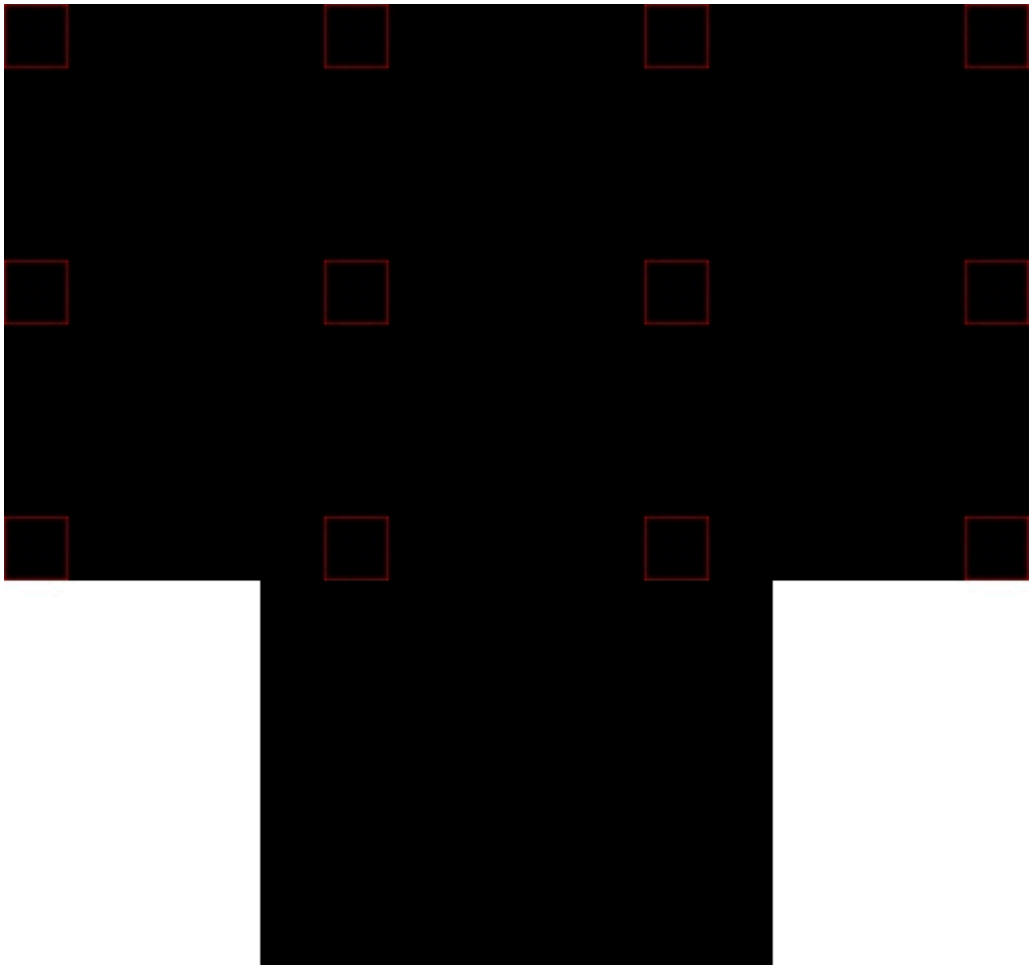
- Set a fixed size of each Bin (also called Tile) (2 to the power of N, the length and width are usually the same, the specific size varies depending on the GPU manufacturer, such as 16x16, 32x32, 64x64), and set the visible data stream according to the Frame Buffer size.



- Convert primitive coordinates. Note that this stage processes index and vertex data. Some GPUs (such as Adreno) use special simplified shaders (rather than complete Vertex Shaders) to process coordinates to reduce bandwidth and energy consumption. Usually only the position of the vertex is valid at this stage, and other vertex data (texture coordinates, normals, tangents, vertex colors) will be ignored.
- Traverse all primitives, mark all blocks covered by the primitives, and write visibility data into the covered block data stream.
- Write the visibility data stream back to system video memory.



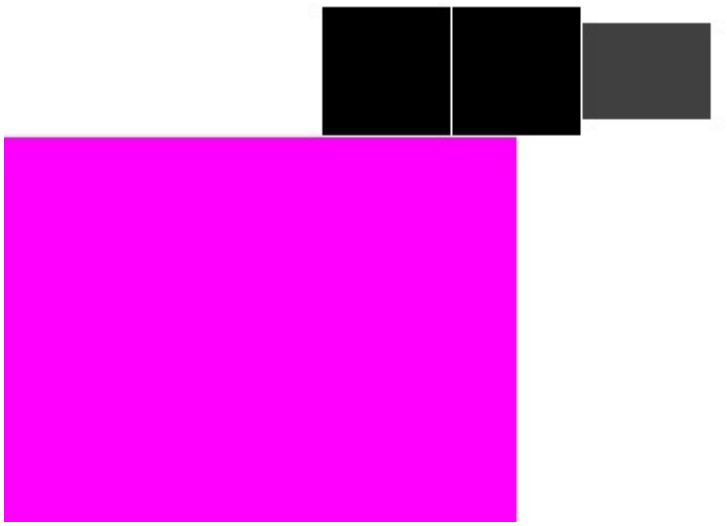
The operation diagram of the Binning stage is as follows:



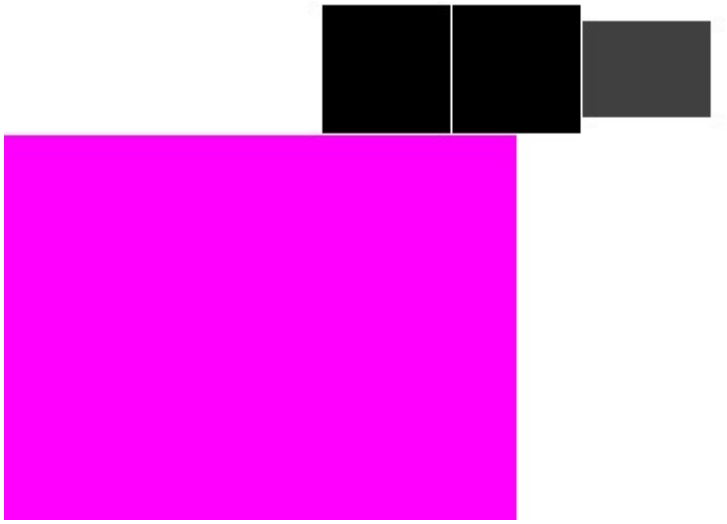
The rendering The process is as follows:

- Initialize rendering Pass.
- Iterate over all chunks and perform the following operations on each chunk:
    - Execute draw calls using the tiled visibility data stream.
    - Rasterize primitives.
    - Pixel operations (pixel shaders, depth-stencil testing, alpha testing, blending). Write pixel data
    - (color, depth, template, etc.) to the buffer on the tiled chip (also known as On-Chip Memory, GMEM, Tiled Memory).

The operation diagram of the Rendering stage is as follows:

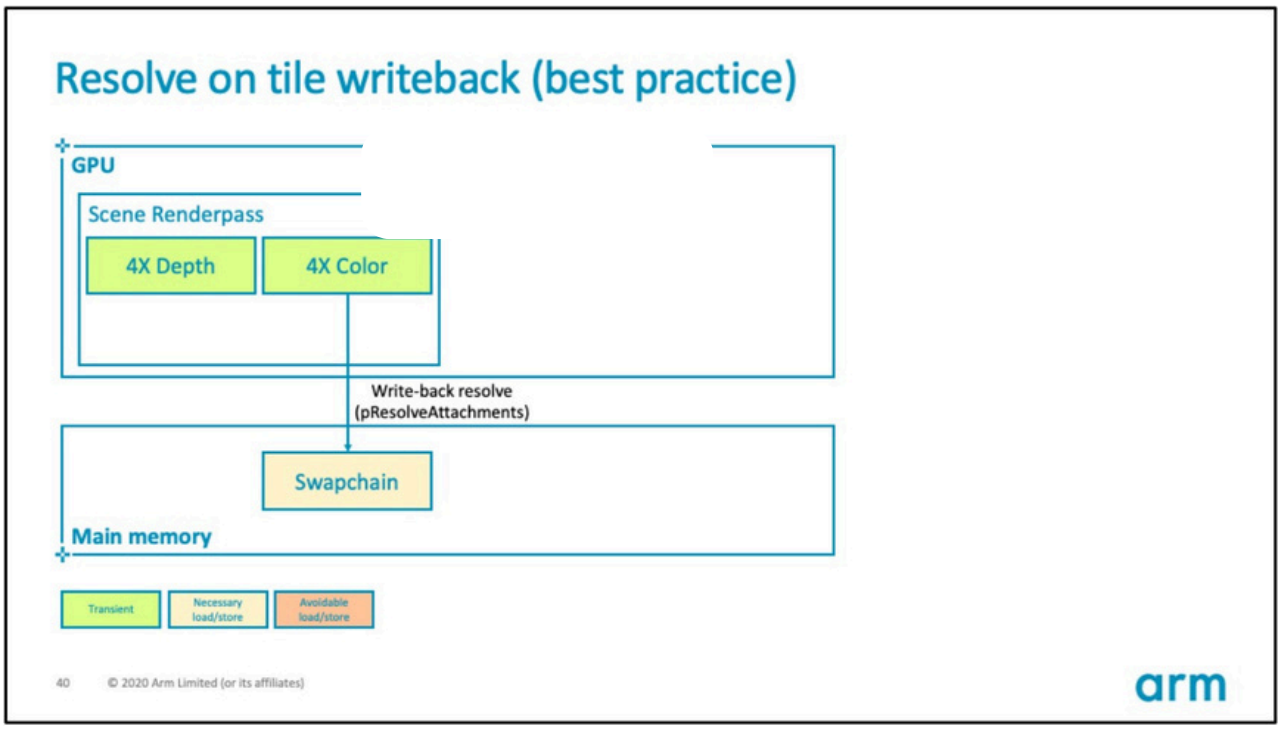


If there are multiple Tile processing units on the GPU, multiple Tiles can be processed simultaneously, and the Tile processing units are independent of each other:



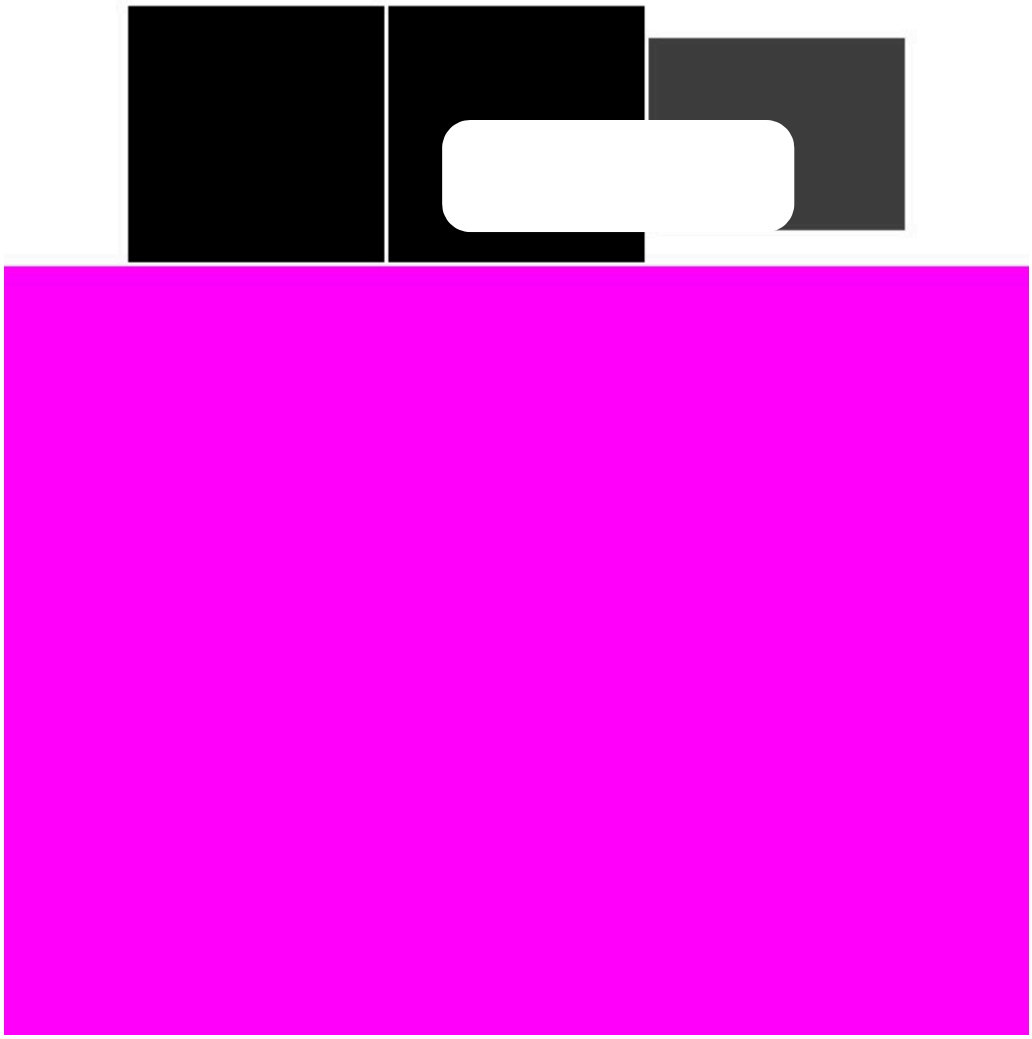
The process of the Resolve Passphase is as follows:

- If MSAA is enabled, the color, depth and other data are parsed (averaged) on GMEM, which can reduce the total amount of data transferred from GMEM to the system memory in the subsequent steps.



- Write all pixel data (color, depth, stencil, etc.) on the tile to system video memory.
- If it is not the last block of the Frame Buffer, continue to execute the next block.
- If it is the last block of the Frame Buffer, the buffer is swapped and the Binning Pass of the next frame is executed.

The following is a schematic diagram of the parsing phase (note that the pixels in the tile contain jagged edges, and the large image below shows the pixels with anti-aliasing after MSAA parsing):



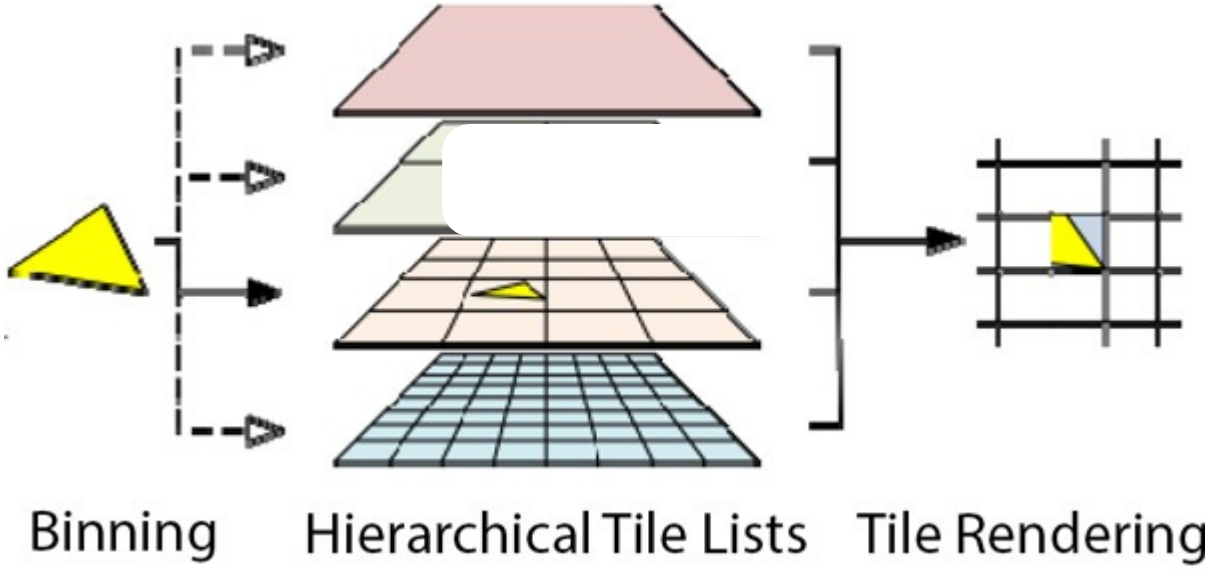
Binning Pass and Rendering Pass are usually processed frame by frame, which means that Rendering Pass will lag behind Binning Pass by one frame to reduce Stall, improve throughput, and improve rendering efficiency.

There are many more optimizations and technologies based on the TB(D)R GPU architecture, which will be discussed later.

12.4.2 Hierarchical Tiling

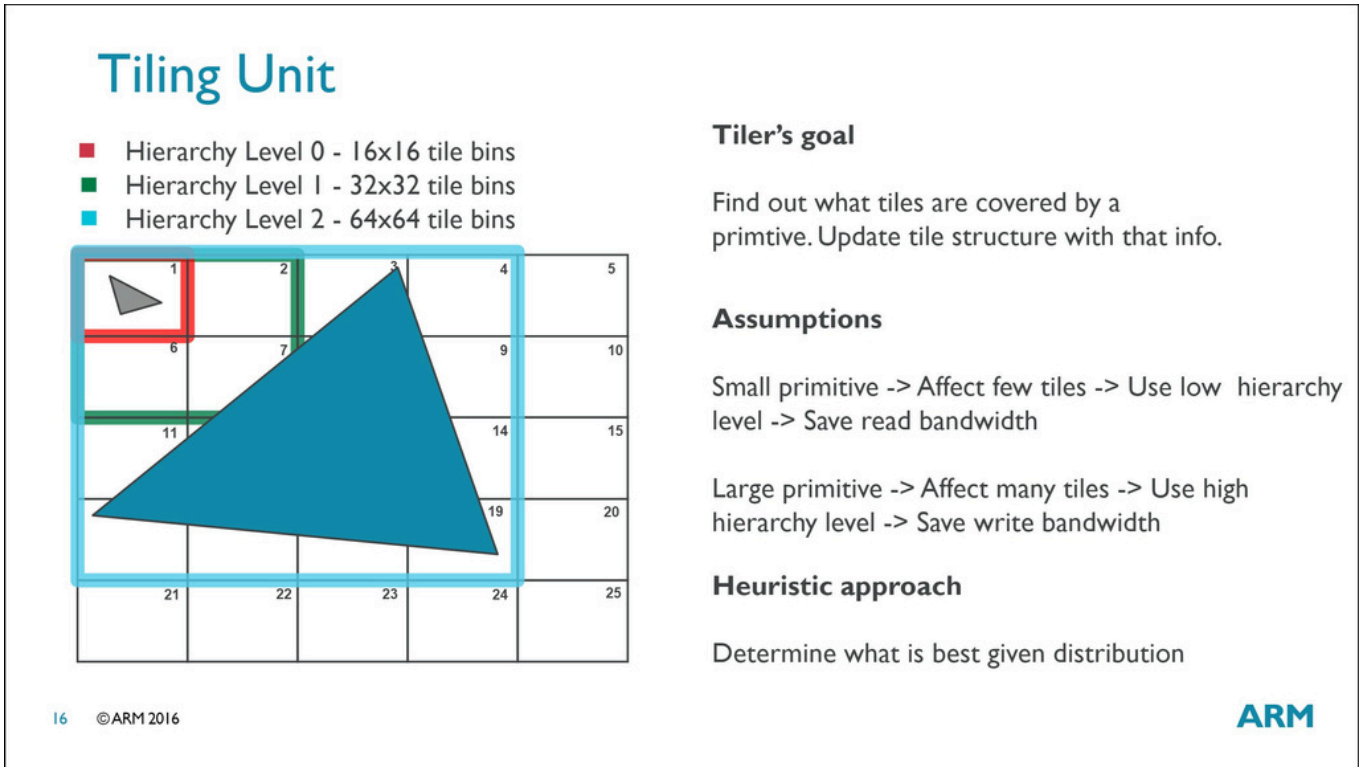
Hierarchical Tilingis translated as hierarchical blocking. It is a blocking technology used for the first time in the Arm Midgard series of chips. As the name suggests, it implements blocking on a hierarchical basis.

In this case, using Hierarchical Tiling allows Midgard to use variable tile sizes, based on the idea of further decomposing tiles (going down the hierarchy, see the image below) until the complexity of the tile reaches the desired size (or the minimum tile complexity is reached). This technique allows Midgard to use small tiles only when necessary, and save resources by using large tiles in scenes with lower complexity.



More specifically, Arm typically sets the bin sizes at different levels to the following:

- Hierarchy Level 0 is set to 16x16 pixels;
- Hierarchy Level 1 is set to 32x32 pixels;
- Hierarchy Level 2 is set to 64x64 pixels;
- Hierarchy Level 3 is set to 128x128 pixels;
- . . . . . .



The goal of the system is to find out the blocks covered by each primitive and update the structural information of the blocks.

If it is a small-area primitive (such as the gray triangle in the figure above), since fewer blocks are affected, low-level blocks are used to save reading bandwidth.
If it is a large-area primitive (such as the blue triangle in the figure above), since many blocks are affected, high-level blocks are used to save write bandwidth.

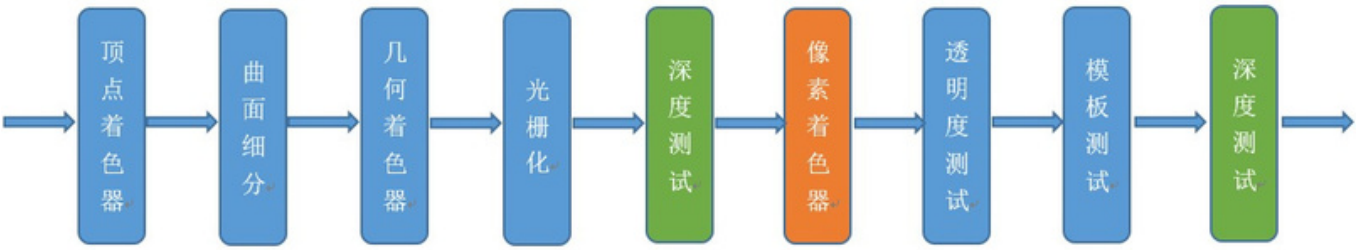
For complex primitives, the GPU uses a heuristic strategy to automatically decide which is the best distribution.

> As for the specific details of the heuristic strategy, I have not found any relevant information or literature yet. If I find it in the future (or if a classmate provides it), I will add to it.

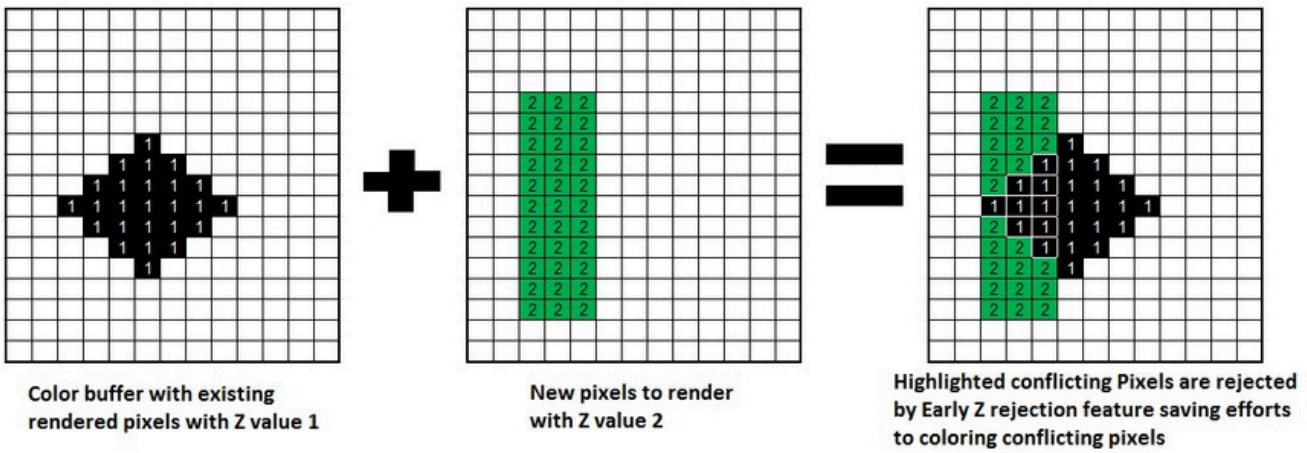
12.4.3 Early-Z

Early-Zis an early depth test that provides a fast occlusion method to remove unnecessary rendering pass objects (pixels that are not visible in the screen space). Adreno GPU can remove occluded pixels at 4 times the drawing pixel fill rate.

Early-Z usually occurs after rasterization and before pixel shading in the Rendering Pass stage. (Figure below)



Early-Z technology can remove many invalid pixels in advance to prevent them from entering the pixel shader with heavy consumption. The minimum unit of Early-Z removal is not 1 pixel, but**pixel quad**.The following is one of the running examples.



Schematic diagram of Early-Z operation. On the left are the rendered values stored in the depth buffer, all of which are 1; in the middle are the areas ready to be rendered with a depth value of 2; on the right, Early- Z technology is used to remove pixels that are larger than the depth buffer.

In order to maximize the Early-Z technology, the rendering engine (such as UE) will use a special Pass (such as UE's PrePass) at the beginning of rendering to render the depth of all opaque objects and give full play to the Early-Z technology of the TBR architecture. This step is not required for GPUs that support the TBDR architecture.

However, the following situations will cause Early-Z to fail:

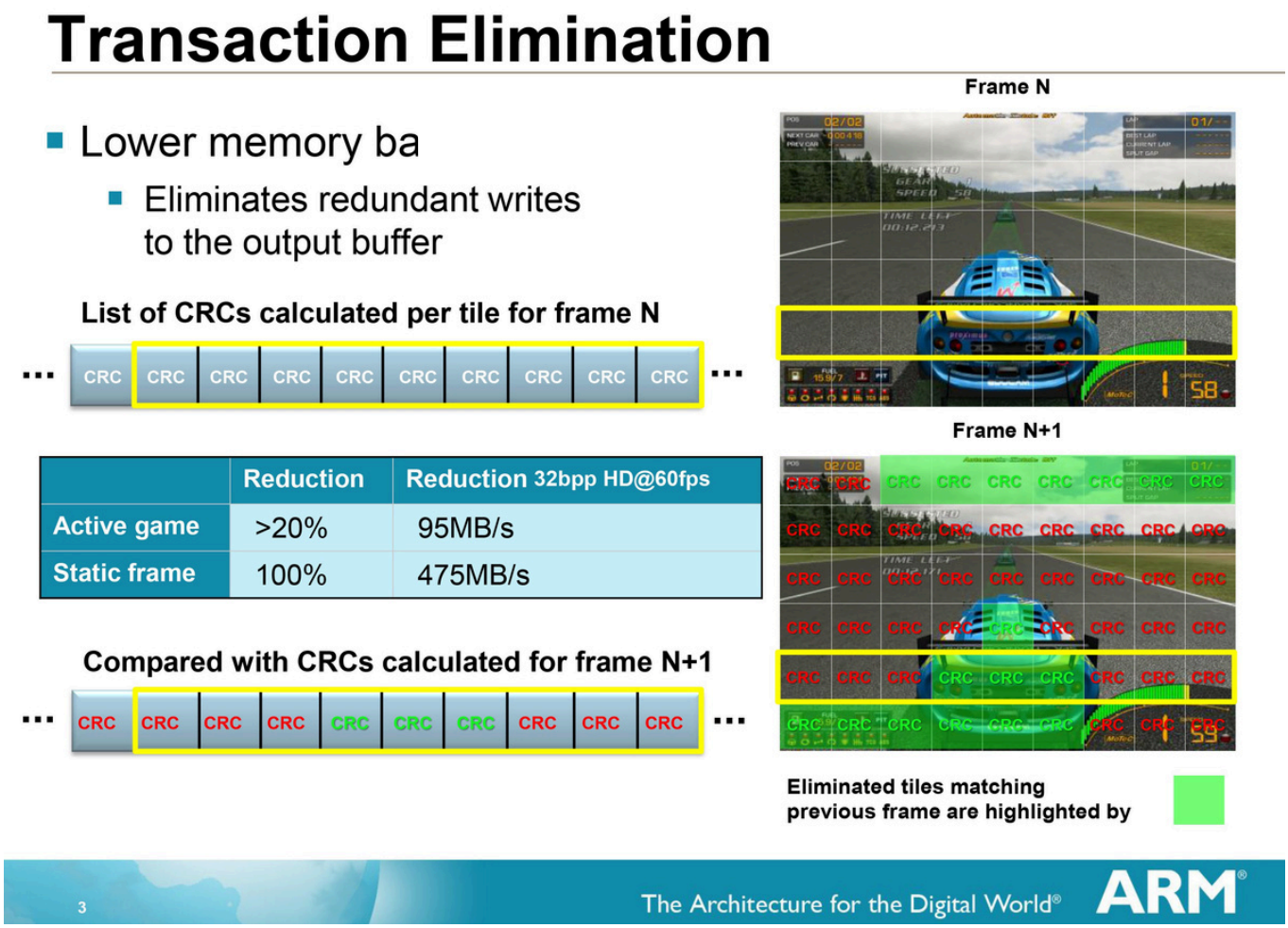
- **Turn on Alpha Test**:Since Alpha Test needs to be compared in the Alpha Test stage after the pixel shader, it is impossible to decide whether the pixel is culled before the pixel shader.
- **Turn on Tex Kill**:that is, there are pixel discard instructions in the shader code (DX's discard, OpenGL's clip).
- **Turn off depth testing**:Early-Z is based on the condition that depth testing is turned on. If depth testing is turned off, Early-Z technology cannot be enabled.
- **Turn on Alpha To Coverage**:Alpha To Coverage will turn on multi-sampling, which will affect surrounding pixels. However, in the Early-Z stage, it is impossible to know whether surrounding pixels are clipped, so they cannot be eliminated in advance.
- And any other operation that results in the need to blend subsequent colors.

12.4.4 Transaction Elimination

Transaction Elimination (TE)is a key bandwidth saving feature of the Mali GPU architecture that can significantly save energy on a System-on-Chip (SoC).

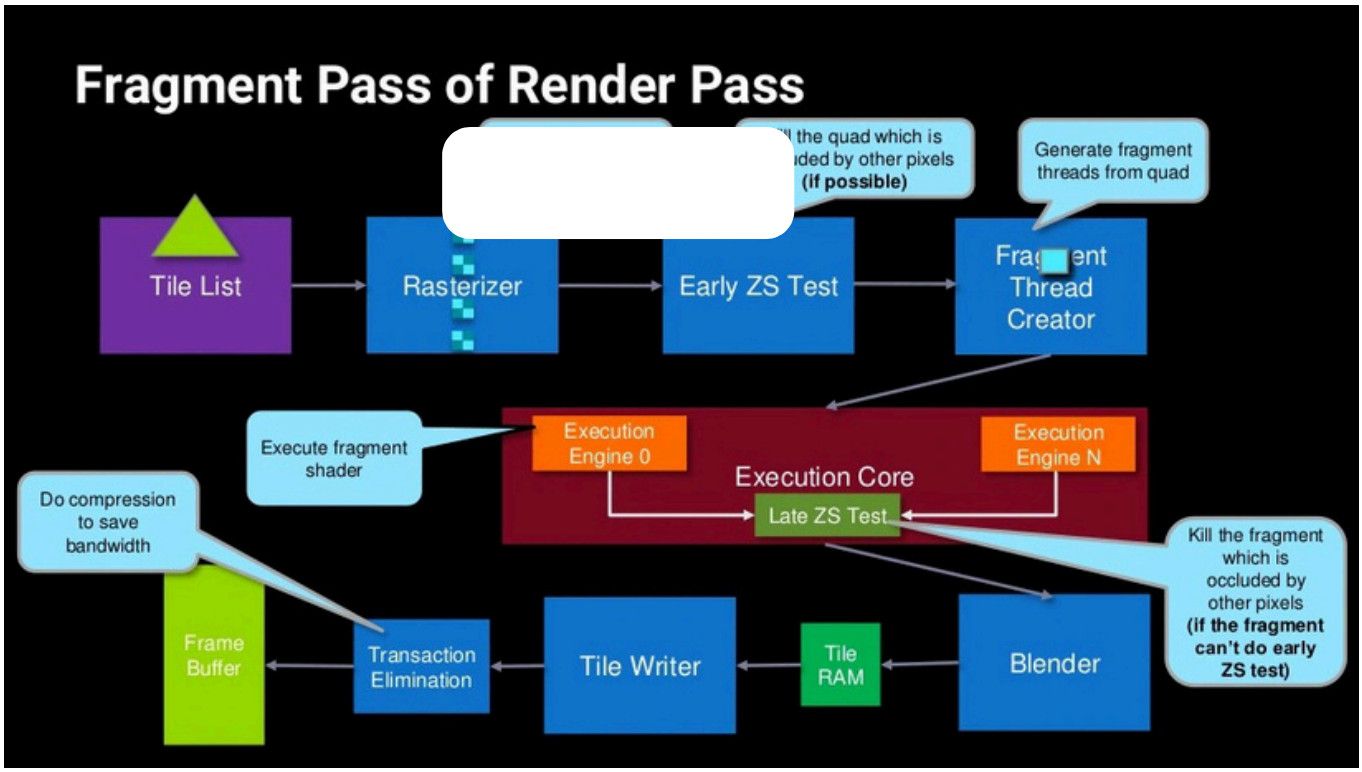
When executing TE, the GPU compares the current frame buffer with the previously rendered frame and only partially updates the modified specific parts, which greatly reduces the amount of data transferred to external memory per frame. The comparison is performed at the tile granularity, and a cyclic redundancy check (CRC) signature is used to determine whether the tile has been modified (Tiles with the same CRC signature are considered to be the same, thereby ignoring the transmission of the tile's data).

> Cyclic Redundancy Check (CRC) is a hash function that generates a short fixed-bit checksum based on network packets, computer files, memory data streams, etc. It is mainly used to detect or verify errors that may occur after data transmission or storage. Here, it is used by the Mali GPU to detect whether the tile data of this frame is the same as the previous frame buffer data.



TE technology operation overview. The current frame will calculate a CRC key value for each block, so that the next frame can compare each block to see if there is any data change, and cancel the data transmission for the unchanged blocks. The green block in the lower right corner of the figure matches the previous frame and does not need to transmit data to the Frame Buffer. For interactive games, the average bandwidth can be reduced by more than 20%.

Executing TE technology has no effect on the final image quality and can be used for all applications of all frame buffer formats supported by the GPU, regardless of the frame buffer accuracy requirements. In addition, it should be noted that TE occurs during the period when the tile writes data to the system memory (Frame Buffer) (lower left of the figure below).

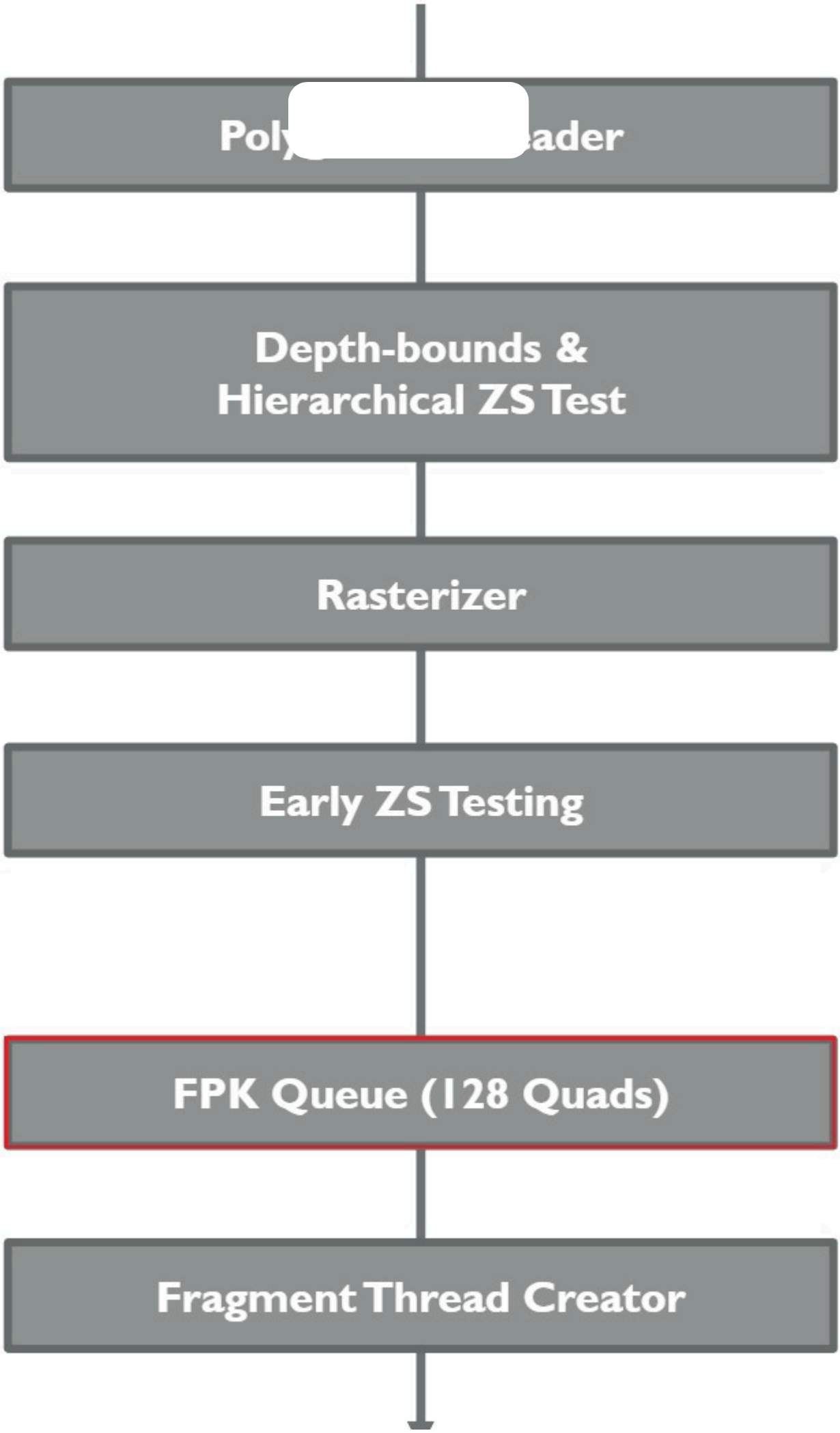


12.4.5 Forward Pixel Kill

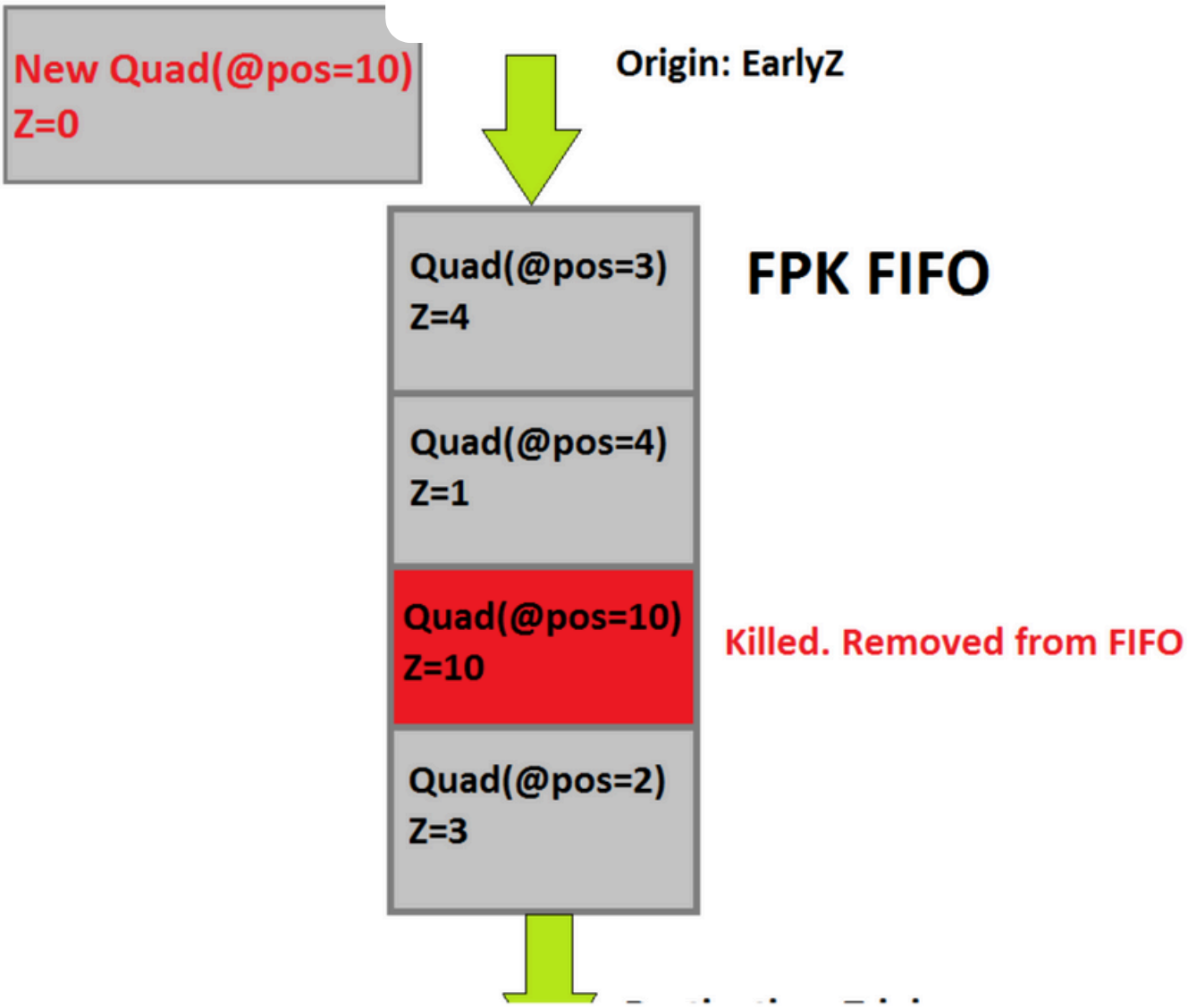
Forward Pixel Kill (FPK)is a technology to reduce OverDraw built into Mali-T62X and T678 and later chips.

In GPUs that support FPK, a thread of pixel shading, even if started, is not irreversibly completed. Ongoing computations can be terminated at any time if the rendering pipeline finds that a later thread will write opaque data to the same pixel location. Because each thread takes a finite amount of time to complete, there is a time window that can be exploited to kill pixels that are already in the pipeline. In effect, the depth of the pipeline is exploited to simulate a foresight effect into the future.

There is a FIFO (first-in-first-out) buffer in the GPU chip that supports FPK (between the Early Z test and the pixel shader, see the figure below), which is used to store Quads that have passed the Early- Z test and are about to enter the pixel shading calculation.


Poly ader
Depth-bounds &
Hierarchical ZS Test
Rasterizer
Early ZS Testing
FPK Queue (128 Quads)
Fragment Thread Creator

Let's take a specific example to illustrate the operating mechanism of FPK, as shown in the following figure:



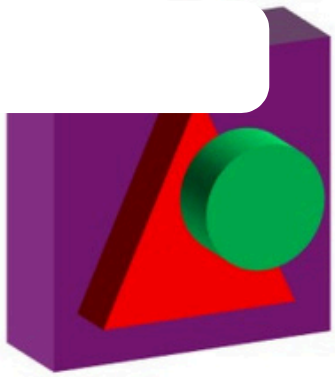
In the above figure, the new Quad (position 10, depth 0) passes the EarlyZ test and is about to enter the FPK FIFO buffer. It is found that there is already a Quad at position 10 and depth 10 in the FIFO. Quad will replace the Quad in the FIFO queue (because the new depth is closer to the screen). In other words, the Quad in the FPK FIFO will be replaced by the new Quad at the same position and smaller (closer) depth.

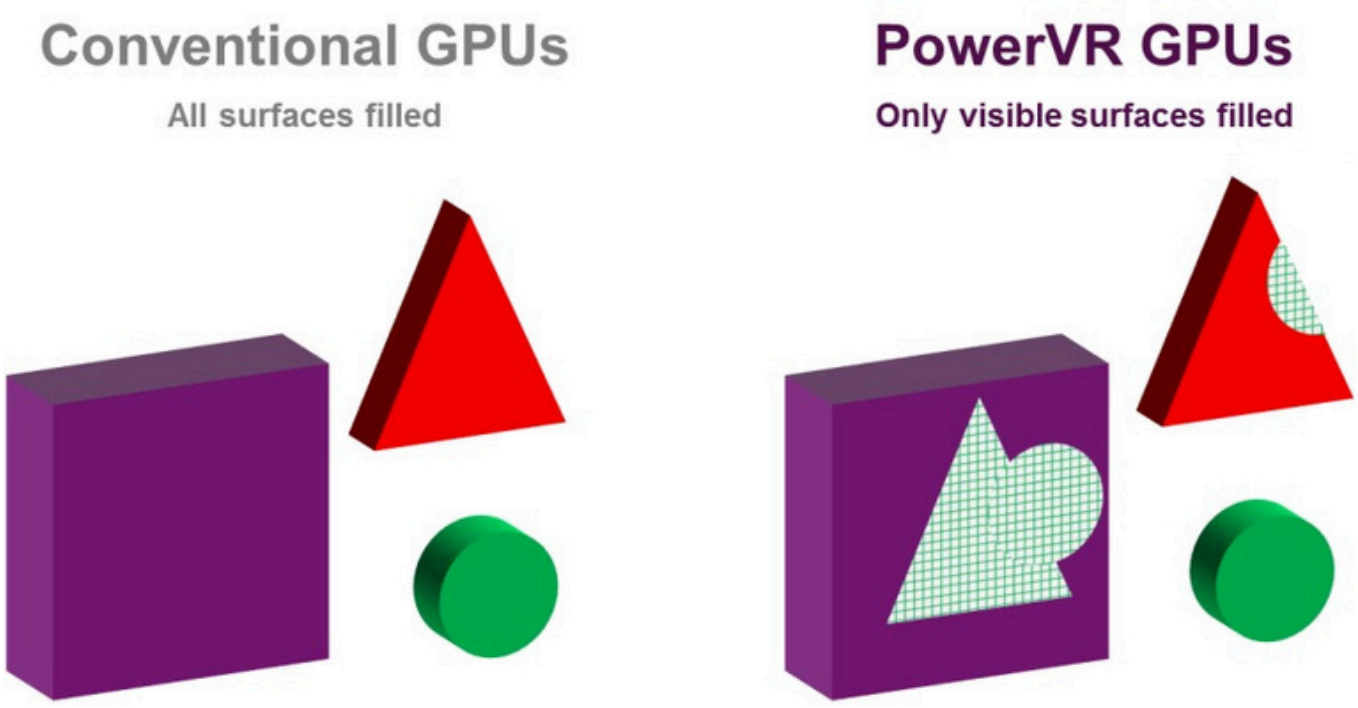
A few additional notes on FPK:

- The FPK culling granularity is Quad (2x2 pixel block). FPK
- is only valid for opaque objects.
- FPK must have depth testing enabled to work.

12.4.6 Hidden Surface Removal

Hidden Surface Removal (HSR)is a dedicated technology for PowerVR chips. Through HSR technology, zero OverDraw can be achieved regardless of the drawing order.



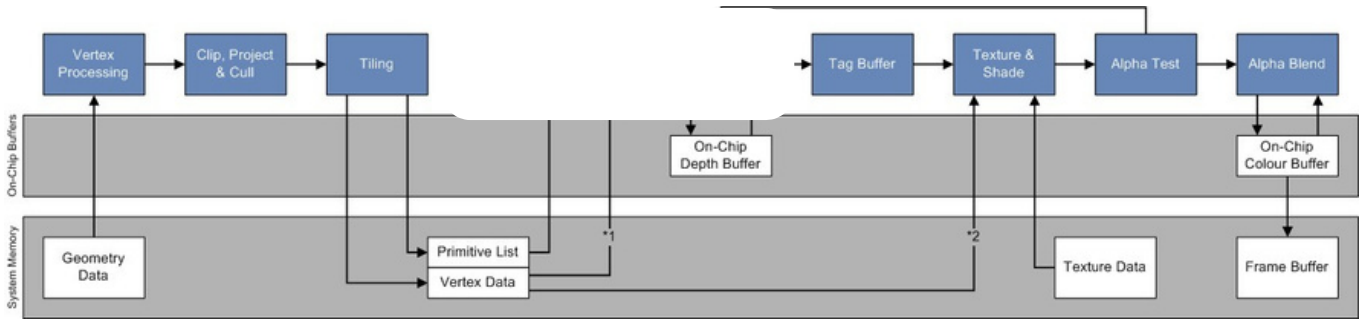


The lower left is a traditional GPU that does not perform culling on occluded pixels, while the lower right shows that PowerVR can perform pixel-level culling using HSR.

On architectures that include Early-Z testing, applications can avoid some OverDraws by submitting draw calls from front to back. Submitting in this order builds up the depth buffer, so occluded pixels that are far from the camera can be culled early. However, this puts additional burden on the application because the draws must be sorted every time the camera or objects in the scene move. It also cannot remove all OverDraws because the per-draw sorting is very coarse. For example, it cannot solve OverDraws caused by object intersections. It also prevents applications from sorting draw calls to keep graphics API state changes to a minimum.

Using PowerVR's TBDR, HSR will completely avoid OverDraw regardless of the order in which objects are submitted (no sorting). HSR can achieve pixel-level culling that does not rely on sequential drawing. The most important approach is to first mark which primitive (triangle) the pixel belongs to after the primitive is rasterized, and store it in the Tag Buffer. After all the primitives in the scene are

processed, it will enter the fragment shader. The HSR stage is after rasterization and before pixel shading:



12.4.7 Low Resolution Z pass

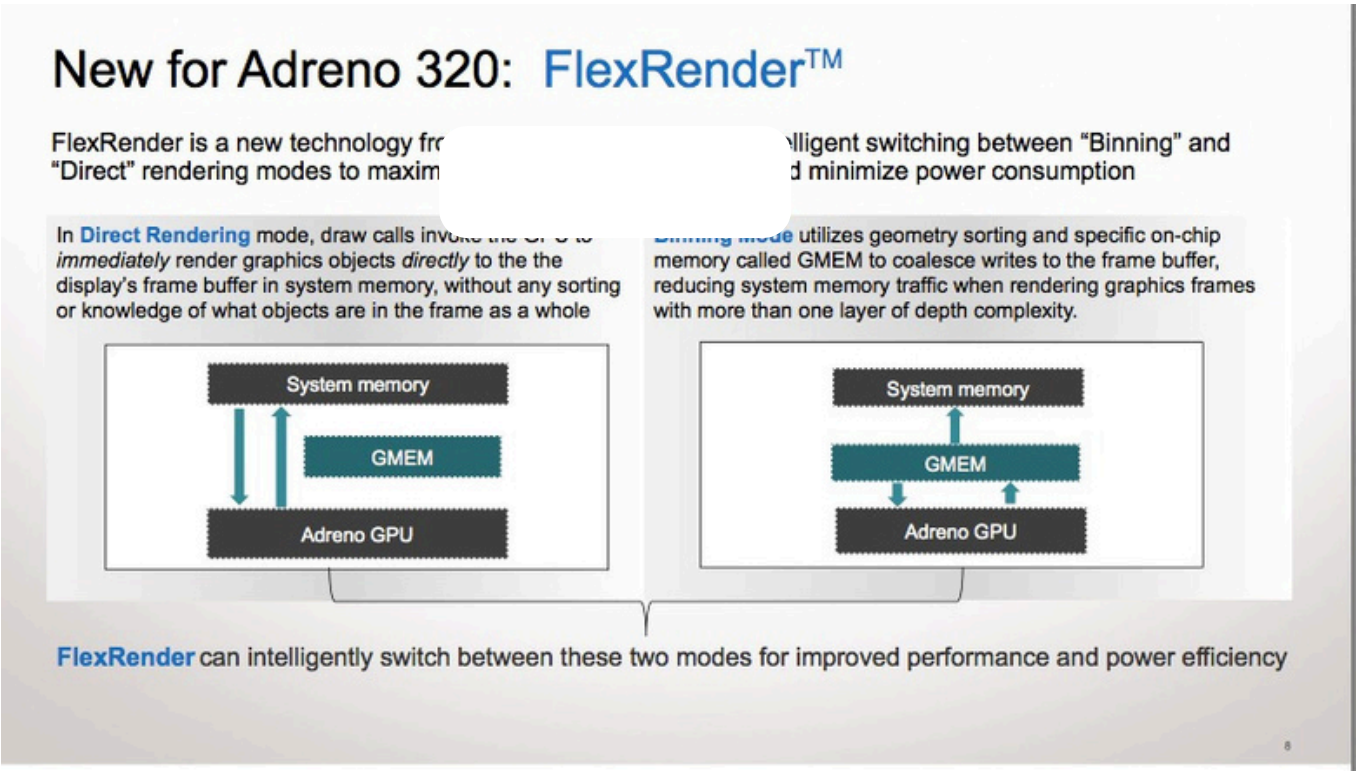
Low Resolution Z pass,referred to as**LRZ**,is an optimization technology for Adreno A5X and above chips when performing Early-Z culling in TBR.

In the Binning Pass stage, the GPU constructs a low-resolution Z buffer and uses LRZ-Tile (note that it is not Bin Tile) as the granularity to cull the occluded area to improve the performance of the Binning stage. Before testing the full -resolution Z buffer, this LRZ can also be used in the Rendering
Pass to effectively cull pixels.
The advantage of this feature is that it reduces memory access and bandwidth, reduces rendering primitives, does not require applications to draw from front to back, and improves frame rate.
However, the following situations will make LRZ technology ineffective:

- Write the depth value in the pixel shader.
- The secondary command buffer of the graphics API (Vulkan) is used. Any
- conditions that require IMR direct rendering.

12.4.8 FlexRender

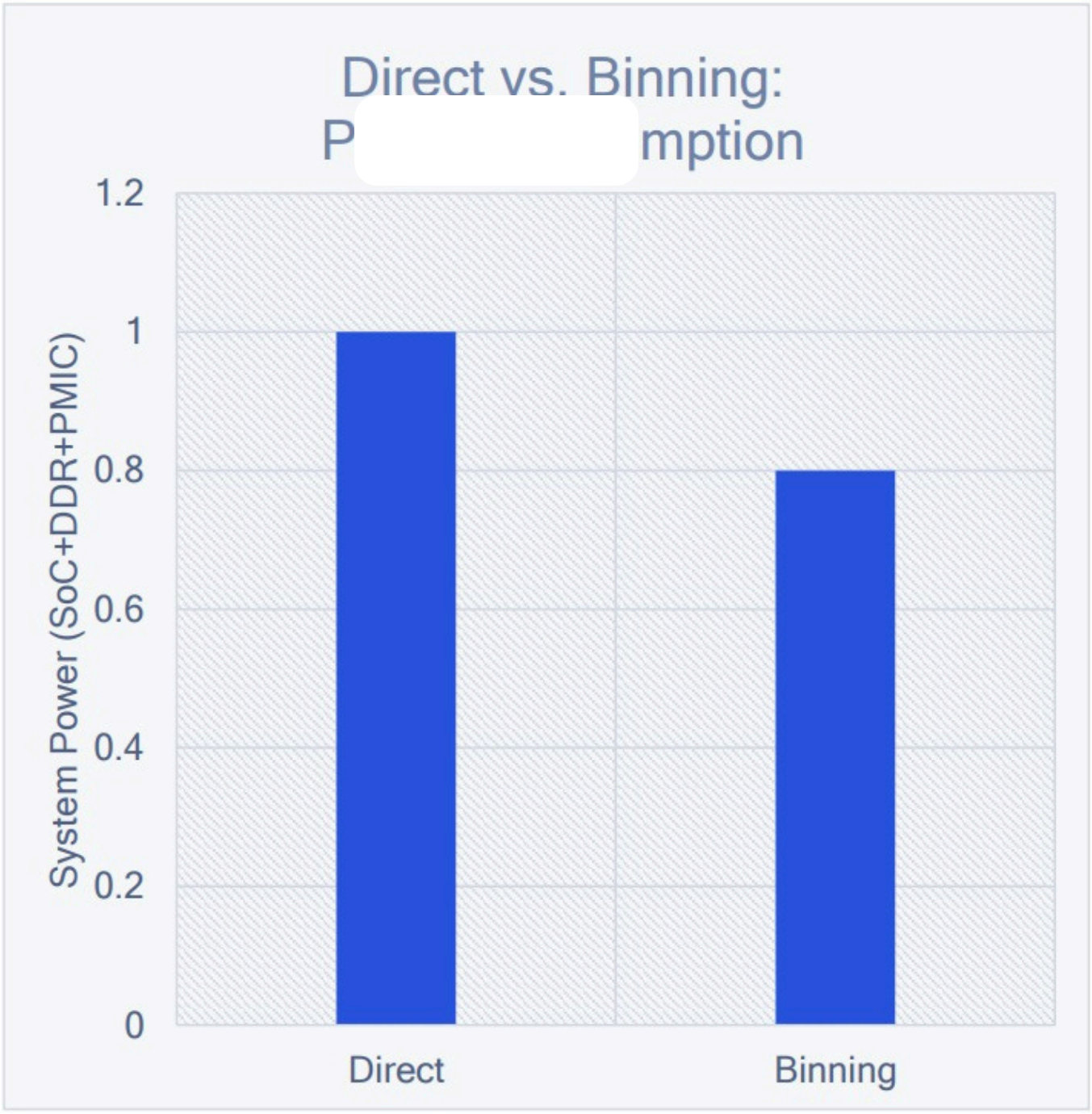
FlexRenderis a unique technology of the Adreno chip. It is a rendering technology that mixes TBR (Binning) and IMR (Direct Rendering) modes, maximizing performance by dynamically switching between the two modes.



FlexRender operation diagram. In Direct Rendering mode, the GPU bypasses GMEM and interacts directly with system memory; in Binning mode, the GPU interacts with system memory through GMEM.

The driver and GPU analyze the rendering parameters of a given render target and automatically select a mode. For example, if the render target size is very small, it will actively switch to IMR mode to reduce rendering overhead (TBR has basic overhead). If occlusion culling is performed, it will also switch to IMR mode (even if it was in TBR mode before).

Generally, IMR mode consumes more energy than TBR mode:



Comparison of energy consumption of Snapdragon SoC in Direct and Binning modes monitored by GFXBench Manhattan 3.0. The latter saves about 20%.

12.4.9 Universal Bandwidth Compression

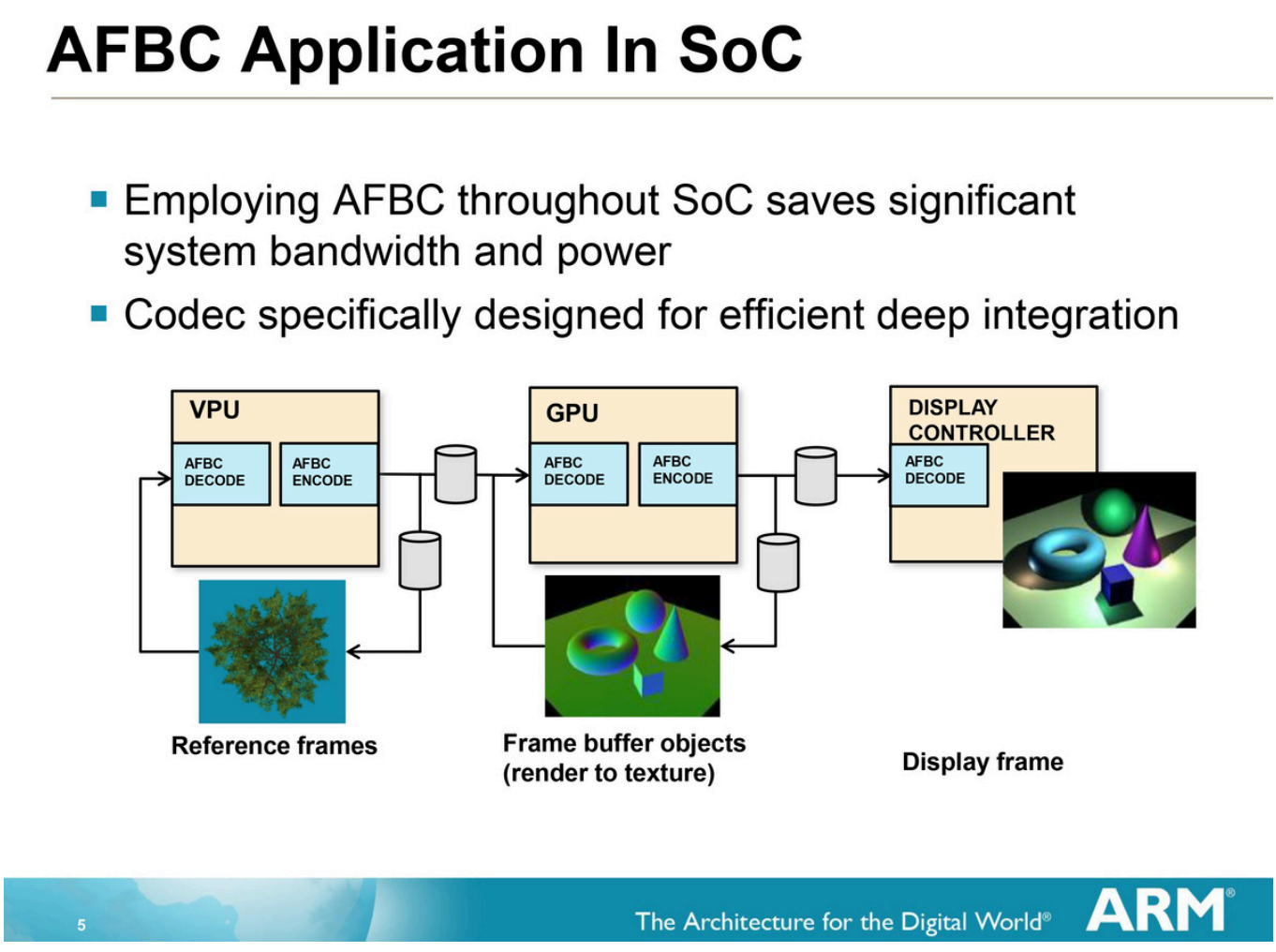
Universal Bandwidth Compression (UBWC)is a universal bandwidth compression technology added to Adreno A5x and later chips. It is a unique predictive band compression scheme that can improve the effective throughput of system memory and achieve significant energy saving by minimizing data bandwidth.

In addition to the GPU chip, UBWC technology is applied to multiple components of the Snapdraggon CPU, such as display, video, camera, etc. Compression supports YUV and RGB formats to reduce memory bottlenecks.

> Although UBWC is used in Qualcomm chips,Google Developer Contributes Universal Bandwidth Compression To Freedreno Driver shows that the technology is actually provided by the Freedreno open source driver. The article does not mention what compression algorithm UBWC uses.

12.4.10 Arm Frame Buffer Compression

Arm Frame Buffer Compression (AFBC)is available exclusively in Arm-designed GPUs, addressing the difficulty of creating increasingly complex designs within the thermal constraints of mobile devices. The most important application is video post-processing, where in many use cases the GPU needs to read video and apply special effects when using the video stream as a texture in a 2D or 3D scene. In this case, AFBC can reduce the overall system-level bandwidth and power cost of transmitting spatially coordinated image data by up to 50%.



AFBC operation diagram.

As a lossless compression protocol and format, AFBC minimizes the amount of data transferred between IP blocks in SoC. Specifically, AFBC has the following features:

- Lossless compression format. The compression format retains the original image accuracy, and the compression rate is comparable to other lossless compression standards.
- Fully supported by Mali GPUs.

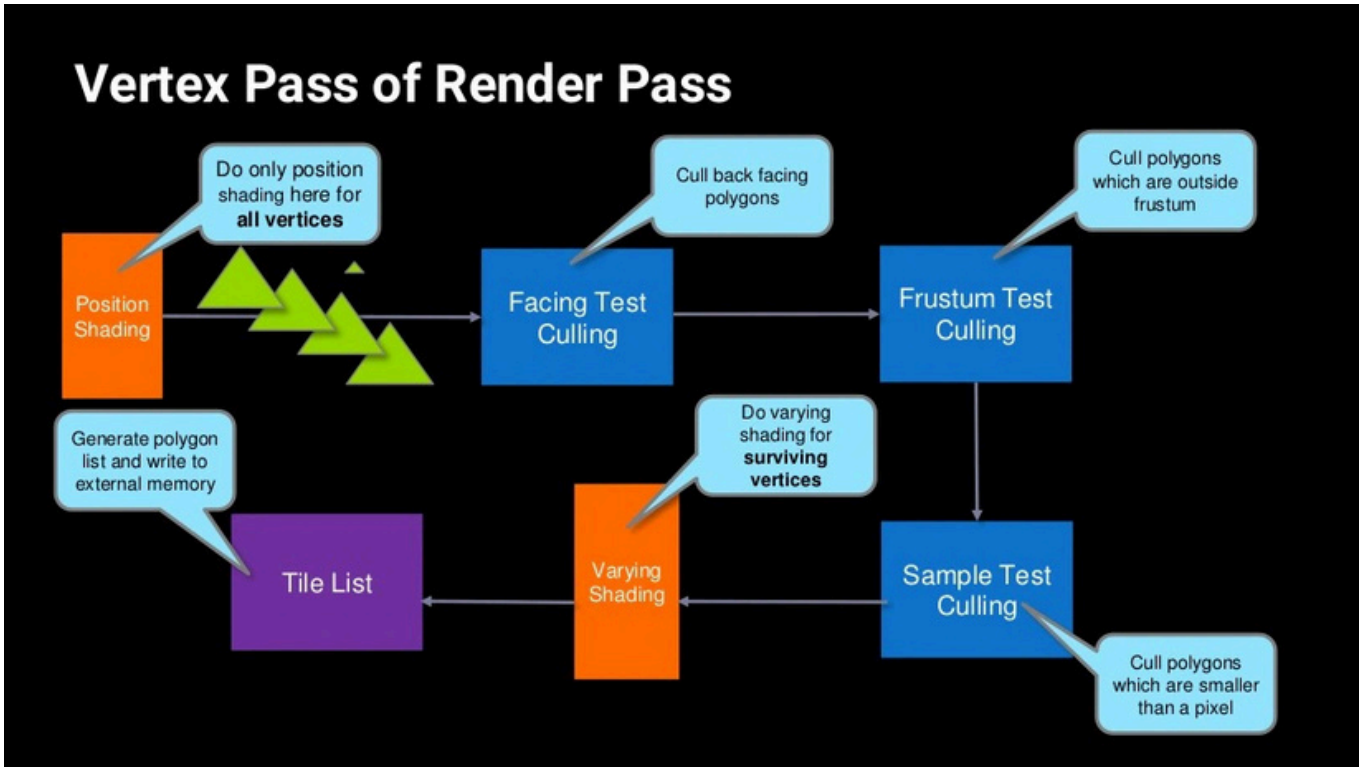
- Reduced energy consumption. Mainly benefits from reduced bandwidth.
- High area efficiency for SoC designs. AFBC can be added in the design with zero area cost.
- Bounded worst-case compression ratio. Worst-case (random access) efficiency drops to 4x4 level.
- Supports YUV and RGB formats. The YUV compression ratio is generally 50%.

12.4.11 Index-Driven Vertex Shading

Index-Driven Vertex Shading (IDVS)is a vertex processing optimization technology in Mali GPU, which occurs in the vertex processing stage of each Render Pass.

The main feature of IDVS is to split the traditional vertex shader into two stages:

- The first stage is**Position Shading**,which occurs before culling of various vertices. This stage only converts the vertex position without performing other operations on the vertex.
- The second stage is**Variable shading**,which occurs after various types of vertex culling. It only processes vertices that pass through various types of cull and performs other operations besides vertex position transformation.



IDVS splits the vertex shader into two stages: Position Shading and Varying Shading.

The advantages of IDVS technology are:

- In most cases, Varying Shading consumes more performance than Position Shading. Invalid vertices are eliminated through the Culling stage of various vertices, thus avoiding entering the expensive Varying Shading.
- By matching the vertex attribute layout of IDVS technology, you can reduce the amount of data read, improve cache hit rate, improve performance, and reduce power consumption. The

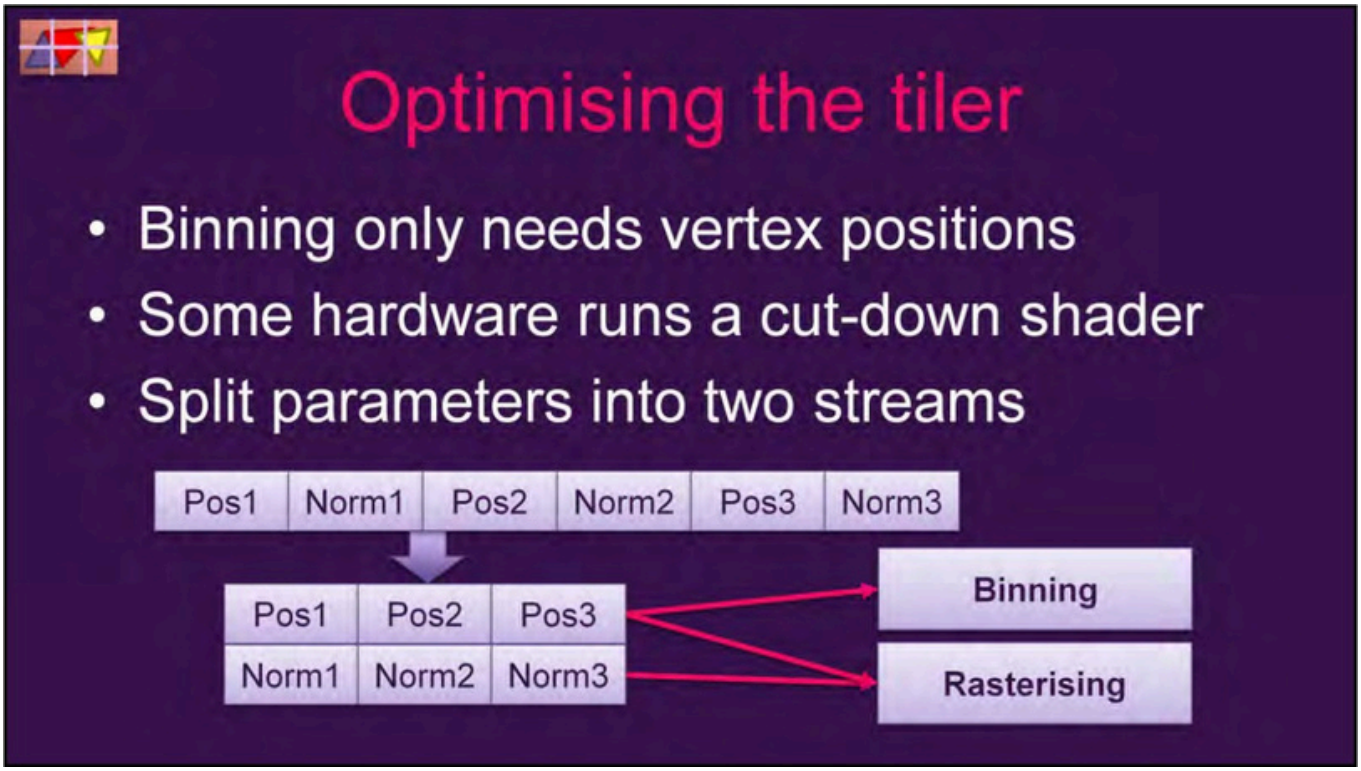
vertex attribute layout matching IDVS technology is as follows:

- The vertex positions are separated into a data stream, and the data stream layout is as follows:

```
xyz | xyz | xyz | ...
```

- Layout the vertex attributes except the position according to SoA (Structure of Array), for example:

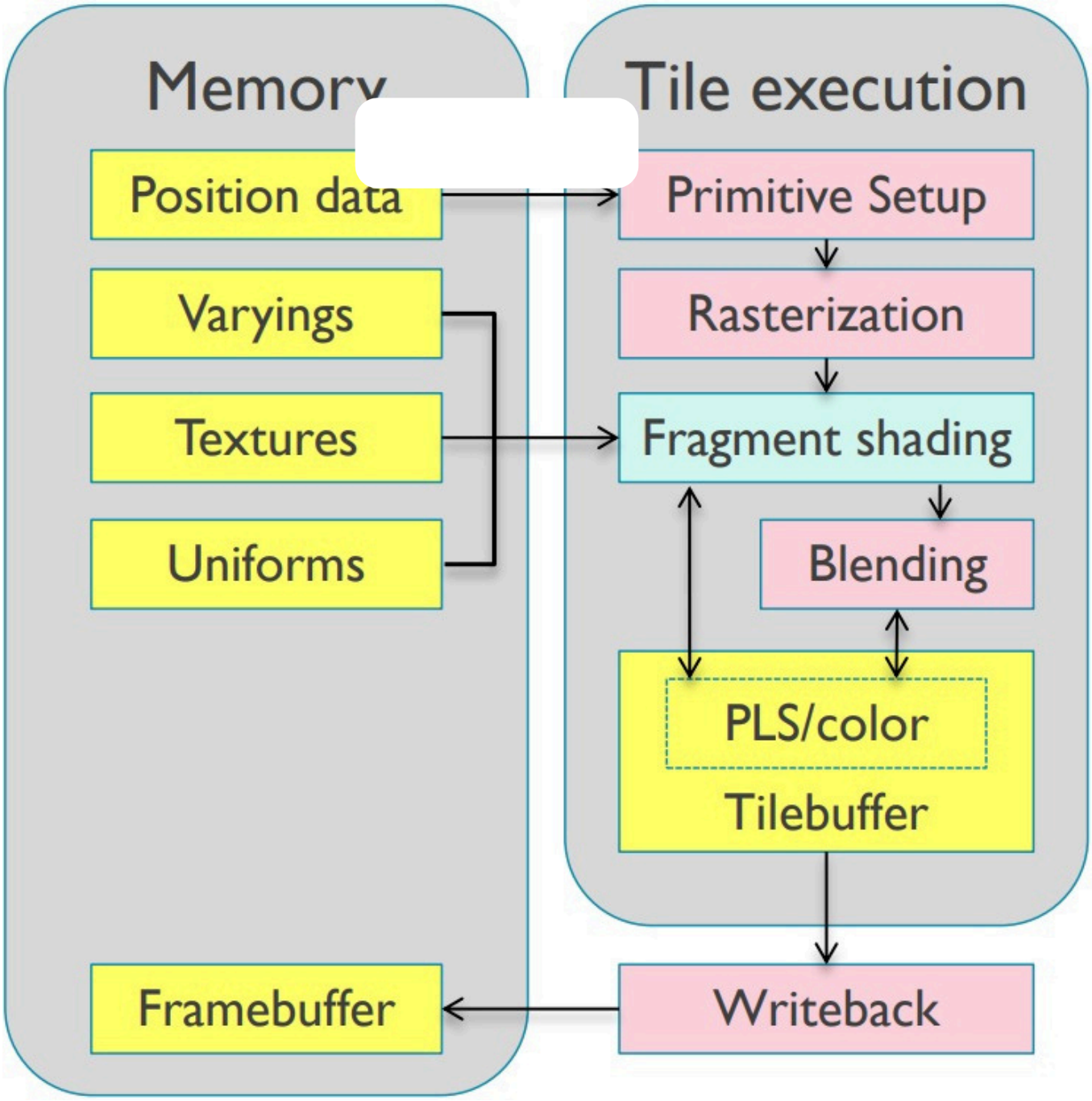
```
color,uv,normal |color,uv,normal |color,uv,normal | ...
```



Schematic diagram of IDVS vertex data flow splitting optimization and interaction.

12.4.12 Pixel Local Storage

Pixel Local Storage (PLS)is a data access method of OpenGL ES. Data declared with PLS will be stored in the GPU's Tile buffer (see the figure below).



When PLS is enabled, the rendering pipeline can efficiently perform color operations, blending, etc. There are three GLSL declaration PLS data keywords, as described in the following table:

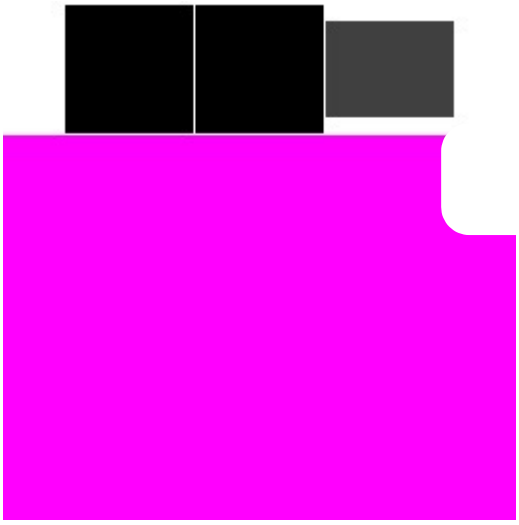
Keywords	effect
__pixel_localEXT	Can read and write data.
__pixel_local_inEXT	Read-only data.
__pixel_local_outEXT	Write data only.

Taking delayed rendering as an example, the application of PLS is as follows:

```
//------GBufferGenerate------ __pixel_local_outEXT
FragData//Write data only
```

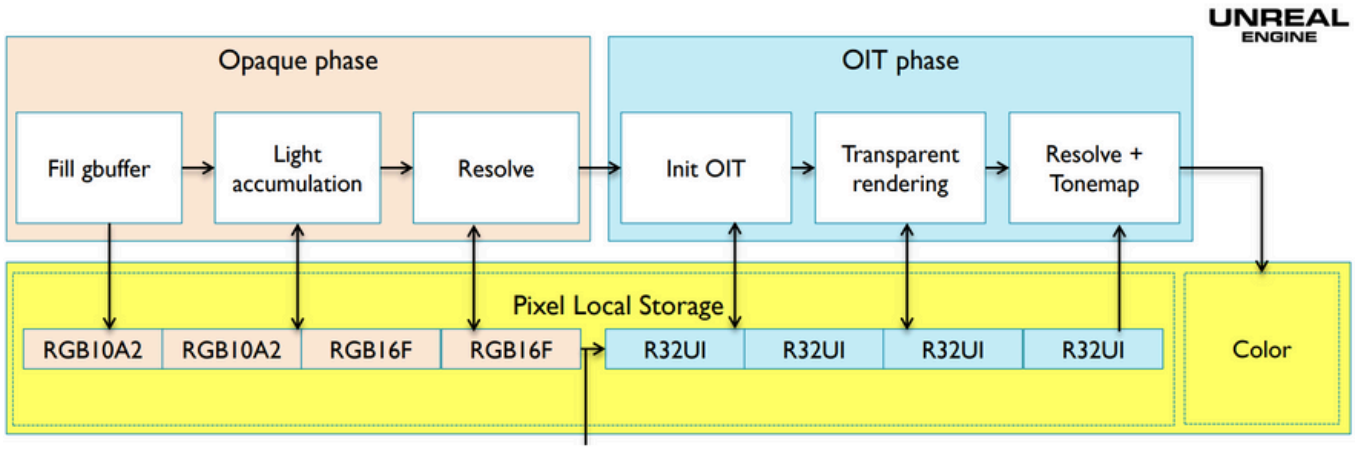
```
{
        layout(rgba8)highpvec4Color; layout(rg16f)
        highpvec2NormalXY;
        layout(rg16f)highpvec2NormalZ_LightingB; layout(rg16f)
        highpvec2LightingRG; }gbuf;


voidmain()
{
        gbuf.Color = CalcDiffuseColor(); vec3
        Normal = CalcNormal(); gbuf.NormalXY =
        Normal.xy; gbuf.NormalZ_LightingB.x =
                                                Normal.Z;
}

//------Light accumulation ------
__pixel_localEXT FragData//Can read and write data {

        layout(rgba8)highpvec4Color; layout(rg16f)highpvec2
        NormalXY; layout(rg16f)highpvec2NormalZ_LightingB;
        layout(rg16f)highpvec2LightingRG; }gbuf;



voidmain()
{
        vec3Lighting = CalcLighting(gbuf.NormalXY, gbuf.NormalZ_LightingB.x); gbuf.LightingRG +=
        Lighting.xy;
        gbuf.NormalZ_LightingB.y += Lighting.z;
}

//------Final coloring------
__pixel_local_inEXT FragData//Read-only data {

        layout(rgba8)highpvec4Color; layout(rg16f)highpvec2
        NormalXY; layout(rg16f)highpvec2NormalZ_LightingB;
        layout(rg16f)highpvec2LightingRG; }gbuf;



out highvec4FragColor;

voidmain()
{
        FragColor = resolve(gbuf.Color, gbuf.LightingRG, gbuf.NormalZ_LightingB.y);
}
```

The schematic diagram of the operation of deferred rendering using PLS is as follows (note that the red square in the upper right corner represents the rendering geometry data stage, and the green represents the rendering lighting stage):

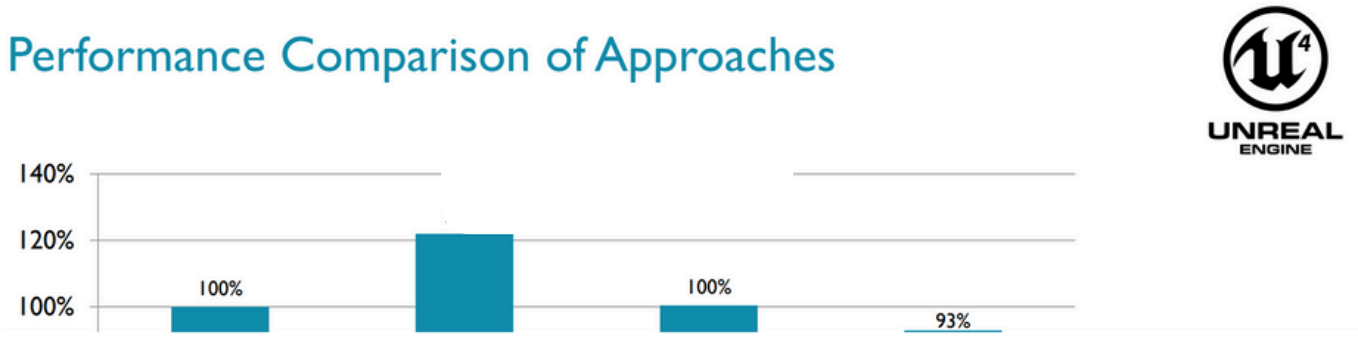


In addition to OpenGL ES, graphics APIs such as Metal, Vulkan, and D3D also provide corresponding interfaces, keywords, or tags to support data operations on GPU tiles.

The above code shows that the GBuffer data required for deferred shading is always in the PLS, and it is best to parse and return the final color without writing the GBuffer back to system memory (below).



PLS can improve performance by about 22%:



UE4 also uses PLS to implement efficient particle soft mixing:

Left: Particles normal blending mode. Right: Particles soft blending mode.

Vulkan also has a similar mechanism, called Subpass, see the following chapter.

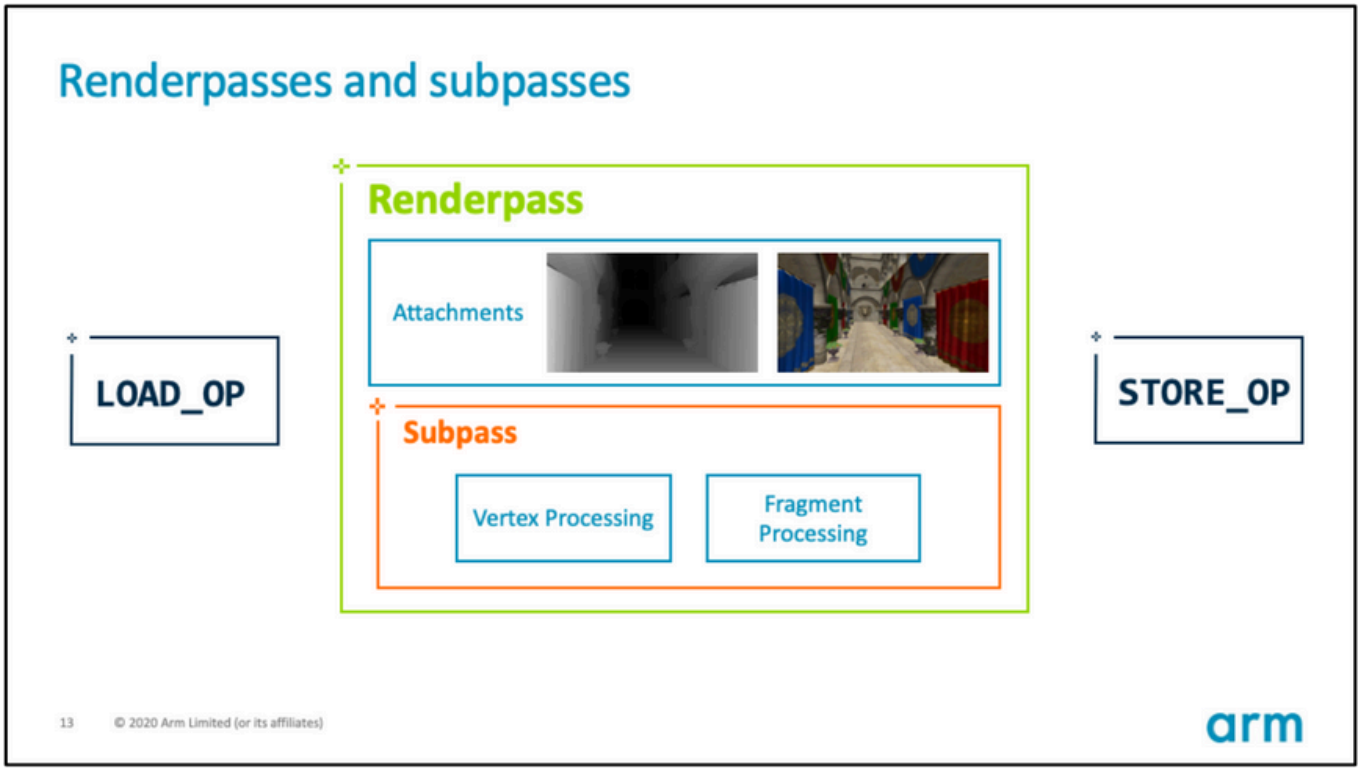
12.4.13 subpass

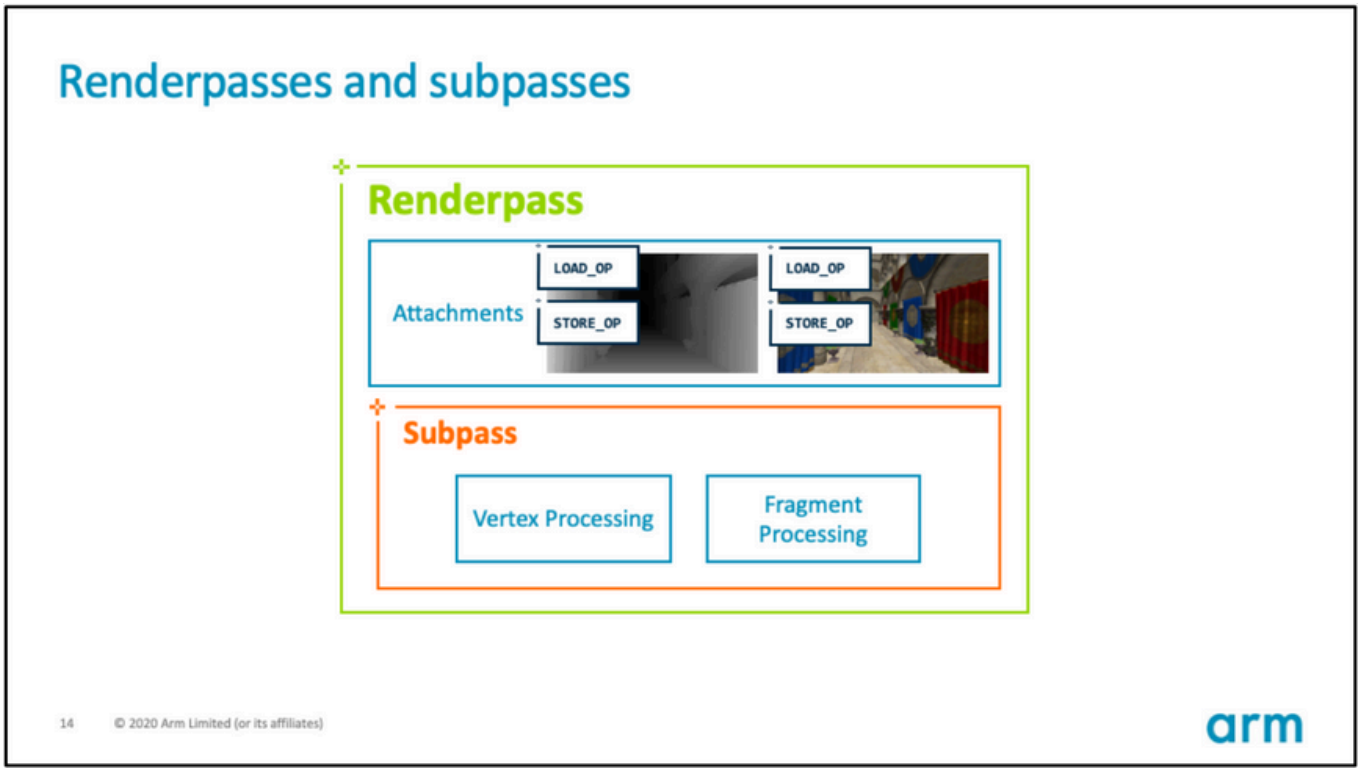
Subpassis a product that conforms to the TB(D)R hardware architecture and is suitable for modern graphics APIs such as Vulkan, DX12, and Metal. The underlying principle is similar to Pixel Local Storage.

Using Subpass requires the following special requirements:

- All subpasses must be in the same Render Pass.
- There is no need to sample surrounding neighborhood pixels. (Otherwise, data will be accessed across tiles, and all data accesses cannot be kept within the same tile.)
- GPU supports TB(D)R hardware architecture. Modern
- graphics APIs such as Vulkan, DX12, Metal, etc.

Each RenderPass and Subpass can specify loadOp and storeOp for each Attachment to precisely control their access behavior:





There are three types of subpass**loadOp tags:**

- **LOAD_OP_LOAD**:Loads an Attachment from global memory to a Tile.
- **LOAD_OP_CLEAR**:Clear the data in the Tile buffer.
- **LOAD_OP_DONT_CARE**:Do not perform any operation on the data in the Tile buffer. Usually, all the data in the Tile will be completely refreshed, which is more efficient than LOAD_OP_CLEAR.

The efficiency of the above three flags is: LOAD_OP_DONT_CARE > LOAD_OP_CLEAR > LOAD_OP_LOAD. Vulkan usage sample code:

```
VkAttachmentDescription colorAttachment = {};
colorAttachment.format          =VK_FORMAT_B8G8R8A8_SRGB;
colorAttachment.samples //Mark=VK_SAMPLE_COUNT_1_BIT;
loadOpforDONT_CARE.
colorAttachment.loadOp =          VK_ATTACHMENT_LOAD_OP_DONT_CARE;
```

There are two types of subpass**storeOp tags:**

- **STORE_OP_STORE**:Store the data in the tile to global memory. **STORE_OP_DONT_CARE**:Do not
- perform any storage operations on the data in the Tile buffer.

The execution efficiency of the above two flags: STORE_OP_DONT_CARE > STORE_OP_STORE. Vulkan usage sample code:

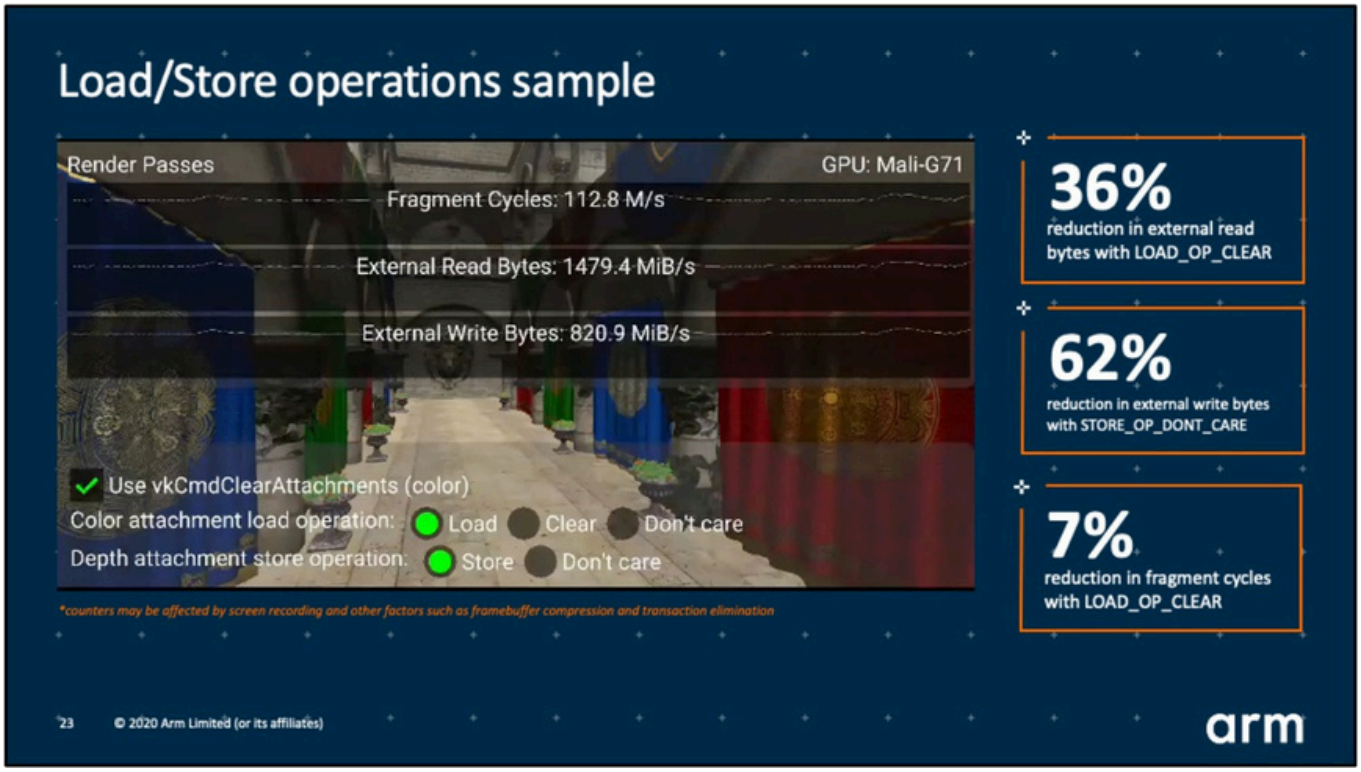
```
VkAttachmentDescription colorAttachment = {};
colorAttachment.format          =VK_FORMAT_B8G8R8A8_SRGB;
colorAttachment.samples //Mark=VK_SAMPLE_COUNT_1_BIT;
loadOpforDONT_CARE.
colorAttachment.loadOp = //MarkVK_ATTACHMENT_LOAD_OP_DONT_CARE;
storeOpforDONT_CARE.
colorAttachment.storeOp =          VK_ATTACHMENT_STORE_OP_DONT_CARE;
```

Unlike OpenGL ES, which has explicit keywords (__pixel_localEXT,__pixel_local_inEXT,)in the Shader to declare variables within a Tile, Vulkan must use the flags**TRANSIENT_ATTACHMENT**and **LAZILY_ALLOCATED**__pixel_local_outEXTin order to store an Attachment within a Tile:

```
VkImageCreateInfo       imageInfo{VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO};
imageInfo.flags             = flags;
imageInfo.imageType         = type;
imageInfo.format            = format;
imageInfo.extent            = extent;
imageInfo.samples           = sampleCount;
// ImageUsageTRANSIENT_ATTACHMENT's mark.
imageInfo.usage             = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;

VmaAllocation memory;
VmaAllocationCreateInfo         memoryInfo{};
memoryInfo.usage                    = memoryUsage;
// ImageMemory usageLAZILY_ALLOCATED's mark. memoryInfo.preferredFlags =
VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT;
```

```
//createImage.
autoresult = vmaCreateImage(device.get_memory_allocator(), &imageInfo, memoryInfo, &handle, &memory,
nullptr);
```

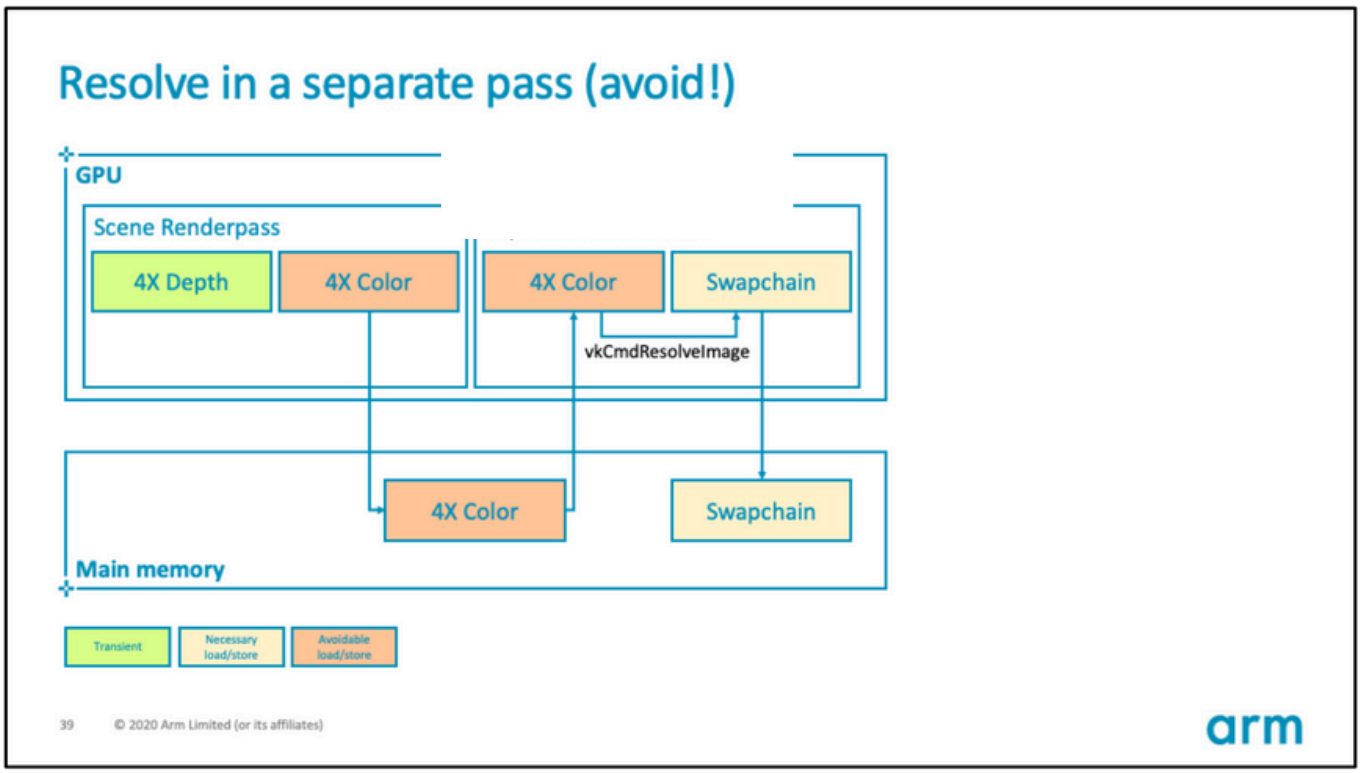
After optimizing with subpass's loadOp and storeOp, Vulkan's official test example shows that it can reduce 36% of global memory reads, 62% of global memory writes, and 7% of fragment execution cycles:

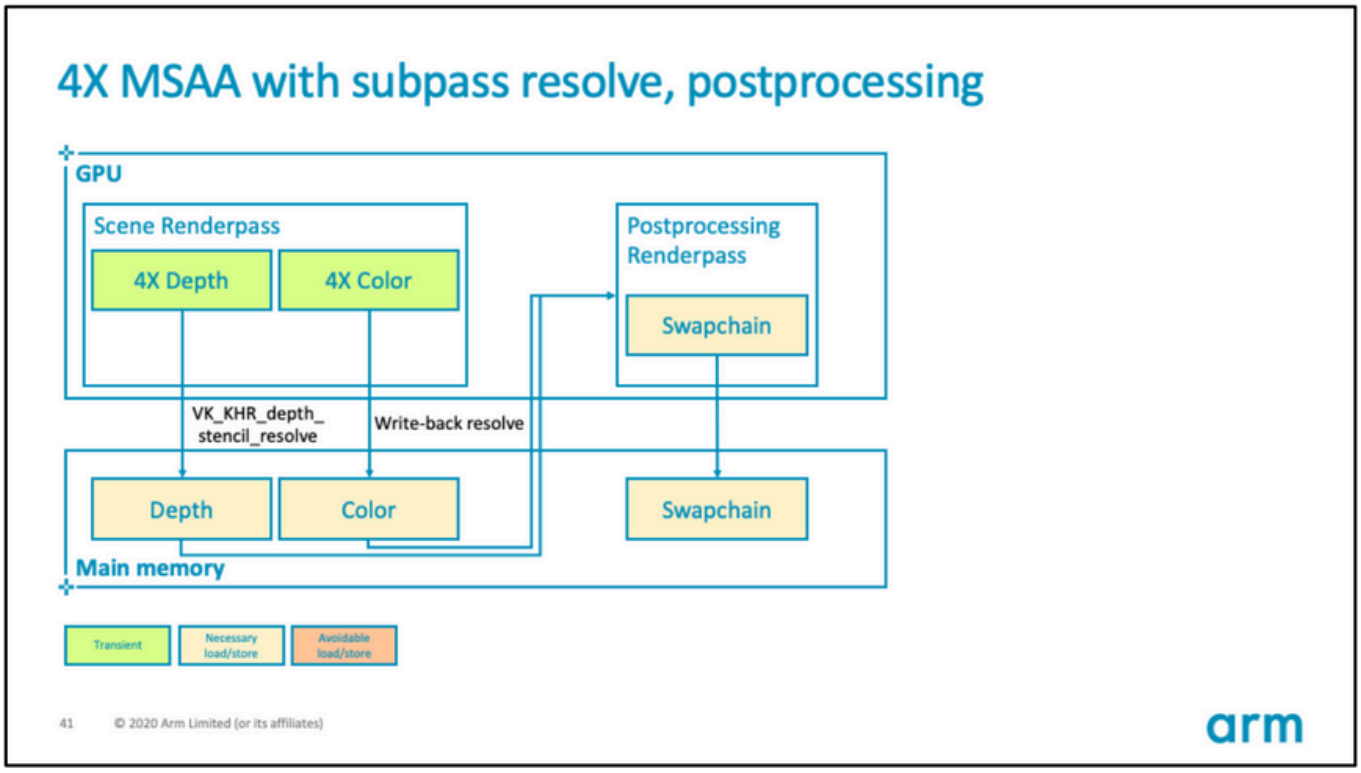


In addition, using the correct storeOp and loadOp can efficiently parse MSAA data within the Tile, as follows:

- The image (or attachment) with MSAA must be transient. The following markup can be used to obtain the resolved MSAA data at the end of the Render Pass:
    - loadOp = LOAD_OP_CLEAR; storeOp =
    - STORE_OP_DONT_CARE; Use the memory tag
    - LAZILY_ALLOCATED; Use the pResolveAttachments
    - tag in the subpass.
- For the depth template attachment, a similar effect can be achieved:
    - Use the VK_KHR_depth_stencil_resolve flag. Only
    - Vulkan 1.2 and above APIs are supported.

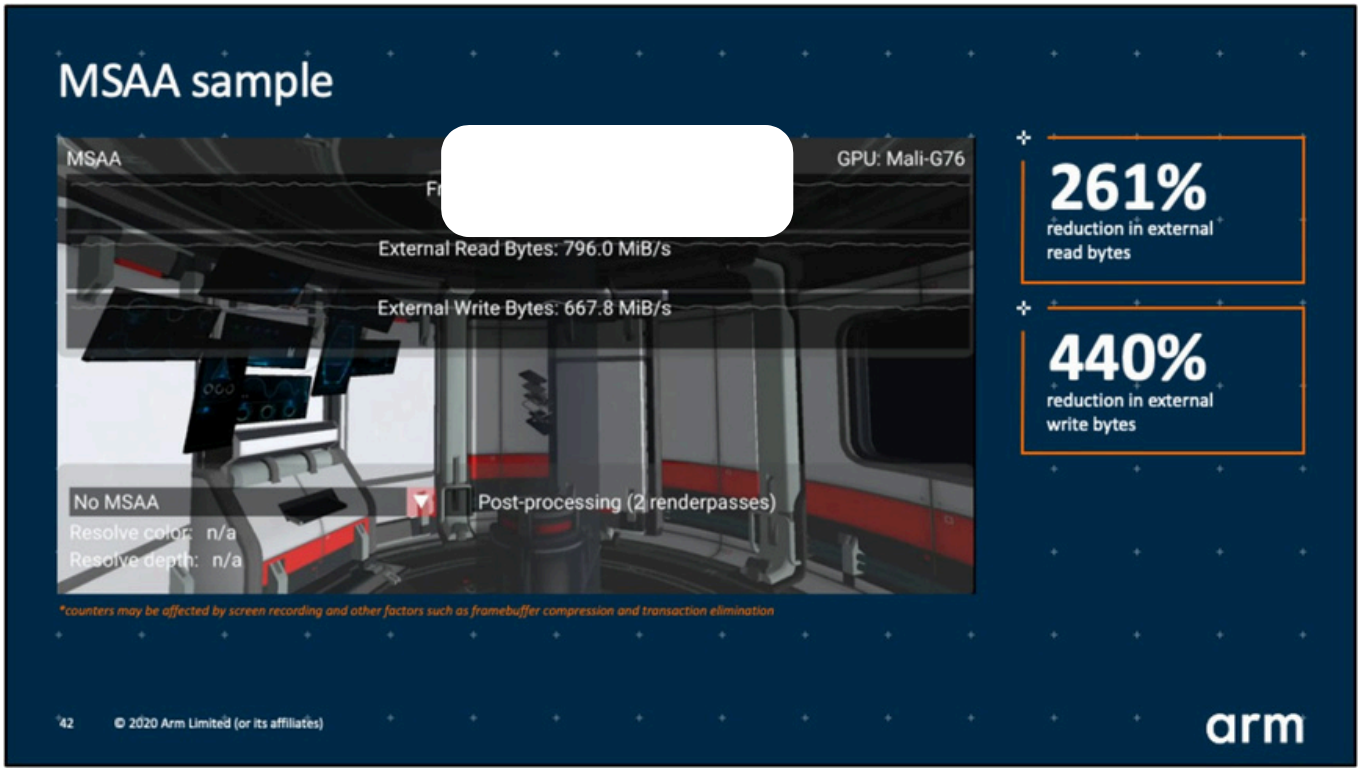
The above method can efficiently resolve the MSAA data in the tile without transferring the MSAA data to the global memory. In addition, it is necessary to**avoid**using the vkCmdResolveImage interface to resolve MSAA:





Top: Incorrect demonstration of using vkCmdResolveImage to resolve MSAA. Bottom: Correct demonstration of using within a Tile to resolve MSAA.

After optimizing MSAA resolution using subpass's loadOp and storeOp, Vulkan's official test sample shows that global memory reads can be reduced by**261%**and global memory writes by**440% ! !**



The optimization effect is obvious! ! What are you waiting for? Just take up the advantageous weapon of subpass to optimize the application! !

For more instructions, see the github of Vulkan official organization KhronosGroup:Appropriate use of render pass attachments .

For information about UE's encapsulation of Subpass, please refer to:10.4.4.2 Subpass rendering .

12.4.14 Adaptive Scalable Texture Compression

Adaptive Scalable Texture Compression (ASTC)is a texture compression format jointly developed by Arm and AMD. Unlike the fixed block size (4x4) of ETC and ETC2, ASTC supports compression of variable block sizes, thereby obtaining flexible texture data with a higher compression rate and reducing GPU bandwidth and energy consumption .

Although ASTC has not yet become a standard format for OpenGL and only exists as an extension, it is currently widely supported by mainstream GPUs and can be said to be a non-standard extension. However, in Vulkan, ASTC is already a standard feature. Specifically, ASTC supports the following features:

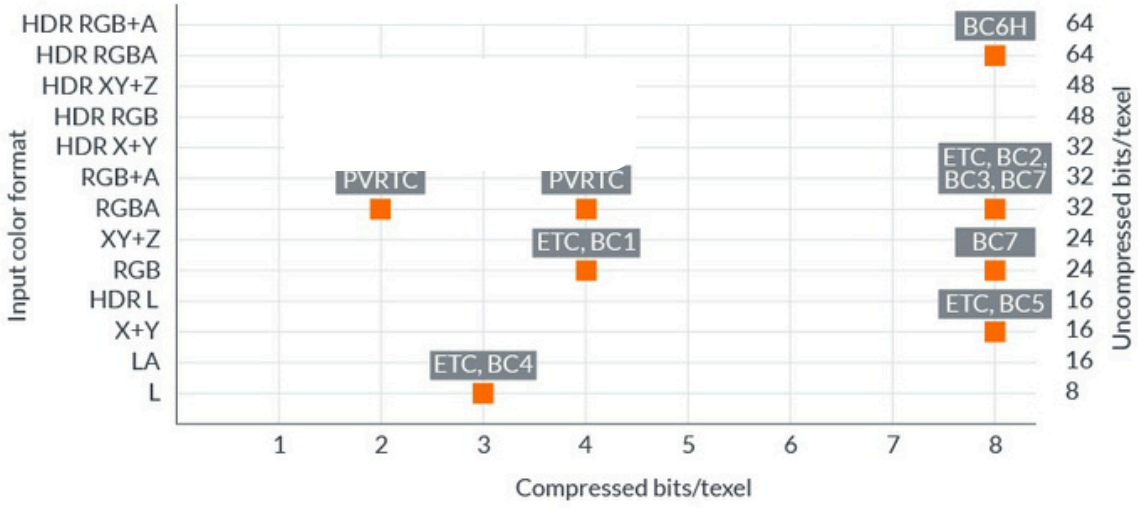
- **Flexible format.**ASTC can compress data between 1 and 4 channels, including one noncorrelated channel, such as RGB+A (correlated RGB, non-correlated alpha). And the block size is variable, such as 4x4, 5x4, 6x5, 10X5, etc.

  > Adreno A5X and above GPU chips support the following ASTC formats with different block sizes (including two-dimensional and three-dimensional):
  >
  > - ASTC_4X4
  >
  >   ASTC_5X4

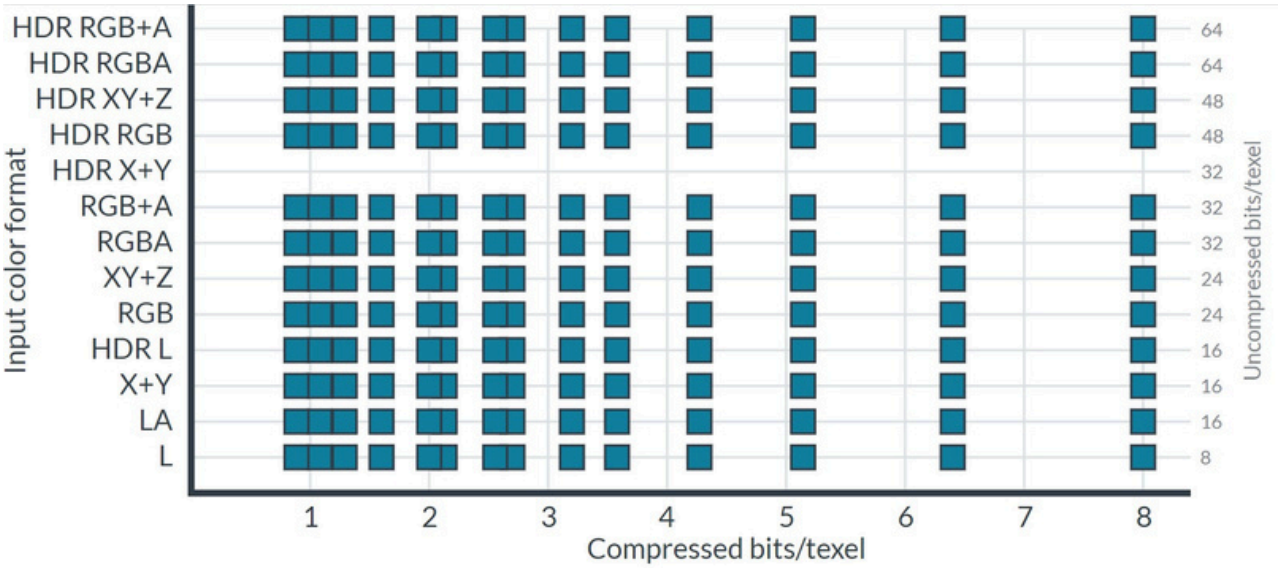
- ASTC_5X5
- ASTC_6X5
- ASTC_6X6
- ASTC_8X5
- ASTC_8X6
- ASTC_8X8
- ASTC_10X5
- ASTC_10X6
- ASTC_10X8
- ASTC_10X10
- ASTC_12X10
- ASTC_12X12
- ASTC_3X3X3
- ASTC_4X3X3
- ASTC_4X4X3
- ASTC_4X4X4
- ASTC_5X4X4
- ASTC_5X5X4
- ASTC_5X5X5
- ASTC_6X5X5
- ASTC_6X6X5
- ASTC_6X6X6

- **Flexible bitrates.**ASTC offers a wide range of bitrates to choose from when compressing images, between 0.89 bits and 8 bits per texel (bpt). The choice of bitrate is independent of the choice of color format. Traditional formats such as ETC can only have integer bitrates.
- **Advanced Format Support.**ASTC can compress images in Low Dynamic Range (LDR), LDR sRGB, High Dynamic Range (HDR) color spaces, and can also compress 3D volumetric textures.
- **Improved image quality.**Despite its high format flexibility, ASTC outperforms almost all traditional texture compression formats (ETC2, PVRCT, BC, etc.) in image quality at the same bit rate.

- **The format matrix is fully covered.**Before ASTC appeared, the traditional texture compression format supported relatively few combinations of color formats and bit rates, as shown in the following figure:



The above formats are also limited by the graphics API or operating system, so the compression options for any single platform are very limited. The emergence of ASTC solves the above problems and almost achieves complete coverage of the required format matrix, providing content creators with a wide range of bitrate options. The following figure shows the available formats and bitrates:



How does ASTC achieve the above goals? The answer lies in the fact that ASTC uses a special compression algorithm and data structure. The key points and explanations of ASTC's algorithm technology are as follows:

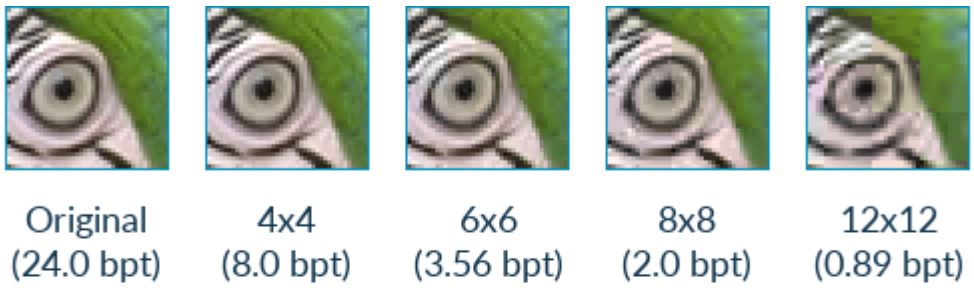
- **Block Compression**

  A compression format for real-time graphics needs to be able to quickly and efficiently convert random samples into textures, so the compression technology must do the following:

  - Given only a sampling coordinate, calculate the address of the data in memory. Ability to
  - decompress random samples without decompressing too much surrounding data.

The standard solution used by all contemporary real-time compression formats (including ASTC) is to split the image into fixed-size pixel blocks, and then each block is compressed into a fixed number of output bits. This guarantees fast shader access to texels in arbitrary order,
with good decompression cost.

2D Block footprints in ASTC range from 4x4 texels to 12x12 texels, all of which are compressed into 128-bit output blocks. By dividing 128 bits by the number of pixels in the occupied space, the format bit rate can be obtained, which ranges from 8 bpt ( ) to 0.89 bpt ( ). Below is a comparison of the image quality of different bit rates:128/(4 · 4)128/(12 · 12)



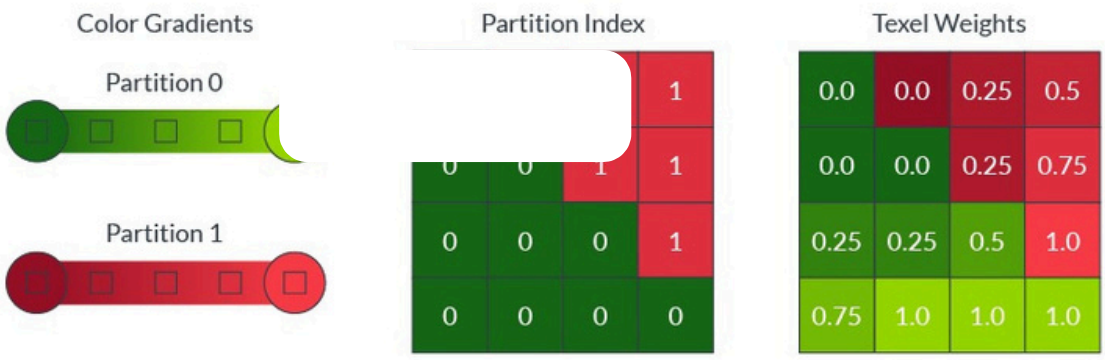
- **Color endpoint**

  The color data for a block is encoded as a gradient between two color endpoints. Each texel chooses a position along the gradient, which is then interpolated during decompression. ASTC supports a 16-color endpoint encoding scheme, called endpoint mode. The endpoint mode options allow changing the following:

  - The number of color channels. For example: luma, luma+alpha, rgb, or rgba.
  - Encoding method. For example: direct, radix+offset, radix+scale, or quantization level.
  - The range of the data. For example: low dynamic range or high dynamic range.

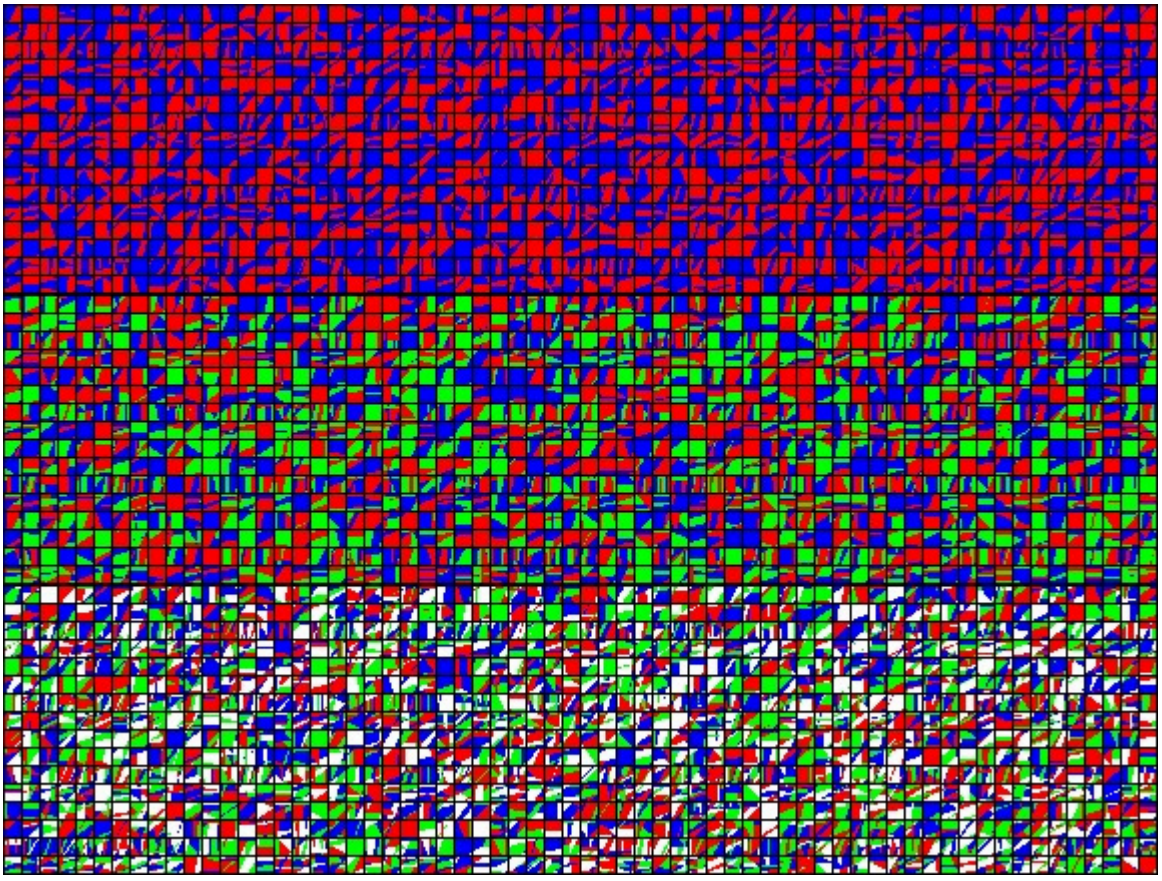
  Allows selection of different endpoint modes and endpoint color BISE quantization levels on a block-by-block basis.

- **Color partition** The colors within a block are often complex, and a single-color gradient often cannot accurately
  capture all the colors within the block. For example, a red ball lying on green grass requires two-color division, as shown in the following figure:



ASTC allows a single block to reference up to four color gradients, called partitions. For decompression, each texel is assigned to a separate partition.

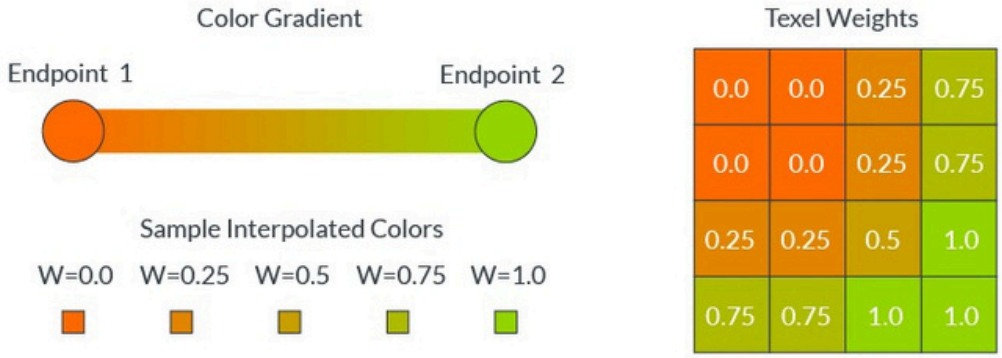
Directly storing the partition assignment for each texel would require a lot of decompression hardware to store all block sizes. Instead, ASTC uses the partition index as a seed value to algorithmically generate a series of patterns. The compression process selects the best matching pattern for each block, and then the block only needs to store the index of the best matching pattern. The following figure shows the patterns generated by 2 (top of the image), 3 (middle of the image), and 4 (bottom of the image) partitions for an 8×8 block size:



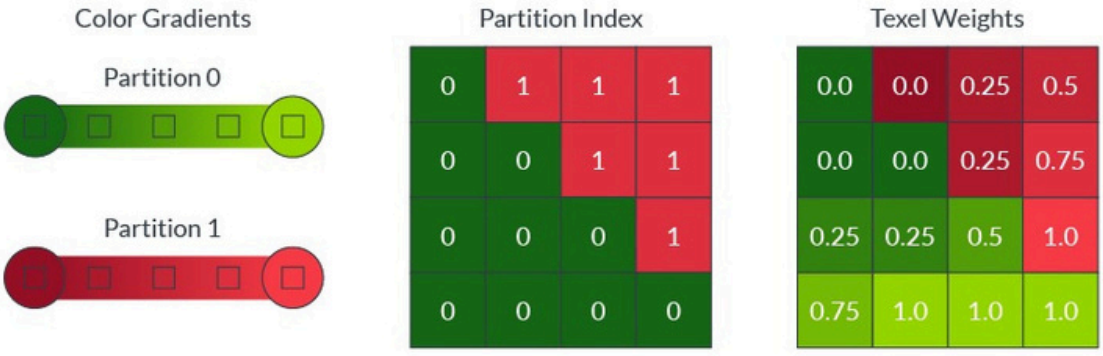
The number of partitions and the partition index can be selected on a per-block basis, and different color endpoint patterns can be selected on each partition.

- **Color Coding**

ASTC uses a gradient to specify the color value of each texel. Each compressed block stores the endpoint colors of the gradient, as well as the interpolation weights for each pixel. During decompression, the color value of each pixel is interpolated between the two endpoint colors based on the weight of each pixel. The following figure shows the interpolation of various texel weights:



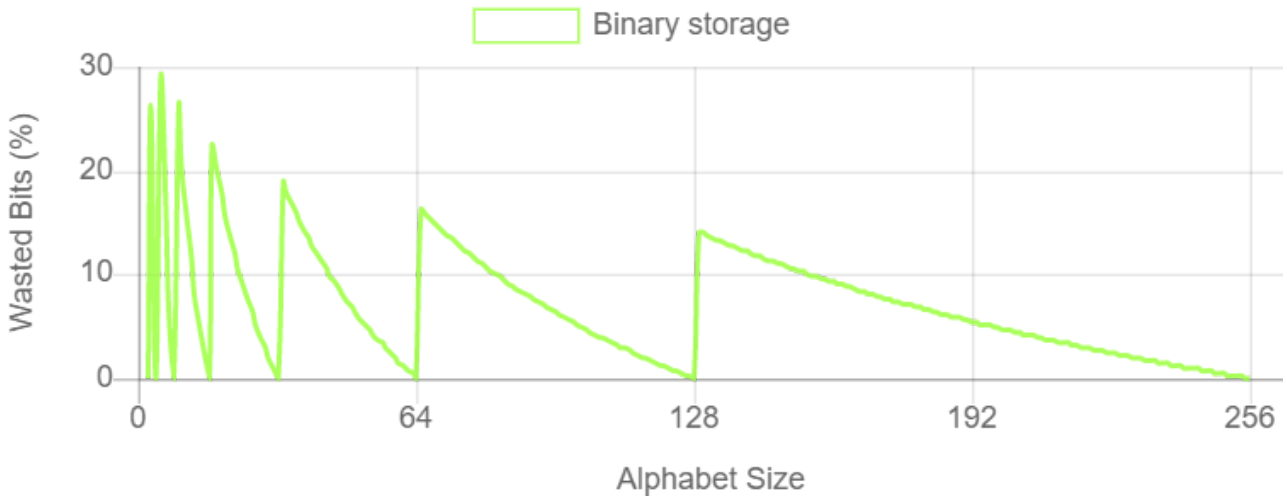
Blocks often contain complex color distributions, such as a red ball on green grass. In these cases, a single color gradient cannot accurately represent all the different texel color values. ASTC allows a block to define up to four different color gradients, called partitions, and can assign each texel to a separate partition. The following figure shows how the partition index specifies the color gradient for each texel (two partitions, one for the red ball pixel and one for the
green grass pixel):



- **Storing alphabet**

  Although the color and weight values for each pixel are theoretically floating point values, there are too few bits to store the actual values directly. To reduce the storage size, these values must be quantized during compression. For example, if you have a floating point weight for each texel in the range 0.0 to 1.0, you can choose to quantize to 5 values: 0.0, 0.25, 0.5, 0.75, and 1.0, and then use integers 0-4 to represent these five quantized values in storage.

In general, if you choose to quantize N levels, you need to be able to efficiently store the characters in a table of N symbols. A table of N symbols contains log2(N) bits of information for each character. If you have a table of 5 possible symbols, each character contains about 2.32 bits of information, but simple binary storage requires rounding to 3 bits, which wastes 22.3% of the storage capacity. The following chart shows the percentage of bit space wasted to store any N-symbol table of characters using simple binary encoding:



The above chart shows that for most character sizes, using an integer number of bits per character wastes a lot of storage capacity. Efficiency is critical for compression formats, so this is a problem that ASTC needs to solve.

One solution is to round the quantization level to the next power of 2, so that no extra bits are wasted. However, this solution forces the encoder to consume bits that could be used elsewhere to gain greater benefits, so this solution reduces image quality and is not an optimal solution.

- **Quint and trit**

  A more efficient solution is to combine three quint characters together instead of one quint character into three bits. Three characters from the five-letter alphabet have combinations, containing 6.97 bits of information. We can store these three quint characters in 7 bits, with a storage waste of only 0.5%.53= 125

  We can also construct a three-symbol alphabet in a similar way, called a trit, and combine the trits into groups of five. Each trit has combinations and contains 7.92 bits of information. We can store these five trit characters in 8 bits with only 1% storage waste.35= 243

- **Bounded Integer Sequence Encoding**

  The Bounded Integer Sequence Encoding (BISE)used by ASTC allows character sequences to be stored using arbitrary characters up to 256 symbols. Each character size is encoded using the most space-efficient bit, tuple, and quintuple.

  - An alphabet containing at most symbols can be encoded using n bits per character.
    $2_n - 1$

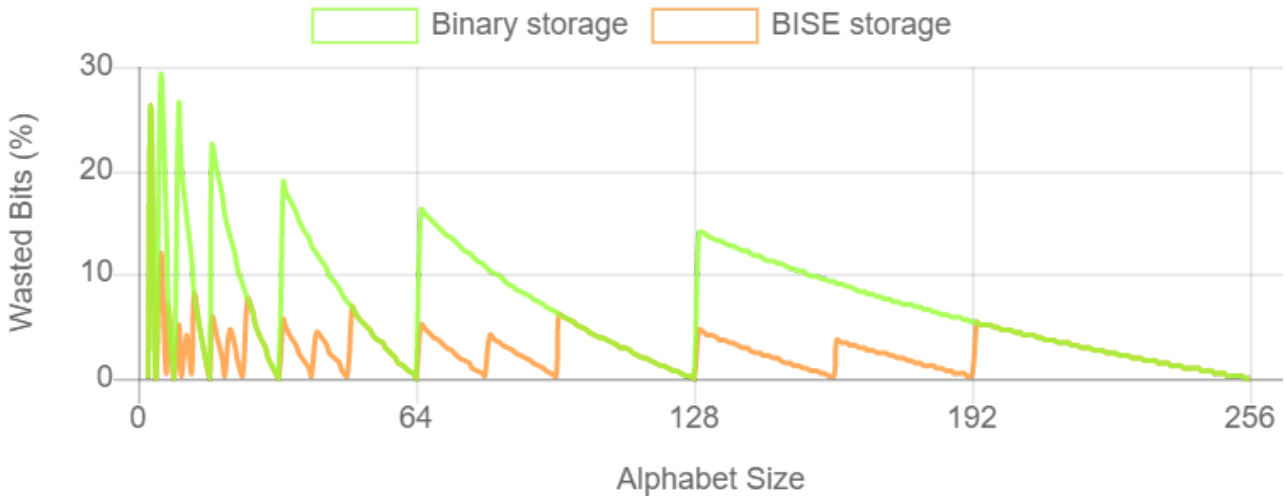
- An alphabet containing at most symbols can be encoded using n bits (m) and a trit (t) per character, and reconstructed using the equation $.3 \cdot (2n-1)(t \cdot 2n) + m$
- An alphabet containing at most symbols can be encoded using n bits (m) and a quint (q) per character and reconstructed $5 \cdot (2n-1)(q \cdot 2n) + m$

When the number of characters in the sequence is not a multiple of 3 or 5, it is necessary to avoid wasting storage space at the end of the sequence, so another constraint is added to the encoding. If the last few values to be encoded in the sequence are zero, then the last few bits of the encoded bit string must also be zero. Ideally, the number of non-zero bits is easy to calculate and does not depend on the size of the previously encoded values. This is difficult to handle properly during compression, but it is possible. It means that there is no need to store any padding after the end of the bit sequence, because we can safely assume that they are zero bits.

With this constraint, BISE encodes S strings from an alphabet of N symbols using a fixed number of bits, through smart packing of bits, trits, and quints:

- The maximum value of S is , using bits.$2N-1N \cdot S$
- The maximum value of S is , using bits.$3 \cdot 2N-1N \cdot S+\text{ceil}(8S/5)$
- The maximum value of S is , using bits.$5 \cdot 2N-1N \cdot S+\text{ceil}(7S/3)$

The compressor chooses the option that produces the smallest storage space for the size of the alphabet being stored. Some use binary, some use bits and trit, and some use bits and quint. The following figure shows the efficiency gain of BISE storage over binary storage:

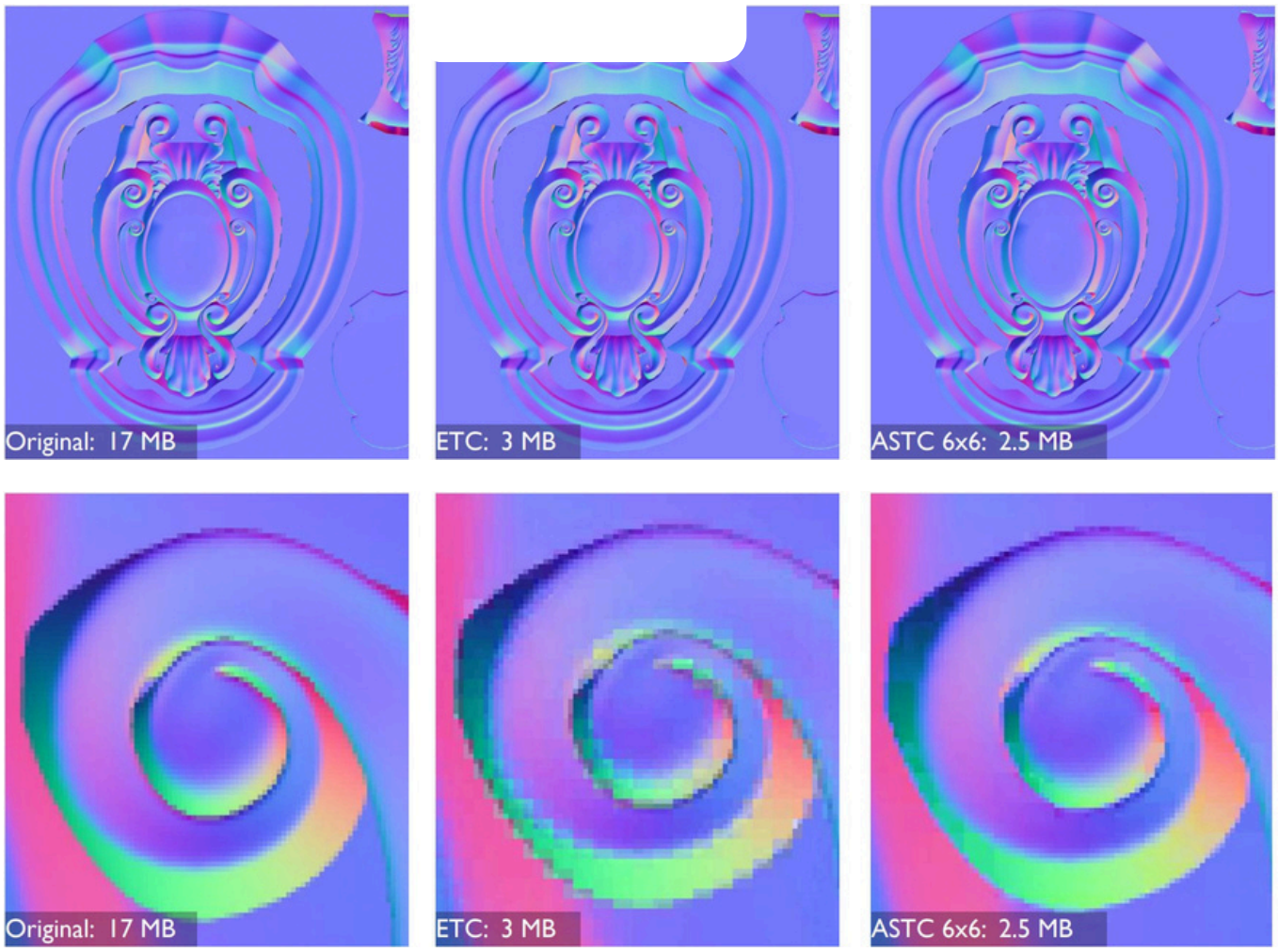


In addition, during the compression process, the best encoding is selected for each block. When calculating the texel weight value, in addition to the above-mentioned BISE, there is also a dualplane weights algorithm.

ASTC is free to use, easy to integrate, and supported by many mainstream systems and hardware. Supporting ASTC requires the following OpenGL extensions:

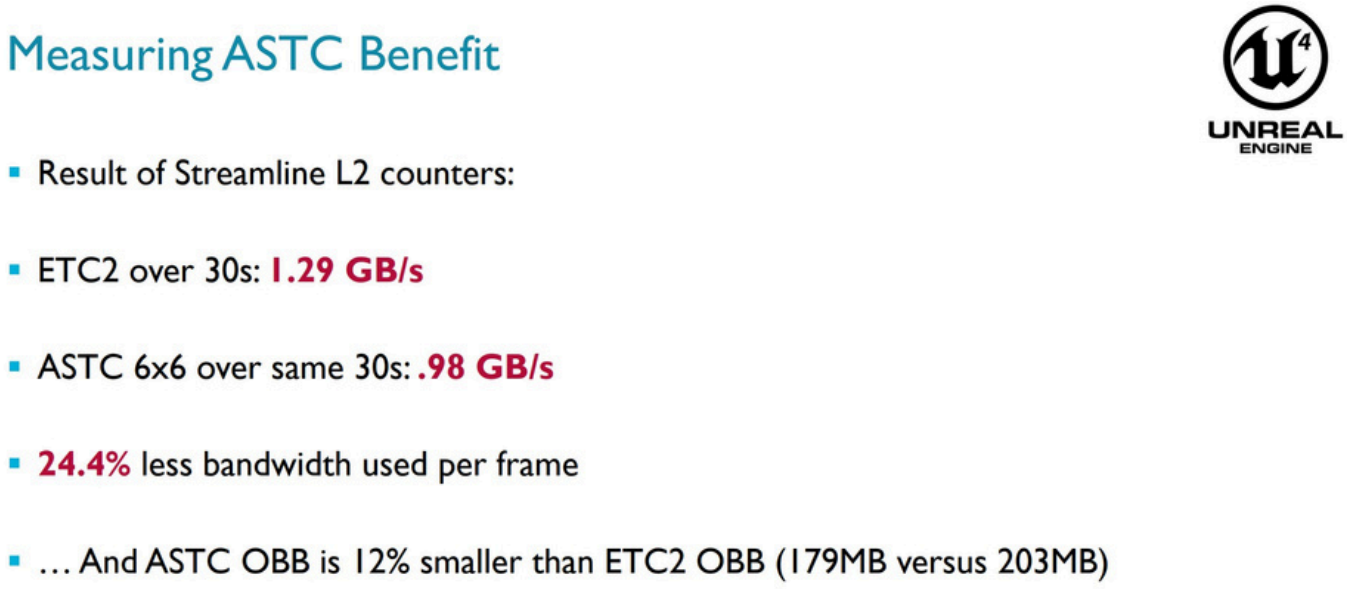
```
GL_AMD_compressed_ATC_texture
GL_ANDROID_extension_pack_es31a
```

Compared with traditional texture compression formats (ETC, BC, PVRTC, etc.), the compression effect of ASTC is very obvious, the image quality is closer to the original image, and the compression rate is higher:



Left: Original normal map; Middle: Compressed to ETC; Right: Compressed to ASTC.

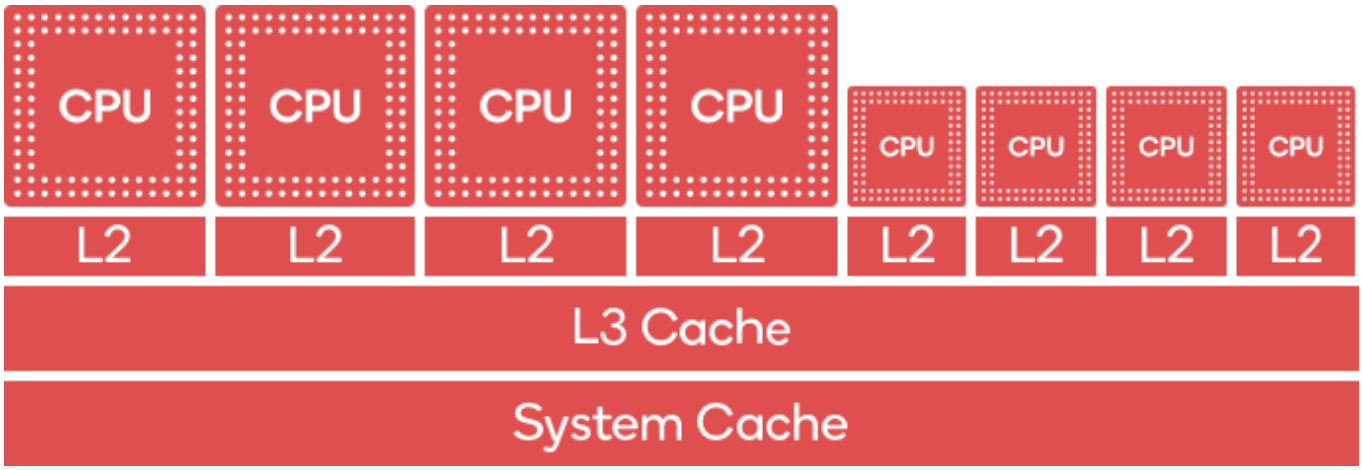
The intuitive benefit brought about by this is that less memory and bandwidth are occupied, and the bandwidth per frame can be reduced by about 24.4%:



For more details about ASTC, seeAdaptive Scalable Texture Compression .

12.4.15 big.LITTLE Core

Mobile CPUs (note that they are not GPUs, such as Qualcomm Keyo CPUs) have a**big.LITTLE** combination architecture, which was first proposed by Arm. This architecture has both big cores and little cores. The big core is optimized for high performance, while the little core is optimized for energy consumption.



Qualcomm Keyo CPU's big.LITTLE architecture. The 4 on the left are big cores, which have high performance but consume more power, while the 4 on the right are little cores, which have lower performance but are more power-efficient.

The big.LITTLE architecture features are as follows:

- By combining two very different processors in one SoC, it responds to the changing performance requirements of smart devices.
- The big.LITTLE software automatically handles the task allocation to the appropriate CPU core. The operating system directly senses the high-performance and high-efficiency cores in the system and can dynamically allocate each task to the appropriate core based on performance requirements.

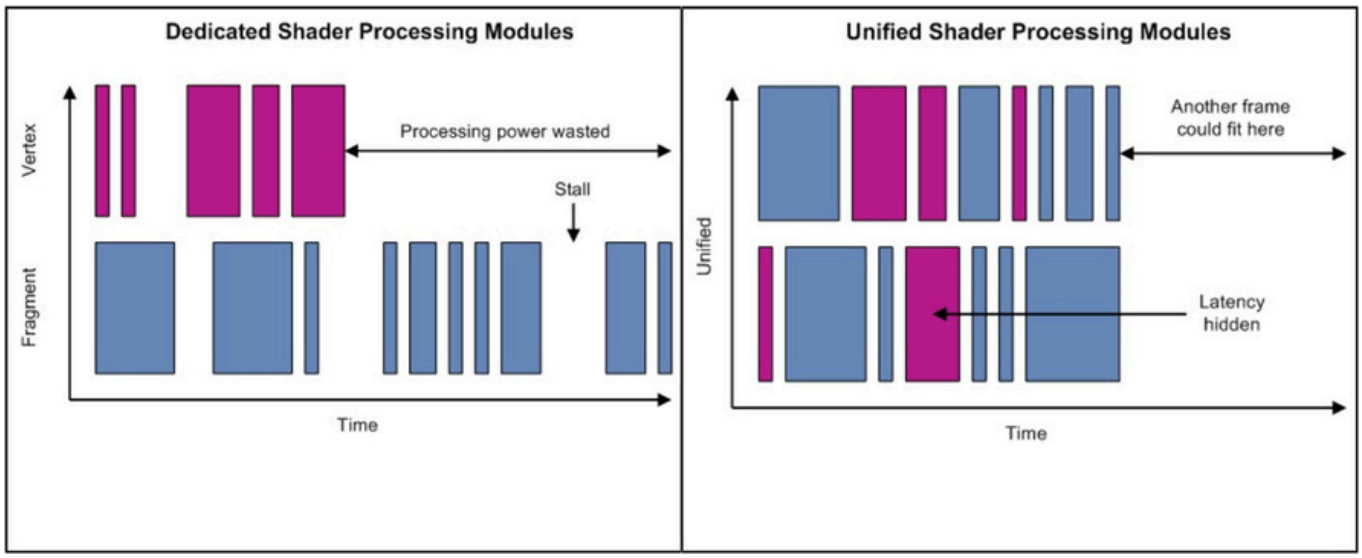
Understanding and how to use the features of this architecture is critical to optimizing performance and power efficiency, which will result in longer gaming sessions and cooler gaming experiences.

To improve the efficiency of big.LITTLE, try to give priority to the use of little cores. Assuming the expected frame time is 16ms (60FPS), developers can use tools (such as Snapdragon Profiler) to identify tasks and move them to LITTLE cores . For example, a game with cloth simulation takes 3 milliseconds to execute on a big core, but may take 10 milliseconds to execute on a little core. As long as this execution time is acceptable (the frame budget in this case is 16ms), it should be moved
to the little core to reduce the utilization of the big core and improve power efficiency.

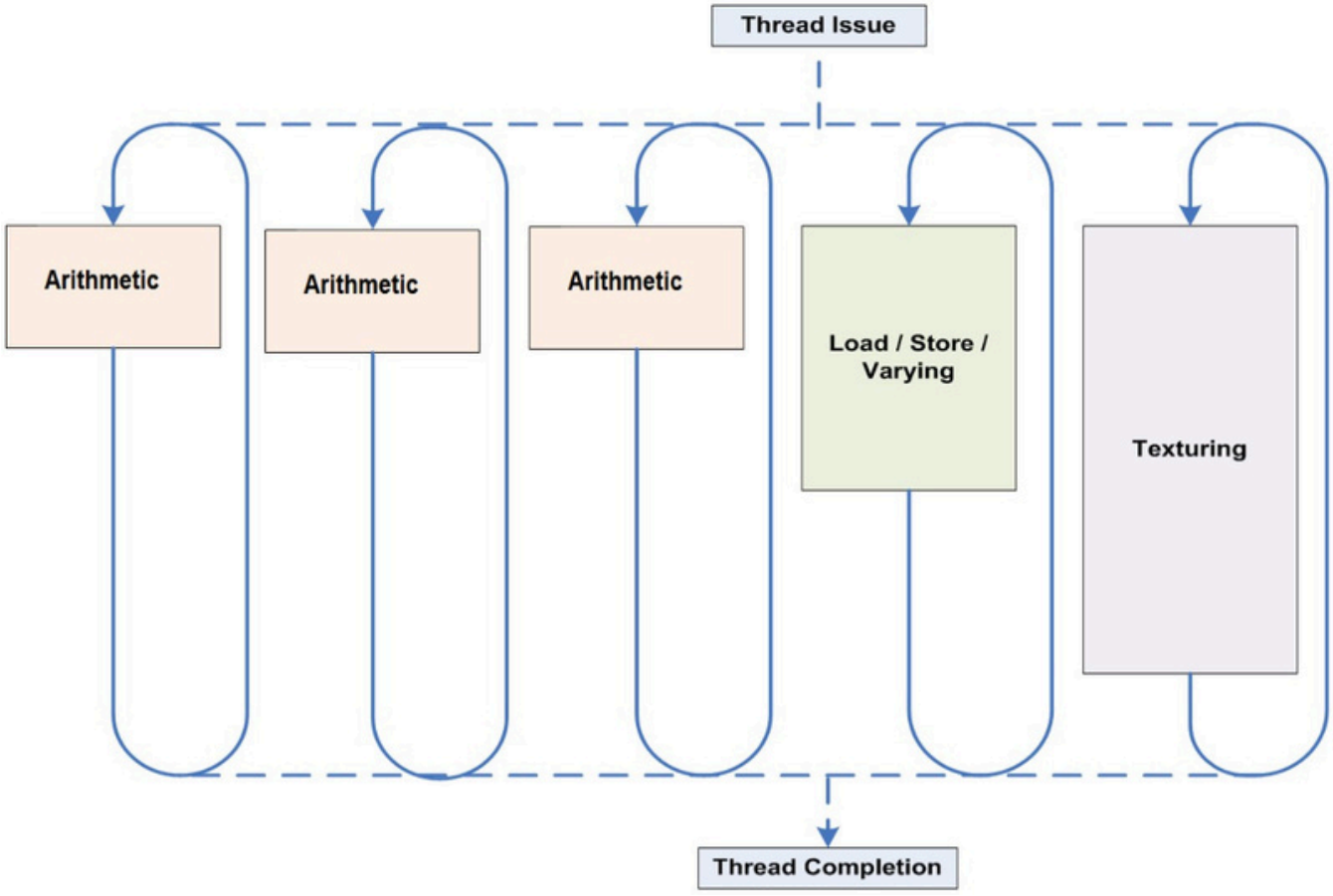
Mobile SoC manufacturers (such as Qualcomm and Arm) usually provide relevant SDKs and APIs for developers to specify which type of CPU core the task runs on. For details, please refer to: Controlling Task Execution .

12.4.16 Other technical points

In addition to the technical points mentioned in the above section, there are actually many other technologies in mobile chips or graphics APIs, such as SIMD, SIMT, Unified shader architecture (see the figure below), Scalar architecture (scalar shader architecture), Tripipe (see the figure below), etc.
For more technical details, you can read my other article about GPU:In-depth GPU hardware architecture and operation mechanism .



Left: Separate Shader Processing Unit, Right: Unified Shader Processing Unit. It can be seen that the processor of the latter is basically running at full capacity, thus reducing waiting and idling, and improving overall computing power.



Schematic diagram of the Tripipe structure in the Mali GPU, which includes 3 computing units, 1 access unit and 1 texture unit, with 128-bit bandwidth, 2 times FP64, 4 times FP32, and 8 times FP16

operating efficiency.
In addition, OpenGL ES has many extensions that can improve performance, such as reading and writing operations on texture sub-regions:

```
KHR_partial_update
EXT_buffer_age
```

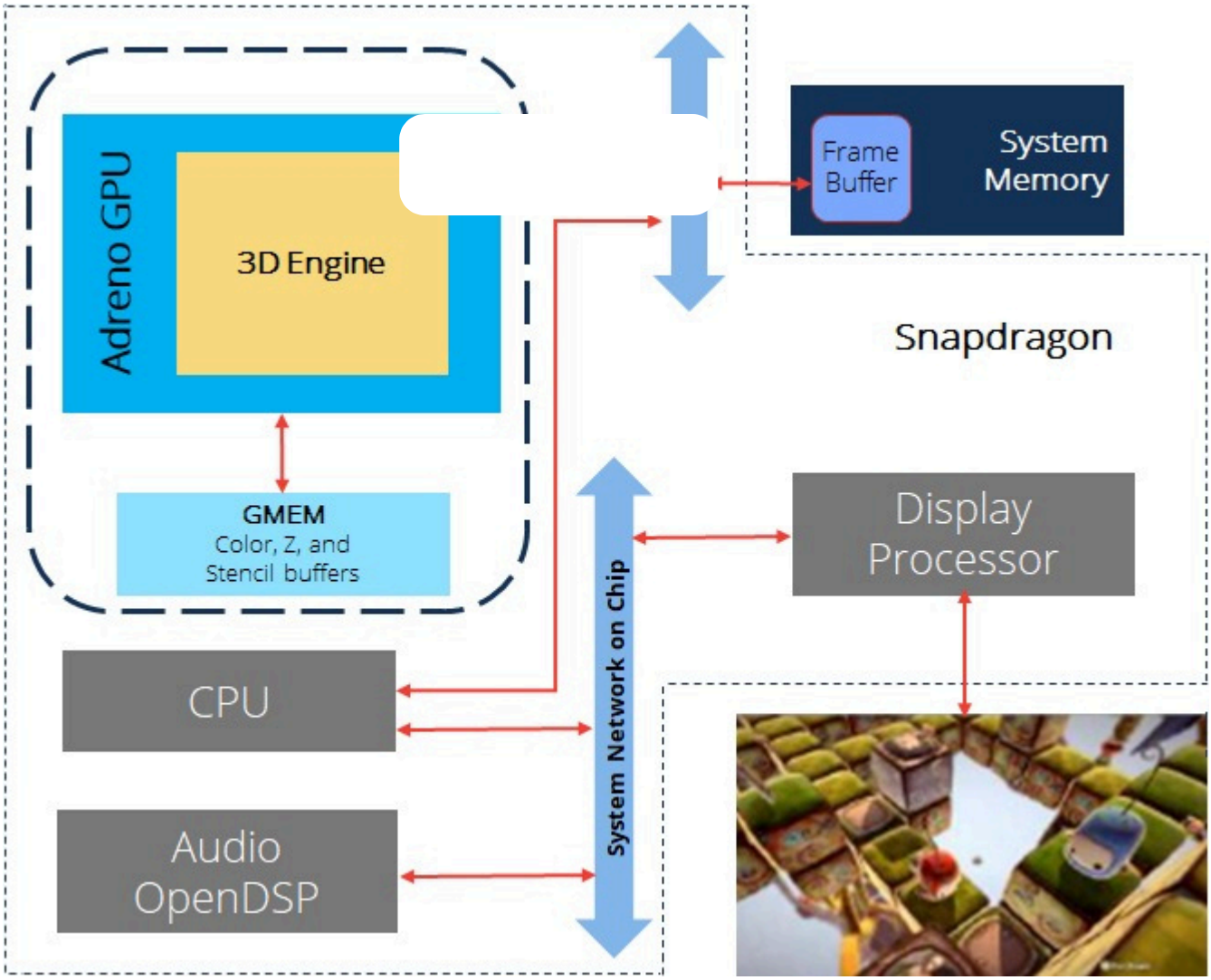
This extension allows the caller to use the Backbuffer time to specify multiple boxes to draw frame content. This technology is similar to TE, but does not write data to the Tile buffer.

12.5 Mobile GPU Architecture and Mechanism

This chapter will explain the hardware architecture and operating mechanism of mobile GPU.

12.5.1 Overview of Mobile GPU

Due to the portability of mobile GPUs, the three indicators of PPA need to be considered. Therefore, designing a high-performance GPU is extremely difficult and challenging. Currently, there are major GPU manufacturers such as Qualcomm, Arm, and Imagination Tech, and their representative products are Adreno, Mali, and PowerVR. Mobile GPUs are usually integrated into SoCs, forming an organic hardware architecture system with CPUs, memory, and other devices.



Snapdragon framework diagram. It includes CPU, Adreno GPU, memory and other components, and exchanges data through Bus, Network, etc.

As time goes by, mobile hardware develops, and more and more new graphics APIs and rendering features are migrated to mobile devices, as shown in:

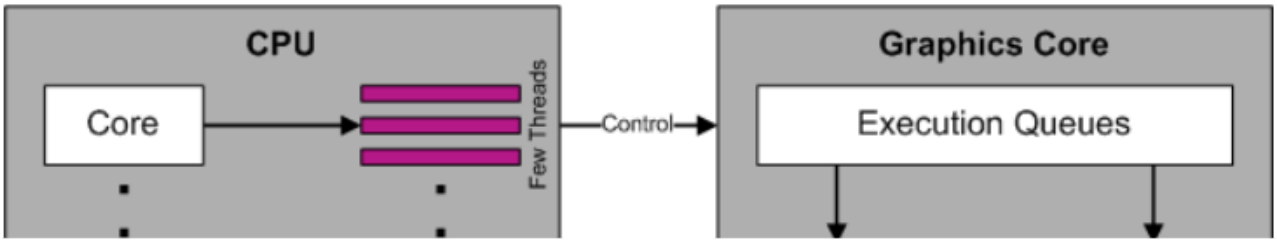
- Mainstream GPUs support graphics APIs such as DX12, Vulkan1.2, OpenGL ES 3.2, and new rendering features such as VRS, Mesh Shading, Ray Tracing, and WaveMath.
- GPU throughput and computing power have been greatly improved, including ALU, Texture, Memory and other aspects:

Qualcomm Adreno 640 GPU performance at a glance, with Xbox One performance data on the right.

- Memory bandwidth is increased and energy efficiency is improved.

- Power-saving features abound.
    - Render Target Compression, FP16 math ops, ASTC, Vulkan Subpasses.
    - UBWC, AFBC, IDVS, PLS, etc.

- Mobile Soc is widely used in VR applications and brings many dedicated optimization technologies.
- With the improvement and enhancement of Compute Shader capabilities, support for OpenCL libraries have become more complete.
- The number of parallel operations increases and the throughput improves.



Schematic diagram of CPU and GPU operation. It can be seen that the GPU cache is small but has a large number of threads.

The relevant concepts and terms in the mobile GPU architecture are explained as follows:

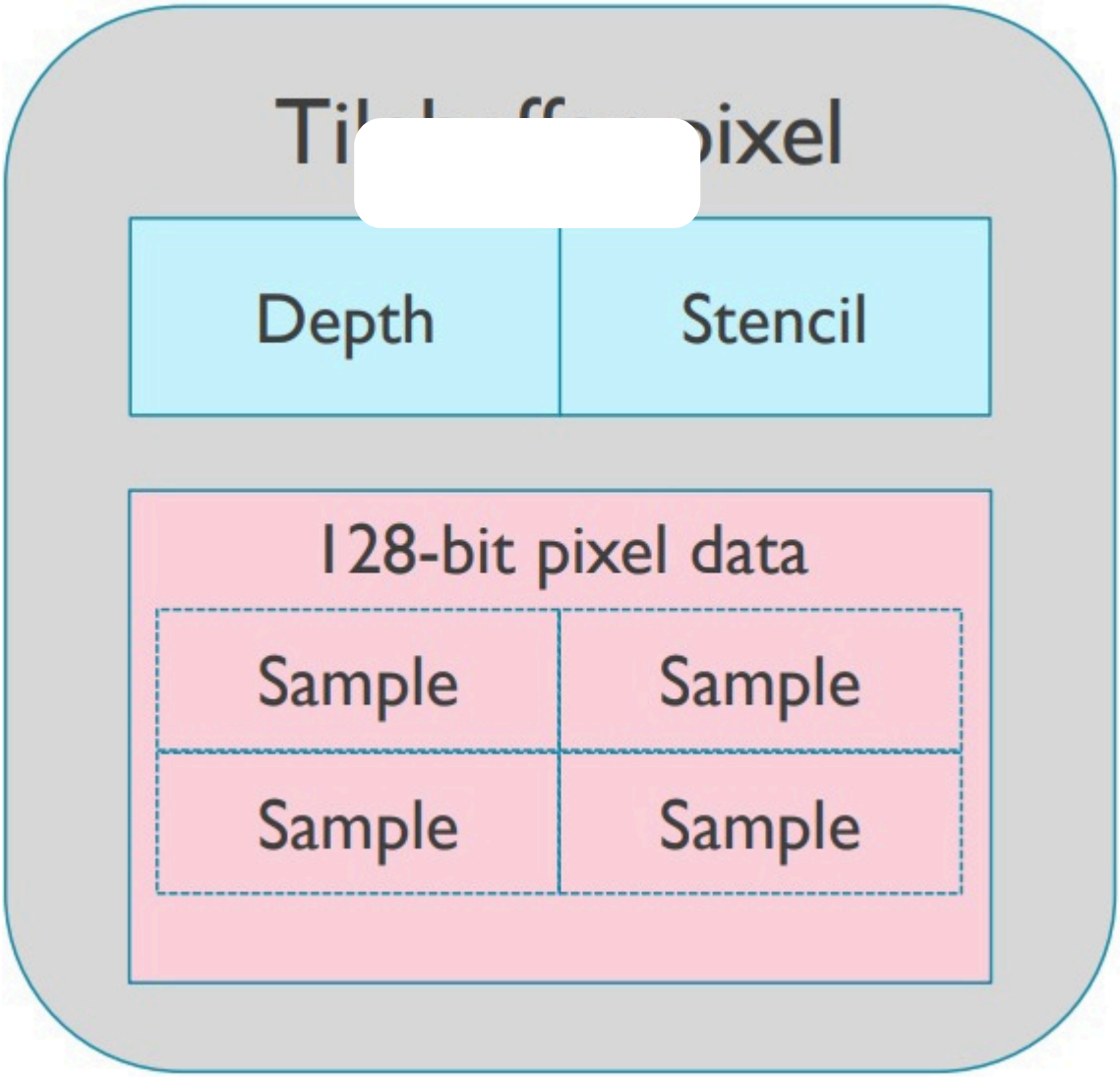
concept	Full name	Analysis
AMBA	Advanced Microcontroller Bus Architecture	Advanced Microcontroller Bus Architecture
AXI	AMBA Advanced eXtensible Interface	AMBA Advanced Extensible Interface
APB	AMBA Advanced Peripheral Bus	AMBA Advanced Peripheral Bus

concept	Full name	Analysis
ACE	AMBA AXI Coherency Extensions	AMBA AXI Coherence Extensions
GPU	Graphics Processing Unit	Graphics Processing Unit
VPU	Video Processing Unit	Video Processing Unit
DPU	Display Processing Unit	Display Processing Unit
ISA	Instruction Set Architecture	Instruction Set Architecture
SIMD	Single Instruction Multiple Data	SIMD
ISP	Image Synthesis Processor	Composite Image Processor
TSP	Texture and Shading Processor	Texture and Shader Processors

12.5.2 Mobile GPU Operation Mechanism

Since the operating mechanism of each GPU manufacturer, each series, and each generation of products may be different, this section takes Mali GPU as an example to explain the GPU operating mechanism of the mobile terminal. First, the parameters of the Arm Mali T880 GPU hardware architecture are explained as follows:

- 16 Shader Cores (SC).
- The tile size is 16x16 (4x4~32x32 internally).



    - Can store depth template buffer, 128-bit pixel data.
    - Each pixel has 16 bytes, raw bit access.

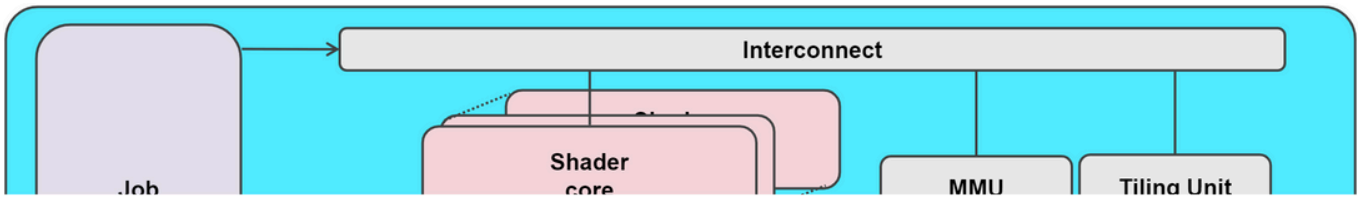
- Supports GLES3.2, Vulkan 1.0, CL 1.2, DX 11.2.
- 4x, 8x, 16x MSAA.

Mali GPU High-Level Architecture

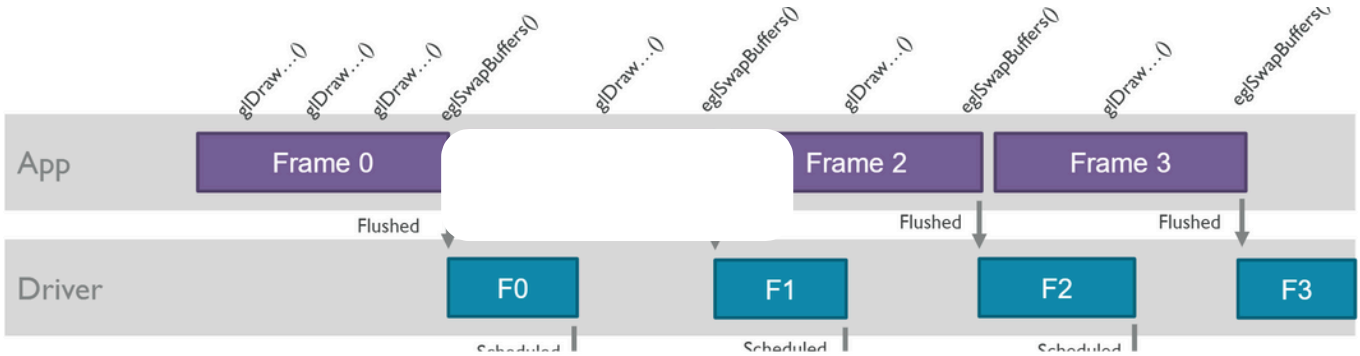
A breakdown of the Mali-T880

Schematic diagram of the Arm Mali T880 GPU hardware architecture and its functional description.

For Mali GPUs, the driver submits drawing tasks through the Job Manager, which creates and submits tasks to the GPU's drawing hardware, and they interact through internal connection elements.



A simplified diagram of the interactions between the application, driver, GPU, DPU and other layers is as follows:



Schematic diagram of the interaction between application, driver, GPU, etc., where eglSwapBuffers indicates the end of the frame, the App will attribute the drawing command to the driver, the driver will schedule the task to the GPU, and the GPU will submit the result to the DPU after the drawing is completed. Note that there is a delay between each layer.

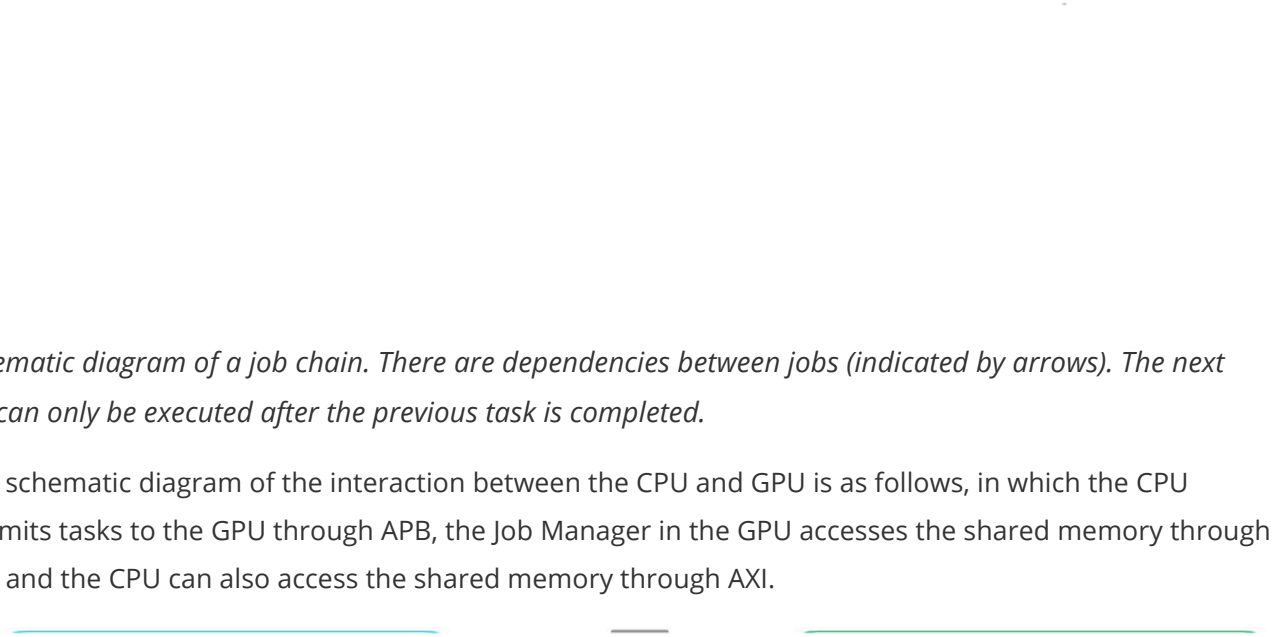
First, let's examine the interaction between the application layer and the driver layer. When an application calls a graphics API (such as OpenGL ES), the driver creates a corresponding resource architecture diagram:

There are several types of jobs in the GPU:

Job Name	abbreviation	describe
Vertex Job	**V**	Executes the vertex shader for a set of vertices.
Tiler Job	**T**	The Tiling Unit (fixed function) splits the transformed primitives into covered tiles.

Job Name	abbreviation	describe
Fragment Job	**F**	Runs work on a single render target across all tiles.
Job Chain	-	Job chain.

The following is one example of a GPU job chain:



Schematic diagram of a job chain. There are dependencies between jobs (indicated by arrows). The next job can only be executed after the previous task is completed.

The schematic diagram of the interaction between the CPU and GPU is as follows, in which the CPU submits tasks to the GPU through APB, the Job Manager in the GPU accesses the shared memory through AXI, and the CPU can also access the shared memory through AXI.

The schematic diagram of the Job Manager creating and assigning tasks in the GPU is as follows:

Job Manager operation diagram. In the diagram, 3 vertex jobs, 1 block job, and 2 shading jobs are assigned. The block job depends on the vertex job.

For Shader Core, Mali's structure is Tripipe, which is a unified shader architecture that can execute VS or PS:

To go a step further, the diagram of the vertex job operation is as follows: The vertex thread does not write to the tile buffer, but directly accesses the main memory. The vertex task contains 4n vertices.

The schematic diagram of the fragment operation is as follows:

Fragment Work is divided into three stages: Front-End, Tripipe, and Back-End. Pixels that have successfully passed rasterization, Early-Z, and FPK will have threads created by Fragment Thread Creator (in units of Quad, ie 2x2 threads), enter Tripipe shading, then enter Late-Z, blending, and finally write to Tile memory.

However, not all mobile GPUs operate in exactly the same way as Mali. For example, PowerVR has many differences:

Schematic diagram of the PowerVR Series 7XT architecture.

Diagram of the PowerVR Series 7XT unified shader cluster architecture.

For more information about PowerVR, see:

- PowerVR Series5 Architecture Guide for Developers PowerVR
- Graphics - Latest Developments and Future Plans

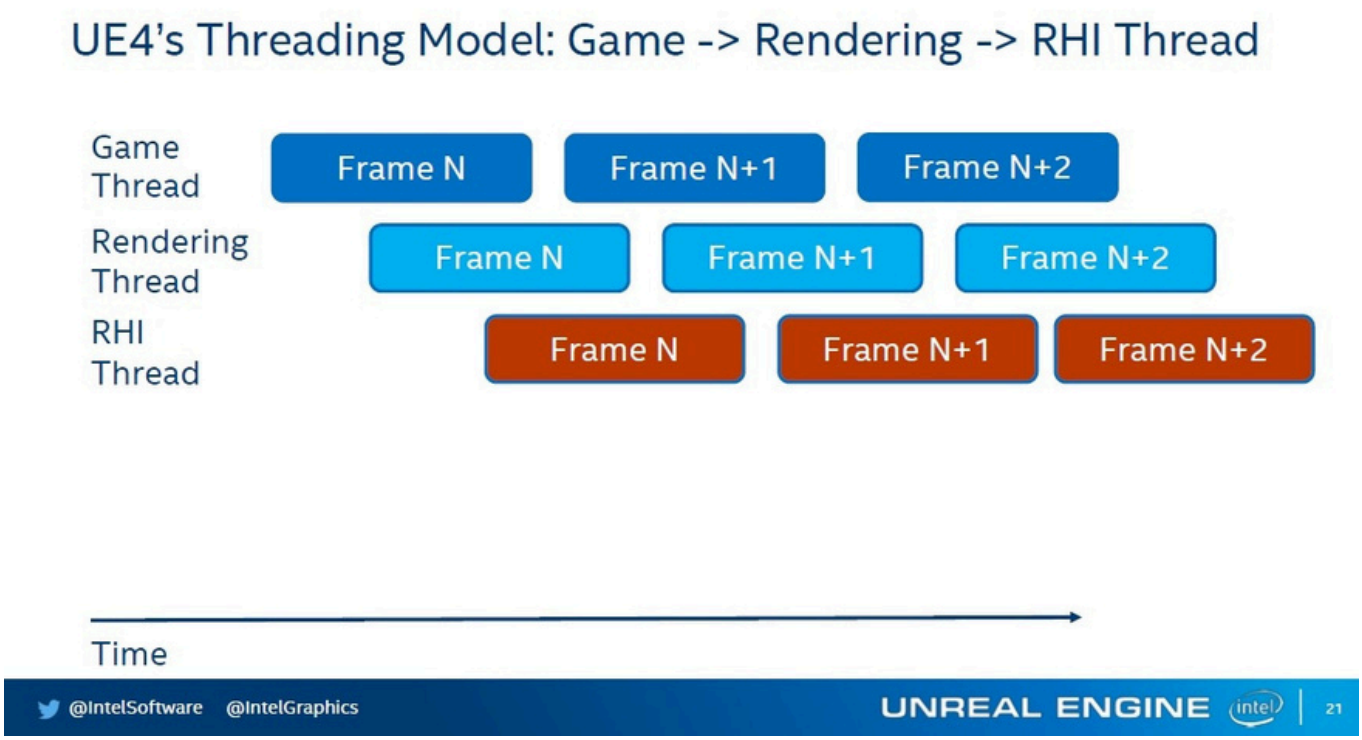
12.5.3 Parallelism, Jamming, and Latency

As Moore's Law slows down, modern mobile SoCs are moving towards multi-core and highparallelism. Whether an application can utilize multi-core performance to improve parallel efficiency largely determines its quality and user experience.

Contrary to parallel efficiency, lag and delay are the natural enemies of real-time applications (such as games). Stall means low frame rate and the application does not run smoothly; delay means that the operation cannot respond in time, which reduces the user experience of the product and even causes serious user loss.

Whether on the PC or mobile side, the scenes that the rendering pipeline needs to handle are becoming more and more complex. Coupled with features such as multithreading, there are more or less waiting and stalling phenomena, which lead to latency. This phenomenon is particularly obvious in the mobile rendering pipeline where TB(D)R is prevalent.

There are both objective and subjective reasons for lag and delay. Objective reasons refer to the cooperative waiting and synchronization of multiple threads, driver optimization, and benign optimization of the internal execution mechanism of the GPU. Subjective reasons refer to those that do not use interfaces , tags, states, or resources that conform to specific rendering mechanisms, which can be avoided and optimized.



UE has game thread, rendering thread and RHI thread. The latter thread is usually delayed a little more than the former thread. There is also synchronization and waiting between them to prevent the former thread from leading too much time.

Schematic diagram of the delay between applications, drivers, GPU, and display. The lower layer will lag behind the upper layer for a period of time.

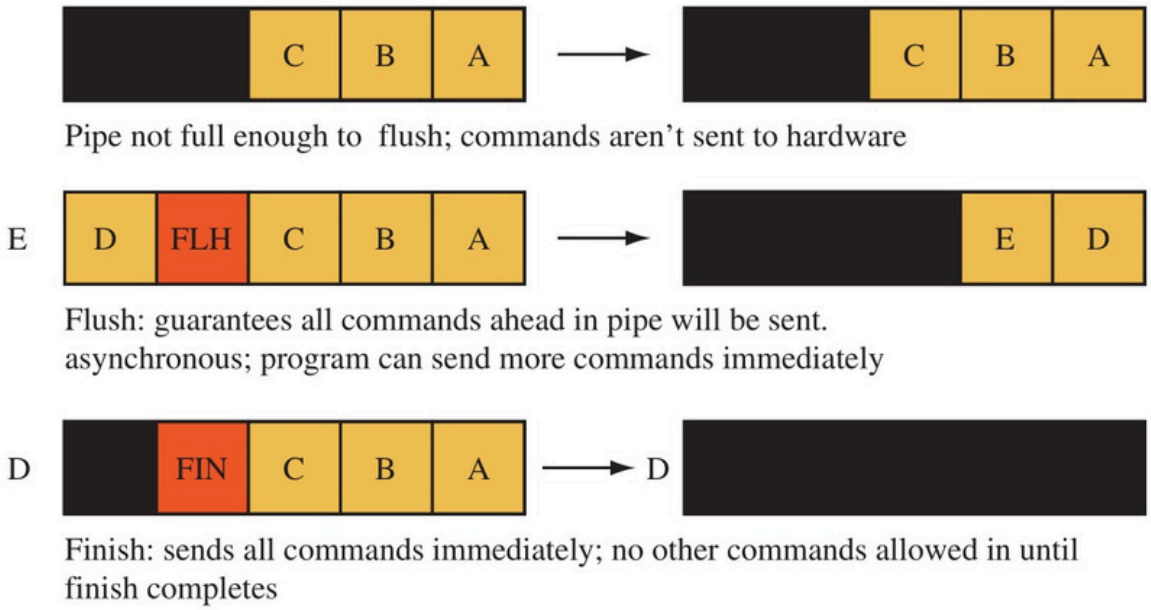


Figure 21.4 Finish versus flush.

Schematic diagram of the execution of glFinish and glFlush in OpenGL. After glFlush is called, the rendering instructions may not be refreshed to the GPU immediately, but only when the driver's rendering command buffer is full, which may also cause delays.

The common practice of TB(D)R on mobile GPUs is to process Binning Pass and Rendering Pass in different frames to improve parallel efficiency, but this will also cause delays:

Schematic diagram of Binning and Rendering error frame processing in the TBR architecture.

The above is a perfect error frame processing situation. If any of the following situations occurs, the execution rhythm of TBR will be disrupted, resulting in more serious stalls and delays:

- Binning relies on the data or resources of the previous frame.

> *The binning of frame n+1 depends on the rendering result of frame n, so it cannot be processed in parallel with the rendering pass of frame n and can only be delayed to the next frame.*

This situation can be solved by delaying the use of time, such as binning N frames using the rendering results of N-1 frames. The following figure is an optimization example of a real-time environment stereogram:

- After submitting and before rendering, you need to modify the data. For example:
    - The pixel shader calculates and writes data to a framebuffer object, using the result to generate the displacement.
    - From the CPU you write to the texture, render with it, then update the texture again, then render the next frame; the pixel shader doesn't execute until the texture update is complete.

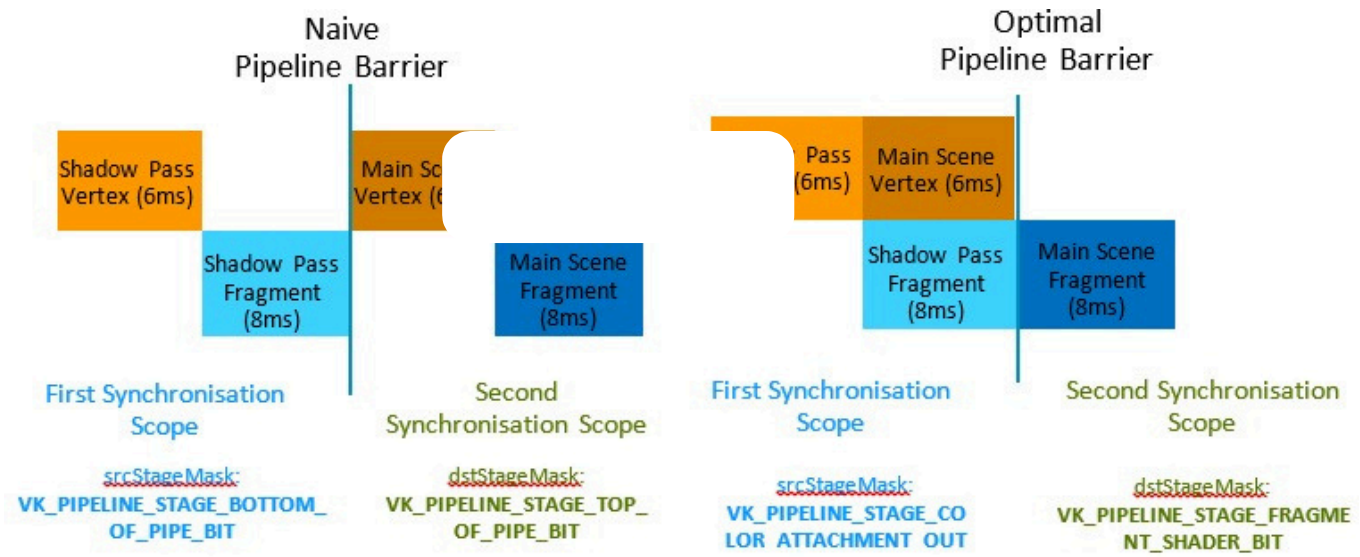
Modern graphics APIs such as Vulkan have a Subpass mechanism. Subpass can be processed in parallel (Overlap) and can also specify data dependencies:

Top: Subpass overlap mechanism i; Bottom: Data dependencies within and between subpasses.

Modern graphics APIs such as Vulkan, Metal, and DX12 can accurately specify the waiting stage of the rendering pipeline barrier. For example, the following figure uses the default PipelineBarrier, which will cause Vertex and Fragment processing to have more idle or waiting time, wasting GPU time cycles:

By modifying the source and target stages that the barrier needs to wait for, this type of stall can be alleviated and the utilization of the shader unit can be improved:

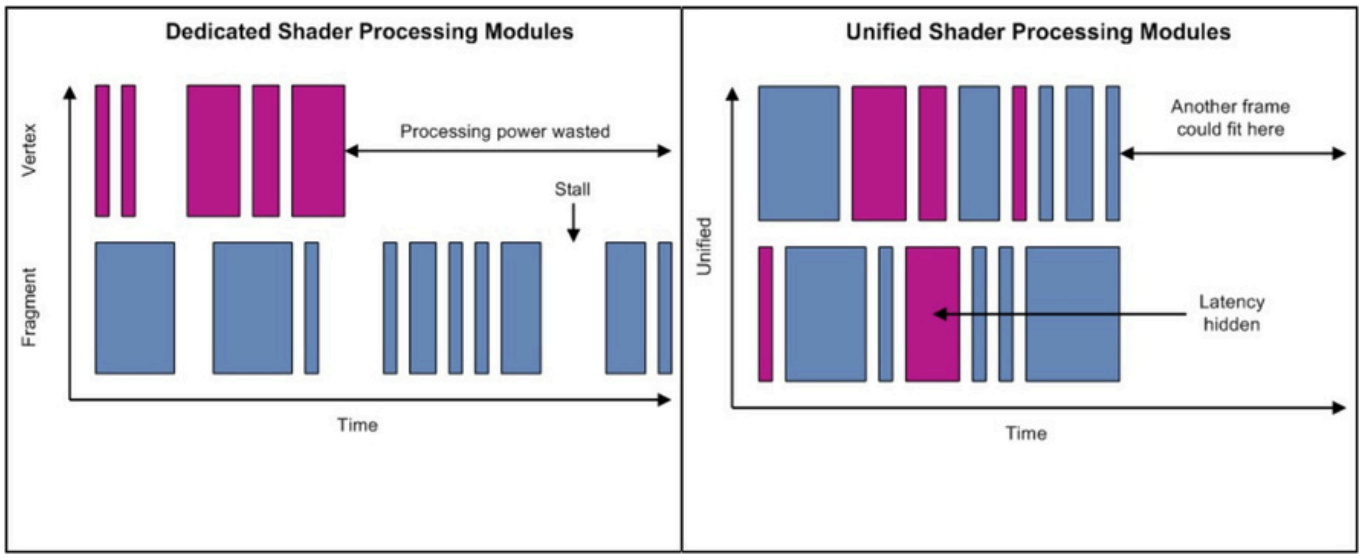
The specific optimization examples of Pipeline Barrier are as follows:



By using Vulkan's Pipeline Barrier to optimize the waiting phase between each Pass, Stall and latency can be reduced. In the figure, it drops from 28ms to 22ms.

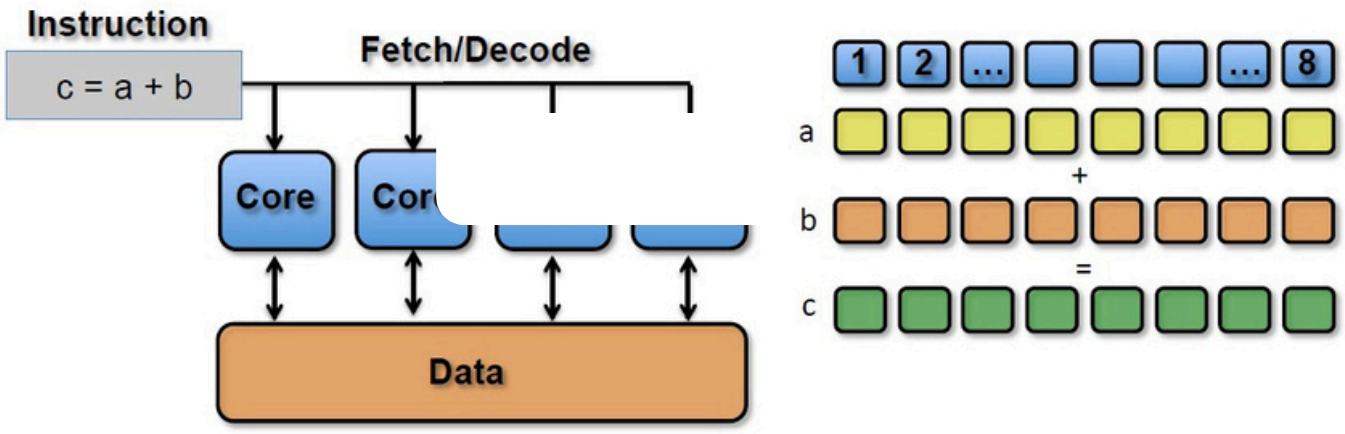
PowerVR's TBDR architecture will not start the rendering phase until all graphics elements in the current frame have processed the Binning data. This may cause more serious delays. If Alpha Test is turned on, the depth will be written back to the HSR stage, thereby interrupting the normal process of HSR and causing unexpected stalls.

Modern GPUs (including mobile ones) generally have a unified shader architecture. The characteristic of this architecture is that all shader cores can execute both VS and PS. In the future, GS, MS, etc. may be unified, so that the GPU can coordinate and balance VS or PS intensive tasks, so that all cores are in a fully loaded state as much as possible, thereby reducing waiting, idleness and delay:

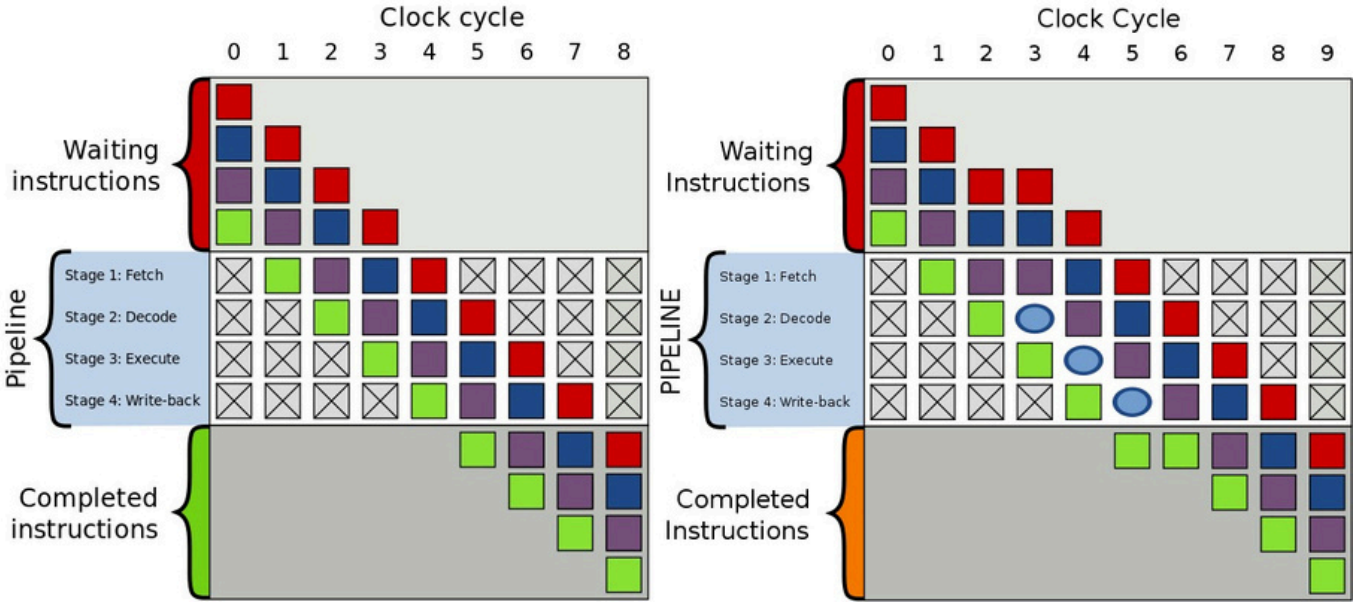


Left: Separate Shader Processing Unit, Right: Unified Shader Processing Unit. It can be seen that the processor of the latter is basically running at full capacity, thus reducing waiting and idling, and improving overall computing power.

Modern GPUs have fully utilized SIMD and SIMT technologies to improve overall parallel efficiency and throughput:



However, if there are data dependencies between GPU instruction groups, the execution time between instruction groups will be extended:



Left: The GPU instruction group is executed normally without waiting. Right: The GPU instruction group is added with a bubble, causing delay.

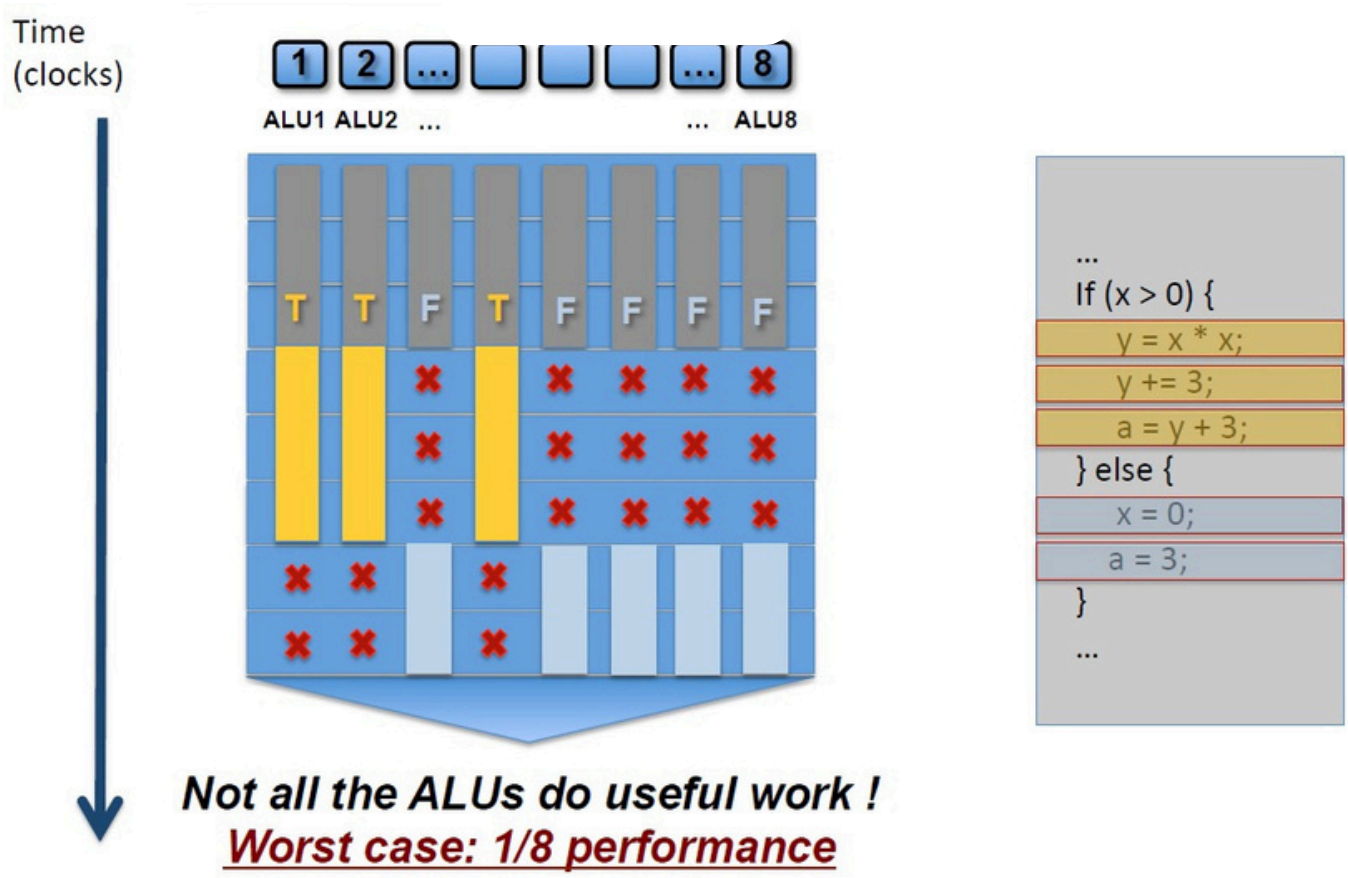
The bubbles in the above figure are generated to resolve the data dependencies between GPU instruction groups:

Bypassing backwards in time	Problem resolved using a bubble

Left: The lower group of instructions depends on the writing of the upper group of data. If it is not processed, old data will be obtained. Right: Bubbles are inserted into the lower group of instructions,

delaying one clock cycle to ensure that the latest data is obtained.

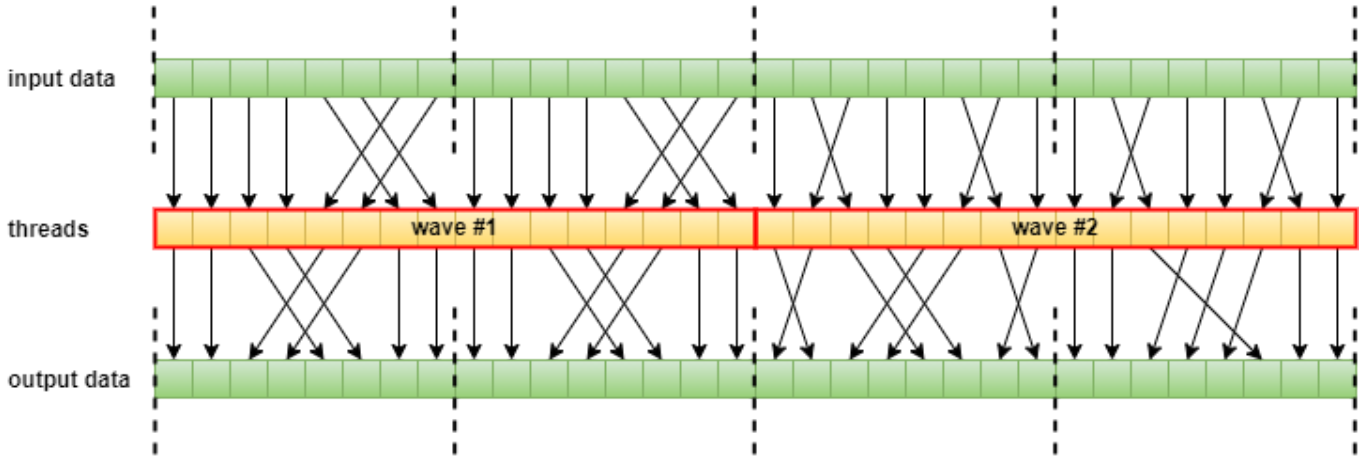
ifDynamic branch loop statements such as and in Shaderforwill reduce the utilization of GPU computing units and prolong the time they take to run instructions:



Instructions for accessing memory can also cause GPU computing units to stall, prolonging computing time:

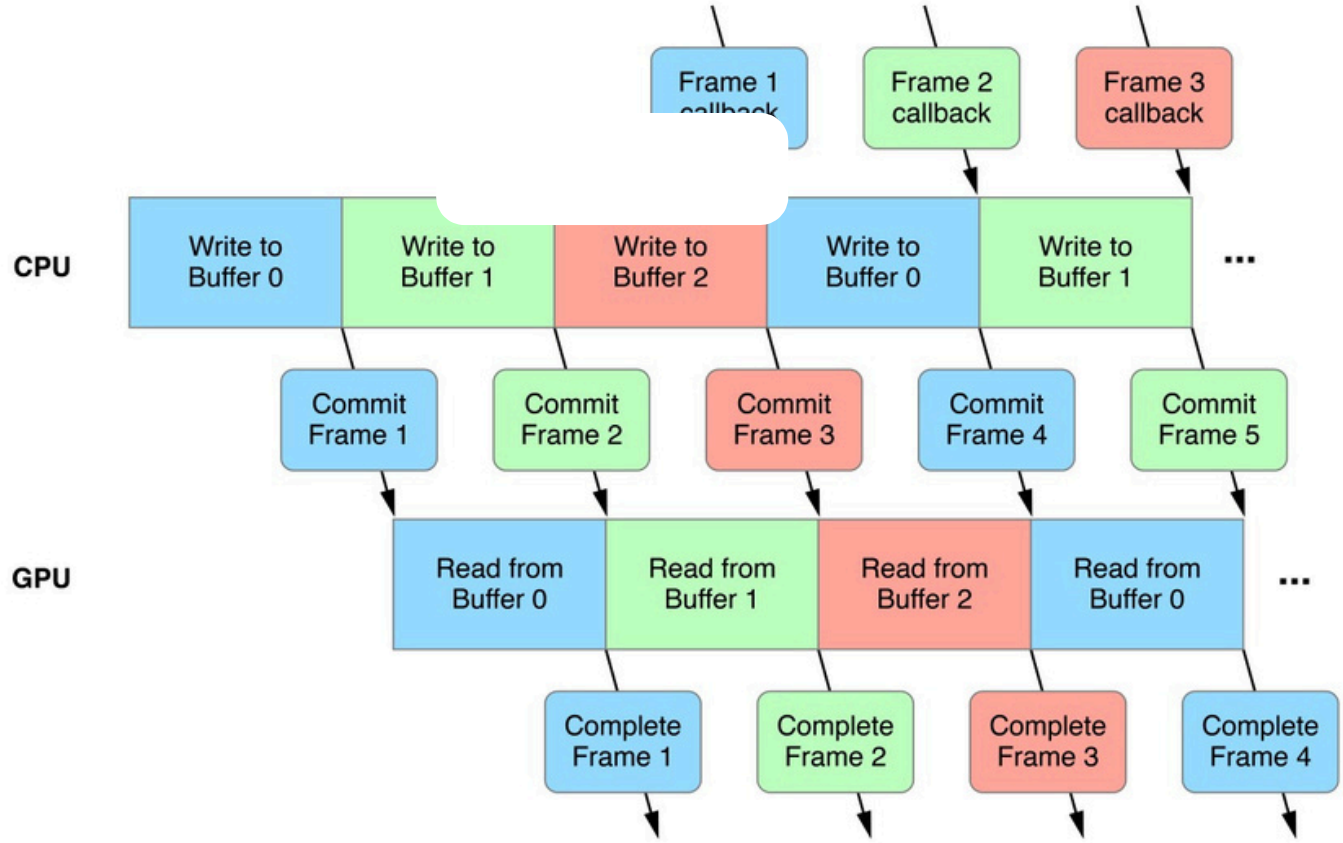
Unlike the low latency and low throughput of the CPU, the GPU is designed for high parallelism and high throughput, but at the same time has a small cache capacity, low cache hit rate, and high latency:

Therefore, if the GPU data structure is not well designed, the cache hit rate will be greatly reduced, thereby increasing the lag and latency of the computing unit. The GPU thread scheduler usually considers data relevance and keeps threads in the same thread group in the same cache line:



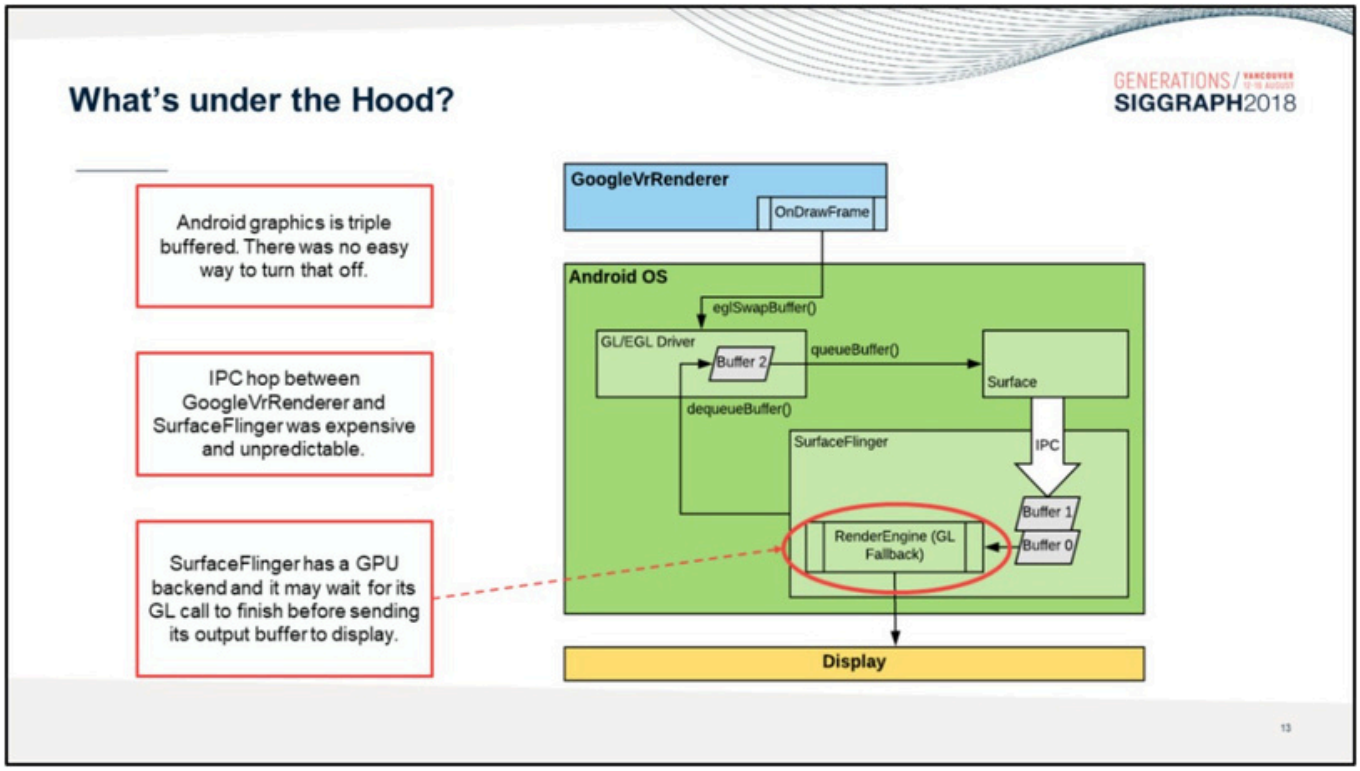
Schematic diagram of the Wave operation with a size of 16. The dotted line indicates that thread groups cannot access data across boundaries to improve the cache hit rate of data access within the thread group.

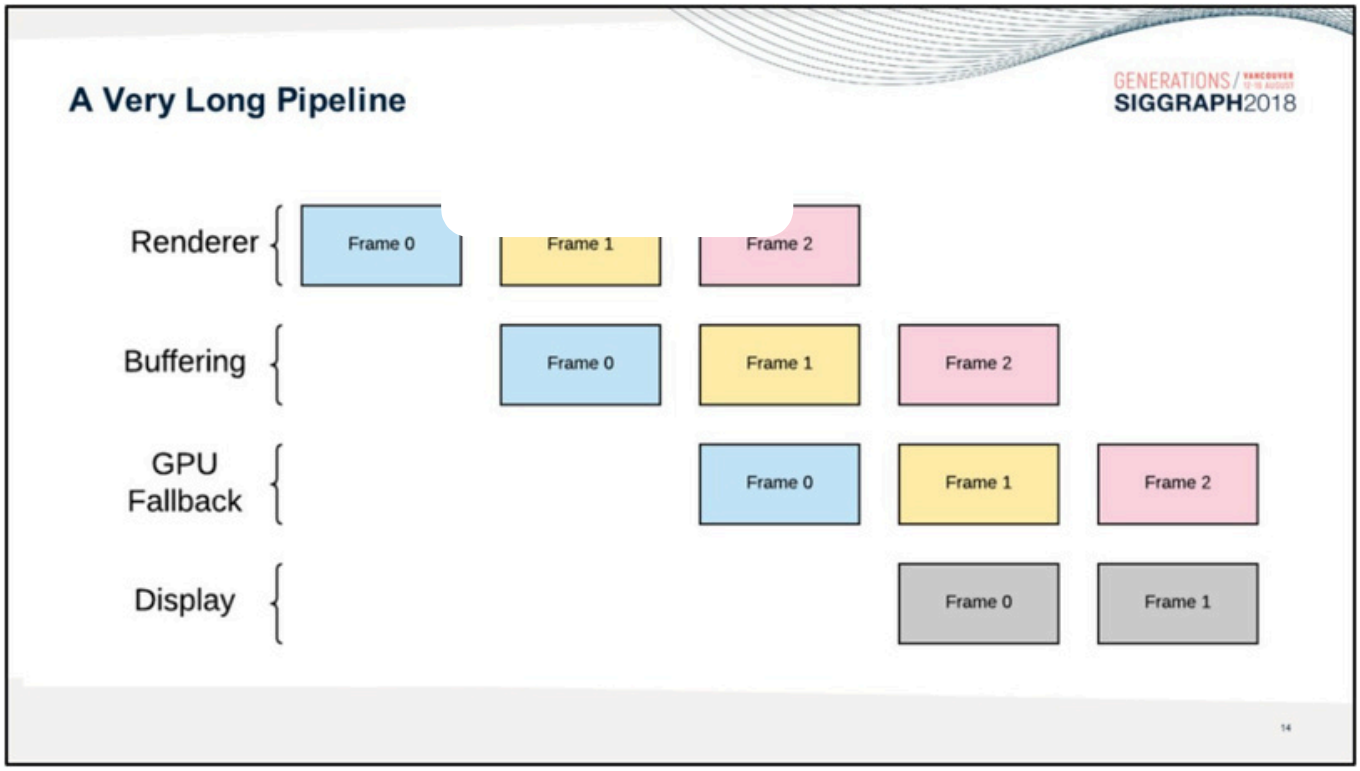
In addition to the above situations, if the system or application uses double buffering, triple buffering, vertical synchronization and other mechanisms, a certain delay will be introduced.



Schematic diagram of the three-buffer execution mechanism.

The Android system rendering module uses multi-level encapsulation and triple buffering mechanism, which always delays the picture by 3 frames:





On the contrary, by making good use of the parallel mechanisms of Async Compute, Copy Engine, Graphic Pipeline, etc., using RDG's automatic processing of resource allocation and dependencies, using the characteristics of subresources and aliased resources, and merging barriers and other operations , you can reduce waiting and delays between and within Passes and improve parallel efficiency.

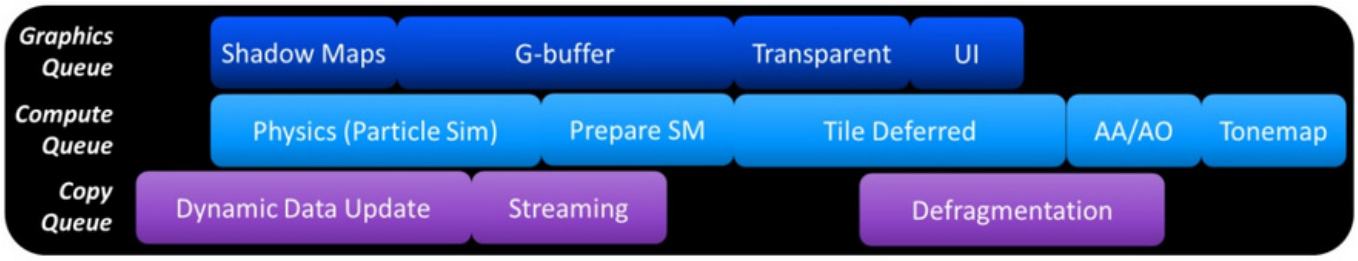
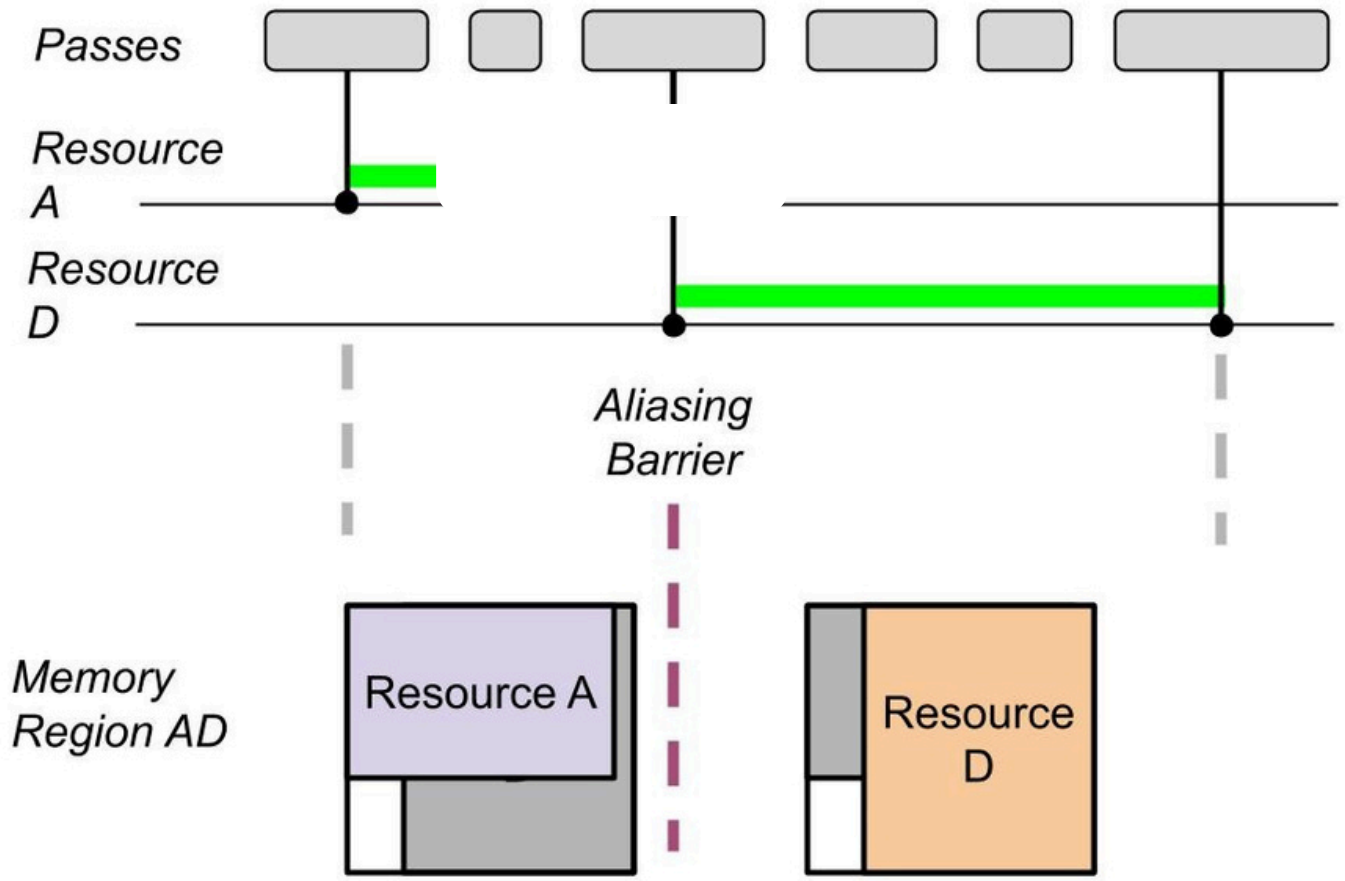


Figure 3: Example of graphics rendering tasks assigned to the different queue types available in DirectX 12

Examples of parallel operation of Async Compute, Copy Engine, and Graphic Pipeline.



Schematic diagram of the operation mechanism of alias resources, where resources A and D occupy the same memory area at different time periods. When using RDG, alias resources can save more than 50% of the used resource allocation space, even though they add additional resource manage complexity to the rendering system.

In short, from the logic update of the CPU App layer, the generation of rendering instructions, the calling and submission of the graphics API, across the driver layer, system layer, GPU internal, to the final display presentation, there may be various problems such as dependency, waiting, freezing and delay. This requires us to take a comprehensive view, identify the bottleneck of the entire rendering pipeline, and prescribe the right remedy to make our programs run efficiently, smoothly and instantly on the mobile SoC.

The UE4 official documentation provides some suggestions and optimization measures for latency, see Low Latency Frame Syncing for details .

- **To be continued**

References

- Unreal Engine Source
- Rendering and Graphics
- Materials
- Graphics Programming
- Mobile Rendering
- Qualcomm® Adreno™ GPU
- PowerVR Developer Documentation

- Arm Mali GPU Best Practices Developer Guide Arm
- Mali GPU Graphics and Gaming Development Moving
- Mobile Graphics
- GDC Vault
- Siggraph Conference Content
- GameDev Best Practices
- Accelerating Mobile XR
- Frequently Asked Questions
- Google Developer Contributes Universal Bandwidth Compression To Freedreno Driver
- Using pipeline barriers efficiently
- Optimized pixel-projected reflections for planar reflectors
- The difference between UE4's mobile and PC graphics and how to minimize the difference
- Deferred Shading in Unity URP
- General techniques for optimizing mobile game performance
- In-depth understanding of GPU hardware architecture and operation mechanism
- Adaptive Performance in Call of Duty Mobile
- Jet Set Vulkan : Reflecting on the move to Vulkan
- Vulkan Best Practices - Memory limits with Vulkan on Mali GPUs A
- Year in a Fortnite
- The Challenges of Porting Traha to Vulkan
- L2M - Binding and Format Optimization
- Adreno Best Practices
- Summary of knowledge on GPU architecture of mobile devices Mali
- GPU Architectures
- Cyclic Redundancy Check Arm
- Guide for Unreal Engine Arm
- Virtual Reality
- Best Practices for VR on Unreal Engine
- Optimizing Assets for Mobile VR
- Arm® Guide for Unreal Engine 4 Optimizing Mobile Gaming Graphics
- Adaptive Scalable Texture Compression
- Tile-Based Rendering
- Understanding Render Passes
- Lighting for Mobile Platforms
- Frame Pacing for Mobile Devices
- ARM Mali GPU. Midgard Architecture ARM's
- Mali Midgard Architecture Explored
- [Unite Seoul 2019] Mali GPU Architecture and Mobile Studio Killing
- Pixels - A New Optimization for Shading on ARM Mali GPUs
- Qualcomm's Quad-Core Snapdragon S4 (APQ8064/Adreno 320) Performance Preview
- Low Resolution Z Buffer support on Turnip
-

- Render Graph and Modern Graphics APIs
- Hidden Surface Removal Efficiency
- Unreal Engine 4: Mobile Graphics on ARM CPU and GPU Architecture
- Low Latency Frame Syncing
- Qualcomm® Snapdragon™ Mobile Platform OpenCL General Programming and Optimization
- Qualcomm Announces Snapdragon 865 and 765(G): 5G For All in 2020, All The Details Introduction to
- PowerVR for Developers
- PowerVR Series5 Architecture Guide for Developers
- PowerVR Graphics - Latest Developments and Future Plans
- PowerVR virtualization: a critical feature for automotive GPUs
- PowerVR Performance Recommendations
- PowerVR Low Level GLSL Optimization Mobile
- GPU approaches to power efficiency
- Processing Architecture for Power Efficiency and Performance
- opengl: glFlush() vs. glFinish() Cramming
- Software onto Mobile GPUs Vulkan on
- Mobile Done Right
- Triple Buffering
- Asynchronous Shaders
- Why Talking About Render Graphs
- NVIDIA Variable Rate Shading
- Introduction to compute shaders
- Introduction to GPU Architecture
- An Introduction to Modern GPU Architecture
- Understanding GPU caches
- Transitioning from OpenGL to Vulkan
- Next Generation OpenGL Becomes Vulkan: Additional Details Released
- Bringing Fortnite to Mobile with Vulkan and OpenGL ES
- Appropriate use of render pass attachments
- Preparing Android for XR
-

https://github.com/pe7yu