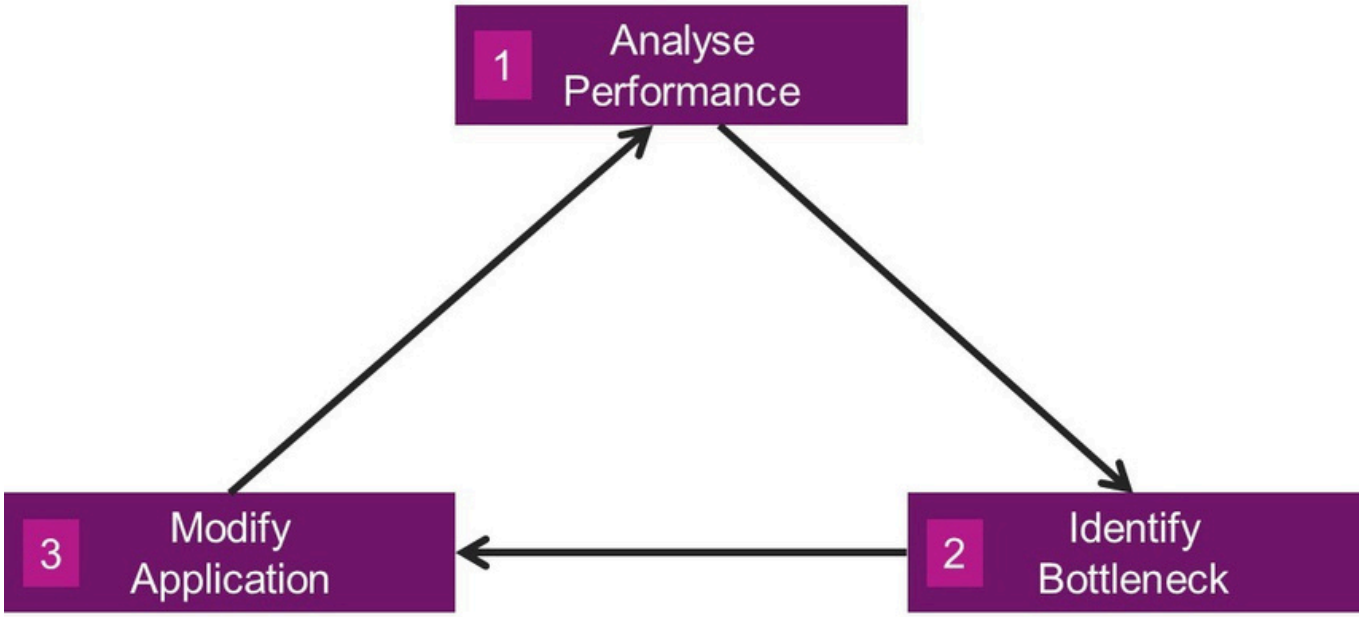
Analysis of Unreal Rendering System (12) - Mobile Special Topic Part 3 (Rendering Optimization)

Table of contents

12.6 Mobile rendering optimization

The previous chapters have thoroughly analyzed the characteristics and mechanisms of the mobile GPU architecture, which can guide us to abstract some principles to obtain high-performance rendering code and applications.

In order to achieve a smooth, efficient, and good experience, every application must pay attention to performance optimization and carry it out throughout. The performance optimization of an application is divided into the following triangular cycles:



The first step is to analyze the overall performance of the application.

The second step is to use tools to locate performance bottlenecks.

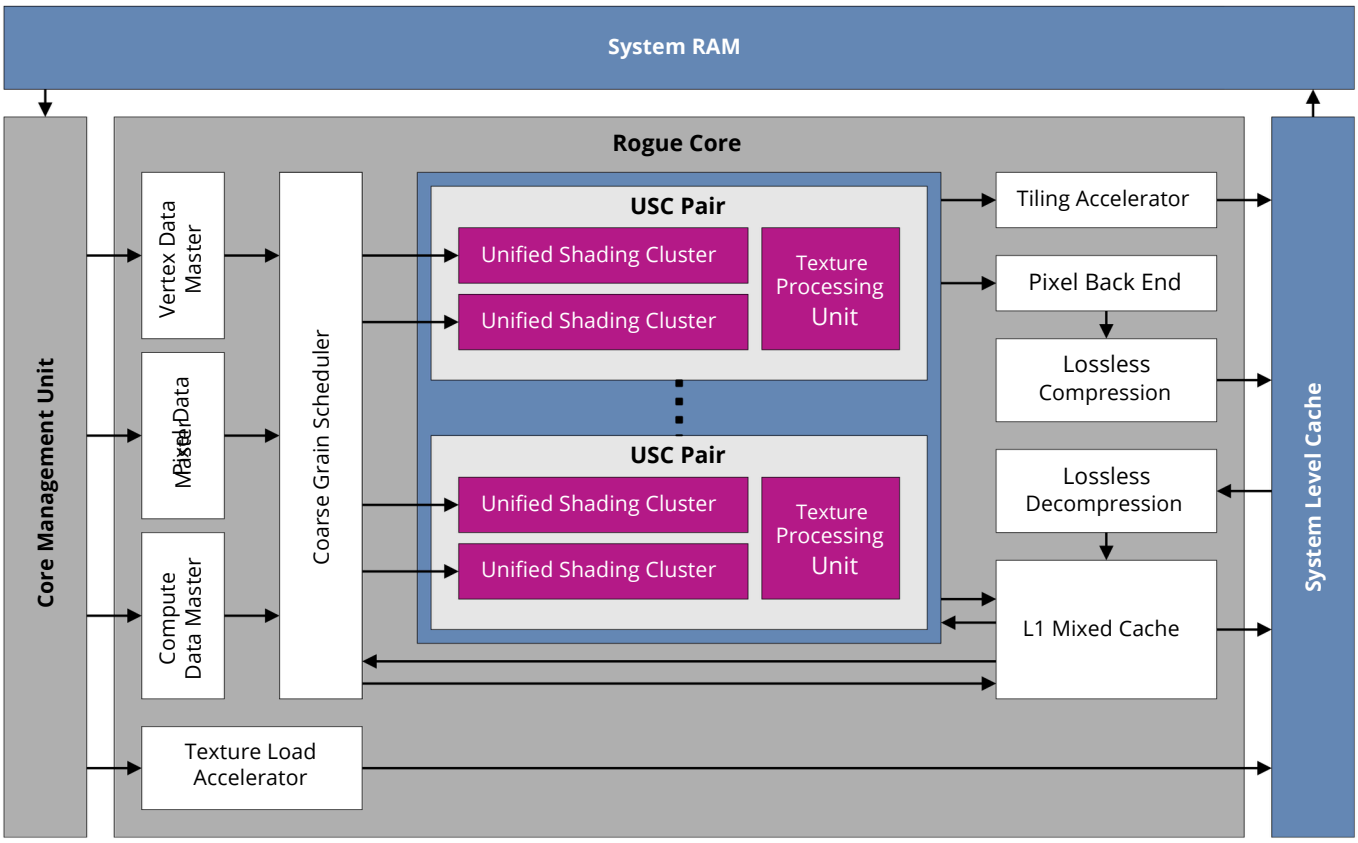
Step 3: Modify the application and go back to step 1 for recursive analysis.

When does this triangle loop stop? That is when the performance of the application has reached the standards specified at the beginning of the project (such as high, medium and low image quality not less than a certain number of frames, DC, number of triangles less than a certain number, etc.), and it is known that the application has reached the efficiency limit, and going down will reach a dead end with a very low input-output ratio.

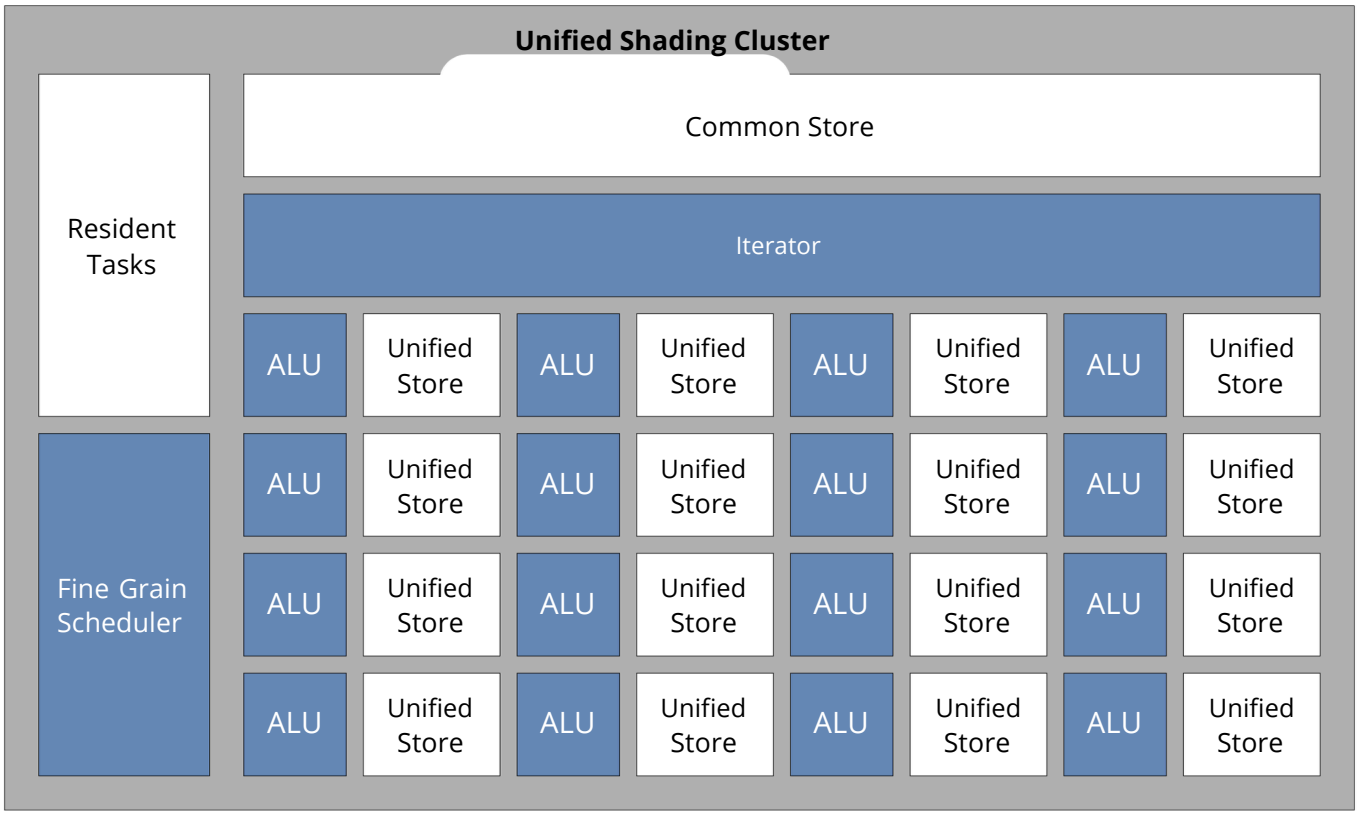
This article will cover the following concepts:

name	Aliases	describe
USC (Unified Shading Cluster	Shading Cluster, Shading Unit, Execution Unit	A semi-autonomous part of the graphics core that can often execute an entire workgroup. Other large components such as the Texture Unit can be shared between USCs.
Core	Processor, Graphics Core	An almost completely autonomous part of the graphics core. Typically, this is a collection of USCs and possibly supporting hardware such as texture units.
Task	Thread Group, Warp, Wavefront	The native grouping of threads executed by the USC, a PowerVR Rogue core consists of 32 threads.
shared	Shared variables	Variables stored in Shared memory.
const / uniform	const / uniform variables, uniform blocks, uniform buffers	Variables, blocks, and buffers stored in Constant memory.

Let's add the PowerVR Rogue hardware architecture and data flow interaction diagram as shown below:



The Unified Shading Cluster (USC) for PowerVR Rogue is shown below:



In addition, let me add the concept of fragments, which is widely used in this chapter:

> **A fragment**is the smallest representation unit formed after the rasterization of the geometry inside the GPU. It can only be written into the rendering texture and become a pixel after a series of fragment operations (alpha test, depth test, template test, etc.). Therefore, a fragment is not a pixel, but it has the probability of becoming a pixel.
> However, within D3D or UE, there is no concept of fragments, and pixels contain fragments.

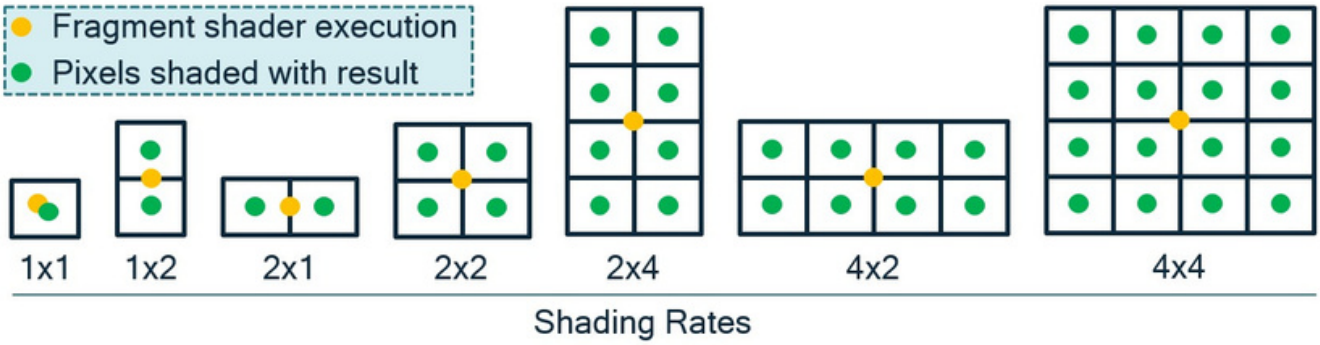
12.6.1 Rendering Pipeline Optimization

12.6.1.1 Using New Features

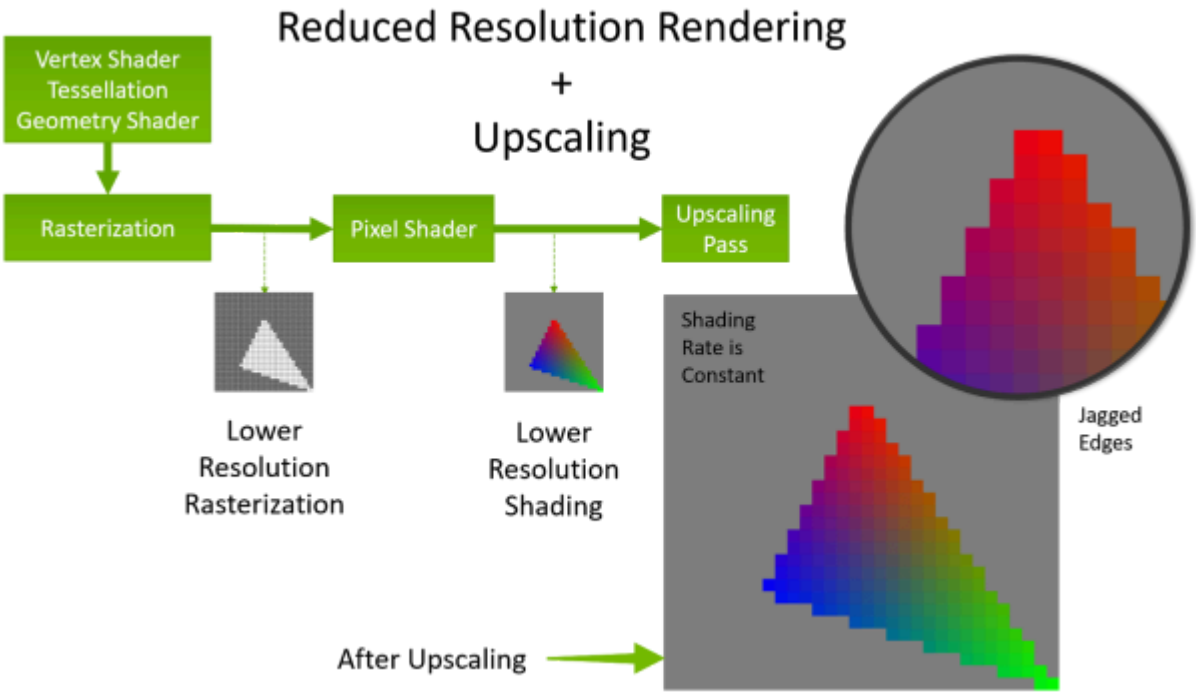
- **Variable Rate Shading**

Variable Rate Shading (VRS)allows a pixel shader to shade one or more pixels at a time, so that one shading calculation can represent a pixel or a group of pixels. VRS is the inverse of anti-aliasing. Anti-aliasing works by smoothing highly varying content, sampling each pixel more frequently to avoid aliasing and jagged edges. However, if the surface to be rendered does not have high color variation or will be blurred in subsequent passes (eg, motion blur), performing a shading calculation at each pixel is usually inefficient.
VRS allows developers to specify shading rates, where only one shader calculation is performed on a pixel, and the resulting operation is applied to the specified pixel group configuration. If used properly, it should not cause visual quality degradation, while significantly reducing the burden of GPU rendering, thereby saving power and improving performance.

VRS diagram. The screen uses different shading rates according to the color change frequency. High shading rates are used for high changes (such as cars), and low shading rates are used for low changes (such as the road surface in the lower left and lower right).



Common shading rates and operating mechanisms supported by VRS. The yellow dot is the shading coordinate, and the green dot is the shading result of directly reusing the yellow dot.



The working mechanism of VRS in the rendering pipeline. VRS performs rasterization at the rasterization stage using a specified shading rate and then amplifies it after entering PS.

UE can set a shading rate for each material in the material attribute template:

The core idea of VRS optimization is to reduce the number of calculations and reuse the results of surrounding calculation points, thereby improving rendering efficiency. Situations suitable for using VRS:

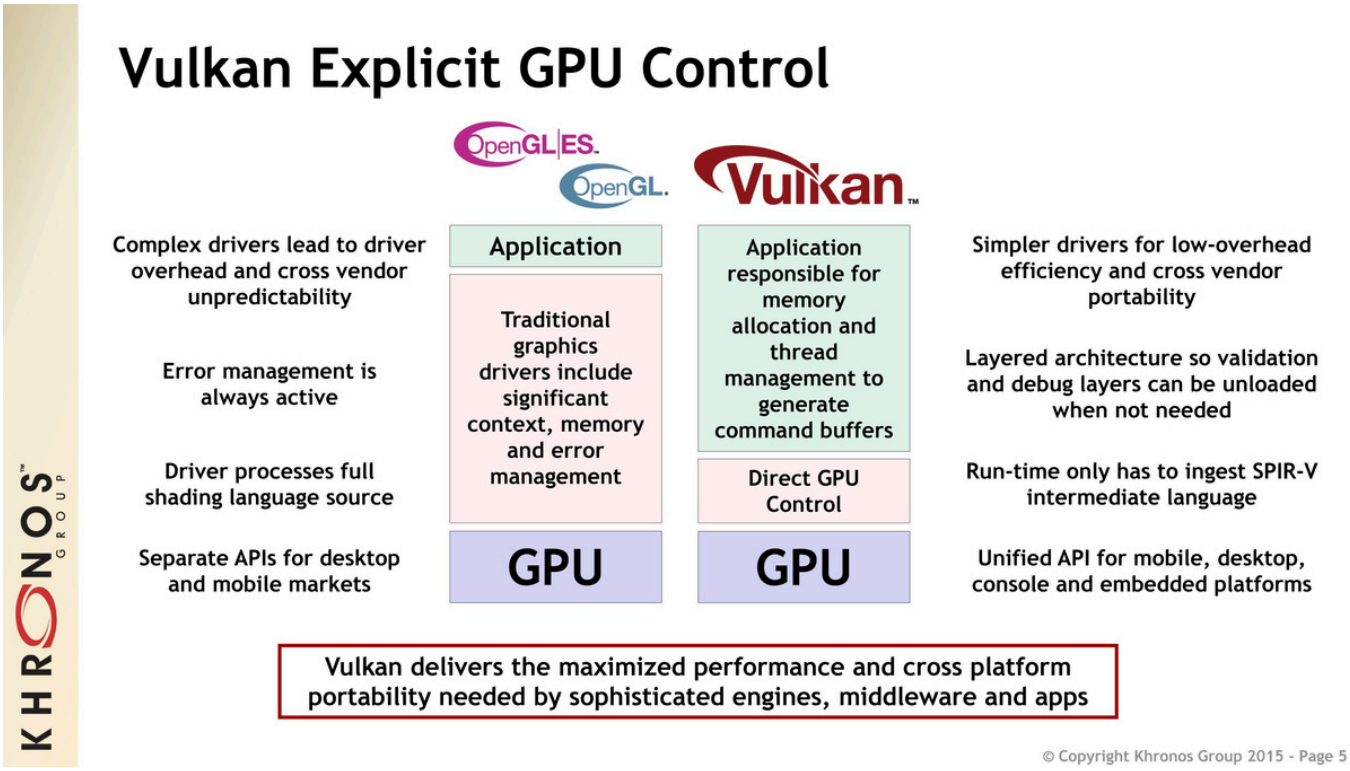
- Objects with a low rate of color change.
- Objects in the motion blurred area.
- Objects outside the depth of field.

Using VRS on mobile devices requires extensions of different graphics APIs:

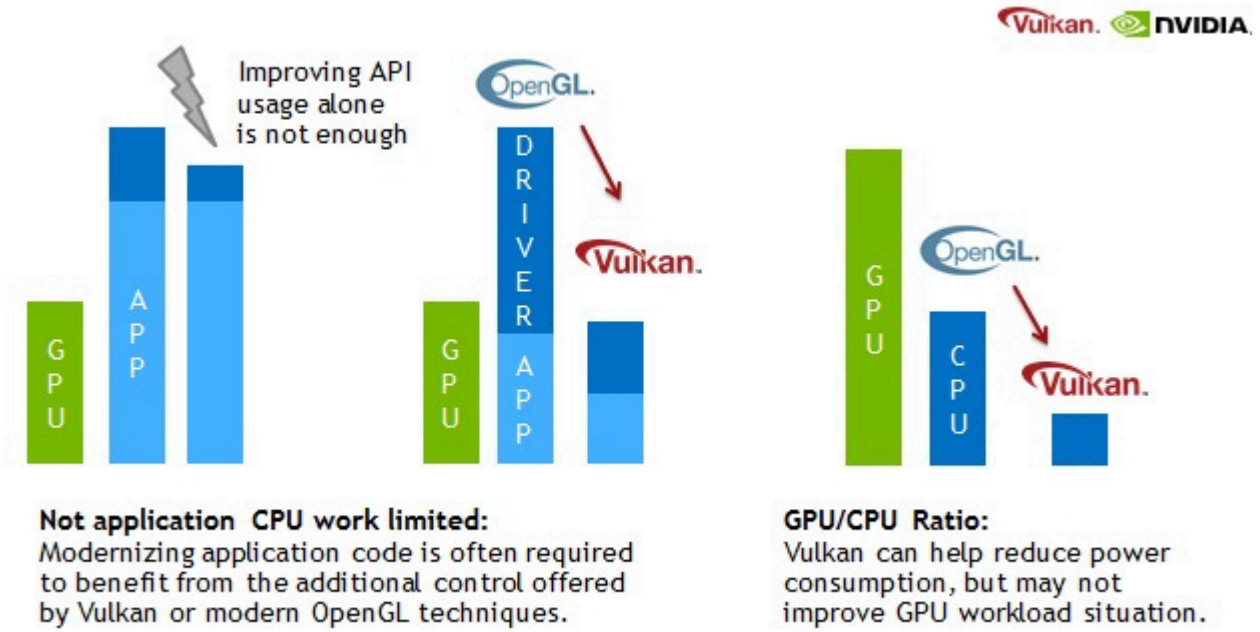
```
// ------ OpenGLES ------ //
Qualcomm
QCOM_shading_rate
GL_SHADING_RATE_1X1_PIXELS_QCOM
GL_SHADING_RATE_1X2_PIXELS_QCOM . . . . . .


// Arm / Imagination Tech (Not
supported)

//------ Vulkan------
VK_KHR_fragment_shading_rate
```

- **Use Vulkan instead of OpenGL.**

Compared with traditional APIs such as OpenGL, Vulkan supports multi-threading and a lightweight driver layer. It can accurately manage GPU memory, synchronization and other resources, avoid runtime verification, is based on a command queue mechanism, has no global state, etc. (see the figure below).



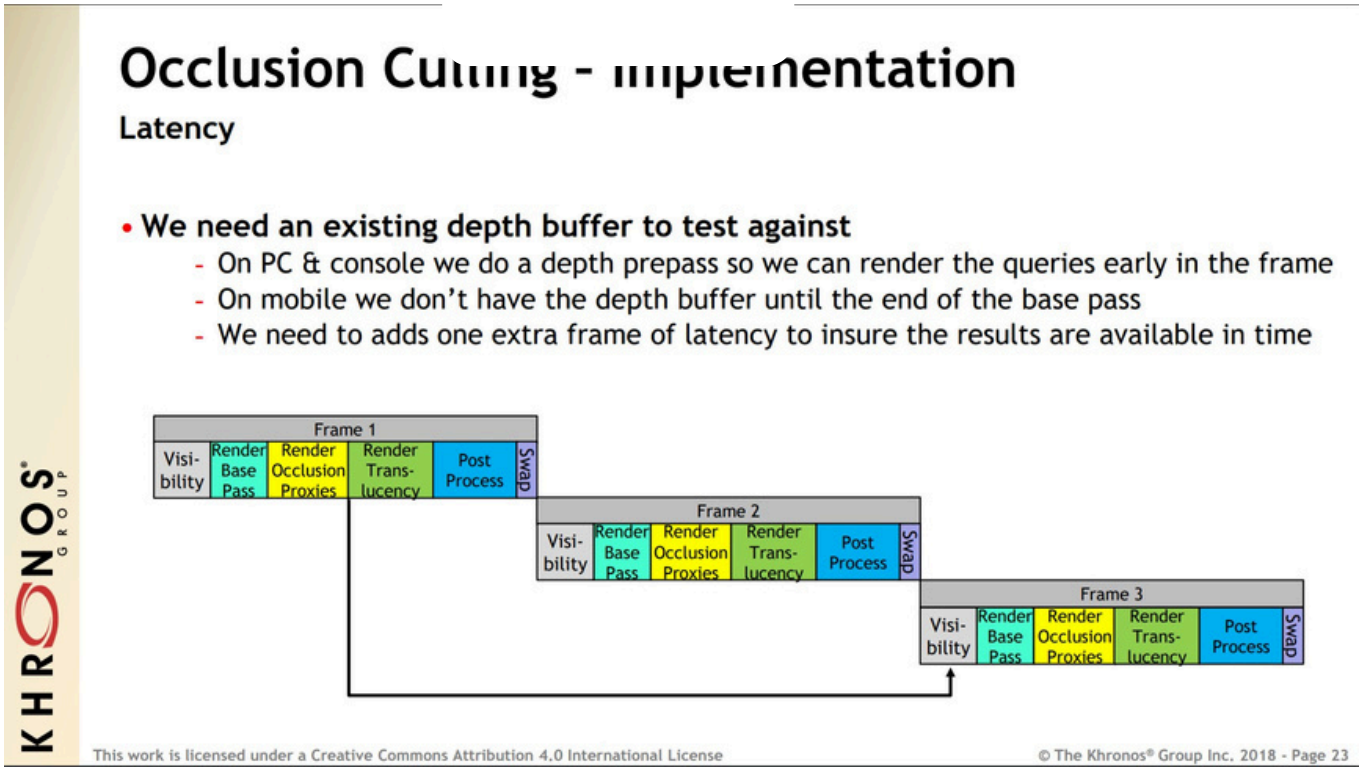
Thanks to Vulkan's advanced design concept, its rendering performance is higher, and it is usually better than OpenGL in terms of CPU, GPU, bandwidth, energy consumption, etc. However, if the CPU or GPU load of the application itself is high, the benefits of using Vulkan may not be so obvious:



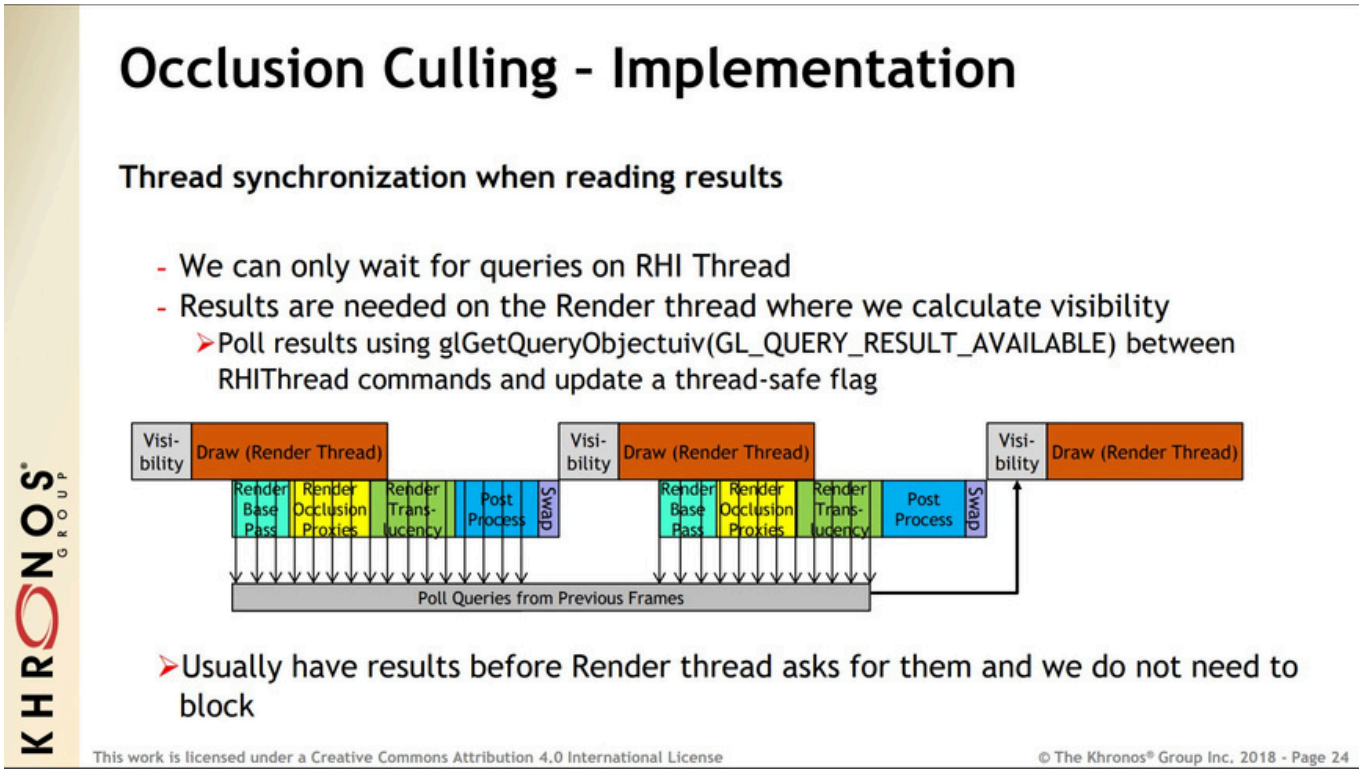
- **Use occlusion culling.**

Occlusion culling remove can obstructed objects or objects that occupy a small portion of the screen in the distance in advance to avoid entering the GPU pipeline and occupying bandwidth and

computing resources. UE's occlusion culling on the mobile side is delayed by two frames (because the depth buffer is available only after the BasePass ends, and an additional frame delay is required to ensure the result is available):



Then the RHI thread waits for the result of the occlusion query, which is used by the rendering thread. Since it is delayed by two frames, the rendering thread does not need to wait when calculating visibility:



When using occlusion culling, you need to follow these recommendations:

1. Return query results only when needed, don't wait for it, because synchronous waiting is very inefficient.

2. For occlusion, use the exact count option only when necessary. For OpenGL ES, use GL_ANY_SAMPLES_PASSED, and for Vulkan, use VK_QUERY_CONTROL_PRECISE_BIT = false unless you really need to know the number of occlusions. 3. Do not modify resources that are being referenced in a draw call. 4. Do not use GL_MAP_INVALIDATE_BUFFER /GL_MAP_INVALIDATE_RANGE with glMapBufferRange(),
because these flags will trigger the creation of an unnecessary resource copy in some versions of the driver.

12.6.1.2 Pipeline Optimization

- **Eliminate sub-pixels during tessellation.**

Tessellation increases the level of detail and can reduce bandwidth memory and CPU cycles by allowing other game subsystems to operate on a lower-resolution mesh representation. However, high levels of tessellation can produce sub-pixel triangles, which results in reduced rasterization utilization. It is important to use distance, screen space size, or other adaptive metrics to calculate tessellation factors that avoid sub-pixel triangles.

- **Enable backface culling during tessellation.**

Backface culling of primitives prevents redundant pixels from entering the pixel shader, thereby improving performance.

- **Delete unused render target or shader resources.**

Operating more RT or shader resources will increase bandwidth and reduce performance. Therefore, try to delete unreferenced resources.

- **Avoid GMEM loading.**

Before each Pass rendering, you need to call the graphics API to explicitly clear RT.

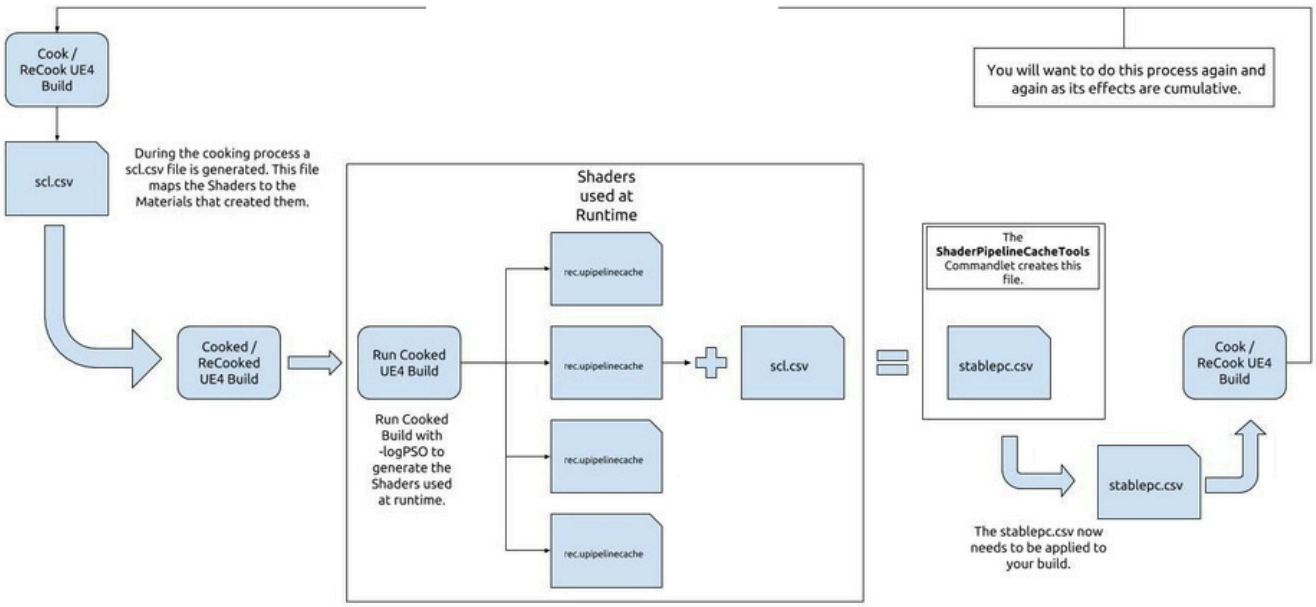
> OpenGL ES: glClear()
> Vulkan: LOAD_OP_CLEAR / LOAD_OP_DONT_CARE

- **Use subpass or PLS.**

Vulkan's subpass (or OpenGL ES's PLS) allows data from multiple passes to be continuously stored in GMEM (Tile buffer), avoiding repeated data transfers between GMEM and global memory, thereby reducing bandwidth and latency.

- **Use PSO cache.**Creating PSO objects at runtime consumes CPU performance. If you collect and compile the shaders used by the material and save them as binary files during the offline stage, so that you can directly read the cache file and convert it into a PSO object when calling

it at the next runtime, you can reduce the CPU load. The following figure is an illustration of UE's PSO cache mechanism:



For more details, please refer to the UE official document:PSO Caching .

- **Use depth-only (Z-only) rendering.**

The GPU has a special mode where it can write Z-only pixels at twice the rate as in normal mode, for example when an application is rendering a shadow map.
There are two ways to put the GPU into this mode:

1. The graphics API explicitly instructs the hardware to enter this special rendering mode.

2. The application prompts the driver with a specific rendering state, such as using an empty fragment shader and disabling the Frame Buffer write mask.
Some rendering programs or engines (such as UE) use a dedicated PrePass to render depth to make full use of Early-Z calculations. However, mobile GPUs need to be treated with caution and actual tests should be used as the standard.

- **Use an indirect indexed drawing interface.**

Indirect draw calls shift overhead from the CPU to the GPU, reducing CPU and GPU bandwidth. For example, draw call parameters are cached at load time so that meshes can be rendered in buffer object storage. This cached data can be used as input parameters toglDrawArraysIndirector glDrawElementsIndirect.

Requires OpenGL ES 3.1 for support.

- **Draw Call optimization.**
    - Merges geometric objects and also merges their materials.

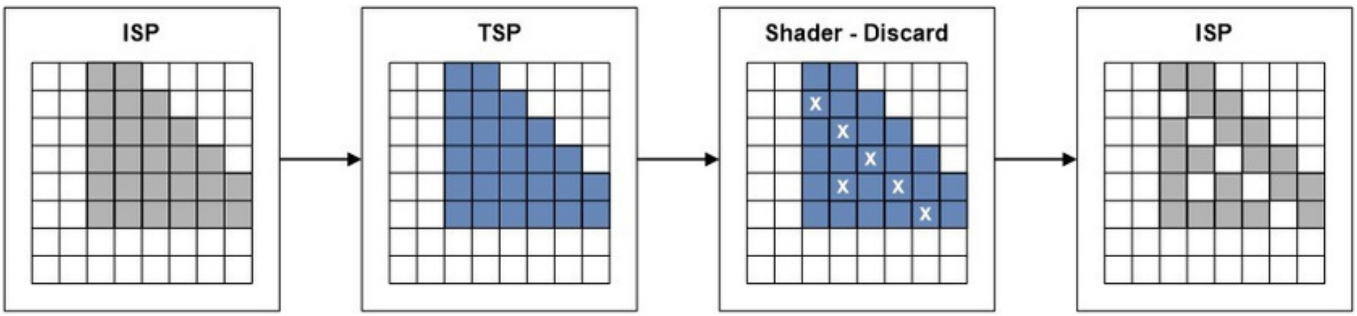
- Using batch processing can reduce energy consumption even when the process is not CPU-bound.
- Use instance.
- Draw using non-direct indexing.
- Avoid drawing small objects multiple times.
- Set a reasonable number of Draw Calls based on high, medium and low image quality.

When using batch processing, pay attention to the total number of vertices, which cannot exceed the expression range of the index (usually up to 65k). In addition, if the bounding box of the object after Merging or batching is too large, it will cause performance degradation because it is impossible to effectively use Frustum Cullinig, occlusion culling and other technologies for culling.
In addition, it is important to note that the submitted geometric objects are adjacent and should fall within the same tile as much as possible to reduce the number of tiles covered, reduce bandwidth, and improve cache hit rate:

Top: Good geometry submission order. Bottom: Incorrect geometry submission order.

- **Disable Alpha Test / Discard.**

Alpha Test will disrupt the normal process of TBR and cause the rendering pipeline to stall, which is particularly obvious on PowerVR (the Alpha Test stage will write the depth back to the HSR stage). Because TB(D)R generally turns on Early-Z technology and special hidden surface elimination technology (HSR, FPK) when rendering opaque objects, depth testing will be turned on at this stage, and the depth of the fragment that passed the depth test will be written. However, if Alpha Test is turned on or Discard is used in Shader, it is impossible to determine whether the depth of the fragment is valid in the Early-Z/hidden surface elimination technology stage, and it is necessary to wait until the PS, Alpha Test and other stages are completed:



This will not fully utilize the advantages of HSR technology, thereby reducing rendering performance.

You can use Alpha Blend instead of Alpha Test. If you really need Alpha Test, the rendering order of objects must follow this order: Opaque -> Alpha-tested -> Blended.

- **Minimize Alpha Blend.**

The reason is that deferred renderers, such as PowerVR GPUs, calculate the visibility of a fragment before the fragment shader processes it, preventing invisible fragments from being processed unnecessarily in the output image. If transparent objects are required, try to reduce the number of transparent objects.

Since Alpha Blend cannot write depth and cannot fully utilize HSR/FPK, it will cause Overdraw and increase bandwidth and data transmission volume.
If necessary, here are some optimization suggestions:

1. Use unorm format first instead of floating point numbers. (Note: This is a suggestion from Arm Mali, other GPUs may be different, based on actual measurements)
2. If it is an opaque object, Blend and alpha to coverage should be disabled.

3. Do not use blending on floating point frame buffers that carry MSAA data.

4. Avoid excessive OverDraw. Monitor the number of blending layers generated on a per-pixel basis. Even for simple shaders, a high number of blending layers will quickly consume clock cycles due to the large number of fragments.

5. Consider dividing large UI elements into opaque and transparent parts. Then you can draw the opaque and transparent parts separately, allowing Early-ZS or FPK/HSR to remove the OverDraw under the opaque part.

6. Don't just set alpha to 1.0 in the fragment shader to disable blending.

- **Make full use of Early-Z and FPK/HSR to cull occluded pixels.**

To take full advantage of Early-Z, objects should be drawn in the following order:

1. Draw opaque objects from front to back.

2. Draw the masked object from front to back.

3. Draw semi-transparent objects from back to front.

For mobile GPUs that widely support the TBR architecture, it is not recommended to enable the Prepass drawing-specific depth, otherwise it will increase bandwidth and Draw Calls.
In addition, when drawing opaque objects, try to do the following:

1. Disable the discard statement.

2. Disable Alpha to Coverage.

3. It is forbidden to modify the depth in the fragment shader.

If any of the above is violated, Early-Z will become invalid and Late-Z will be forced to be used, thus reducing rendering efficiency.

- **Turn on cropping and testing fully.**

Clipping techniques include occlusion culling, frustum clipping, Scissor, distance clipping, LOD, etc.

Tests include back-face test, depth test, stencil test, etc., but transparency test is disabled.

- **Disable Z-Prepass.**

Mobile GPUs usually have built-in pixel-level culling based on the TBR structure, so there is no need to draw depth again. UE disables Z-Prepass by default on mobile devices.

- **Minimize stencil buffer updates.**

1. If the values are the same, use KEEP instead of REPLACE.

2. Some renderers (such as UE) use lighting drawing Pass pairs: the first Pass is used to create the template buffer, and the second Pass is used to color the unmasked fragments. The template value can be reset in the second Pass to prepare for the next lighting pairing, which can avoid a separate template cleanup operation.

> UE's mobile scene renderer uses this template cleanup optimization method when drawing lighting.

- **Call the graphics API correctly.**
    - Do not use the API in a way that causes the GPU to idle unless you are meeting your performance targets.
    - Do not wait for query results on fence and query objects too early in the rendering pipeline.
    - *Use the GL_MAP_UNSYNCHRONIZED*flag to enable asynchrony when calling glMapBufferRange() to prevent the rendering pipeline from stalling.
    - Avoid calling the following interfaces synchronously:
        - However, some GPUs (such as PowerVR) do not block the calling thread because they use a double buffering mechanism.
        - glFinish()
        - glReadPixels()
        - glWaitSync()
        - glClientWaitSync()
        - eglClientWaitSync()
        - glMapBufferRange() without*GL_MAP_UNSYNCHRONIZED*flag

      Avoid unnecessary calls to the above interfaces. The fewer calls, the better.
    - Avoid using glFlush() to split rendering passes, as the driver (Mali) will automatically flush when needed.
    - Perform Clear whenever possible. Use glClear/glDiscardFramebufferEXT/ glInvalidateFramebuffer to clean up the render texture

before drawing or at the beginning of the rendering pass to prevent the GPU from reading the data of the previous frame into the Tile buffer and save bandwidth. Vulkan uses loadOp.

    - Whenever possible, use glColorMask to mask color channels that do not need to be written.

Failure to follow the above recommendations may result in the following consequences:

1. If the pipeline is exhausted, the GPU is partially idle during the bubble generation period, resulting in performance loss.
2. There may be some performance instability depending on the interaction with the system dynamic voltage and frequency scaling power management logic.

- **Optimize Command Buffer.**

1. For best performance, set the ONE_TIME_SUBMIT_BIT flag. Do not set SIMULTANEOUS_USE_BIT unless you really need it.
2. Build per-frame command buffers instead of using synchronous command buffers.

3. If the alternative is to replay the same sequence of commands each time in the application logic, use SIMULTANEOUS_USE_BIT. It is more efficient than the application manually replaying the commands, but less efficient than submitting the buffer all at once.

4. Do not use a command pool with RESET_COMMAND_BUFFER_BIT set. This will increase memory management overhead because the driver cannot use a single large allocator for all command buffers in the pool.

5. Use secondary command buffer to allow construction of multi-threaded rendering passes.

6. Minimize the number of calls to the secondary command buffer per frame.

- **Optimize descriptor sets and layouts.**

1. Pack as much descriptor set binding space as possible.

2. Update descriptor sets that have been allocated but no longer referenced, instead of resetting the descriptor pool and reallocating new descriptor sets.
3. Reuse pre-allocated descriptor sets to avoid updating the same information.

4. Use VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC to bind the same UBO or SSBO, but at different offsets. Another option is to build more descriptor sets.

5. Do not leave blank spaces in the descriptor set, as this will waste space and block access continuity.
6. Do not leave unused entries, because copying and merging still consumes resources.

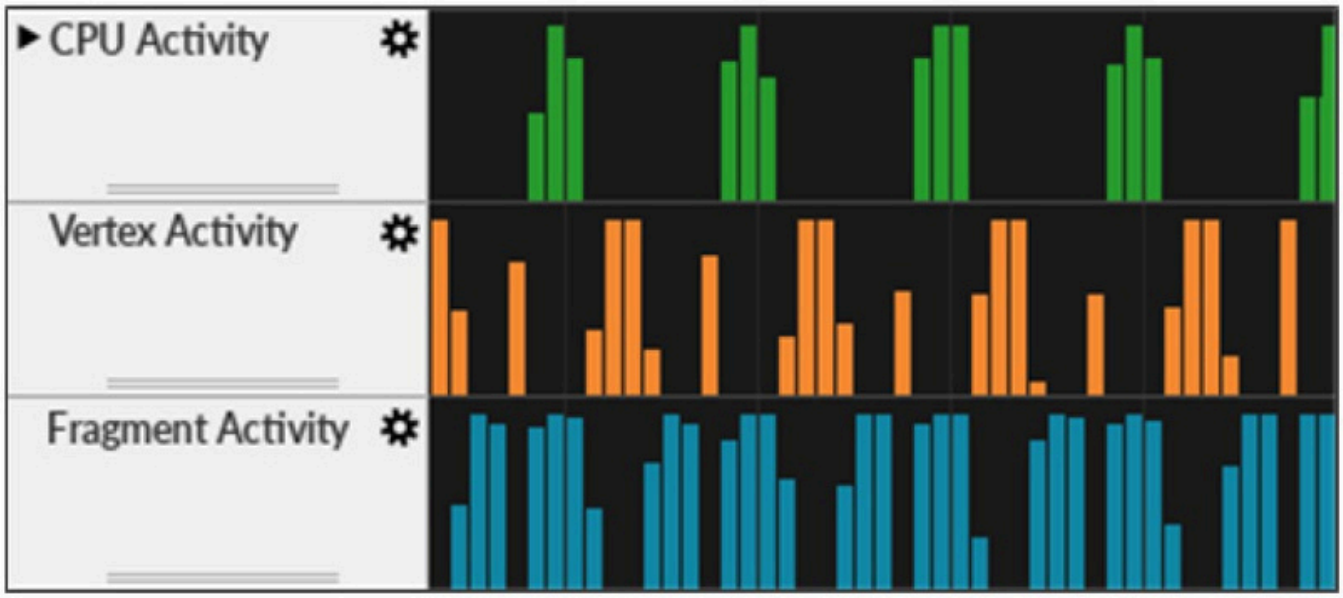
7. Do not allocate descriptor sets from the descriptor pool on performance-critical code paths.

8. If you don't plan to change the binding offset, don't use DYNAMIC_OFFSET UBOs/SSBOs, because there is a small additional cost to handle dynamic offsets. Inefficient descriptor sets and layouts The negative impact of unoptimized Vulkan descriptor sets and layouts can increase the CPU cost of draw calls.

- **Avoid render pipeline bubbles (idleness).**

Rendering pipeline bubbles may occur in the following situations:

1. Command Buffer submission is not frequent enough. Infrequent submission of command buffers will reduce the workload in the GPU processing queue and limit potential scheduling opportunities.
2. Data dependency. Suppose there are rendering passes M and N, with M at a later stage. When N is used by M earlier in the pipeline, data dependency occurs. Data dependency causes latency, during which enough work must be done to hide the latency in result generation.



Schematic diagram of rendering pipeline bubbles. The figure shows that there are bubbles in CPU, VS, and PS.

The following suggestions can reduce air bubbles in the tubing:

1. Submit the Command Buffer frequently. For example, after each major rendering pass in the frame, but not during the rendering pass.
2. If something is causing the bubbles, try filling the bubbles with techniques, for example, by inserting independent workloads between two rendering passes.
3. Consider generating dependent data in an earlier pipeline stage than the stage that uses the dependent data. For example, the compute stage is suitable for generating input data for the vertex shading stage. The fragment stage is not suitable because it is executed later than the vertex shading stage pipeline, otherwise it will cause lag and delay.

4. Consider processing dependent data at a later stage in the pipeline. For example, it is better for fragment shading to use the output from other fragment shading than for compute shading to use fragment shading.

5. Use fences to asynchronously read GPU data back to the CPU. Never call the interface that reads data from the GPU to the CPU synchronously, otherwise the entire rendering pipeline may be seriously stalled.

Additionally, the following suggestions can optimize the rendering pipeline:

1. Don't wait for GPU data unnecessarily anywhere in the pipeline.

2. Don't wait until the end of the frame to submit all rendering passes.

3. Don't create any backwards data dependencies in the pipeline without enough intermediate work to hide the latency.

4. Do not use vkQueueWaitIdle() or vkDeviceWaitIdle().

Use pipeline synchronization correctly.

-

Modern graphics APIs such as Vulkan have very fine-grained pipeline stages:

```
typedef enumVkPipelineStageFlagBits {

    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT =0x00000001,

    //Vertex Stages
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT          =0x00000002,
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT          =0x00000004,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT          =0x00000008,
    VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT          =0x00000010,
    VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT          =0x00000020,
    VK_PIPELINE_STAGE_GEOMET RY_SHADER_BIT        =0x00000040,
    // Fragment Stages
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT        =0x00000080,
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT          =0x00000100,
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT          =0x00000200,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT //          =0x00000400,
    Compute Stages
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT =0x00000800,
    VK_PIPELINE_STAGE_TRANSFER_BIT =0x00001000,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT =0x00002000,


    VK_PIPELINE_STAGE_HOST_BIT =0x00004000,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT          =  0x00008000,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT          =  0x00010000,

    (......)
} VkPipelineStageFlagBits;
```

Modern graphics APIs (such as Vulkan) also include many synchronization objects:

1. Subpass dependencies, Pipeline Barriers, Events, etc. are used for fine-grained synchronization within a single Queue.

2. Semaphore (signal) is used for heavier dependencies across queues.

There are two variables for pipeline dependencies:*srcStage*and*dstStage*.*srcStage*indicates the pipeline stage that must be waited for, and*dstStage*indicates the pipeline stage that must be waited for synchronization before processing begins.

For better parallel efficiency and fewer pipeline bubbles,*srcStage*should be as early as possible and

dstStage as late as possible. The worst performance is achieved if*srcStage*is *VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT* .

Semaphore can use pWaitDstStages to specify specific stages.

More specifically, you can achieve better rendering efficiency by following these guidelines:

1. The earlier srcStageMask is set, the better.

2. dstStageMask is set as late as possible.

3. Check whether the dependency is forward (eg srcStageMask is a vertex or calculation, dstStageMask is a fragment) or backward (eg srcStageMask is a fragment, dstStageMask is a vertex or calculation). Try to minimize the use of backward dependencies.

4. If backward dependencies are indeed necessary, add enough delay between producing and consuming resources to hide the scheduling bubble caused by the backward dependency.
5. Use srcStageMask = ALL_GRAPHICS_BIT and dstStageMask = FRAGMENT_SHADER_BIT to synchronize the two rendering passes with each other.
6. Zero-copy algorithms are the most effective, so minimize the use of TRANSFER copy operations. Pay close attention to the impact of TRANSFER copies on the hardware pipeline.
7. Use intra-queue barriers only when necessary, and schedule as much work as possible between barriers.

8. Don't leave your hardware idle.

9. Don't forget to handle overlapping vertices/calculations and fragments.

10. Do not use the following srcStageMask to dstStageMask synchronization combinations, as they will completely drain the pipeline:

```
BOTTOM_OF_PIPE_BIT    toTOP_OF_PIPE_BIT
ALL_GRAPHICS_BIT      to  ALL_GRAPHICS_BIT
ALL_COMMANDS_BIT      to  ALL_COMMANDS_BIT
```

11. If you merge pipeline barriers, be careful not to introduce false dependencies. Make sure not to break vertex/fragment overlaps and create an unnecessary bubble.

12. Instead of using VkEvent to signal and immediately wait for the event, use vkCmdPipelineBarrier().

13. Do not use VkSemaphore for dependency management in a single queue.

14. Do not leave too much idle time in the rendering pipeline (otherwise it will reduce performance), and do not leave too little idle time in the rendering pipeline (otherwise it may cause errors).

- **Properly handle pipeline resources.**

OpenGL ES provides a synchronous rendering model for application developers, even though the underlying execution may be asynchronous, and must reflect the state of data resources at the time of the draw call. If an application modifies a resource while a pending draw call still references it, the driver must take avoidance actions to ensure correctness.
The driver's synchronization behavior for these resources from GPU vendor to GPU vendor. For example, the Mali driver avoids blocking and waiting for resource reference counts to reach zero, as doing so would exhaust the pipeline and cause poor performance. The Mali GPU will create a brand new version of the resource, and the old or ghost version of the resource will remain until the pending draw call is completed and its reference count drops to zero. Other drivers (such as PowerVR) will block the rendering pipeline for the current frame and delay processing until the next frame, causing performance degradation.

This behavior is expensive, requiring memory to be allocated for the new resource and cleaned up when the empty resource is complete. If the update is not a full replacement, it also requires copying from the old resource buffer to the new resource buffer.

To optimize resources, follow these recommendations:

1. Avoid modifying resources referenced by queued draw calls. Use N-buffered resources and perform dynamic resource updates through pipelines.
2. Use the GL_MAP_UNSYNCHRONIZED flag to allow glMapBufferRange() to fill in unreferenced areas of the buffer that are still referenced by dynamic draw calls. Do not use GL_MAP_INVALIDATE_BUFFER / GL_MAP_INVALIDATE_RANGE with glMapBufferRange(), as these flags trigger the creation of an unnecessary resource copy in some versions of the driver.

- **Efficiently upload texture assets.**

When uploading texture resources to the hardware graphics, for uncompressed textures, they are uploaded linearly by scan line, and for compressed textures, they are uploaded block by block.
Some GPUs (such as PowerVR) use a unique layout internally to improve memory access locality and cache efficiency. The reformatting of data is done on the chip by dedicated hardware, so it is very fast. If you can follow the following steps, you can improve performance even more:

1. Upload textures during non-performance critical periods, such as initialization. This helps avoid frame rate drops associated with texture loading.

2. Avoid uploading mid-frame texture data to a texture object that is already used for that frame.

3. Perform a warm-up step after texture upload is complete. This still helps avoid frame rate drops related to texture loading.

The warm-up step mentioned above ensures that the texture is fully uploaded immediately. By default, glTexImage2D does not immediately perform all the processing required for an upload, and the texture is fully uploaded the first time it is used. You can force an upload by drawing a series of triangles on the screen or by binding the texture object in question.

12.6.1.3 Bandwidth Optimization

- **Pay attention to where the data is stored,**such as RAM, VRAM, Tile Buffer, GPU Cache, and reduce unnecessary data transfer.
- **Pay attention to the data access type,**such as whether it is read-only or write-only, whether atomic operations are required, and whether cache consistency is required.
- **Pay attention to the feasibility of caching data. The hardware can cache data for quick access by subsequent GPU operations.**The cache hit rate can be improved through the following points:
    - Improve transfer speeds by ensuring that client vertex data buffers are used in as few draw calls as possible. Ideally, the application should never use them.
    - Reduce the amount of data that the GPU needs to access when executing a schedule or draw call. This allows as much data as possible to fit into the cache line, increasing the hit rate.

- **Use texture compression formats.**ASTC is preferred, followed by ETC, PVRTC, BC and other compression formats. GPU hardware usually supports such compression formats, can encode and decode them quickly, and can read more texel content into the GPU cache line at one time, improving the cache hit rate.
- **Use a pixel format with fewer bits.**For example, RGB565 has 8 fewer bits than RGB888, and ASTC_6X6 replaces ASTC_4x4. For pixel formats supported by Adreno, seeSpec Sheets . **Use half-**
- **precision (such as FP16) instead of high-precision (FP32) data,**such as model vertex and index data, and use SOA (Structure of Array) data layout instead of AOS. **Render at a lower**
- **resolution and then enlarge it later.**This can reduce bandwidth, computing effort, and device heat generation.
- **Try to reduce the number of draws.**Reducing the number of draws can reduce the bandwidth and consumption between the CPU and GPU and within the GPU.
- **Ensure that data is stored on-chip.**

By using the features of PLS and Subpass, you can achieve delayed rendering, particle soft mixing, etc. on mobile terminals. The following table shows the relationship between the use of different bits and performance when PowerVR GX6250 implements delayed rendering:

Configuration	Time/frame (ms)
96bit + D32	20
128bit + D32	twenty one
160bit + D32	twenty three
192bit + D32	twenty four
224bit + D32	28
256bit + D32	29
288bit + D32	39

As can be seen from the above, when the bit number is greater than 256, which exceeds the maximum bit number of GX6250, the data cannot be completely stored in the On-Chip and will overflow to the global memory, causing the frame time to increase by 10ms, an increase of 34.5%.

Therefore, carefully assembling, optimizing and compressing the data of each pixel to ensure that the data can be completely accommodated on the On-Chip can effectively improve performance and save bandwidth.

- **Avoid redundant copies.**

Ensures that hardware components (CPU, graphics core, camera interface, video decoder, etc.) that use the same memory all access the same data without any intermediate copying.

- **Use the correct flags to create buffers, textures, etc.**Some Mali GPUs (such as Bifrost) perform the following flag combinations:

1. DEVICE_LOCAL_BIT | HOST_VISIBLE_BIT | HOST_COHERENT_BIT

2. DEVICE_LOCAL_BIT | HOST_VISIBLE_BIT | HOST_CACHED_BIT

3. DEVICE_LOCAL_BIT | HOST_VISIBLE_BIT | HOST_COHERENT_BIT | HOST_CACHED_BIT

4. DEVICE_LOCAL_BIT | LAZILY_ALLOCATED_BIT

The memory types of HOST_VISIBLE_BIT | HOST_COHERENT_BIT | HOST_CACHED_BIT are described as follows:
1. Provide cache storage on the CPU that is consistent with the GPU view of memory, without the need for manual synchronization.
2. If the chipset supports the hardware consistency protocol between the CPU and GPU, then the GPU supports this tag combination.

3. Due to hardware consistency, it avoids the overhead of manual synchronization operations. When available, cached, consistent memory is preferred over cached, inconsistent memory types.

4. Resources that must be mapped and read by application software on the CPU.

5. Hardware consistency consumes very little power, so it cannot be used for write-only resources on the CPU. For write-only resources, the CPU cache can be bypassed by using the Not Cached, consistent memory type.

Description of LAZILY_ALLOCATED memory type:

1. It is a special type of memory that initially only supports GPU virtual address space, not physical memory pages. If the memory is accessed, physical pages are allocated as needed.
2. Must be used with a transient attachment created with VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT. The purpose of a transient Image is to be used as a framebuffer attachment, which only exists in a single rendering process and can avoid using physical memory.

3. Data cannot be written back to global memory.

The following are recommended usage of Vulkan memory tags:

1. For immutable resources, use HOST_VISIBLE | HOST_COHERENT memory.

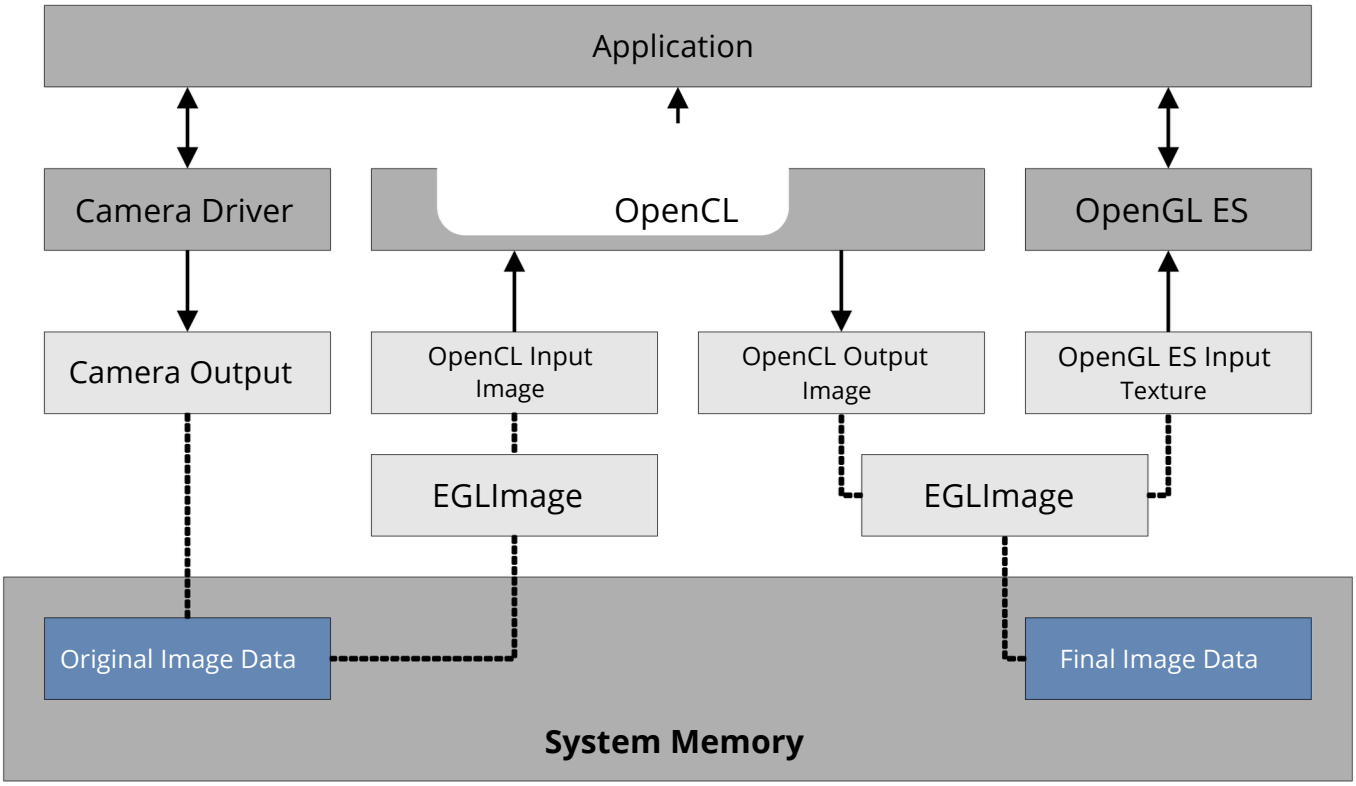
2. For write-only resources on the CPU, use HOST_VISIBLE | HOST_COHERENT memory.

3. Write updates to HOST_VISIBLE | HOST_COHERENT memory using memcpy(), or write sequentially for best efficiency with the CPU write-combine unit.
4. Use HOST_VISIBLE | HOST_COHERENT | HOST_CACHED memory to read resources back to the CPU. If this combination is not possible, use HOST_VISIBLE | HOST_CACHED.
5. Use LAZILY_ALLOCATED memory for temporary framebuffer attachments that only exist in a single rendering process.
6. Use only LAZILY_ALLOCATED memory for TRANSIENT_ATTACHMENT framebuffer attachments.

7. Mapping and unmapping buffers consumes CPU performance. Therefore, frequently accessed buffers should be persistently mapped, such as uniform buffers, data buffers, or dynamic vertex data buffers.

- **Use zero-copy paths whenever possible.**

As shown in the figure below, by using EglImage to realize that Camera and OpenCL share Original Image Data, OpenCL and OpenGL ES share Final Image Data, thus achieving zero copy:



- **Group memory accesses.**

The compiler uses several heuristics that can identify memory access patterns within the kernel that can be grouped together into bursts of read or write operations. In order for the compiler to perform this optimization, memory accesses should be grouped as closely together as possible.

For example, placing reads at the beginning of a kernel and writes at the end of the kernel gives the best efficiency. Accesses to larger data types like vectors are also compiled into single transfers whenever possible, loading 1 float4 is better than loading 4 separate float values.

- **Reasonable use of Shared/Local memory.**

In the early stage of Shader (such as initialization), frequently accessed data can be read into Shared/Local memory to improve access speed.

- **Access memory in row-major order.**

The GPU usually pre-reads row-adjacent data into the GPU cache. If the shader algorithm accesses data in a row-first manner, the cache hit rate can be improved and the bandwidth can be reduced.

- **GPU specific bandwidth optimizations.**

Mali's Transaction elimination only applies in the following situations:

1. The sampling data is 1.

2. The mimap level is 1.

3. image uses COLOR_ATTACHMENT_BIT.

4. The image does not use TRANSIENT_ATTACHMENT_BIT.

5. Use a single color attachment. (Mali-G51 GPU and later do not have this limitation)

6. The effective tile size is 16x16 pixels. The pixel data storage determines the effective tile size.

Mali GPUs also support AFBC textures, which can reduce video memory and bandwidth.

12.6.2 Resource Optimization

12.6.2.1 Texture Optimization

- **Use a compressed format.**

ASTC has become the preferred texture compression format due to its excellent compression rate, image quality closer to the original image, and adaptability to more platforms. Therefore, whenever possible, try to use ASTC. Unless some old devices cannot support ASTC, consider using texture compression formats such as ETC and PVRTC. See12.4.14 Adaptive Scalable Texture Compression for details .

- **Use Mipmaps whenever possible.**

Texture Mipmaps provide memory usage reduction to reduce the amount of data when sampling textures, thereby reducing bandwidth, increasing buffer hit rate, and improving image quality. Why not have the best of both worlds? Specifically, it is reflected in the following aspects:

1. Greatly improve texture cache efficiency to improve graphics rendering performance, especially in the case of strong reduction, texture data is more likely to be installed in Tile Memory.
2. Improve image quality by reducing aliasing caused by insufficient texture sampling without mipmapping.
However, using Mipmaps will increase memory usage by 33%. Avoid using Mipmaps in the following situations:
1. Filtering cannot be reasonably applied, for example to textures containing non-image data (indexed or depth textures).
2. Textures that are never scaled down, such as UI elements, where texels always map one-to-one to pixels.

- **Use a packed atlas.**

After packing the atlas, it is possible to batch render or instance render to reduce CPU and GPU bandwidth.

- **The size remains a power of 2.**

Although current graphics APIs already support non-power-of-2 (NPOT) textures, there are good reasons to keep texture sizes in powers of 2 (POT):

1. In most cases, POT textures should be preferred over NPOT textures, as this provides the best opportunity for hardware and driver optimization work. (eg texture compression, Mimaps generation, cache line alignment, etc.)

2D applications should not experience a performance loss from using NPOT textures (except possibly when uploading). 2D applications can be browsers or other applications that render UI elements, where NPOT textures are displayed with a one-to-one texel to pixel mapping.

3. Make sure the length and width of the texture are multiples of 32 pixels so that the texture upload can be optimized by the hardware.

- **Minimize texture size.**
- **Minimize texture bit depth.**
- **Minimize the number of texture components.**
- **Use texture channels to pack multiple maps.**For example, pack the roughness, specular, metalness, AO and other maps of the material into the RGBA channel of the same texture.

12.6.2.2 Vertex Optimization

- **Use staggered vertex layout with split positions. See**12.4.11 Index-Driven Vertex Shading for the reasons .
- **Use appropriate vertex and index storage formats.**Reducing data precision can reduce memory and bandwidth and increase the amount of computational units. The vertex formats currently supported by mainstream mobile GPUs are:

```
GL_BYTE
GL_UNSIGNED_BYTE
GL_SHORT
GL_UNSIGNED_SHORT
GL_FIXED
GL_FLOAT
GL_HALF_FLOAT
GL_INT_2_10_10_10_REV
GL_UNSIGNED_INT_2_10_10_10_REV
```

- **Consider instancing geometric objects**.Modern mobile GPUs generally support instanced rendering, which reduces bandwidth by submitting a small amount of geometric data and drawing it multiple times. Each instance is allowed to have its own data, such as color, transformation matrix, lighting, etc. It is often used for objects such as trees, grass, buildings, and groups of soldiers.
- **The primitive type uses triangles.**Modern GPUs are designed to process triangles, and if it is a quadrilateral or something like that, the efficiency is likely to be reduced.
- **Reduce the size of the index array,**such as using strip format instead of simple list format, and using raw valid indices instead of degenerate triangles.
  For the post-transform cache, optimize the index locally.
-

- **Avoid using index buffers with low spatial consistency.**This will reduce cache hit rates.
- **Use instance attributes to work around any uniform buffer size limitations.**For example, 16KB uniform buffers.
- **Each instance uses 2N vertices.**
- **Prefer indexed lookups using gl_InstanceID to uniform buffers or shader storage buffers over per-instance attribute data.**

12.6.2.3 Grid Optimization

- **Use LODs.**

Using LOD for meshes can improve rendering performance and reduce bandwidth. Conversely, not using LOD can create performance bottlenecks.

Wireframe mode with different LODs of the same mesh.

The following are examples of wasted computation and memory resources:

1. Objects that use a large number of polygons do not cover a small area on the screen, such as a distant background object.
2. Use polygons for details that will never be seen due to camera angle or clipping (such as objects outside the field of view).
3. Use a large number of primitives for the object. In fact, you can use fewer primitives to draw it and still ensure that the visual effect is not lost.

- **Simplify the model and merge vertices.**By merging adjacent vertices, the number of mesh vertices can be effectively reduced. Using mesh simplification technology, good LOD data can be generated.
- **Offline merging of small grids close together,**such as sand, vegetation, etc.
- **The number of vertices in a single mesh cannot exceed 65k.**This is mainly because the vertex index precision on mobile devices is 16 bits, and the maximum value is 65535.
- **Remove invisible primitives,**such as triangles inside a box.
- **Use simple geometric objects and add details with normal maps and bump maps.**
- **Avoid small triangles.**

The Quad drawing mechanism will greatly increase OverDraw for small triangles. On PowerVR hardware, for triangles covering less than 32 pixels, it will affect the efficiency of rasterization and cause performance bottlenecks.

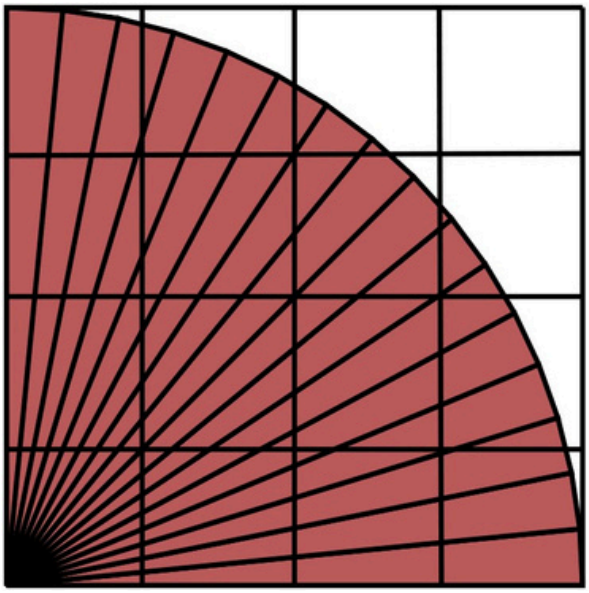
Submitting many small triangles may cause the hardware to spend a lot of time processing them in the vertex stage, where the number of triangles rather than their size is the main factor. In particular,

this can cause bottlenecks in tile accelerator (TA) fixed-function hardware. A large number of small triangles will result in increased accesses to the parameter buffer located in system memory, increasing memory bandwidth usage.

- **Make sure that each primitive in the grid can create at least 10 to 20 pixels.**
- **Use almost equilateral triangles**to maximize the ratio of area to side length and reduce the number of generated quads.
- **Avoid long, thin triangles.**

Similar to small triangles, thin and long triangles (shown in red in the figure below) will also generate more invalid pixels, occupy higher GPU resources, and increase Overdraw.

- **Avoid using fan-shaped or similar geometric layouts.**The center point of the triangle fan has a high triangle density, so that each triangle has very low pixel coverage. Tile axis-aligned cutting can be considered, but it will introduce more triangles. (Figure below)



8	6	3	0
11	9	6	3
14	12	9	6
16	14	11	8

The number of triangles generated after the fan (left) is cut in a tile-axis-aligned manner (right).

12.6.3 Shader Optimization

12.6.3.1 Statement Optimization

- **Use appropriate data types.**

Using the most appropriate data type in your code allows the compiler and driver to optimize your code, including pairing of shader instructions. Using the vec4 data type instead of float may prevent the compiler from performing optimizations.

```
int4 ResultOfA(int4 a)
{
    returna +1;// int4andintAdding, only1Instructions.
}

int4 ResultOfA(int4 a)
{
    returna +1.0;// int4andfloatAdd, need3Instructions:int4 -> float4 ->Add ->int4
}
```

- **Reduce type conversions.**

```
uniform    sampler2DColorTexture; TexC;
invec2
vec3light(invec3amb,invec3diff) {

    //Texture Sample Returnsvec4,will be implicitly converted tovec3,
    Extra1Instructions. vec3Color =Texture(ColorTexture, TexC); Color
    *= diff + amb;
    returnColor;
}

//In the following code, input parameters/temporary variables/return values arevec4,No implicit type conversion, less code than above1Instructions. uniform
        sampler2DColorTexture; TexC;
invec2
vec4light(invec4amb,invec4diff) {

    vec4Color =Texture(Color, TexC); Color *= diff
    + amb;
    returnColor;
}
```

- **Packed scalar constant.**

Filling scalar constants into a vector consisting of four channels greatly improves hardware acquisition efficiency. In GPU skeletal animation systems, the number of skinned bones can be increased.

```
floatscale, bias;//twofloatvalue. vec4a = Pos * scale + bias;
//Two instructions are required.
```

```
vec2scaleNbias;//Put twofloatValues are packaged into avec2
vec4a = Pos * scaleNbias.x + scaleNbias.y;//An instruction (mad)Finish.
```

- **Use scalar operations.**

Be careful about vectorizing scalar operations, as the same vectorized output will take more cycles. For example:

```
highpvec4v1, v2; highp
floatx, y;

// Bad!!
v2 = (v1 * x) * y;// vector*scalarthenvector*scalarTotal8Scalarmuladd. // Good!!

v2 = v1 * (x * y);// scalar*scalarthenvector*scalarTotal5Scalarmuladd.
```

12.6.3.2 State Optimization

- **Use const whenever possible.**

The const keyword can provide significant performance improvements if used correctly. For example, a shader with a const array declared outside of the main() block will perform much better than one without.

Another example is referencing an array member by const value. If the value is const, the GPU can know in advance that the number will not change, and the data can be pre-read before running the shader, thus reducing stall.

- **Keep shader instruction count reasonable**.

Too long shaders are usually inefficient, for example if you need to include many instruction slots in a shader relative to the number of texture fetches, you can consider splitting the algorithm into several parts. The values generated by part of the algorithm can be stored in a texture and then retrieved by sampling the texture. However, this approach is expensive in terms of memory bandwidth. The following situations can also reduce the efficiency of texture sampling: 1. Use trilinear, anisotropic filtering, wide texture formats, 3D and cube map textures, texture projection;
2. Texture lookup using different Lod gradients; 3. Gradient calculation across pixel Quad.

- **Minimize the number of shader instructions.**

Modern shader compilers often perform specific instruction optimizations, but it is not automatic and effective. Oftentimes manual intervention is required to analyze the shader and reduce instructions as much as possible, even if saving one instruction is worthwhile.

- **Avoid using uber-shaders.**

An uber-shader combines multiple shaders into a single shader using static branching. This makes sense if you are trying to reduce state changes and batch draw calls. However, it usually increases the GPR count, which hurts performance.

- **Sampling textures efficiently.**

There are many ways to sample (filter) textures, and performance is usually inversely proportional to effect:

Some filtering types of textures and their corresponding effect pictures.

To sample textures efficiently, the following rules must be followed:

1. Avoid random access and keep sampling within the same 2x2 pixel Quad, which has a high hit rate and makes the shader more efficient.
2. Avoid using 3D textures. Fetching data from volume textures is usually expensive due to the need to perform complex filtering to calculate the resulting values.
3. Limit the number of Shader texture samples. Using four samplers in a shader is acceptable, but sampling more textures may cause performance bottlenecks.
4. Compress all textures. This allows better memory usage, translating into fewer texture stalls in the rendering pipeline.
5. Consider turning on Mipmaps. Mipmaps help merge texture fetches and help improve performance at the expense of increased memory usage. They also reduce bandwidth and improve cache hit rates.
6. Try to use simple texture filtering. The sampling methods from high to low performance (from low to high effect) are: nearest, bilinear, cubic, tri-linear, anisotropic. The more complex the sampling method, the more data will be read, thereby increasing memory access bandwidth, reducing cache hit rate, and causing greater latency. You need to pay special attention to this.
7. Give priority to using texelFetch / texture(), which is usually more efficient than texture sampling (but requires tool analysis and verification).
8. Be cautious with pre-calculated texture LUTs. In real-time rendering, it is common to encode the results of complex calculations into textures and use them as lookup tables (such as irradiance maps for IBLs, pre-integrated maps for skin subsurface scattering). This approach will only improve performance when the shader is the bottleneck. If the function parameters and texture coordinates in the lookup table differ greatly between adjacent fragments, cache efficiency will be affected. Performance profiling should be performed to determine whether this method actually improves performance.
9. Use mediump sampler instead of highp sampler, the latter is half the speed of the former.

10. Anisotropic Filtering (AF) optimization suggestions:

(1) Use 2x anisotropy first and evaluate whether it meets the quality requirements. Higher sample counts can improve quality, but they also have diminishing returns and are often not commensurate with the performance cost.

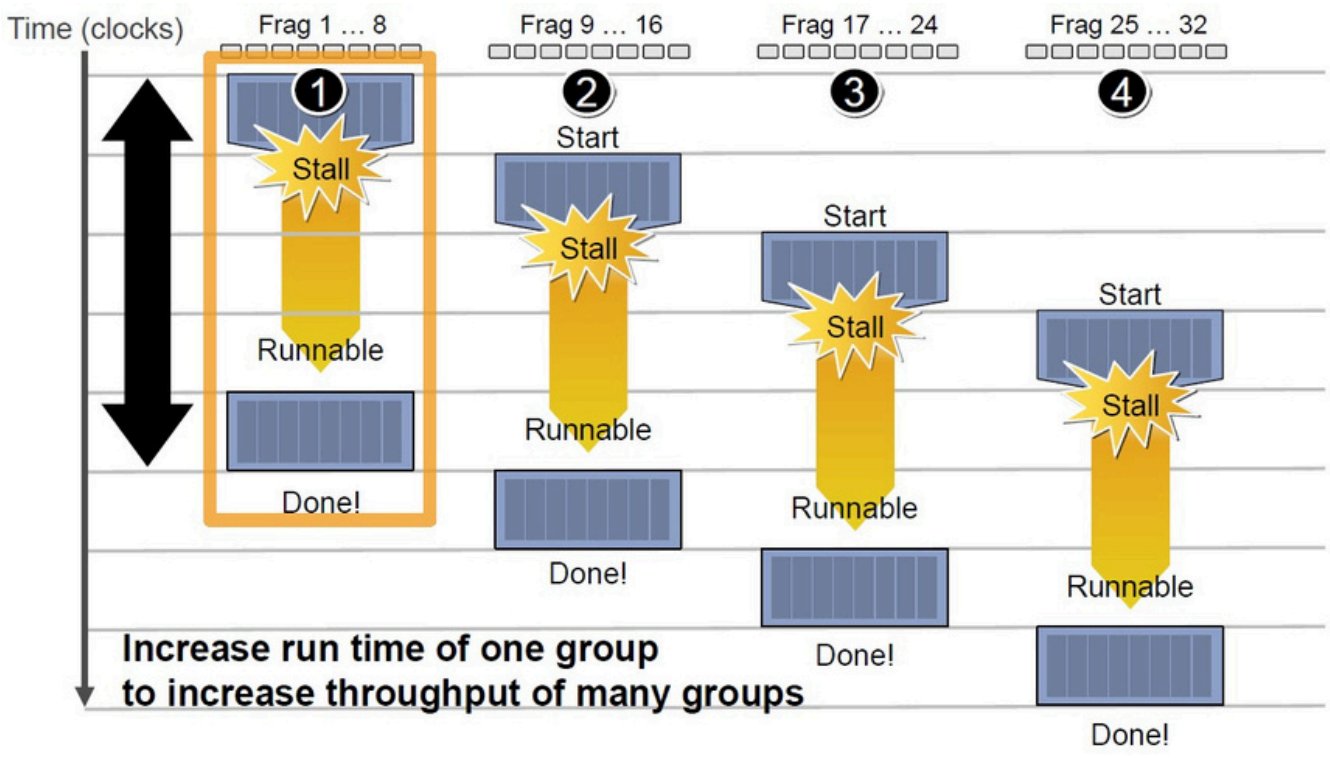
(2) Consider using 2x bilinear anisotropy instead of trilinear isotropy. In areas of high anisotropy, the 2x bilinear algorithm is faster and produces better image quality. Note that by switching to bilinear filtering, seams can be seen at the transition points between mipmap levels .

(3) Use anisotropic and trilinear filtering only for objects that benefit the most. Note that 8x trilinear anisotropy is 16 times more expensive than simple bilinear filtering!

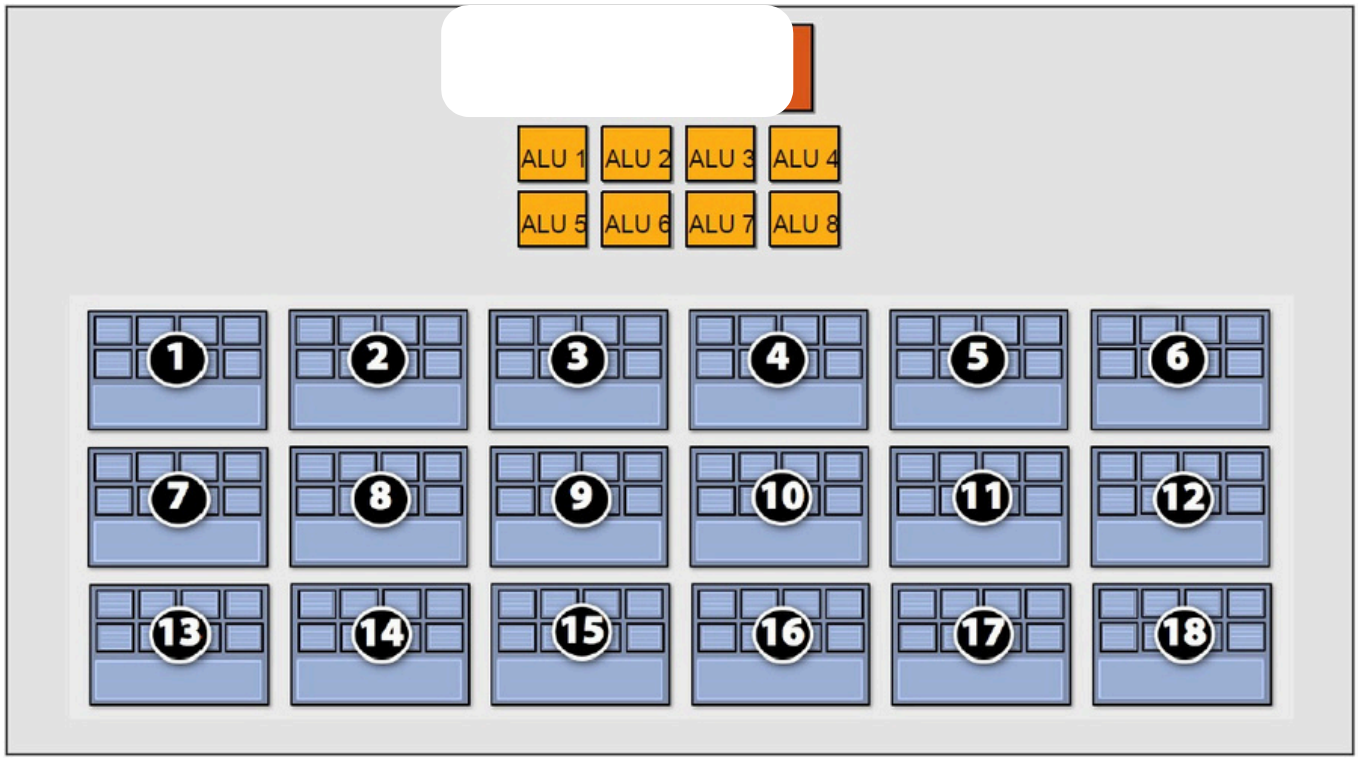
- **Try to avoid dependent texture reads.**

Dependent texture reads are a special type of texture read where the texture coordinates depend on some calculation in the shader (rather than changing in some regular pattern). Since the value of this calculation cannot be known in advance, it is impossible to pre-fetch texture data, thus reducing the cache hit rate during shader processing, causing lag.
Vertex shading texture lookups are always treated as dependent texture reads, just like channel-z dependent texture reads in fragment shading. On some drivers and platform versions, Texture2DProj() can also act as a dependent texture read if given an invalidwVec3 or Vec4.

The costs associated with dependent texture reads can be offset to some extent by hardware thread scheduling, especially for shaders that involve a lot of math calculations. This process involves the thread scheduler pausing the current thread and swapping in another thread to process on the USC. This swapped thread will do as much processing as possible, and once the texture fetch is complete, the original thread will be swapped back (Figure 2).



The GPU's Context needs to access the cache or memory, which will cause a delay of several clock cycles. At this time, the scheduler will activate the second set of Context to utilize the ALU.



The more contexts available on a GPU, the higher the throughput of the computing unit can be. The architecture with 18 groups of contexts in the above figure can maximize the throughput.

Although the hardware will do its best to hide memory latency, for good performance you should avoid relying on texture reads as much as possible. Applications should try to calculate texture coordinates before the fragment shader executes.

- **Avoid using dynamic branches**.

Dynamic branches will delay shader instruction time, but if the branch condition is a constant, the compiler will optimize it. Otherwise, if the conditional statement is related to uniform or mutable variables, it cannot be optimized. Other suggestions:

1. Minimize dynamic branching among spatially adjacent shading threads.

2. Use built-in functions such as min(), max(), clamp(), mix(), saturate() to avoid branch statements.

3. Check the benefits of branching relative to the calculation. For example, skipping the lighting calculation for pixels above a threshold distance from the camera is usually faster than directly calculating it.

- **Pack shader interpolation data.**

Shader interpolation requires**GPR (General Purpose Register)**to pass data to the pixel shader. The number of GPR is limited, and if it is full, it will cause stall, so try to minimize their use.
Don't use varyings when you can use uniforms. Pack values together, since all varyings have four components regardless of whether they are used or not, like putting two vec2 texture coordinates

into a vec4. There are other more creative ways to pack and compress data on the fly.

- **Reduce shader GRP usage.**

The more**GPRs (General Purpose Registers)**are occupied, the more computation is required. If there are not enough available registers, register overflow may occur, resulting in poor performance. The following measures can reduce GPR occupancy:

1. Use simpler shaders.

2. Modify GLSL to reduce even one instruction, which can sometimes reduce the occupancy of one GPR.

3. Unrolling loops can also save GPRs, but it depends on the shader compiler.

4. Configure the shader according to the target platform to ensure that the final solution chosen is the most efficient.
5. Unrolling loops tend to push texture fetches to the top of the shader, requiring more GPRs to hold multiple texture coordinates and fetch the results simultaneously.
6. Minimize the number of global and local variables. Reduce the scope of local variables.

7. Minimize the data dimension. For example, don't use 3D if you can use 2D.

8. Use data types with lower precision, such as FP16 instead of FP32.

- **Avoid constant math in shaders**.

Almost every game released since shaders have been spending unnecessary math instructions on shader constants. These instructions need to be identified in the shader to move these calculations to the CPU. It might be easier to identify the math for shader constants in the compiled code.

- **Avoid using discard statements in pixel shaders.**

Some developers believe that manually discarding (also known as killing) pixels in the pixel shader can improve performance. In fact, it is not that simple, for the following reasons:
1. If some pixels in a thread are killed, but other pixels in the same Quad are not, the shader still executes.
2. Depends on how the compiler generates microcode.

3. Some hardware architectures (such as PowerVR) will disable TBDR optimization, causing stall and data write-back in the rendering pipeline.

- **Avoid modifying depth in pixel shaders**.

The reason is similar to the previous one.

- **Avoid sampling textures in VS.**

Although the current mainstream GPUs have adopted a unified shader architecture, the execution performance of VS and PS is similar. However, it is still necessary to ensure that the texture operations in VS are local and the textures use a compressed format.

- **Split up special draw calls**.

If a shader bottleneck is GPR and/or texture cache, splitting the Draw Call into multiple Passes can actually increase performance. However, the results are difficult to predict and should be based on actual performance testing.

- **Use low-precision floating point numbers whenever possible**.

The computing performance of FP16 is usually twice that of FP32, so low-precision floating-point numbers should be used as much as possible in the shader.

```
precision mediumpfloat;

# ifdef GL_FRAGMENT_PRECISION_HIGH
      #define    NEED_HIGHP highp
# else
      #define    NEED_HIGHP mediump
# endif

varyingvec2vSmallTexCoord; varyingNEED_HIGHP
vec2vLargeTexCoord;
```

UE also encapsulates floating-point numbers so that the precision of floating-point numbers can be switched freely on different platforms and image qualities.

- **Try to migrate PS operations to VS.**

Typically, the number of vertices is significantly smaller than the number of pixels. By moving calculations from the pixel shader to the vertex shader, the GPU workload can be reduced and redundant calculations can be eliminated. For example, splitting the diffuse and specular reflections of lighting calculations, migrating the diffuse
reflections to VS, while retaining the specular reflections in PS, can achieve a lighting result with a good balance between effect and efficiency.

- **Optimize Uniform / Uniform Buffer.**

1. Keep uniform data as small as possible. No more than 128 bytes, so that any given shader runs well on most GPUs.
2. Change the uniform to a compile-time constant with #define for OpenGL ES, a dedicated constant for Vulkan, or a static syntax in the shader source.
3. Avoid constants in uniform vectors or matrices, such as elements that are always 0 or 1.

4. Prefer using glUniform() to set uniforms instead of loading them from a buffer.

5. Do not dynamically index uniform arrays.

6. Do not overuse instantiation. Using gl_InstanceID to access instanced uniforms is a dynamic index, and register-mapped uniforms cannot be used.
7. Move Uniform-related calculations to the CPU application layer as much as possible.

8. Try to use uniform buffers instead of shader storage buffers. Use uniform buffers whenever possible, as long as there is enough space. If uniform buffer objects are statically indexed in GLSL and are small enough, the driver or compiler can map them into the same hardware constant RAM used for default uniform block global variables.

- **Keep the UBO footprint as small as possible.**

If the UBO is smaller than 8k, it can be put into constant memory, which will achieve higher performance. Otherwise, it will be stored in global memory, which will significantly increase the access time cycle.

- **Choose a better shading algorithm.**

Choosing a better and more efficient algorithm is more important than low-level (instruction-level) optimization, because the former can significantly improve performance.

- **Choose an appropriate coordinate space.**

A common mistake in vertex shaders is to perform unnecessary conversions between model space, world space, view space, and clip space. If the model-world transformation is a rigid body transformation (containing only rotations, translations, mirroring, lighting, or similar) , then it can be calculated directly in model space.
Avoid converting the position of each vertex to world or view space. It is better to convert uniforms (such as light position and direction) to model space because it is a per-mesh operation and requires less calculation. In cases where a specific space must be used (such as cube map reflections), it is best to use this space throughout the shader to avoid using multiple coordinate spaces in the same shader.

- **Optimize interpolation (Varying) variables.**

Reduce the number of interpolation variables, reduce the dimensions of interpolation variables, delete useless (unused by the fragment shader) interpolation variables, pack them compactly, and use medium and low precision data types as much as possible.

- **Optimize Atomic.**

Atomic operations are common in many computational algorithms and some fragment algorithms. With some minor modifications, atomic operations allow many algorithms to be implemented on highly parallel GPUs that would otherwise be serial.

The key performance issue of atomics is contention. Atomic operations from different shader cores need to reach the same cache line, which requires data consistency access to the L2 cache.

Atomic operations are avoided by keeping them within a single shader core, and atomic are most efficient when the shader core controls the necessary cache lines in L1. The following are specific optimization recommendations: 1. Consider how to avoid contention when using atoms in algorithm design. 2. Consider setting the atomic spacing to 64 bytes to avoid multiple atoms competing on the same cache
line.
3. Consider whether you can amortize contention by accumulating into shared memory atoms. Then, have one of the threads push a global atomic operation at the end of the work group.

- **Make full use of the instruction cache.**

The shader core instruction cache is an often overlooked factor that affects performance. Due to the large number of concurrently running threads, the importance of instruction cache to performance is of great importance. The optimization suggestions are as follows:

1. Use shorter shaders with more threads instead of longer shaders with fewer threads. Shorter shader instructions are more likely to hit in the cache.
2. Use shaders without dynamic branches. Dynamic branches will reduce temporal locality and increase cache pressure.
3. Don't unroll loops too aggressively, although some unrolling may help.

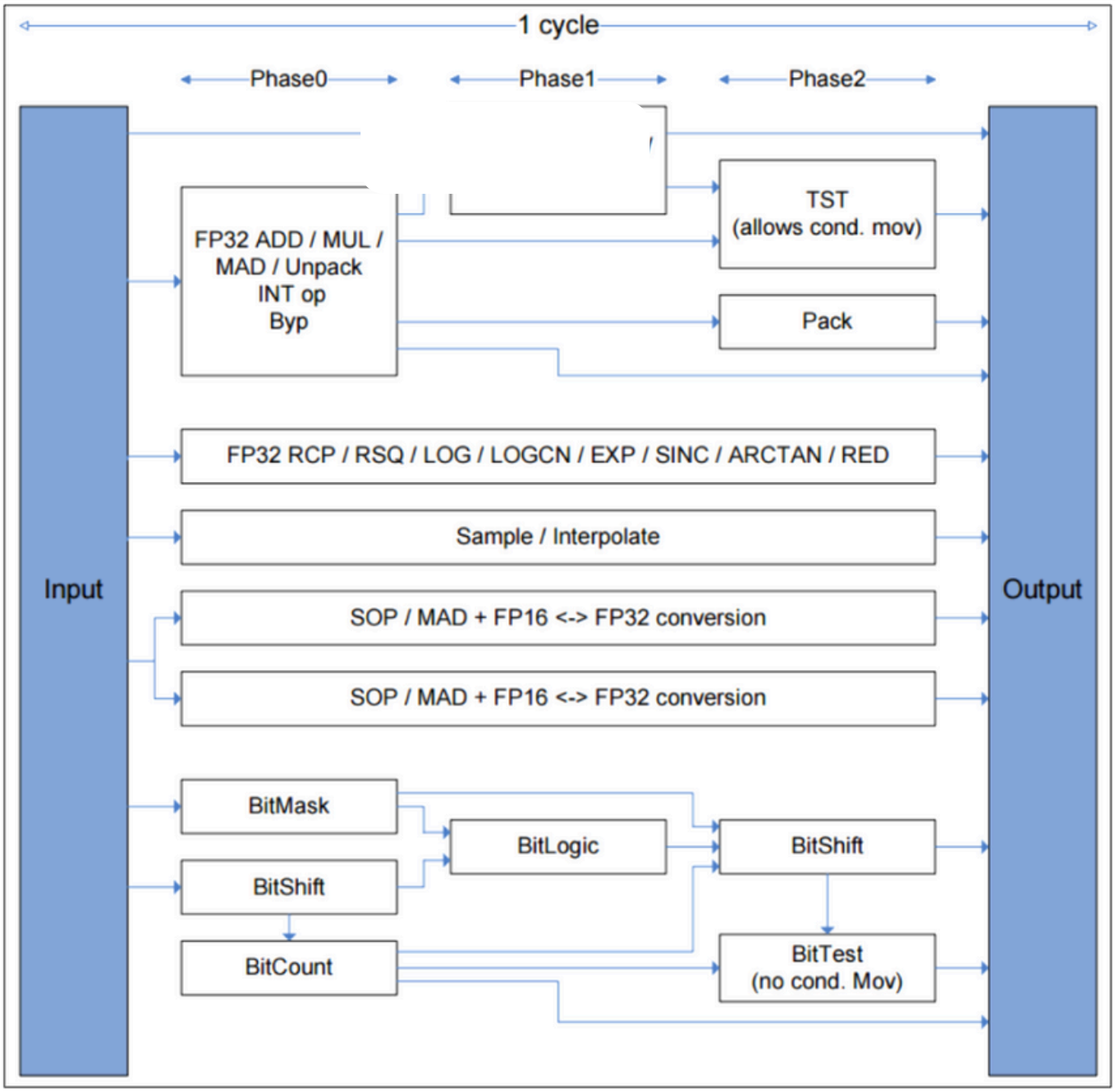
4. Do not generate duplicate shaders or binaries from the same source code.

5. Be careful when there are multiple visible fragment shaders in the same tile (ie Overdraw). All fragment shaders that are not culled by Early-ZS or FPK/HSR must be loaded and executed, increasing cache pressure.

12.6.3.3 Assembly-level optimization

It is recommended that low-level optimization of Shaders be focused on and performed only in extremely performance-sensitive areas or in the later stages of optimization, otherwise it may be counterproductive.

For GPU instruction sets, many instructions can be completed in 1 clock cycle, but some instructions require multiple cycles. The following figure shows some instructions that can be completed in 1 clock cycle for PowerVR:



For the measurement of peak performance, if we take PowerVR 500MHz G6400 as an example, the peak performance data of common instructions are as follows:

Data Types	operate	Single instruction operand	Single instruction clock	Theoretical throughput
16-bit float	**Sum-Of-Products**	6	1	(0.5 × 4 × 16 × 6) ÷ 1 =**192 GFLOPS**
float	**Multiply-and-Add**	4	1	(0.5 × 4 × 16 × 4) ÷ 1 =**128 GFLOPS**

Data Types	operate	Single instruction operand	Single instruction clock	Theoretical throughput
float	**Multiply**	2	1	(0.5 × 4 × 16 × 2) ÷ 1 =**64 GFLOPS**
float	**Add**	2	1	(0.5 × 4 × 16 × 2) ÷ 1 =**64 GFLOPS**
float	**DivideA**	1	4	(0.5 × 4 × 16 × 1) ÷ 4 =**8 GFLOPS**
float	**DivideB**	1	2	(0.5 × 4 × 16 × 1) ÷ 2 =**16 GFLOPS**
int	**Multiply-and-Add**	2	1	(0.5 × 4 × 16 × 2) ÷ 1 =**64 GILOPS**
int	**Multiply**	1	1	(0.5 × 4 × 16 × 1) ÷ 1 =**32 GILOPS**
int	**Add**	1	1	(0.5 × 4 × 16 × 1) ÷ 1 =**32 GILOPS**
int	**Divide**	1	30	(0.5 × 4 × 16 × 1) ÷ 30 =**1.07 GILOPS**

Performance estimates are calculated based on theoretical peak values. In practice, due to various dependencies, frequency reduction, context switching, etc., the actual peak value may not be reached.

By default, the compiler implements floating point division as two range reductions followed by a reciprocal and a multiply instruction, requiring 4 cycles.

In addition, it is important to mention integer division, which is extremely inefficient and should be avoided. You can convert it to float first and then divide it.
For more information about the cost of instructions, seeComplex Operations .

The following are common low-level optimization measures (using PowerVR as an example; other GPUs are similar but not exactly the same, so actual measurements should prevail).
1. To fully utilize the USC core, mathematical expressions must always be written in multiply-add (MAD) form. For example, changing the following expression to use MAD form can reduce cycle

costs by 50%:

```
fragColor.x = (tx + ty) * (tx - ty);// 2 cycles {sop,
        sop, sopmov}
{sop,    sop}
- - >
fragColor.x = tx * tx + (-ty * ty);// 1 cycle {sop, sop}
```

2. It is usually better to write division in reciprocal form, because the reciprocal form is directly supported by the instructions. Simplification of mathematical expressions can further improve performance. RCP

```
fragColor.x = (tx * ty + tz) / tx;// 3 cycles {sop,
        sop, sopmov}
{frcp}
{sop,    sop}
- - >
fragColor.x = ty + tz * (1.0/tx);// 2 cycles {frcp}

{sop, sop}
```

3.sign(x) The results may be the following:

```
if(x >0)
{
    return    1;
}
else if(x <0) {

    return    - 1;
}
else
{
    return    0;
}
```

But using signit is not the best choice to get symbols:

```
fragColor.x =sign(tx) * ty;// 3 cycles {mov, pck, tstgez,
mov} {mov, pck, tstgez, mov} {sop, sop}


- - >
fragColor.x = (tx >=0.0?1.0:-1.0) * ty;// 2 cycles {mov, pck, tstgez, mov} {sop,
sop}
```

4. Useinversesqrt insteadsqrt:

```
fragColor.x =sqrt(tx) >0.5?0.5:1.0;// 3 cycles {frsq}
```

```
{frcp}
{mov, mov, pck, tstg, mov}
- - >
fragColor.x = (tx *inversesqrt(tx)) >0.5?0.5:1.0;// 2 cycles {frsq}

{fmul, pck, tstg, mov}
```

5. Negation optimization of normalize:

```
fragColor.xyz =normalize(-t.xyz);// 7 cycles {mov,
        mov, mov}
{fmul,
{fmad, mov}
{fmad, mov}
{frsq}
{fmul, fmul, mov, mov} {fmul,
mov}
- - >
fragColor.xyz = -normalize(t.xyz);// 6 cycles {fmul,
         mov}
{fmad,  {fmad,  {frsq}  {fmul,
fmul, mov, mov} {fmul, mov}
```

6. Optimization of abs, dot, neg, clamp, saturate, etc.:

```
//abs
fragColor.x =abs(tx * ty);// 2 cycles {sop,
        sop}
{mov,   mov, mov}
- - >
fragColor.x =abs(tx) *abs(ty);// 1 cycle {sop, sop}


// dot
fragColor.x = -dot(t.xyz, t.yzx);// 3 cycles {sop,
        sop, sopmov}
{sop,   sop}
{mov, mov, mov}
- - >
fragColor.x =dot(-t.xyz, t.yzx);// 2 cycles {sop,
        sop,   sopmov}
{sop,   sop}

// clamp
fragColor.x =1.0-clamp(tx,0.0,1.0);// 2 cycles {sop,
        sop, sopmov}
{sop,   sop}
- - >
fragColor.x =clamp(1.0- tx,0.0,1.0);// 1 cycle {sop, sop}


// min / clamp
fragColor.x =min(dot(t, t),1.0) >0.5? tx : ty;// 5 cycles
```

```
{sop, sop, sopmov}
{sop,     sop}
{mov, fmad, tstg, mov} {mov, mov,
pck, tstg, mov} {mov, mov, tstz,
mov}
- - >
fragColor.x =clamp(dot(t, t),0.0,1.0) >0.5? tx : ty;// 4 cycles {sop,
            sop, sopmov}
{sop,     sop}
{fmad, mov, pck, tstg, mov} {mov,
mov, tstz, mov}
```

7. Exp, Log, Pow:

```
// exp2
fragColor.x =exp2(tx);// one cycle {fexp}


//exp
floatexp(floatx ) {

        return exp2(x *1.442695);// 2 cycles {sop, sop}

        {fexp}
}

// log2
fragColor.x =log2(tx);// 1 cycle {flog}


// log
floatlog(floatx ) {

        return log2(x *0.693147);// 2 cycles {sop, sop}

        {flog}
}

// pow
floatpow(floatx,floaty ) {

        return exp2(log2(x) * y);// 3 cycles {flog}

        {sop, sop}
        {fexp}
}
```

Execution efficiency from high to low: exp2 = log2 > exp = log > pow.

8. Sin, Cos, Sinh, Cosh:

```
// sin
fragColor.x =sin(tx);// 4 cycles {fred}

{fred}
{fsinc}
```

```
{fmul, mov}// plus conditional

// cos
fragColor.x =cos(tx);// 4 cycles {fred}

{fred}
{fsinc}
{fmul, mov}// plus conditional

// cosh
fragColor.x =cosh(tx);// 3 cycles {fmul, fmul, mov,
mov} {fexp}

{sop, sop}

// sinh
fragColor.x =sinh(tx);// 3 cycles {fmul, fmul, mov,
mov} {fexp}

{sop, sop}
```

Execution efficiency from high to low: sinh = cosh > sin = cos.

9. Asin, Acos, Atan, Degrees, and Radians:

```
fragColor.x =asin(tx);// 67 cycles fragColor.x =acos(tx);// 79 cycles
fragColor.x =atan(tx);// 12 cycles (Many judgment conditions)


fragColor.x =degrees(tx);// 1 cycle {sop, sop}


fragColor.x =radians(tx);// 1 cycle {sop, sop}
```

From the above, we can see that acos and asin are extremely inefficient, taking up to 79 clock cycles; followed by atan, 12 time cycles; the fastest are degrees and radians, 1 clock cycle.

10. Vectors and matrices:

```
fragColor = t * m1;// 4x4 matrix, 8 cycles {mov}

{wdf}
{sop,   sop,   sopmov}
{sop,   sop,   sopmov}
{sop,   sop}
{sop,   sop,   sopmov}
{sop,   sop,   sopmov}
{sop,   sop}

fragColor.xyz = t.xyz * m2;// 3x3 matrix, 4 cycles {sop,
        sop,   sopmov}
{sop,   sop}
{sop,   sop,   sopmov}
{sop,   sop}
```

Vectors and matrices are more efficient when they have fewer dimensions, so try to reduce their dimensions as much as possible.

11. Scalar and vector operations:

```
fragColor.x =length(tv);// 7 cycles fragColor.y =
distance(v, t); {sopmad, sopmad, sopmad, sopmad}
{sop, sop, sopmov}

{sopmad, sopmad, sopmad, sopmad} {sop,
        sop, sopmov}
{sop, {frsq} {frcp} - - > fragColor.x =length(tv);//
9 cycles fragColor.y = distance(t, v); {mov}
        sop}
```

```
{wdf}
{sopmad, sopmad, sopmad, sopmad} {sop,
        sop,   sopmov}
{sop,   sop,   sopmov}
{sop,   sop}
{frsq}
{frcp}
{mov}

fragColor.xyz =normalize(t.xyz);// 6 cycles {fmul,
          mov}
{fmad, {fmad, {frsq} {fmul, fmul, mov, mov} {fmul, mov} - - >
fragColor.xyz =inversesqrt(dot(t.xyz, t.xyz) ) * t.xyz;// 5 cycles {sop,
        mov}
        mov}
```

```
        sop,   sopmov}
{sop,   sop}
{frsq}
{sop,   sop}
{sop,   sop}

fragColor.xyz =50.0*normalize(t.xyz);// 7 cycles {fmul,
          mov}
{fmad, {fmad, {frsq} {fmul,
fmul, mov, mov} {fmul, fmul,
mov, mov} {sop, sop}
        mov}
        mov}
```

```
- - >
fragColor.xyz = (50.0*inversesqrt(dot(t.xyz, t.xyz) )) * t.xyz;// 6 cycles {sop,
        sop,   sopmov}
{sop,   sop}
{frsq}
{sop,   sop,   sopmov}
```

```
{sop,    sop}
{sop,    sop}
```

The following is the expanded form of some built-in functions of GLSL:

```
vec3cross(vec3a,vec3b ) {

      returnvec3( ay * bz - by * az,
                        az * bx - bz * ax,
                        ax * by - by * ay );
}

floatdistance(vec3a,vec3b ) {

      vec3tmp = a – b;
      return sqrt(dot(tmp, tmp) );
}

floatdot(vec3a,vec3b ) {

      returnax * bx + ay * by + az * bz;
}

vec3faceforward(vec3n,vec3I,vec3Nref ) {

      if(dot( Nref, I ) <0) {

         return    n;
      }
      else
      {
         return    – n:
      }
}

floatlength(vec3v ) {

      return sqrt(dot(v, v) );
}

vec3normalize(vec3v ) {

      returnv /sqrt(dot(v, v) );
}

vec3reflect(vec3N,vec3I ) {

      returnI -2.0*dot(N, I) * N;
}

vec3refract(vec3n,vec3I,floateta ) {

      floatk =1.0-eta * eta * (1.0-dot(N, I) *dot(N, I)); if(k <0.0)

            return0.0;
      else
```

```
        returneta * I - (eta *dot(N, I) +sqrt(k)) * N;
}
```

12. Group operation.

Grouping scalars and vectors at once can improve efficiency:

```
fragColor.xyz = t.xyz * tx * ty * t.wzx * tz * tw;// 7 cycles {sop,
            sop, sopmov}
    {sop, sop, sopmov}
{sop,    sop}
{sop, sop, sopmov}
{sop,    sop}
{sop, sop, sopmov}
{sop,    sop}
- - >
fragColor.xyz = (tx * ty * tz * tw) * (t.xyz * t.wzx);// 4 cycles {sop,
        sop,   sopmov}
{sop,   sop,   sopmov}
{sop,   sop}
{sop,   sop}
```

The above assembly instructions use PowerVR GPU as an example. Others are similar but may not be exactly the same. They need to be optimized depending on the specific platform.

12.6.4 Comprehensive Optimization

12.6.4.1 Lighting and Shadow Optimization

Forward rendering is suitable for simple scenes with only a small number of dynamic light sources.

Traditional deferred rendering is suitable for scenes with many dynamic light sources (especially small-scale local light sources). However, it is limited by the bandwidth bit number of the buffer in the tile and cannot store too much geometric surface information. Compute Shader-based lighting technologies (such as tiled deferred, cluster deferred, and forward+) will write data to global memory, causing huge access cycles and delays, because the MRT data is likely to exceed the Tile buffer. It is not recommended for mobile terminals. There are many shadow techniques, but the shadow technique that best suits the TBR hardware architecture is**stencil shadowing** .Because TBR's GPU hardware is very good at processing stencil buffers, the data is stored in Tile memory and does not need to be written to system memory. If hard shadows are acceptable, the stencil shadow algorithm should be used first.

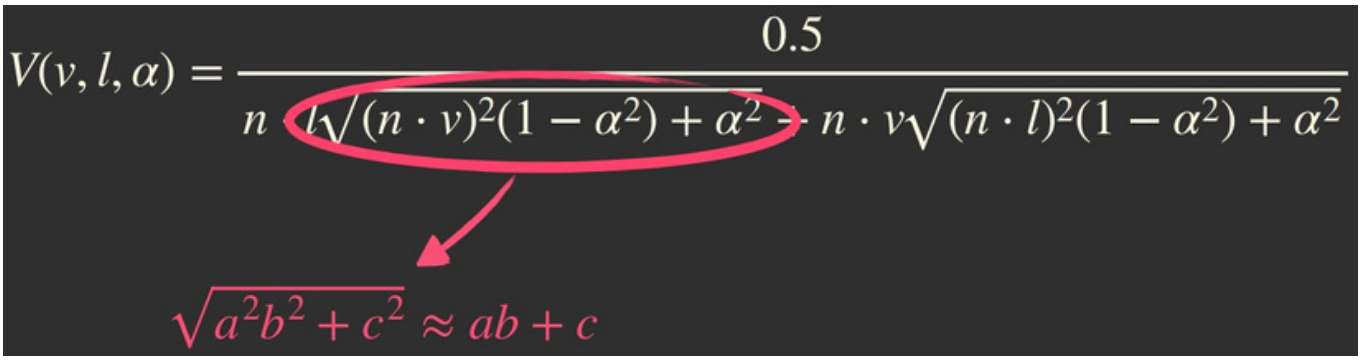
Techniques that require writing results to off-chip memory, such as shadow maps, generally perform worse than techniques that are computed entirely in tile memory.

If you want to use SSAO technology, in order to prevent frequent random high-span access to the depth buffer, it is best to use HZB (Hierarchical Z-Buffer) for acceleration.

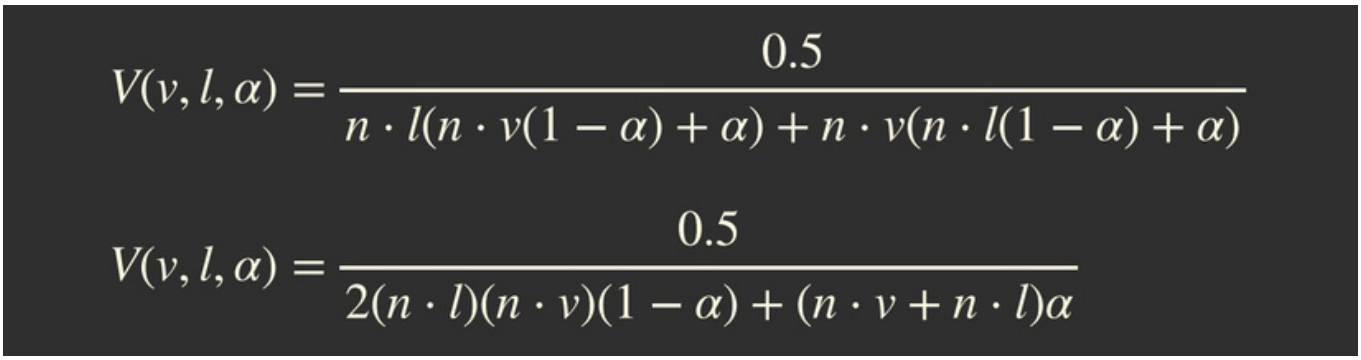
If you need to use SSR technology, downsample the scene color Frame Buffer in advance (using the OpenGL ES interfaceglFramebufferTexture2DDownsampleIMG).

Prioritize the use of template clipping lighting algorithm instead of traditional block and cluster lighting.

Special optimizations are made for mobile lighting. For example, Filament simplifies the visibility function of lighting:



The simplified visibility formula is as follows:

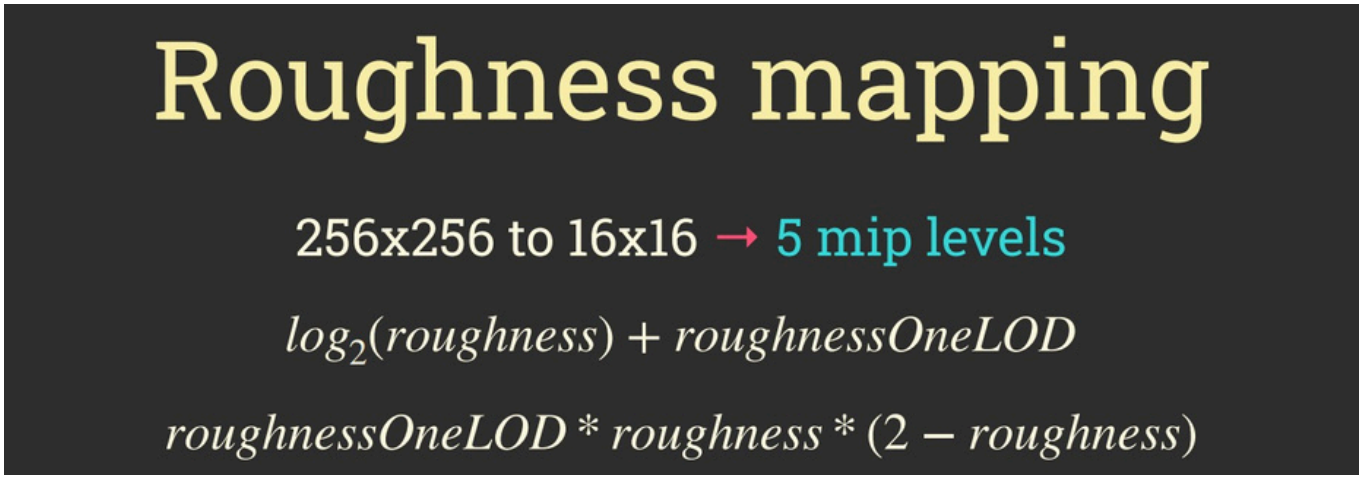


The corresponding implementation code:

```
floatV_SmithGGXCorrelated_Fast(floatroughness,floatNoV,floatNoL)

    // Hammon 2017, "PBR Diffuse Lighting for GGX+Smith Microsurfaces" return0.5/mix(2.0*
    NoL * NoV, NoL + NoV, roughness);
}
```

For the IBL lighting part, Filament abandoned the Diffuse Map and directly adopted the Specular Map (Mimap level when the roughness is 1) to simulate:

But the Specular Map has only 5 levels, and the smallest size is 16x16:



Mimap levels are mapped to roughness as follows:

Mipmap levels	Roughness
0	0.000
1	0.018
2	0.086
3	0.250
4	1.000

The IBL texture is stored in R11G11B10F format instead of RGBM (because the quality is not up to standard), which is reorganized into RGBA8888 and stored in PNG format.

For metal objects, in order to solve the energy non-conservation problem of traditional PBR lighting, Filament adopted the solution of Lagarde & Golubev:

Traditional PBR has the problem of non-conservation of energy when calculating metal materials.
The top row is the non-conservation illustration, the darker it is, the more energy is lost, and the bottom row is the illustration after Filment repairs it (you have to look very carefully to see it clearly).

Filament fixes the non-conservative BRDF formula for metal lighting.

The implemented code is as follows:

```
//TraditionalIBLCalculation code
constfloatV = Visibility(...) * NoL * (VoH / NoH); constfloatF = pow5(
1.0f - VoH); rx += V * (1.0f - F);

ry += V * F;

// FilamentFixed code
constfloatV = Visibility(...) * NoL * (VoH / NoH); constfloatF = pow5(
1.0f - VoH); rx += V * F;

ry += V;
```

In terms of AO, Filament simulates the multi-bounce effect:

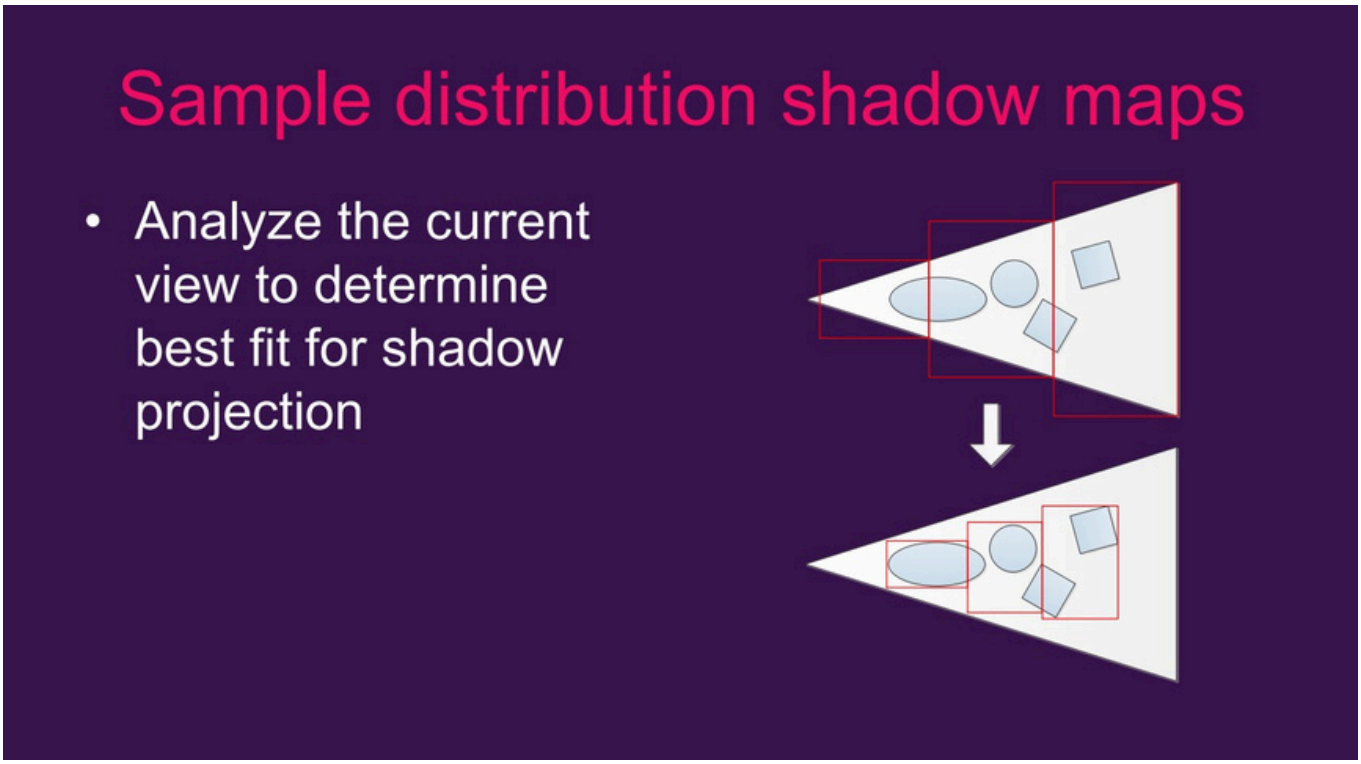
Comparison of Filament's multi-bounce AO effects. Top: Multi-bounce is off; Bottom: Multi-bounce is on. Note how the eyes and ears are slightly brighter.

The simulation code for AO multi-bounce is as follows:

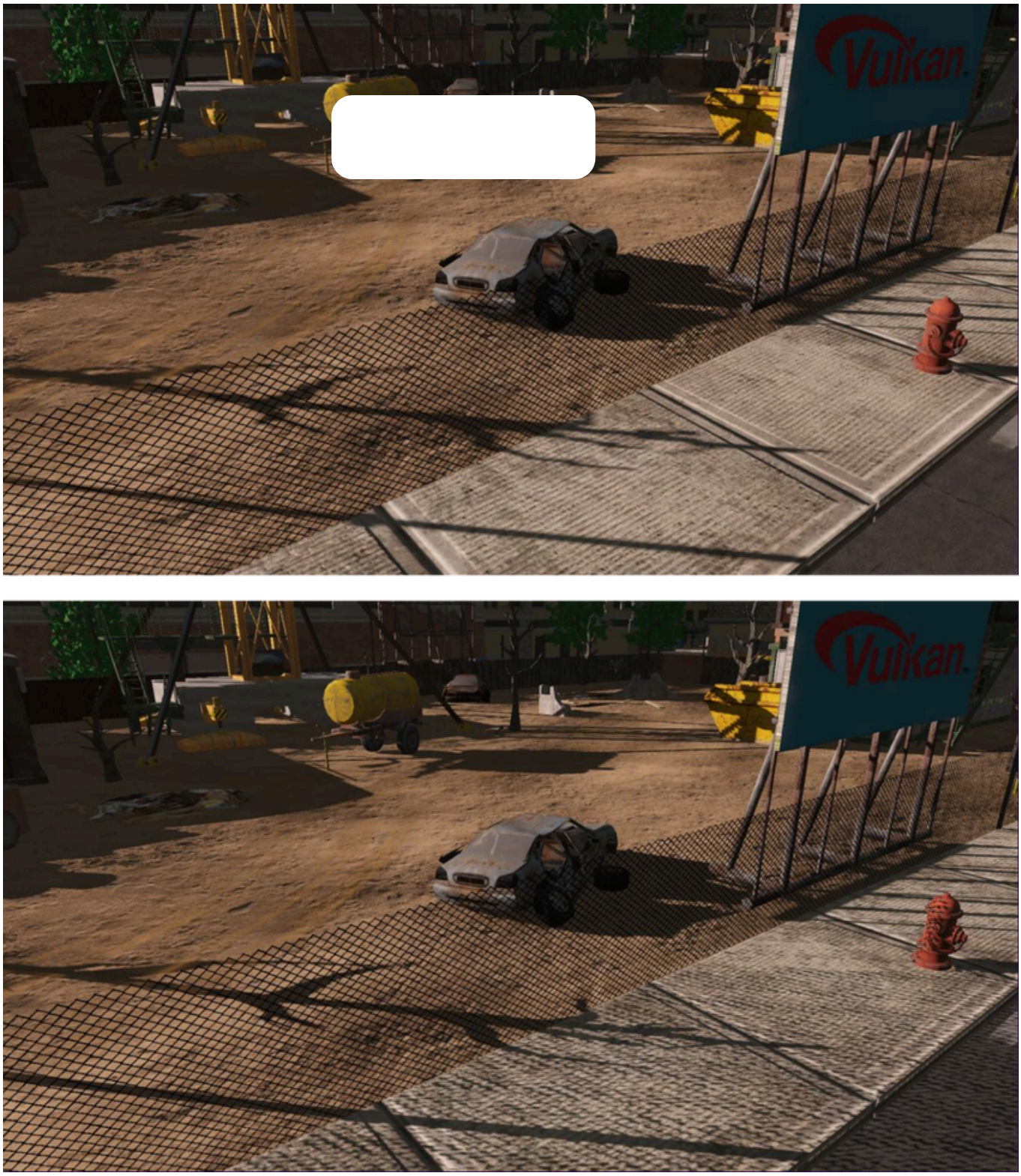
```
vec3gtaoMultiBounce(floatvisibility,constvec3albedo)

  // Jimenez et al. 2016,
  // "Practical Realtime Strategies for Accurate Indirect Occlusion" vec3a =2.0404*albedo -
  0.3324; vec3b =-4.7951*albedo +0.6417; vec3c =2.7552*albedo +0.6903;
```

```
  return max(vec3(visibility), ((visibility * a + b) * visibility + c) * visibility);
}

diffuseLobe *= gtaoMultiBounce(ao, diffuseColor);
```

There are also many optimization techniques for shadows. For example, the following figure shows the **Sample distribution shadow map (SDSM)**technique, which reduces the size of the shadow map by calculating the object bounding box instead of the view frustum bounding box:

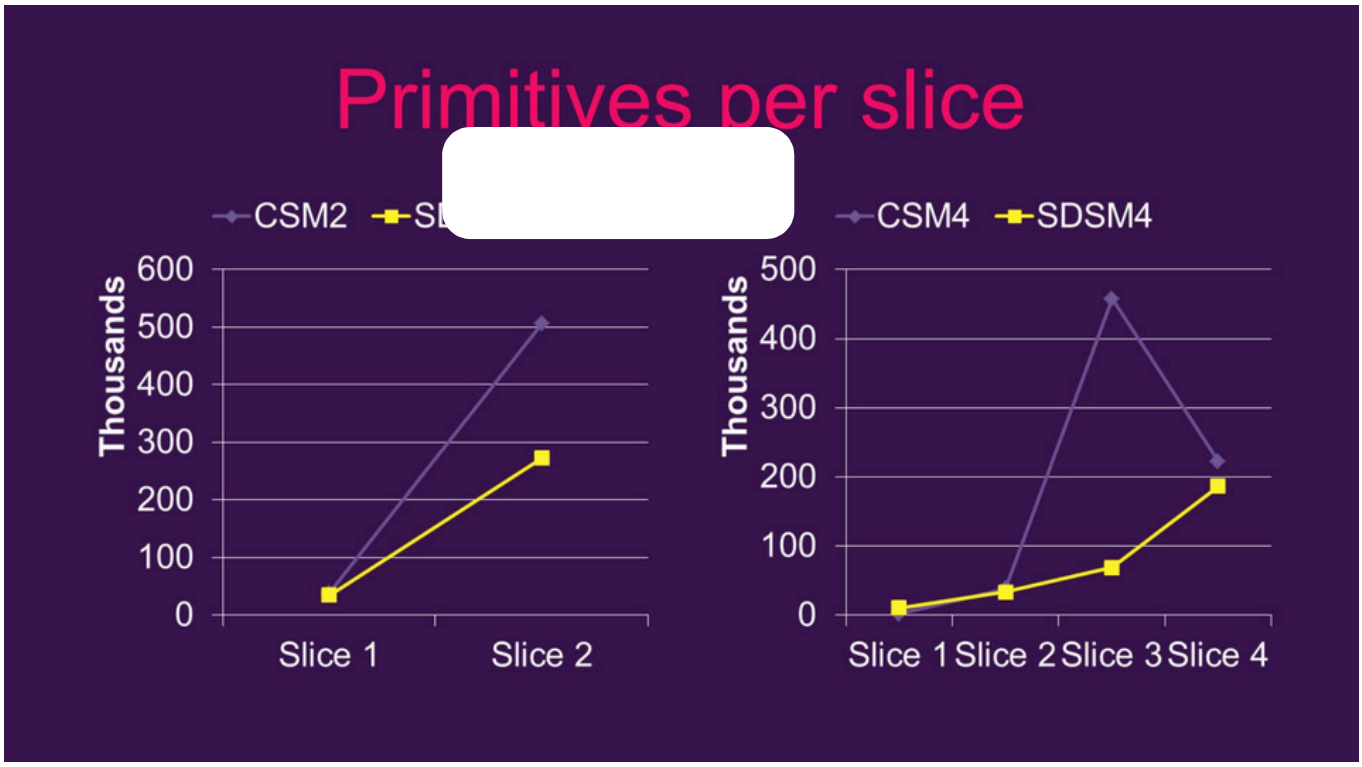


SDSM also avoids GPU lag by constructing HZB and using the HZB of the previous frame. It uses CS to generate the distance of the cascaded sub-shadow map. The generated HZB can be used to quickly clip the cascaded sub-shadow map. Through these optimization measures, SDSM can balance the number of primitives in each cascade, balance the shadow map resolution and output
resolution, and obtain similar shadow effects to non-SDSM methods with a smaller resolution. The following is a comparison of the effects of SDSM and non-SDSM:



Top: normal CSM shadow; Bottom: SDSM shadow.

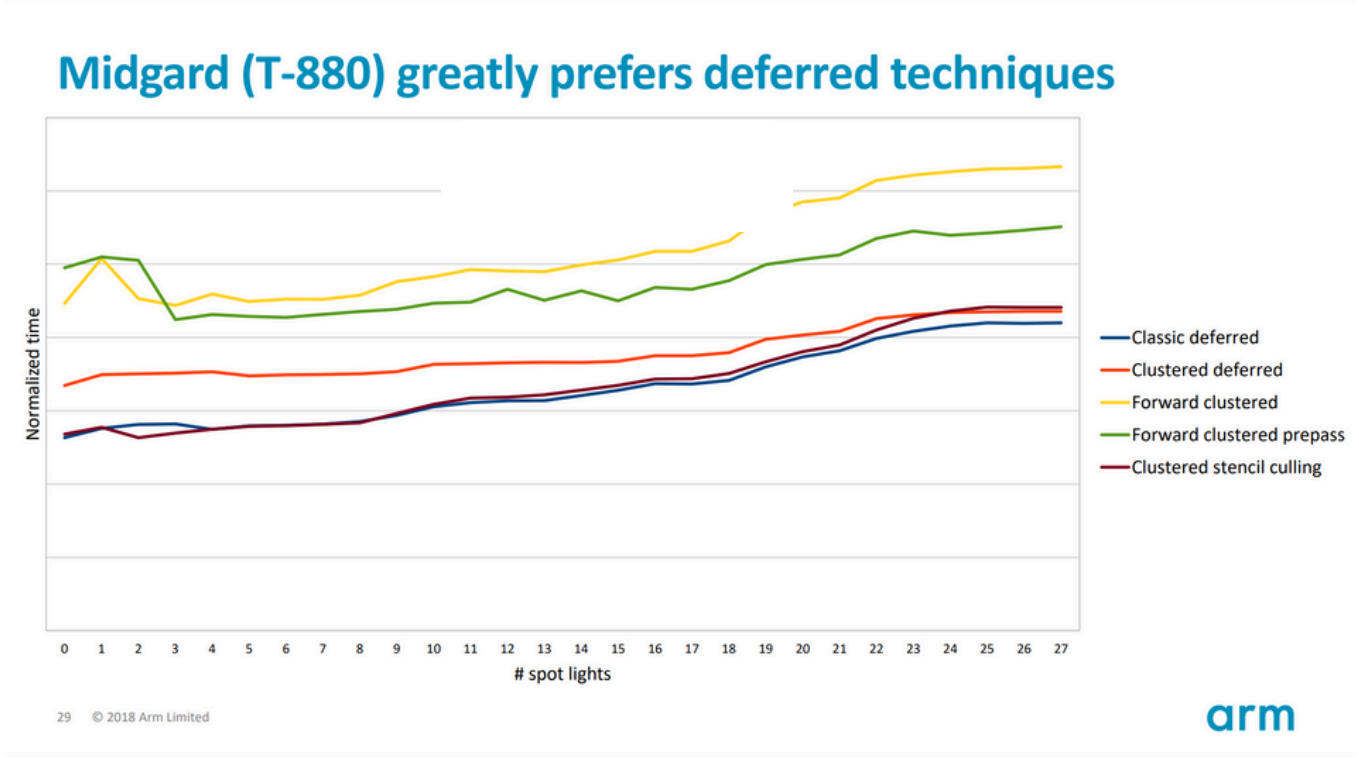
In terms of performance, SDSM also outperforms:

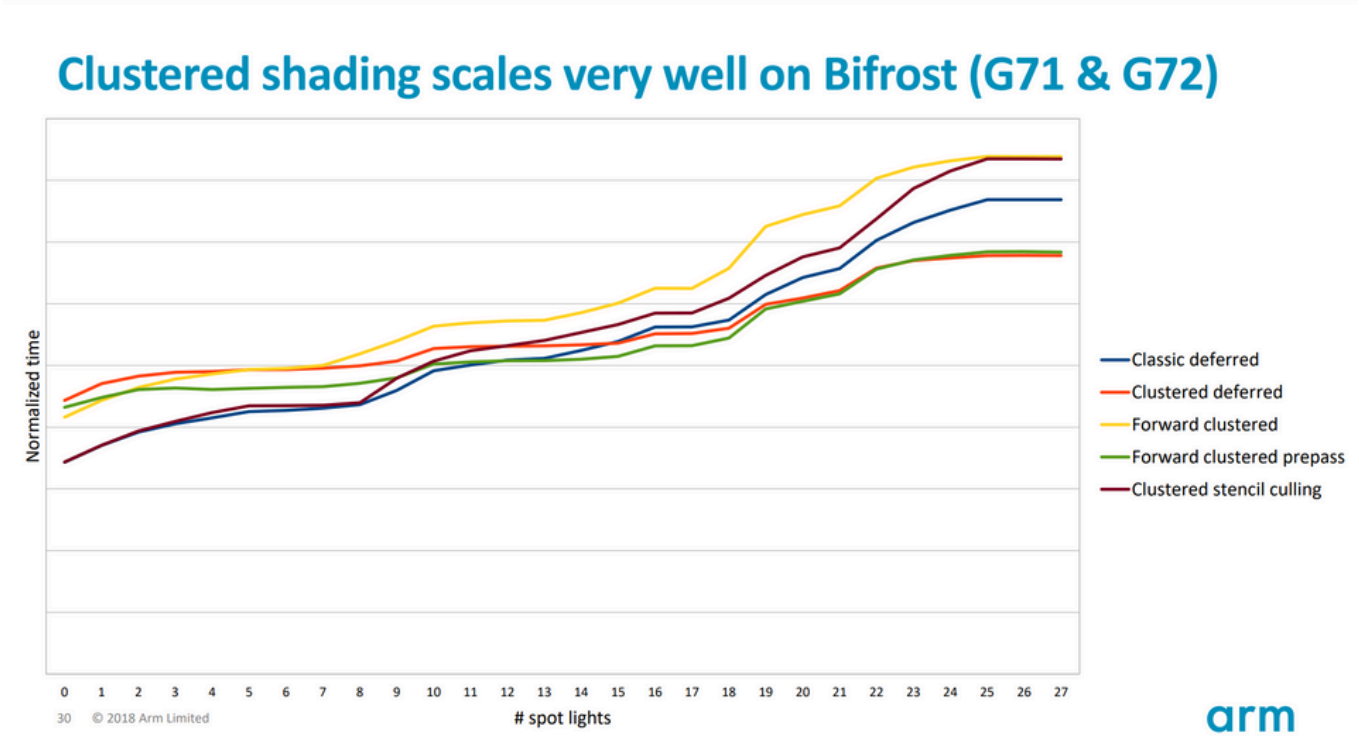


If you are on a low-end device, you can try using blob shadows instead of shadow maps:

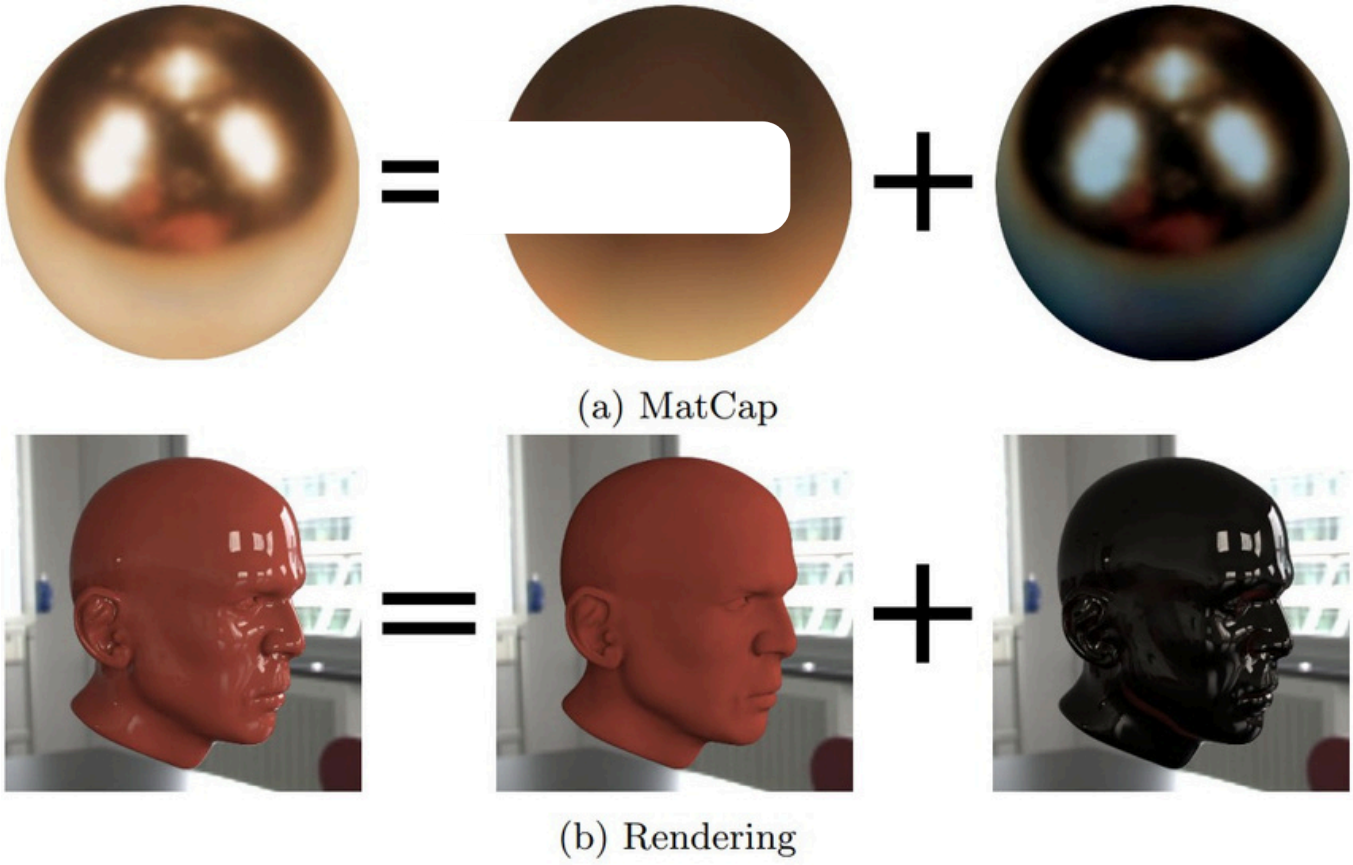
Left: Shadow map; Right: Blob shadow.

For advanced lighting, you can try lighting rendering technologies such as Forward+ and Light Prepass. The following is a comparison chart of various lighting technologies on mobile GPUs:





In addition, you can try to use MatCap technology to achieve IBL effects, which can achieve a good balance between performance and effect:



Rendering effect achieved using MatCap technology.

12.6.4.2 Post-processing optimization

Post-processing effects will take up more bandwidth, so try to turn off all post-processing effects unless necessary.

If post-processing is indeed necessary, common optimization methods are as follows:

1. Combine multiple post-processing effects into one Shader.

2. Reduce the resolution and calculate the post-processing effect.

3. Try to keep post-processing data access within the Tile.

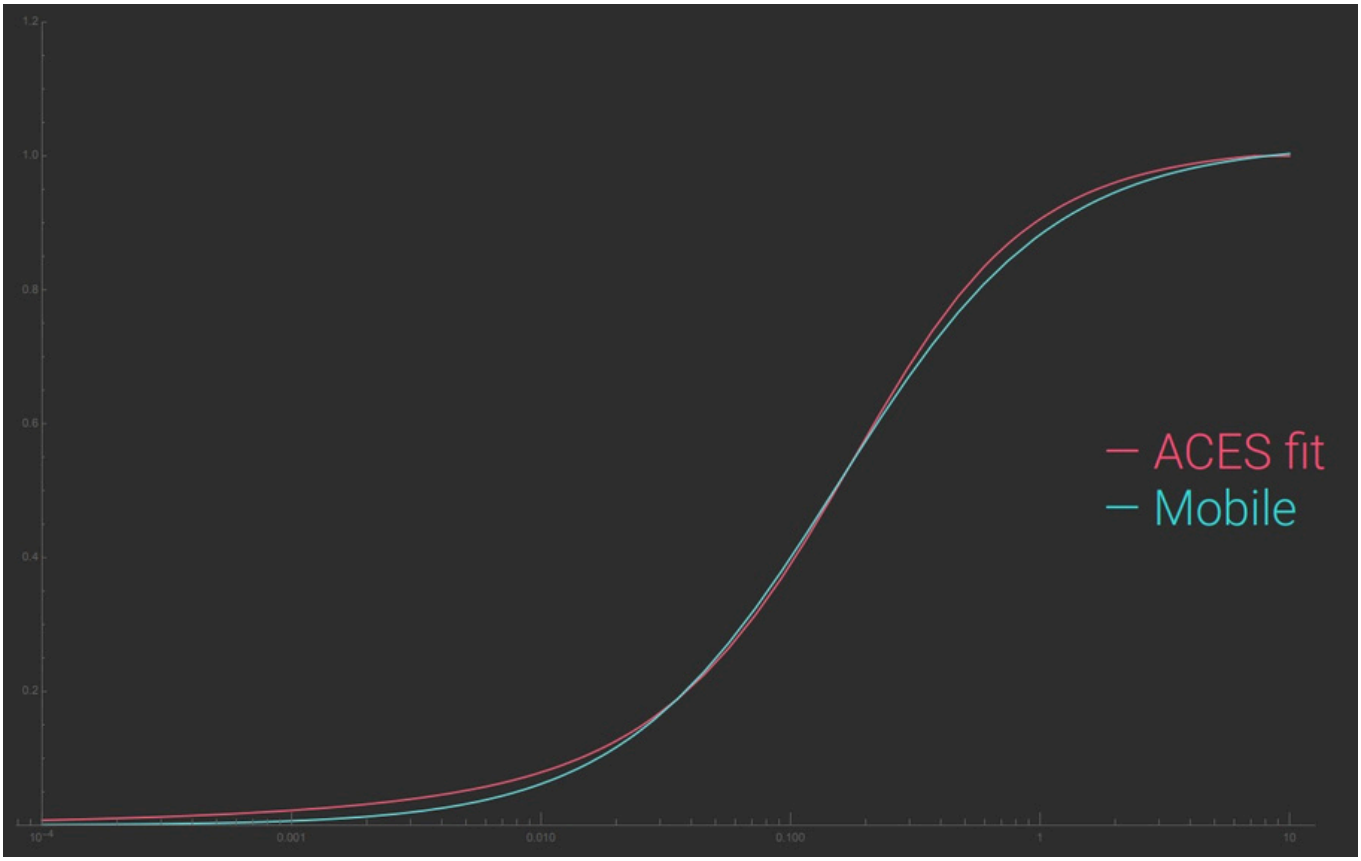
4. Try not to access surrounding pixel data. If necessary, try to maintain locality and timeliness to improve cache hit rate.

5. Dedicated algorithm optimization. For example, splitting Gaussian blur into horizontal blur + vertical blur (separate convolution kernel). Filament has optimized tone mapping for mobile terminals:

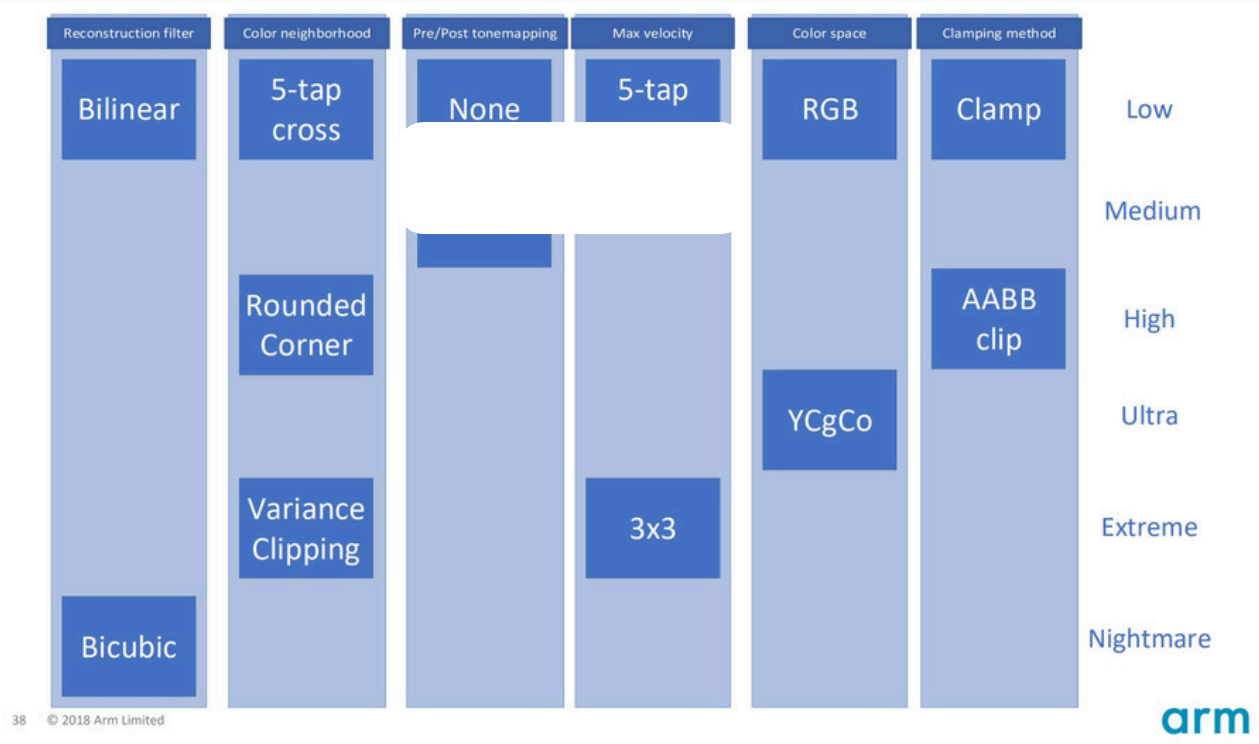
```
//OriginalACESTone mapping.
vec3Tonemap_ACESconstvec3x) {

    // Narkowicz 2015, "ACES Filmic Tone Mapping Curve" constfloata =
    2.51; constfloatb =0.03; constfloatc =2.43; constfloatd =0.59; const
    floate =0.14;
```

```
    return(x * (a * x + b)) / (x * (c * x + d) + e);
}

//Mobile version of tone mapping
vec3Tonemap_Mobileconstvec3x) {

    // Transfer function baked in, // don't use
    with sRGB OETF! returnx / (x +0.155) *
    1.019;
}
```

The simplified tone mapping curve is very close:



Arm provides technical references for commonly used post-processing effects at different quality levels:



12.6.4.3 Sprite Rendering Optimization

Common methods for optimizing sprite rendering include: controlling the number of sprites, controlling the area of the sprites on the screen, and reducing blank areas.

Modern GPUs are not so sensitive to the increase in the number of primitives, but are more sensitive to the increase in the blank area of Sprites with Alpha Blend turned on, because a lot of invalid fragment shading processing will be wasted. An effective way to avoid the waste of blank area is to increase the geometric complexity of drawing sprites. By increasing the geometric complexity to reduce the waste of transparency, performance can be significantly improved.

In the past, when using sprites to simulate particle effects, a quadrilateral was used to draw a circle. At this time, the area around the quadrilateral was blank, and the waste ratio reached an astonishing 22%. If the 4-sided polygon is increased to 12-polysidedgons, the number of wasted fragments processing can be reduced to 3%. The formulas and results are compared as follows:

Assuming radius r of 64

$$A = 1 - \frac{\pi r^2}{(2r)^2}$$

$$A = 0.214$$

$$A = 21.4\%$$

vs

$$A = 1 - \frac{\pi r^2}{12(2-\sqrt{3})r^2}$$

$$A = 0.029$$

$$A = 2.9\%$$

Comparison of the proportion of wasted fragment processing between 4-gon and 12-gon. The left figure is a 4-gon with a wasted ratio of 21.4%; the right figure is a 12-gon with a wasted ratio of 2.9%.

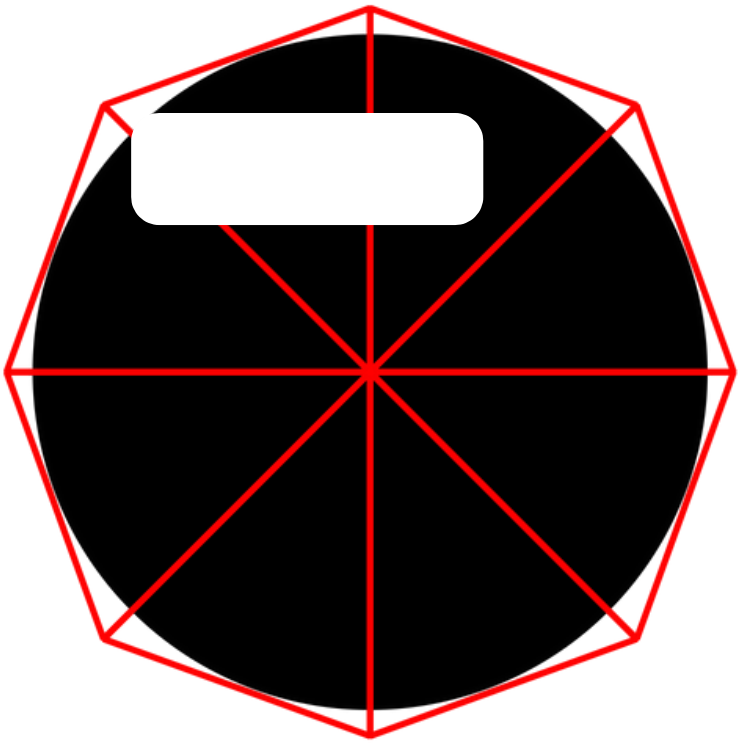


Illustration of using an octagon to draw a circular semi-transparent object.

In addition, you can split opaque and semi-transparent objects (such as UI elements) into separate drawing submissions. The drawing submission order is recommended as follows:

1. Opaque scene sprite elements.

2. Translucent scene sprite elements.

3. Translucent UI elements.

For particle effects, the sharpness of distant particles is not important and a simpler texture filtering method (such as nearest point) can be used.

12.6.4.4 Balancing GPU Workload

For GPU-intensive applications, bottlenecks often occur on the GPU side, and the reason for this is that the workload of various GPU components is unbalanced, resulting in performance bottlenecks.

By properly allocating the workload of each component of the GPU, bottlenecks can be effectively eliminated, the GPU can be fully utilized, and rendering performance can be improved. The following are GPU resources that can be used to distinguish workloads:

1. ALU (Logical Arithmetic Unit)

2. Texturing Load

3. ISP Load (Image Integrated Processor Load)

4. Renderer Active

5. Tiler Active

Through the Profiler provided by GPU manufacturers (such as Snapdragon Profiler, PVRMonitor), their dynamics can be effectively monitored. The following is an optimization description of balanced workload:

1. Using pre-calculation and storing the results in a lookup table (LUT), the work of the ALU can be shifted to Texturing Load.
2. Use procedural texture functions instead of texture acquisition to transfer the work of Texturing Load to the ALU.
3. Use depth and stencil testing to reduce shader calls, thereby reducing texture load or ALU workload.
4. ALU-based Alpha test can be used to exchange depth prepass and depth test, which will increase draw call and geometry overhead, but can significantly reduce ALU workload and register pressure.
5. Alpha test and noise function are used together to achieve LOD transition effect, which will greatly increase the ALU workload. In this case, template prepass can be performed to transfer the ALU workload to ISP.
6. High-fidelity image quality can be improved by providing more complex shaders, higher resolution textures, and increasing the number of polygons. When rendering reaches a bottleneck, it is better to increase the number of polygons rather than increasing the complexity of the fragment shader.

12.6.4.5 Compute Shader Optimization

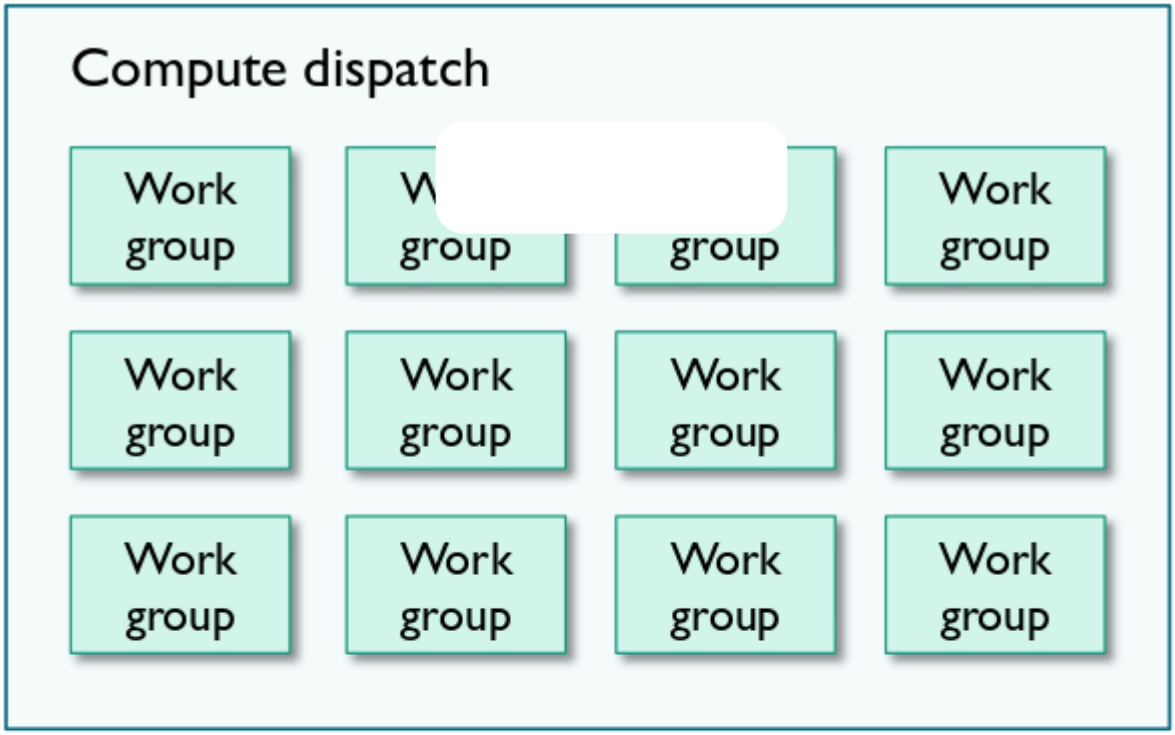
Before**Compute Shaders (CS)**,there were multiple ways to expose embarrassingly parallel computation in OpenGL ES:

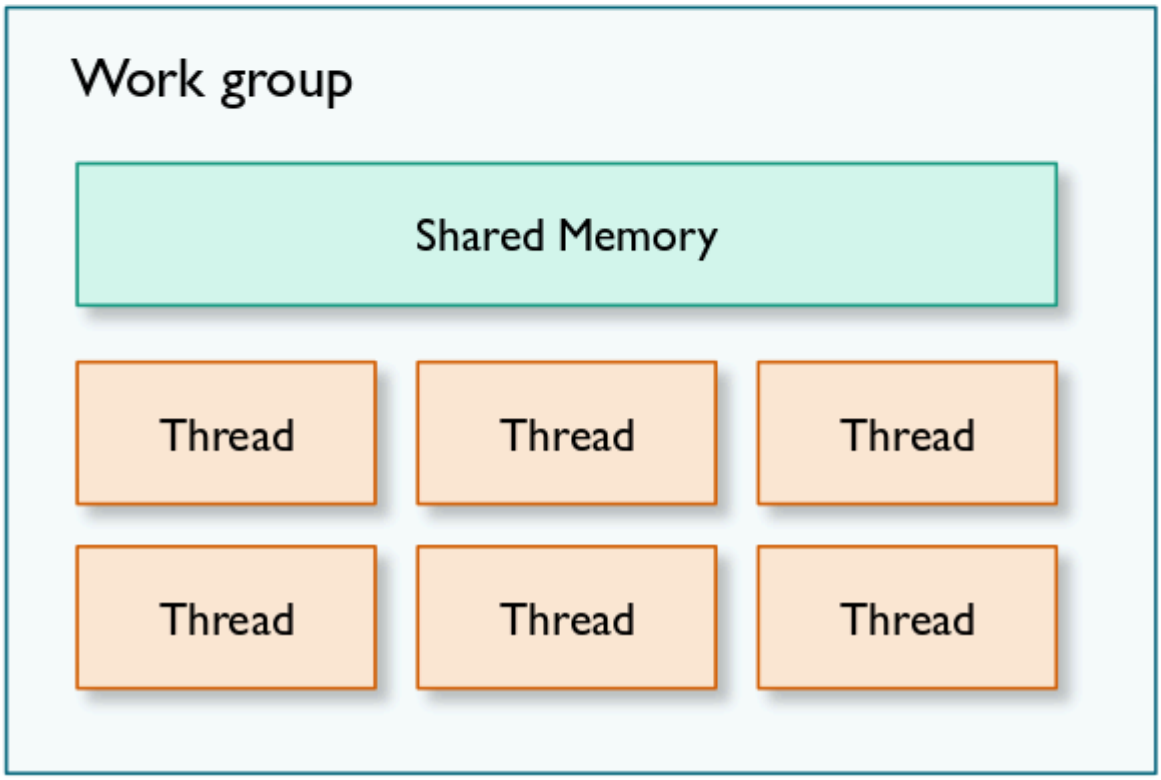
1. Rasterize a quad and perform any calculations in the pixel shader, then write the result to a texture.
2. Use transform feedback to perform arbitrary calculations in the vertex shader.

These methods have many limitations, such as shaders cannot perceive other shaders, and the target of writing data is restricted (VS can only write to gl_Position and variable registers, PS can only write to the specified RenderTarget).

Compute Shader does not have the above limitations. It can specify any input and output data sources and does not need to run the traditional rendering pipeline. It can run custom calculations conveniently, efficiently and flexibly.

When each Compute Shader dispatches a task, you can specify the number of Work Groups and the number of threads in each Work Group.





Top: Schematic diagram of the Work Group that dispatches the Compute Shader each time; Bottom: Each Work Group has several threads, and these threads have a Shared Memory.

The pseudo code for Compute Shader operation is as follows:

```
for(intw =0; w < NUM_WORK_GROUPS; w++) {

    //Ensure parallel operation.
    parallel_for (inti =0; i < THREADS_IN_WORK_GROUP; i++) {

        execute_compute_thread(w, i);
```

```
    }
}
```

The following are recommended for workgroup size:

1. Use 64 as the baseline size of the work group. Do not use more than 64 threads per work group.

2. Use a multiple of 4 as the size of the work group.

3. Try smaller workgroup sizes before larger ones, especially when using barriers or shared memory.

4. When processing images or textures, use square execution dimensions (eg 8x8) to exploit optimal 2D cache locality.

5. If a work group is to complete the work of each work group, consider splitting the work into two channels. Doing so can avoid barriers and idle gaps for most threads in the kernel. Barriers for small work groups also have performance costs.

6. Compute Shader performance is not always intuitive, so continue to measure performance.

For Shared memory, the following recommendations are made:

1. Use shared memory to share important or complex calculations between threads in a workgroup.

2. Keep shared memory as small as possible, as this reduces drastic changes in data cache.

3. Reduce the precision and data width to reduce the size of the shared memory required.

4. Barriers need to be set to synchronize access to shared data. Shader code ported from desktop development sometimes ignores some barriers due to GPU-specific assumptions. However, this assumption is not safe to use on mobile GPUs.

5. Compared with inserting barriers, splitting the algorithm into multiple shaders is more efficient.

6. Smaller work groups have lower costs for obstacles.

7. Do not copy data from global memory to shared memory, as this will reduce the cache hit rate.

8. Do not use shared memory to implement code. For example:

```
if(localInvocationID ==0) {

    common_setup();
}

barrier();
(.....)//          Thread by threadshaderlogic
barrier();

if(localInvocationID ==0) {

    result_reduction();
}
```

In the above code,common_setuponlyresult_reductionone thread is needed, and other threads in the working group will be waiting, resulting in Stall and Idle.

Splitting the above code into three shaders is better becausecommon_setupitresult_reduction requires fewer threads.

> However, the embarrassing thing is that this merged code is widely used in UE's CS code, such as TAA, SSGI and so on.

The following are suggestions for processing Image (or Texture):

1. When using variational interpolation, texture coordinates will be interpolated using fixed function hardware. This, in turn, frees up shader cycles for more useful workloads.
2. Writing to memory can be done in parallel with shader code using Tile-Writeback hardware.
3. No range checking is needed for imageStore() coordinates. This may cause problems when using workgroups that do not fully subdivide a frame.
4. Frame buffer compression and transaction elimination (Mali GPU only) can be performed.

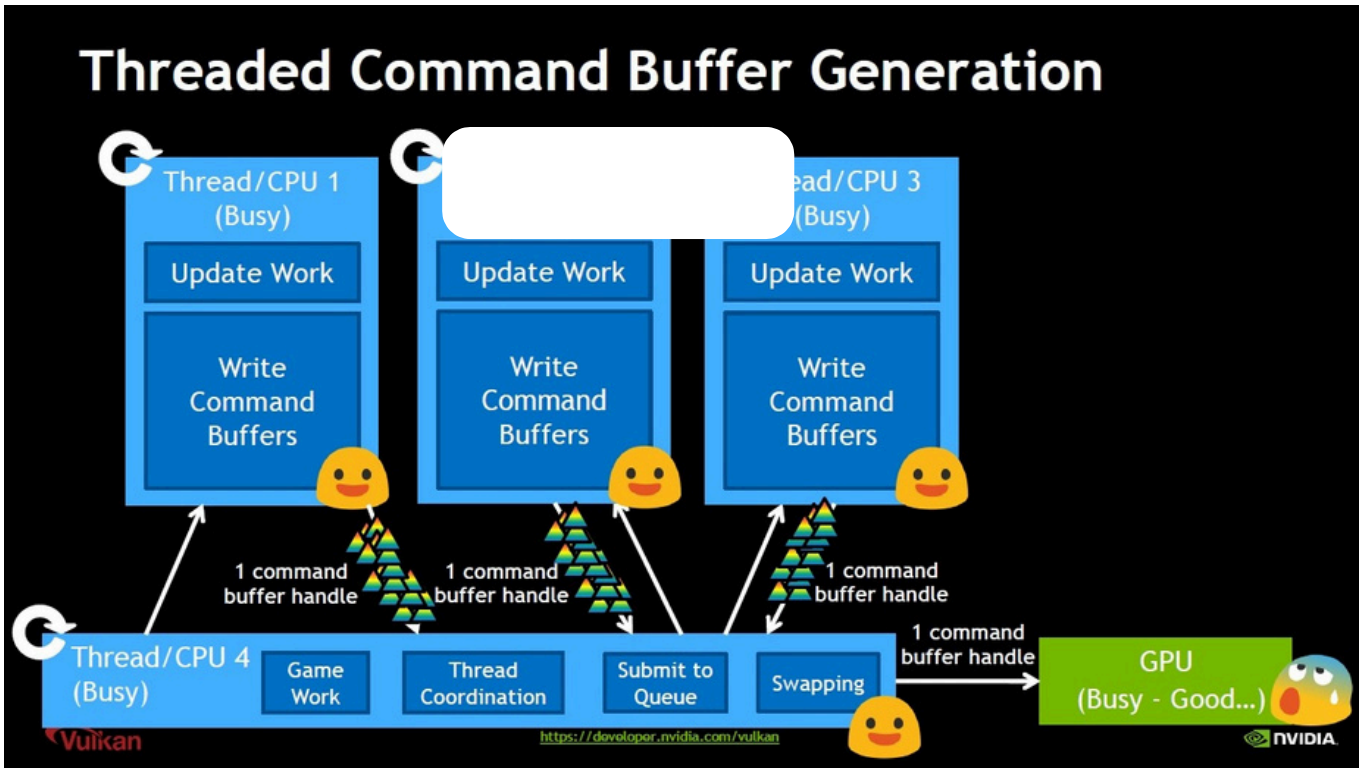
Here are some advantages of using compute for image processing:

1. Shared data sets between adjacent pixels can be utilized to avoid additional transmission of some algorithms.
2. It is easier to use a larger working set in each thread, thus avoiding extra transfers for some algorithms.
3. For complex algorithms such as FFT (Fast Fourier Transform) that require multiple fragments rendering channels, they can usually be merged into a single calculation dispatch.

12.6.4.6 Multi-core Parallelism

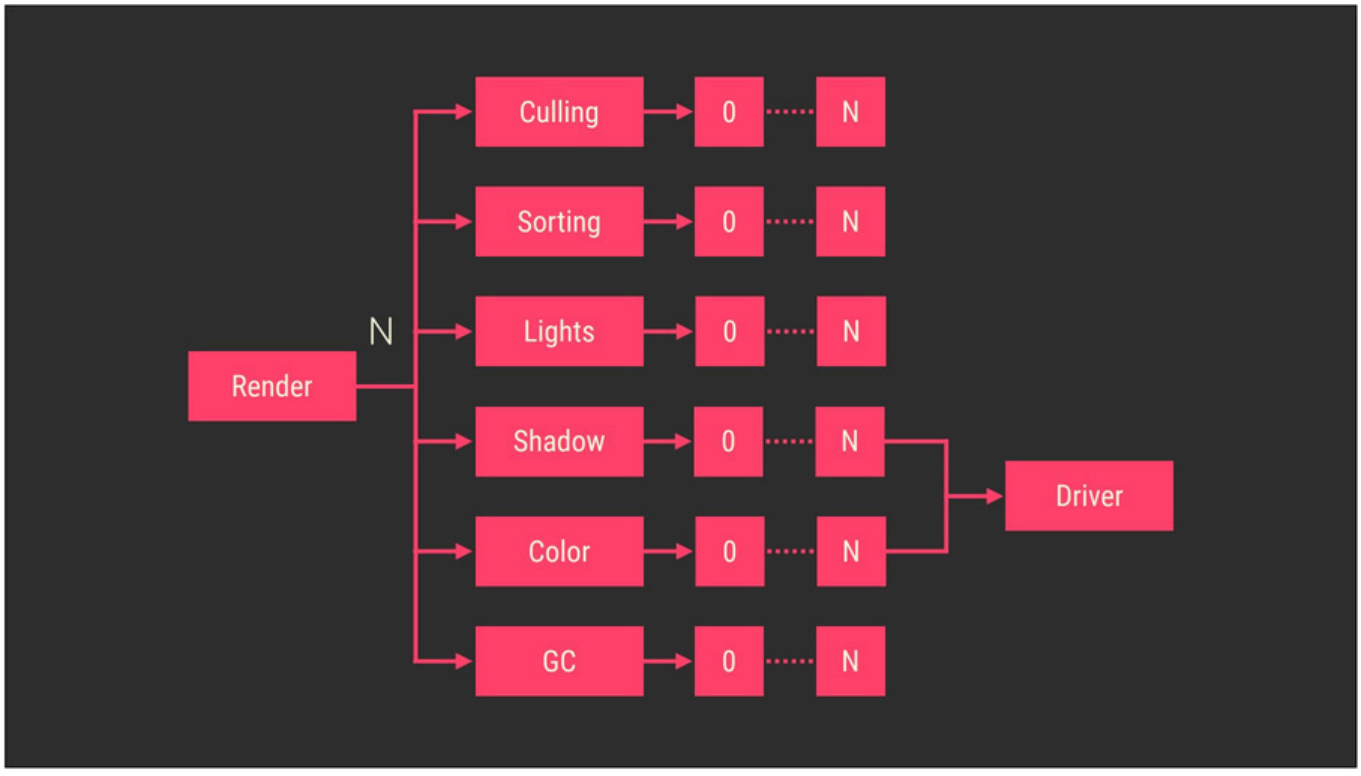
Multi-core is already a standard feature of mainstream CPUs in current mobile SoCs (most CPUs have 8 cores or more). How to use the parallel capabilities of multiple cores to improve rendering effects is a very large and challenging task.

First, make full use of the features of modern graphics APIs (DirectX12, Vulkan, Metal) that allow multi-core creation and execution of Command Buffer to improve parallel efficiency:



A diagram showing how the Vulkan graphics API generates command buffers in parallel.

The parallel rendering diagram of Filament's operating system is as follows:



Simplified diagram of the filamente job system. Each block represents a new parent job, which itself can spawn N jobs. Every loop in this system is multithreaded and jobified.

UE uses TaskGraph for parallel rendering. For more technologies in this area, please refer to:Analysis of Unreal Rendering System (02) - Multi-threaded Rendering .

12.6.4.7 Other comprehensive optimizations

- **System integration optimization**

Most mobile platforms use a vertical sync signal to prevent screen tearing when swapping buffers. If the GPU renders slower than the vertical sync period, a swap chain containing only two buffers can easily cause the GPU to stutter. Recommendations for optimizing the swap chain:

1. If your application**always**runs**slower**than vsync , then don't use two surfaces in the swap chain.

1. If the application**always**runs**faster**,then using two surfaces in the swap chain can reduce memory consumption.
2. If the application**sometimes**runs**slower**than vsync, then using three surfaces in the swap chain can provide the best performance for the application.

- **Using MRT efficiently**

MRT (Multiple Render Target)is widely supported on mobile devices. A common use case is deferred rendering. MRT is needed to store surface geometry information (base color, normal, depth, material) in the geometry pass stage. TBDR can utilize tile buffers (such as PLS and Subpass) to keep MRT data in cache, thereby improving
memory data access speed and reducing latency.
In order to work well on most mobile GPUs, the MRT per-pixel data size is controlled within 128 bits (16 bytes) + a depth template buffer. On some newer GPUs, it can be increased to 256 bits (32 bytes) + a depth template buffer. If it exceeds, the buffer in the tile is insufficient, and the GPU will force the data to be saved in the global memory, greatly reducing the data operation speed.
In addition to memory transactions and performance considerations, when the render target spills out of system memory, not all render target formats are supported at full rate on the system memory bus. Therefore, depending on the formats and texture processing units (TPUs) available in the GPU, the transfer rate may be further reduced. For PowerVR GPUs , the following formats and rates are related as follows: 1.RGBA8Can read at full speed. 2.RGB10A2Can read at nearly full speed. 3.RG11B10Can only read at half speed. 4.RGBA16FCan only read at half speed. 5.RGBA32FOnly 1/4 full speed reading (no bilinear filtering).

- **Choosing the right HDR pixel format**

For HDR, there are several formats to choose from, taking into account factors such as memory bandwidth, precision (quality), alpha support, etc. For HDR texture formats that are natively supported by the hardware, RGB10A2 or RGBA16F can be used, but the bandwidth will increase. These textures provide a good balance between quality, performance (filtering) , and memory bandwidth usage.

RGBMBoth the RGBA8 andRGBdiv8HDR Color texture formats require developers to implement encoding and decoding functions in the shader, requiring additional USC cycles because they are not supported by the hardware. If the application is USC-limited, these formats should not be used. Their advantage is that the memory bandwidth is very low, with the same bandwidth cost as RGBA8. If the application is limited by memory bandwidth, it may be useful to study these formats.RGBMThe encoding and decoding code between HDR Color and HDR Color is as follows:

```
//WillHDRColor codingRGBM.
float4 RGBMEncode( float3 color ) {

	float4 rgbm;
	color *=1.0/6.0;
	//WillHDRThe color coefficients are encoded intoAlphaIn the channel.
	rgbm.a = saturate(max(max( color.r, color.g ),max( color.b,1e-6) ) ); rgbm.a =ceil( rgbm.a *255.0) /
	255.0; rgbm.rgb = color / rgbm.a;


	returnrgbm;
}

//decodingRGBMbecomeHDRColor.
float3 RGBMDecode( float4 rgbm ) {

	return6.0* rgbm.rgb * rgbm.a;
}
```

Common HDR formats are described in detail in the following table:

Texture Format	Bandwidth consumption	USC consumption	filter	Accuracy	Alpha
RGB10A2	1x RGBA8	none	Hardware EP Slightly slower than RGBA8	Higher precision for RGB channels, at the expense of alpha precision	Only four values
RGBA16F	2x RGBA8	none	Hardware accelerated, 0.5x RGBA8	Much higher precision than RGBA8	support
RG11B10F	2x RGBA8	none	Hardware accelerated, 0.5x RGBA8	Equivalent to RGBA16F	Not supported

Texture Format	Bandwidth consumption	USC consumption	filter	Accuracy	Alpha
RGBA32F	4x RGBA8	none	Hardware EP 0.25x RGBA8, only supports the nearest point	Much higher precision than RGBA16F	support
RGBM (RGBA8)	1x RGBA8	Encoding/decoding data	The hardware does not support this format filtering	RGB value range is higher than RGBA8	Not supported
RGBdiv8 (RGBA8)	1x RGBA8	Slightly more complex than RGBM	The hardware does not support this format filtering	Same as above	Not supported

Packed 32-bit formats (RGB10_A2, RGB9_E5) may be preferred over FP16 or FP32.

- **Choose the right anti-aliasing**

MSAA is suitable for forward rendering, and TAA is suitable for deferred rendering. In addition, there are many morphological analysis anti-aliasing technologies, and the efficiency is from high to low: FXAA, CMAA, MLAA, SMAA.

Therefore, choose appropriate anti-aliasing according to the project situation, or choose different anti-aliasing technologies according to high, medium and low image quality.
When using MSAA, use 4x first to achieve a better balance between effect and efficiency. Use MSAA resolution within the tile and avoid using explicit resolution using interfaces such as glBlitFramebuffer().

Monitor, analyze, and compare performance with and without anti-aliasing.

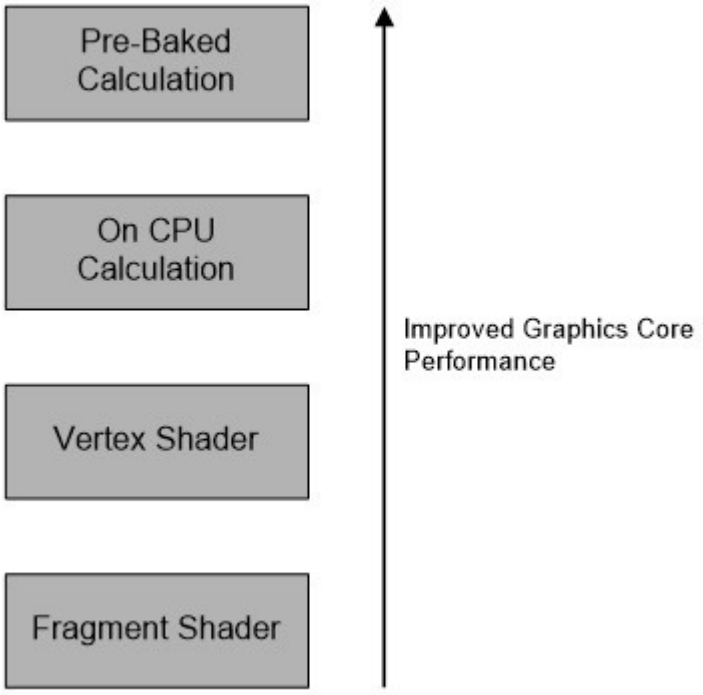
Filament implements a special anti-aliasing filtering algorithm for the highlights (modifies the roughness):

```
floatnormalFiltering(floatperceptualRoughness,constvec3worldNormal) {

  // Kaplanyan 2016, "Stable specular highlights"
  // Tokuyoshi 2017, "Error Reduction and Simplification for Shading Anti-Aliasing"
```

```
    // Tokuyoshi and Kaplanyan 2019, "Improved Geometric Specular Antialiasing" vec3du =f
    (worldNormal); vec3dv =dF(worldNormal);

    floatvariance = specularAntiAliasingVariance * (dot(du, du) +dot(dv, dv)); floatroughness =
    perceptualRoughnessToRoughness(perceptualRoughness); floatkernelRoughness =min(2.0* variance,
    specularAntiAliasingThreshold); floatsquareRoughness = saturate(roughness * roughness +
    kernelRoughness); returnroughnessToPerceptualRoughness(sqrt(squareRoughness));

}
materialRoughness = normalFiltering(materialRoughness, getWorldGeometricNormalVector());
```

- **Move calculations forward as much as possible**

By moving them to earlier in the pipeline, when fewer instances need to be processed, the total number of calculations can be reduced. The calculation chain is as follows:



The efficiency from high to low is: pre-calculation, CPU application layer calculation, vertex shader, pixel shader.

Taking lighting calculation as an example, the rendering efficiency is from high to low: light map, IBL, vertex-by-vertex lighting, and pixel-by-pixel lighting.

But high efficiency means poor controllability, and a balance must be struck between efficiency and effectiveness.

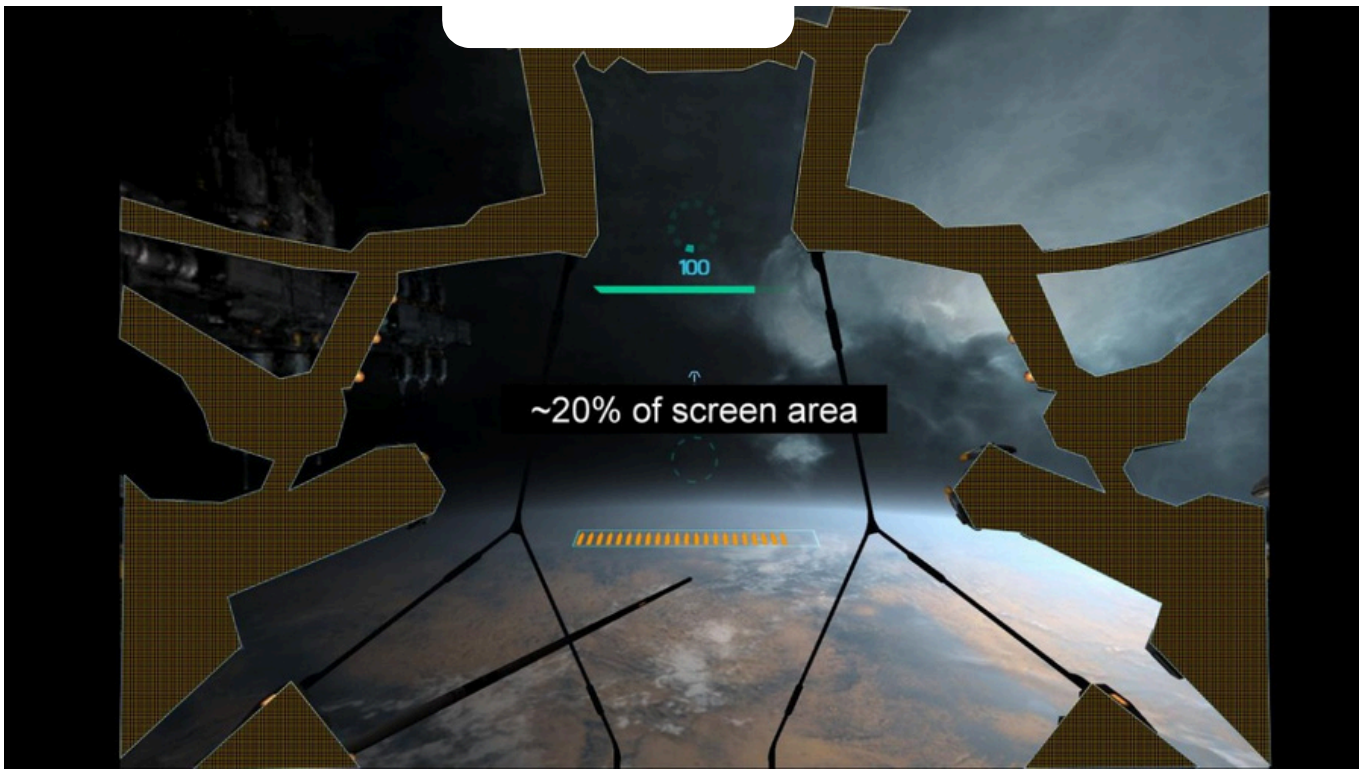
- **Miscellaneous optimizations**

Pay attention to power consumption and device temperature to prevent CPU or GPU throttling from causing performance degradation.

Consider reducing resolution or frame rate, or dynamically adjusting based on some strategy.

Load data with high IO load in advance and cache it. Precompute tasks with high IO load as much as possible.

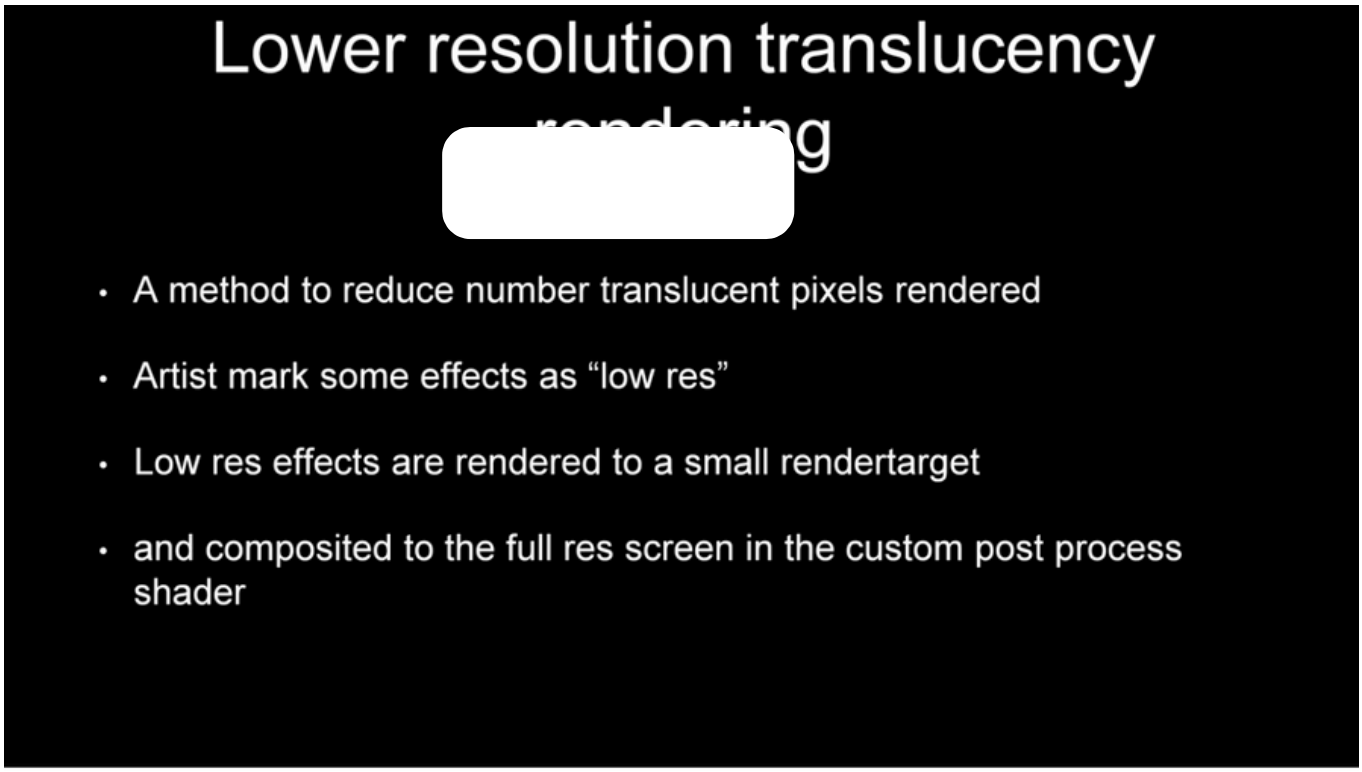
Hide objects behind the UI interface. (below)

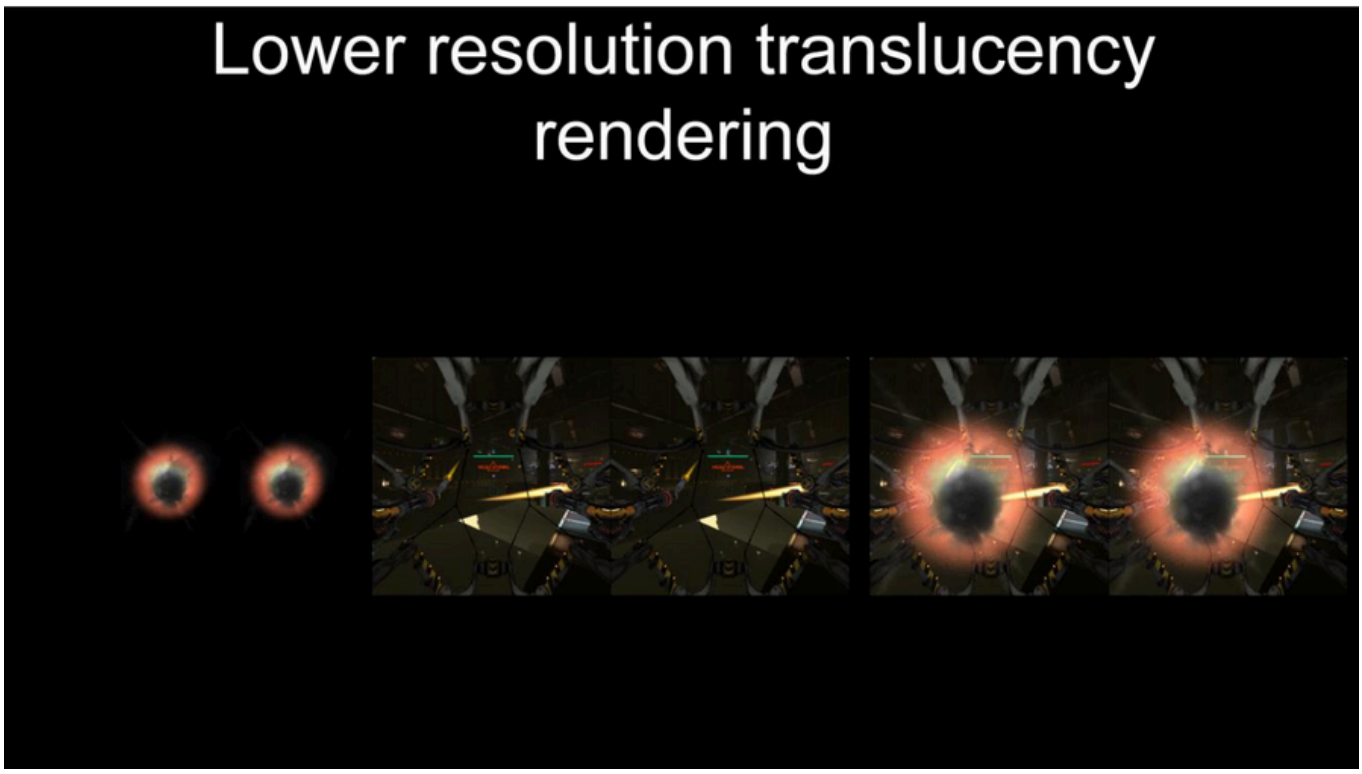


Divide quality grades, formulate parameter specifications, and select technologies with different consumption according to grades.

Consider dynamic grid batching (UE does not have this function and needs to be implemented by yourself).

Rendering semi-transparent objects at a reduced resolution, then upscaling them to blend into the scene color. (below)





In addition, you also need to pay attention to consumption and optimization in multi-thread synchronization, occlusion culling queries, barriers, rendering channels, creating GPU resources and uploading, static static, memory usage and leakage, video memory usage, VS and PS performance ratio, etc.

Here are some of the optimizations that Fornite made on mobile duringA Year in a Fortnite:

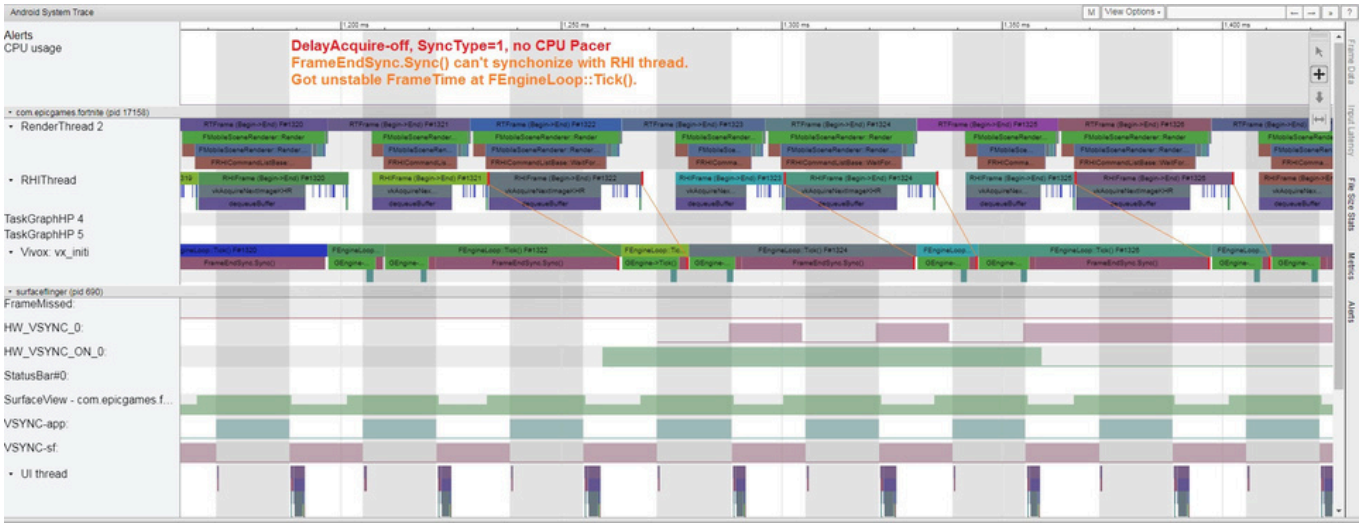
FPSChart **Base (Note9, Mali, Siop00, TypeB case)**

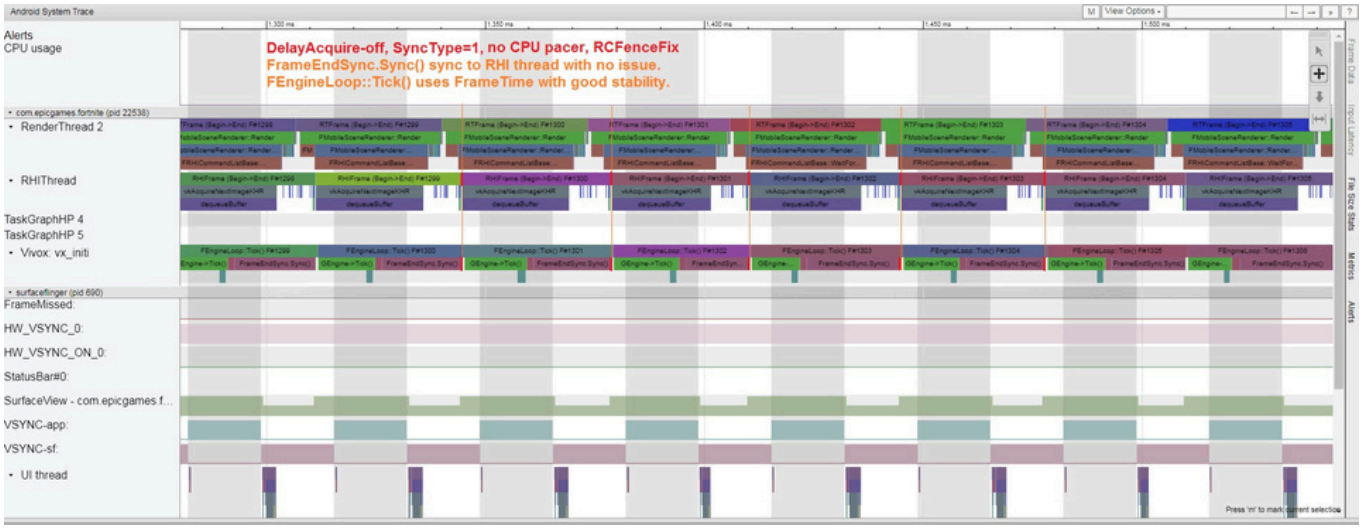
Section Name	Num Frames	Hitches/Min	MVP30	Frametime Avg (ms)	GameThreadtime Avg (ms)	RenderThreadtime Avg (ms)	RHIThreadTime Avg (ms)	GPUtime Avg (ms)	MemoryFreeMB Min	PhysicalUsedMB Max
Entire Run	26999	52.10	3.16	36.30	15.00	9.53	15.44	0.00	1970.44	1587.52
SkyDiving	759	122.44	23.85	41.19	13.34	8.81	18.28	0.00	2186.45	1262.58
OnTheGround	25163	44.58	2.05	34.00	15.08	8.31	15.41	0.00	1970.44	1587.52
MatchWithoutLobby	25924	47.33	2.69	34.00	15.03	8.32	15.49	0.00	1970.44	1587.52

FPSChart **DSCache (Note9, Mali, Siop00, TypeB case)**

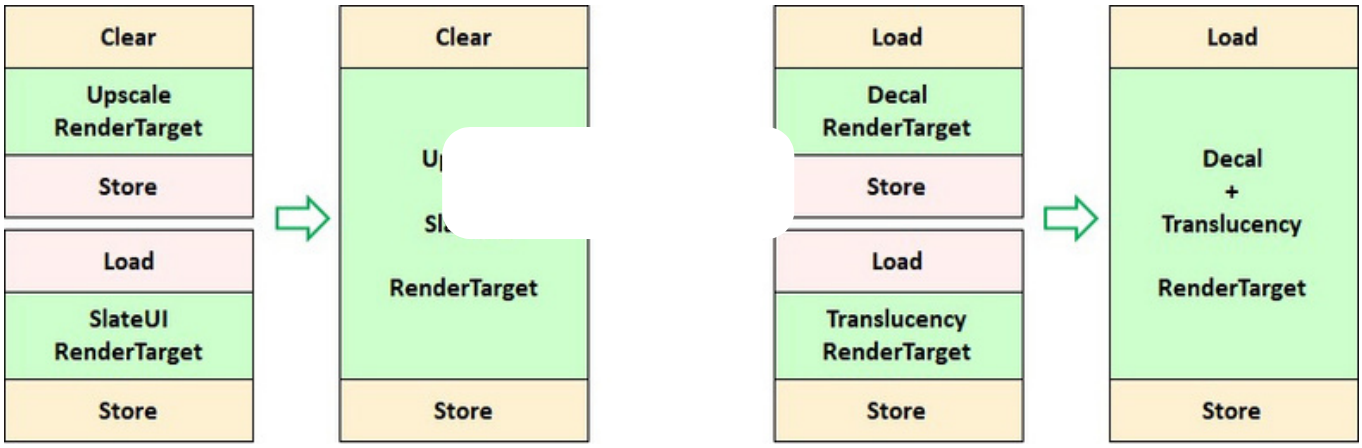
Section Name	Num Frames	Hitches/Min	MVP30	Frametime Avg (ms)	GameThreadtime Avg (ms)	RenderThreadtime Avg (ms)	RHIThreadTime Avg (ms)	GPUtime Avg (ms)	MemoryFreeMB Min	PhysicalUsedMB Max
Entire Run	26999	87.25	1.45	36.01	14.97	8.31	11.83	0.00	1987.04	1585.98
SkyDiving	756	134.89	3.54	41.19	13.34	8.81	14.32	0.00	2201.82	1244.14
OnTheGround	25161	83.66	1.16	33.72	15.08	8.31	11.77	0.00	1987.04	1585.98
MatchWithoutLobby	25919	85.46	1.81	33.94	15.03	8.32	11.84	0.00	1987.04	1585.98

Performance comparison of Fornite after optimizing (caching) descriptor tables. Top: before optimization; bottom: after optimization.

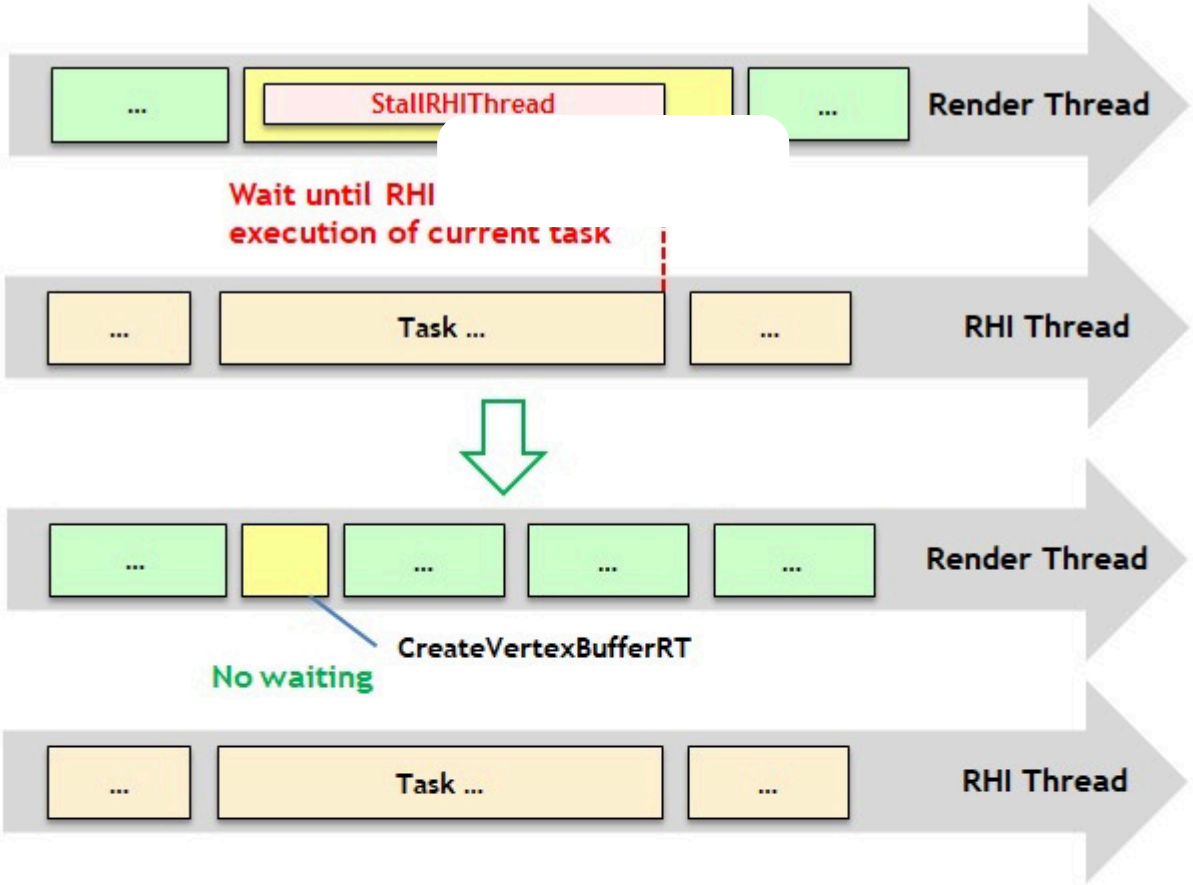




Fornite before and after optimization of synchronization consumption. Top: before optimization; bottom: after optimization.



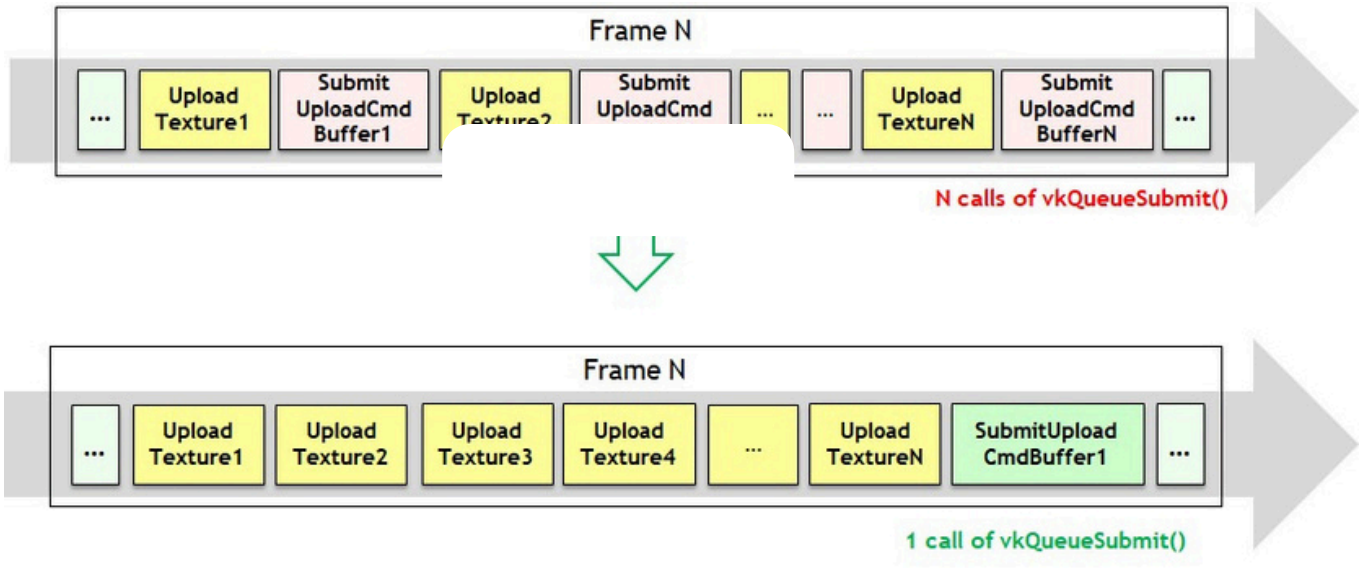
Fornite before and after optimizing the rendering pipeline. Left: Before optimization; Right: After optimization.



Number of Hitches

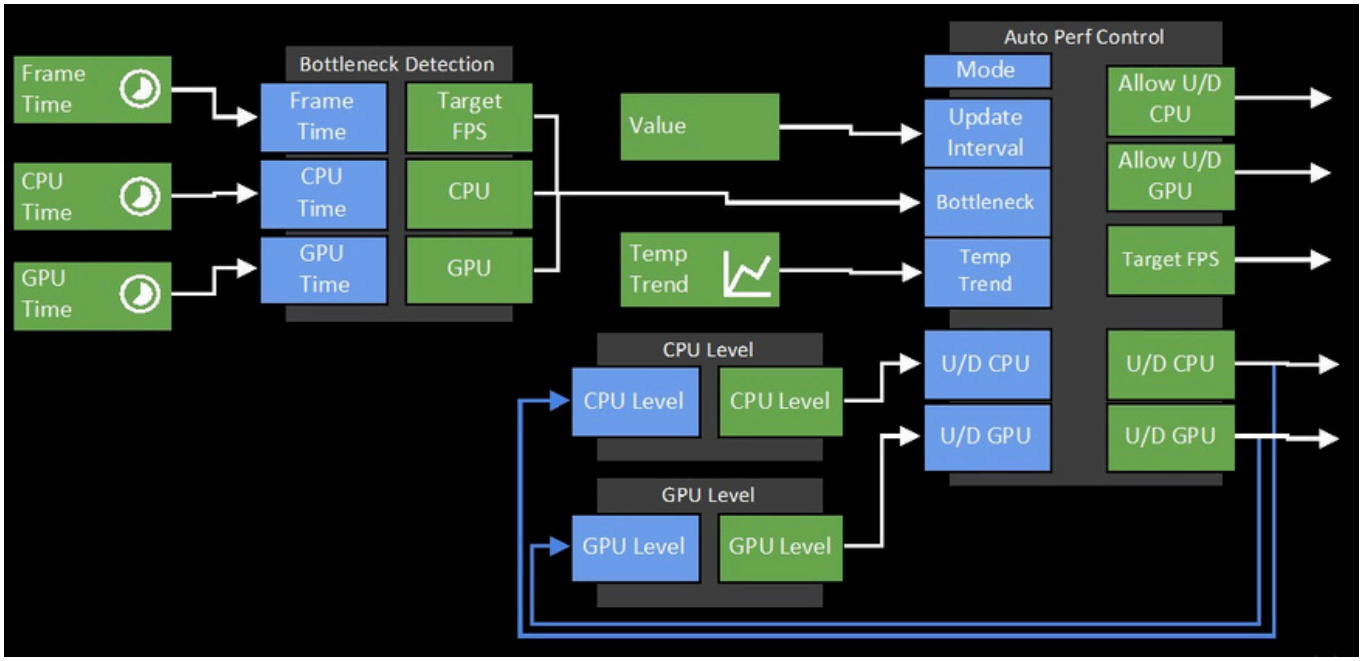
	Original	AsyncBuffer Creation
Frame Time	92	89(-3)
Render Thread Time	38	16(-22)
RHI Thread Time	41	38(-3)

Comparison of the effects of Fornite after asynchronously creating vertex and index buffers.



A comparison of the effects of Fornite optimizing texture uploads (packaging them together).

The Challenges of Porting Traha to VulkanComparison chart of Pipeline Barrier optimization.



An illustration of how Adaptive Performance in Call of Duty Mobile monitors and dynamically adjusts performance, energy consumption, temperature, and other parameters.

12.6.5 XR Optimization

XR rendering usually has the following characteristics: 1. High resolution. The resolution of XR devices (1536x1536, 2K, 4K) is higher than that of ordinary mobile devices (720p, 1080p).

2. Higher refresh rate. The refresh rate of ordinary mobile devices is usually 60Hz or less (such as 30Hz), while XR devices must maintain a refresh rate of 60Hz or even higher (72Hz, 100Hz, 120Hz) to provide a better experience and prevent users from 3D dizziness. 3. Anti-aliasing is required. Without anti-aliasing technology, the rendering quality of XR devices will have severe jagged edges and flickering (because the screen is closer to the eyes).

4. Each frame needs to be rendered twice (people have two eyes). The above special settings result in the bandwidth required by XR devices being more than 9 times that of ordinary mobile devices. Therefore, coupled with the limitations of power and heat dissipation, XR devices are extremely demanding in terms of performance, and the optimization technology requirements are even more stringent.

Below are common XR rendering optimization techniques.

12.6.5.1 Foveated Rendering

Since the human eye requires higher clarity at the center of the gaze point, the required clarity decreases as the distance from the center point increases:

The principle of foveated rendering. The principle is that the closer to the eye's gaze point, the smaller the area covered by the same solid angle (more pixels are needed), and vice versa (fewer pixels are needed).

Qualcomm's XR-specific chips use foveated rendering technology to improve performance by 25% and increase rendering resolution:

Qualcomm uses extensions to OpenGL ES or Vulkan to allow developers to precisely control the details of foveated rendering using detailed parameters:

The effect of foveated rendering and the clarity curve of off-focus points are shown below:

Mali can reduce the frame buffer size by 35%, the total consumption by 20%, and the fragment shader consumption by 40% after using the foveated rendering technology, but it will increase the vertex shader consumption by 52%:

12.6.5.2 Multiview

Qualcomm's XR-specific chip implements multi-view rendering in Advanced mode:

Comparison chart of MultiView for optimizing VR rendering. Top: Rendering without MultiView mode, each eye submits drawing commands separately; Middle: Basic MultiView mode, reuses submitted commands, and copies an extra Command List at the GPU layer; Bottom: Advanced MultiView mode, which can reuse DC, Command List, and geometry information.

Using Multiview rendering technology, you can save 33-49 % of CPU time and reduce Energy consumption by 5-33%:

The Multiview implementation of Mali GPU is different from Qualcomm. Before the Fragment Shader, the same data is shared, while after the Fragment Shader, the left and right eyes are distinguished:

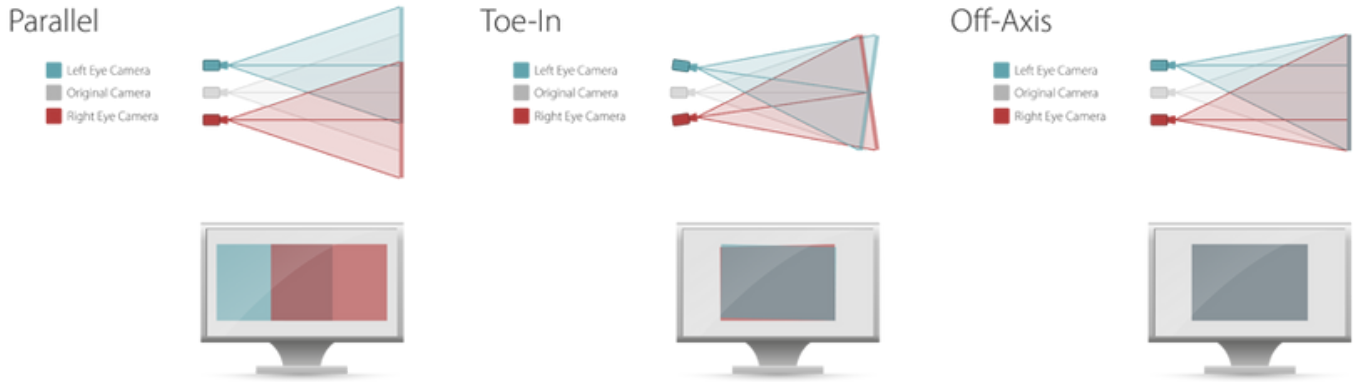
Vulkan also provides optimization technology in Multiview, which is reflected in recording only the different commands for the two eyes, providing multiview-related rendering channels, and using the VIEW_LOCAL tag to improve the utilization rate and cache hit rate within the tile:

Other platforms or graphics APIs also implement Multiview differently:

12.6.5.3 Stereo Rendering

Stereo rendering is to merge the two eyes into one pass to perform rendering, thereby reducing draw calls. The first thing to do is to merge the frustum of the two eyes into one:

Then use the merged view frustum to enter the regular rendering process. According to the camera merging strategy, it can be divided into three types: parallel, transposition, and axis shift:



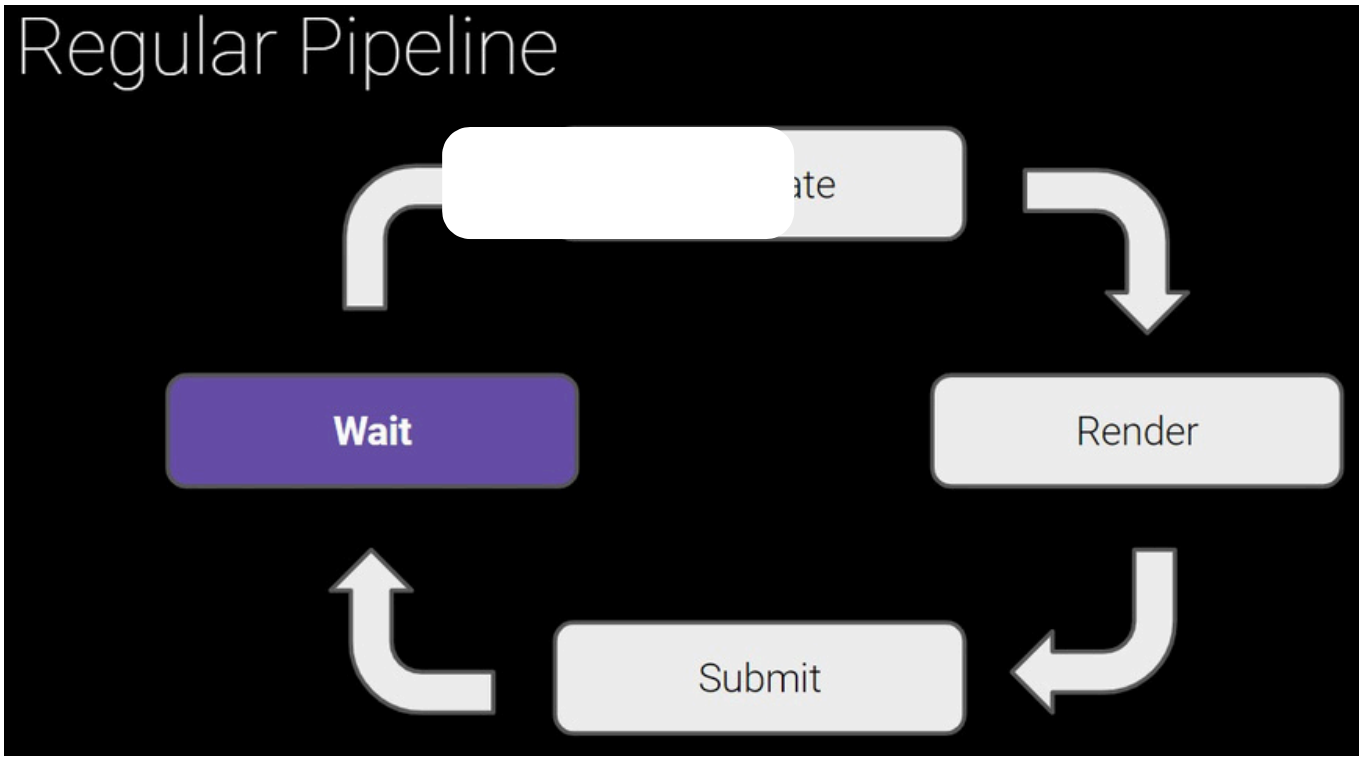
Three camera merging strategies for stereo rendering. Left: parallel; middle: rotation; right: tilt-shift.

The following figure shows the parallel 3D rendering effect (note that the left and right images are slightly offset):

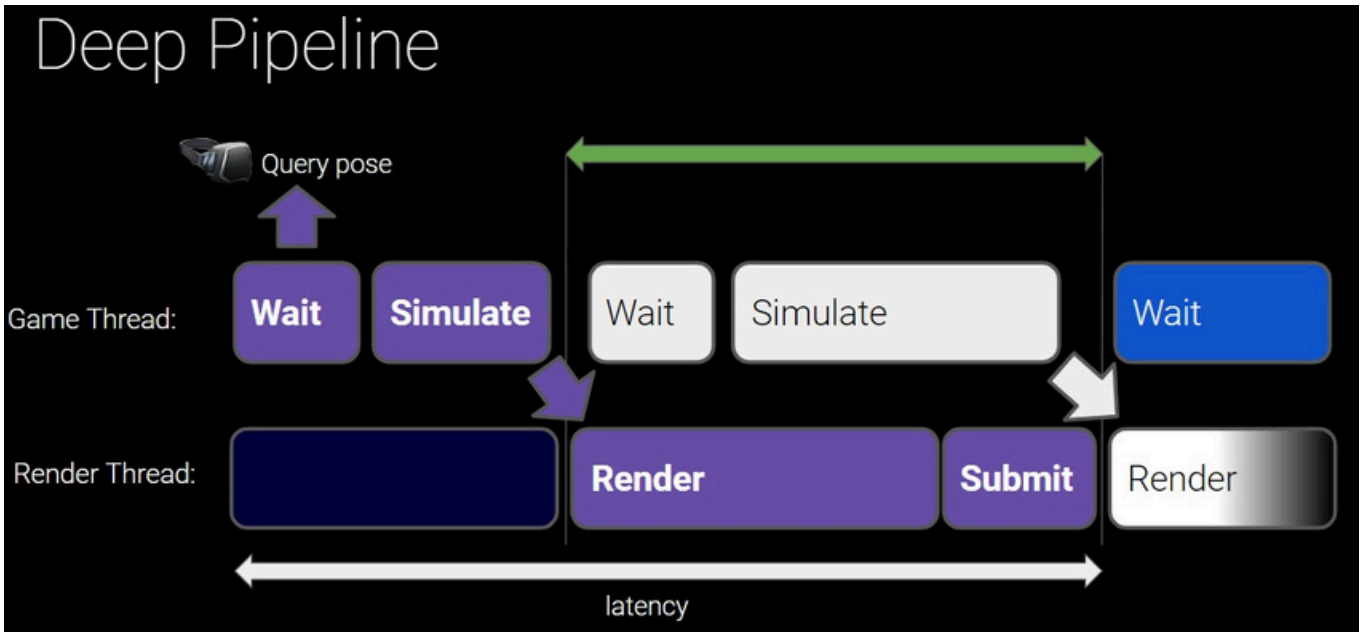
12.6.5.4 Hiding Delay

For XR devices, 20ms is the maximum acceptable latency, which means that the time from obtaining logical data (such as device posture) to display presentation cannot exceed 20ms. Assuming the device runs at 60fps, if there is a two-frame delay, the total time from the first frame to presentation of logical data will reach 50ms! ! (Figure below)

XR applications are different from ordinary simple applications. In order to take advantage of multicore advantages, multi-threaded rendering must be introduced, so there is waiting and delay between each thread.

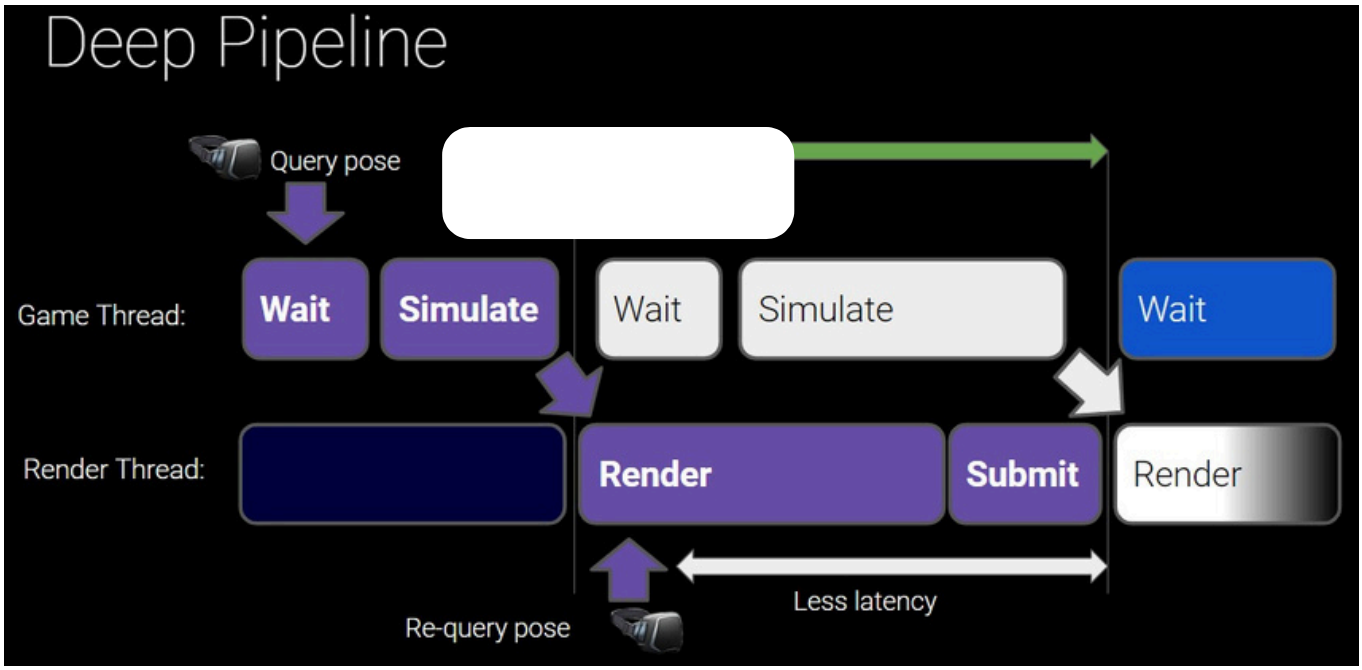


A common application rendering flow chart.



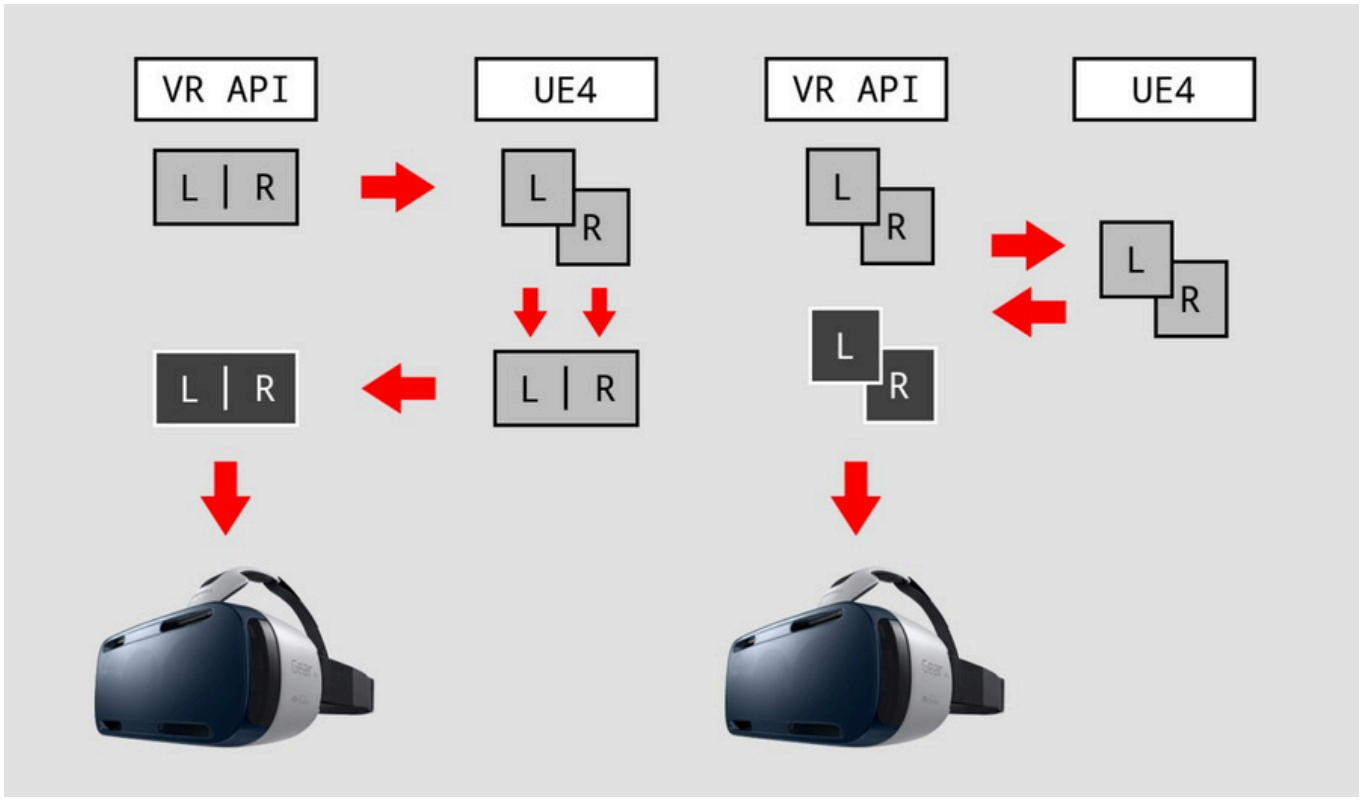
The model that separates the game thread and the rendering thread can separate the logic simulation and rendering layers, but it will cause delays.

If you query the pose information of the XR device in the game thread, there will be a large delay, which will cause the user to feel dizzy and lag. The solution to this problem is also very simple, which is to query the pose information again in the early stage of the rendering thread to reduce the delay:



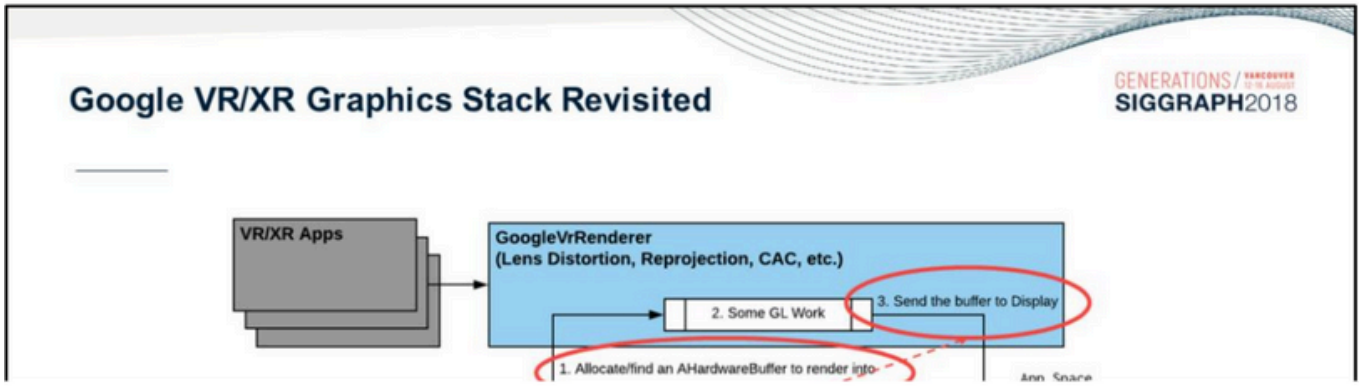
Of course, increasing the frame rate, arranging parallel tasks reasonably to shorten the duration, and using single buffering can also reduce latency.

In addition, Arm has tried to optimize UE4 for VR applications, reducing the original 3-frame delay to 1-2 frames:



- **System-level optimization.**

For example, Google's Android system engineers optimized Android's original triple buffering into double buffering, thereby reducing rendering delay from 3 frames to 1 frame.



Comparison of rendering delays after Android optimization. Top: No optimization; Bottom: Optimization performed.

12.6.5.5 Develop technical specifications

Because XR devices are more stringent than ordinary mobile devices, the standards that can be set will be more stringent.

Here are some recommendations from Oculus regarding technical specifications:

1. Draw Call

Equipment	Number of Draw Calls	Scene complexity
Quest 1	50~150	high
Quest 1	150~250	middle
Quest 1	200~400	Low
Quest 2	80~200	high
Quest 2	200~300	middle
Quest 2	400~600	Low

2. Number of triangle faces

Equipment	Number of triangles
Quest 1	350,000~500,000
Quest 2	750,000~1,000,000

In addition to the above two parameters, you also need to pay attention to technical parameters such as frame rate, resolution, memory, video memory, freeze, power consumption, battery life, device temperature, etc.

Mobile devices require careful attention to the balance of heat and energy.

12.6.5.6 Other XR Optimizations

Optimized Rendering Techniques Based On Local Cubemaps proposes a rendering technique based on Local Cubemap optimization, which can efficiently achieve dynamic soft shadows, reflections and other effects:

The concept and calculation process of Local Cubemap.

Key illustrations and implementation of dynamic soft shadows based on Local Cubemap.

Dynamic reflection key illustration and implementation based on Local Cubemap.

How Crytek Builds 3-Dimensional UI for VR mentions that there are three technologies for rendering fonts in VR: rendering to texture and then mapping to model, distance field, and 3D modeling, and compares the advantages and disadvantages of these methods in detail.

Rendering to texture is not optimized for small fonts and requires a higher resolution.

Distance field fonts have delicate transitions and good anti-aliasing, but they are complex to implement and costly.

3D grid fonts. Requires good anti-aliasing support, and may be the best choice in the long run.

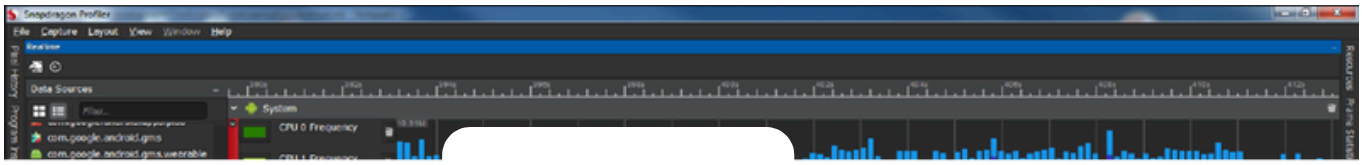
12.6.6 Debugging Tools

Performance optimization and debugging and analysis tools are closely related. As the saying goes, if you want to do your work well, you must first sharpen your tools.

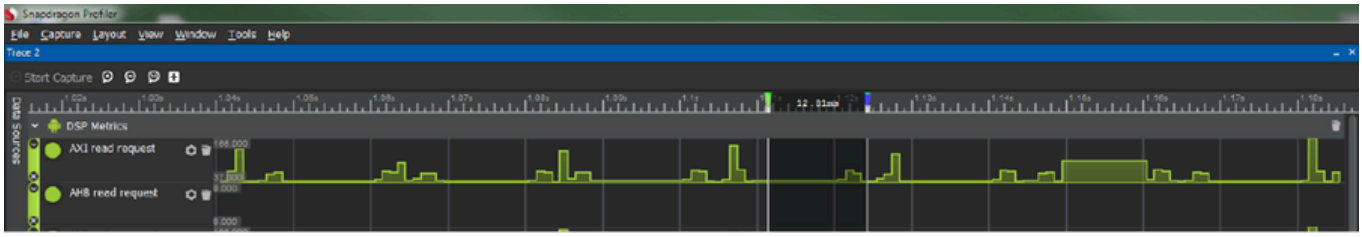
In addition to the conventional performance analysis provided by software or IDEs such as RenderDoc, PIX, Visual Studio, and XCode, GPU manufacturers provide more professional and indepth analysis tools for their own hardware. The following is a table of commonly used manufacturers and corresponding analysis tools:

GPU manufacturers	GPU	Analysis software
Qualcomm	Adreno	Snapdragon Profiler
Arm	Mali	Arm Mobile Studio
Imagination Tech	PowerVR	PowerVR Graphics Tools

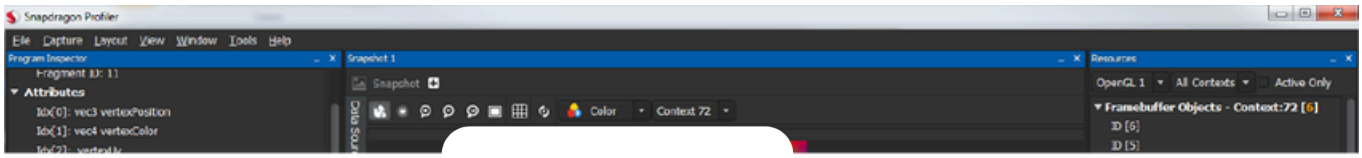
Taking Qualcomm's Snapdragon Profiler as an example, it can monitor the real-time activities of the SoC (Realtime), track the system and driver workload over a period of time (Trace), and analyze the specific rendering status, process and resources of a frame (Snapshot).



Snapdragon's Realtime page.



Snapdragon Trace page.



Snapdragon's Snapshot page.

Using Snapdragon's powerful monitoring function, you can view the consumption of threads, drivers, and GPU components, find performance bottlenecks, and optimize freezes and power consumption. For specific optimization cases, see:Identify application bottlenecks .

12.7 Summary

This article mainly describes the main process of the UE mobile scene renderer, the process of forward and deferred rendering, the rendering characteristics and lighting algorithms of the mobile terminal. The following two parts go beyond UE and elaborate on the current dedicated rendering technology involved in the mobile terminal, the mobile terminal GPU architecture and operation mechanism, and finally give detailed rendering optimization suggestions.

For the optimization of mobile games, you can refer to my other articleGeneral Techniques for Mobile Game Performance Optimization as a supplement.

The mobile terminal topic is divided into three parts, with a total of nearly 60,000 words, and references more than 100 various documents, materials and papers, making it the one with the most references. It took more than a month from organizing, planning, studying the paper to write,
revising and publishing it.

When it's late at night and everyone should be sleeping, or relaxing on weekends, I'm still writing furiously. Although I've almost exhausted all my spare time, I feel a great sense of accomplishment. I hope this article will be helpful and a reference for all of you to learn UE and mobile rendering, and work together for the rise of domestic graphics rendering technology.

12.7.1 Thoughts on this article

As usual, this article also arranges some small thoughts to help understand and deepen the mastery and understanding of UE and mobile rendering:

- Please explain the main process of the UE mobile scene renderer.
- Please explain the main process of forward and deferred rendering on UE mobile terminals.
- Please explain the algorithm of lighting and shadow on UE mobile terminal and the optimizations made.
- Please explain the current mobile-specific rendering technologies, such as TBR, Subpass, etc. What
- are some common rendering optimization techniques for mobile terminals? Please give some examples.

-
-
-
-

-
-
-
-
-

-

-
-
-
-

References

- Unreal Engine Source
- Rendering and Graphics
- Materials
- Graphics Programming
- Mobile Rendering
- Qualcomm® Adreno™ GPU
- PowerVR Developer Documentation
- PowerVR Performance Recommendations
- PowerVR Graphics Techniques
- 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 Compute Development 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 VkPipelineStageFlagBits(3) Manual Page RGBM color encoding The Mechanics of Robust Stencil Shadows Optimized Stencil Shadow Volumes
- PSO Caching
- High Quality, High Performance Graphics in Filament
- Analyzing Modern Rendering
- Advanced Real-time Shadowing Vulkan Multi-Threading Oculus Quest: Testing and Performance Analysis Making EVE: Gunjack Real-time 2D manipulation of plausible 3D appearance using shading and geometry buffers Profiling with Stereo Rendering Introduction to Stereo Rendering
- Multiview Rendering for VR Pipeline
- stall
- Improving Visual Rendering Quality in Mobile Virtual Reality on Unreal Engine
- Performance Guidelines for Mobile Devices
- Virtual Reality Presentations
- The future of mobile VR
- Optimized Rendering Techniques Based On Local Cubemaps
- Optimizing Assets for Mobile VR
- High Quality Mobile VR with Unreal Engine with Oculus
- Snapdragon Profiler
- Understanding and resolving Graphics Memory Loads
- Qualcomm® Snapdragon™ Profiler User Guide Identify
- application bottlenecks
- How Crytek Builds 3-Dimensional UI for VR
-
-
-
-
-
-
-
-
-
-