KHRONOS® GROUP

GDC

GDC 2019
#KhronosDevDay

Galaxy GameDev

DEVELOPER DAY

Bringing Fortnite to Mobile with Vulkan and OpenGL ES

Jack Porter, Epic Games

Kostiantyn Drabeniuk, Samsung Electronics

KHRONOS® GROUP GDC

GDC 2019
#KhronosDevDay

Agenda

- **Part 1 - Fortnite Mobile Challenges and Solutions - Jack Porter, Epic Games**
  - Scope of the problem to bring PC & console cross-play to mobile
  - Performance
  - Memory
  - Recent UE improvements

- **Part 2 - Vulkan for Fortnite Mobile - Kostiantyn Drabeniuk, Samsung Electronics**
  - Vulkan advantages
  - Performance optimizations
  - Hitching and memory optimizations

The Same Game, Not a Port.

From the very start we set out to support cross-play for all platforms including mobile

- The same map is used on all platforms (with regular simultaneous content updates)
- Anything that affects gameplay must be supported
- The engagement distance must be the same across platforms
- Code must not diverge from base Unreal Engine 4

Fortnite Rendering Features - PC & Console

Deferred Renderer

Movable Directional Light
Cascaded Shadow Maps
Ray-traced Distance Field Shadows

Movable Skylight
Distance Field Ambient Occlusion
Screen Space Ambient Occlusion

Local Lights
Point + Spot
Shadows
Shadow Caching

Materials
Physically Based
Subsurface Scattering
Two-sided foliage

Effects
Volumetric Fog
Light Shafts
GPU Particle Simulation
Soft Particles
Decals
Foliage Animation

Post Processing
Bloom
Object Outlines
ACES Tonemapper

Anti-aliasing
Temporal AA
~~MSAA~~

Fortnite Rendering Features - Mobile

Forward Renderer

Movable Directional Light
Cascaded Shadow Maps
~~Ray-traced Distance Field Shadows~~

Movable Skylight
~~Distance Field Ambient Occlusion~~
~~Screen Space Ambient Occlusion~~

Local Lights
Point ~~+ Spot~~
~~Shadows~~
~~Shadow Caching~~

Materials
Physically Based (w/approx)
~~Subsurface Scattering~~
~~Two-sided foliage~~

Effects
~~Volumetric Fog~~
Light Shafts
GPU Particle Simulation
Soft Particles
Decals
~~Foliage Animation~~

Post Processing
Bloom
~~Object Outlines~~
~~ACES~~ Tonemapper

Anti-aliasing
~~Temporal AA~~
MSAA

Scaling Content for Mobile

- **Destructible Hierarchical LOD**
  - Aggregate individual assets into a hierarchy of proxy objects
    - Replace individual assets with proxies at a distance
  - Fortnite is a game where everything is destructible
    - Tag in vertex color to allow the vertex shader cull destroyed geometry from the proxy assets

No HLOD

HLOD

Fortnite Mobile - By the Numbers

Geometry

- 80,000 objects on the island
  - 10,000 typically loaded
- 800 draw calls average, 2000+ peak
- 600,000+ triangles (high end)

Shaders / PSOs

- 4,300 PSOs actually needed for rendering!
  - gathered using automated and manual gameplay
  - from a pool of 28,000 shader programs

Memory

- 1.2GB – 2GB
  - Varies depending on device profile and rendering API and shader allocation strategy

Challenges

- Performance
- Memory
- Device Compatibility

Performance

CPU cost

- **Draw call cost - graphics API overhead**
  - Add an RHI Thread
  - Use Vulkan instead of OpenGL ES
- **Reducing draw calls & state change**
  - Improving occlusion culling
  - Sorting
  - Instancing

GPU cost

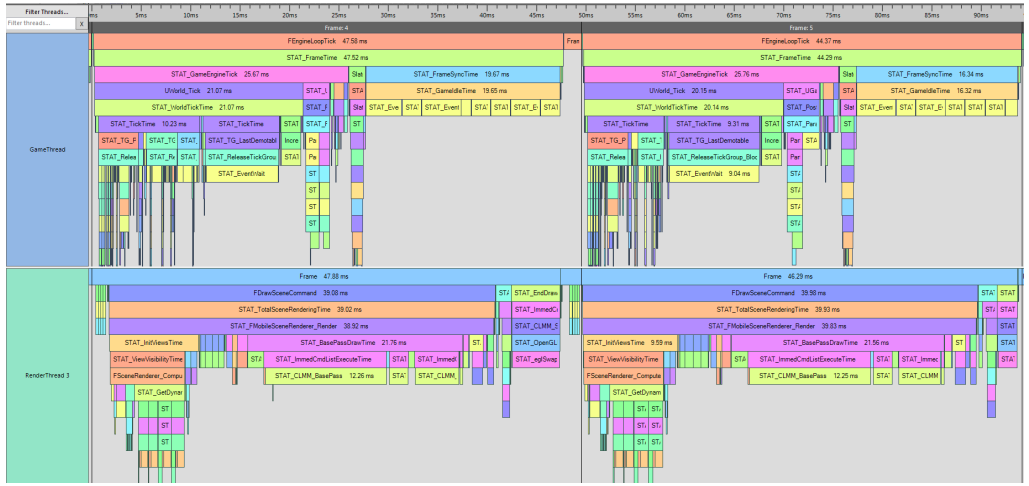
- **Content changes**
- **Resolution and frame rate scaling**
- **Rendering code improvements**
  - Collapsing render passes

Draw Call Cost - Renderer Threading

- **Unreal Engine 4 has two main threads**

Game Thread

- Update game state from player input, network and physics simulation
- Enqueue game object state change
- Enqueue resource changes
- Send command to render scene

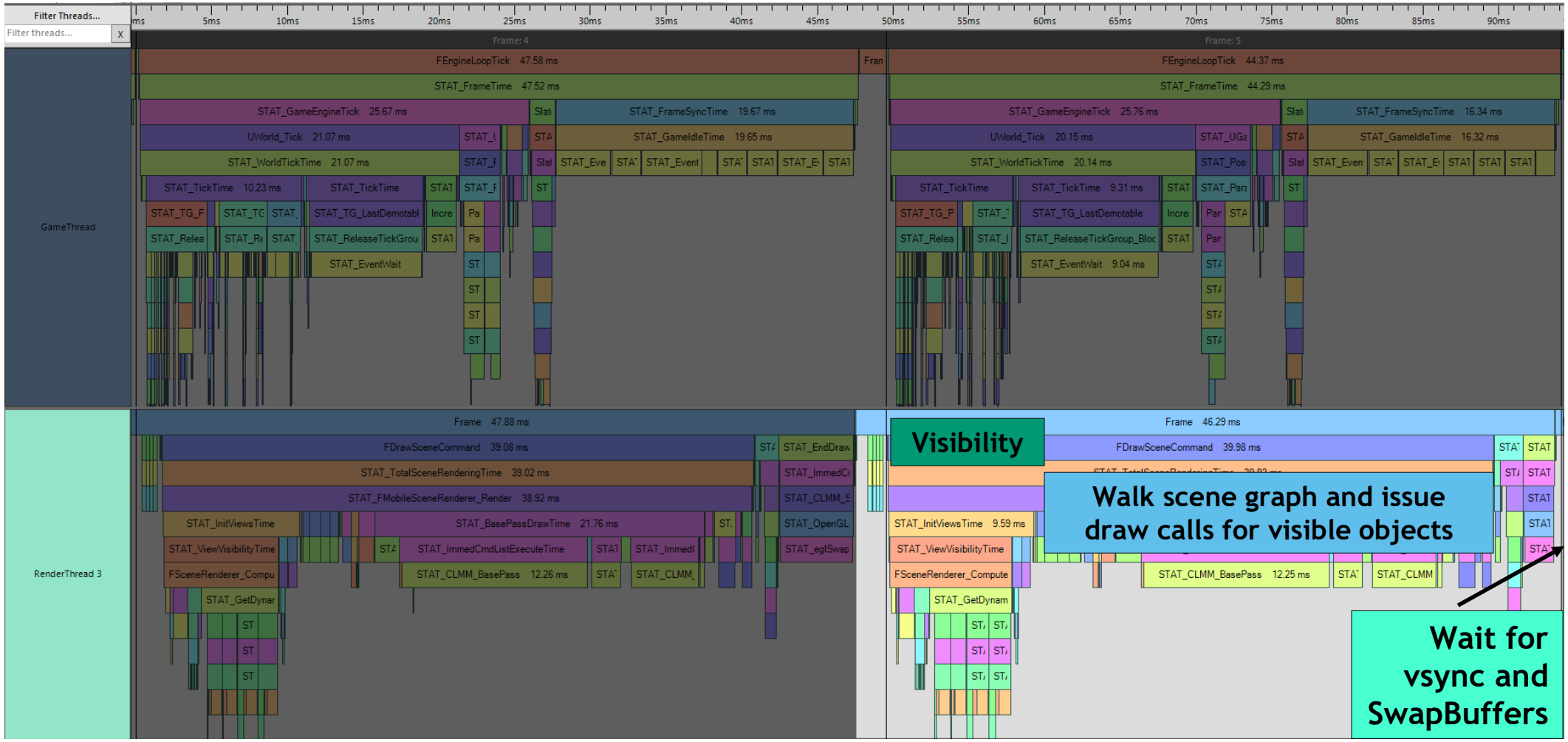


Render Thread

- Dequeue state change into game object render proxies
- Create or update render resources
- Render scene

1. Retrieve occlusion queries from a previous frame
2. Calculate object visibility
3. Render shadow maps
4. Render opaque geometry including lighting and shadows (base pass)
5. Render occlusion queries testing on depth layed down by base pass
6. Render translucency pass
7. Render post-process and tonemap
8. Render UI

Game / Render Thread



Add RHI Thread

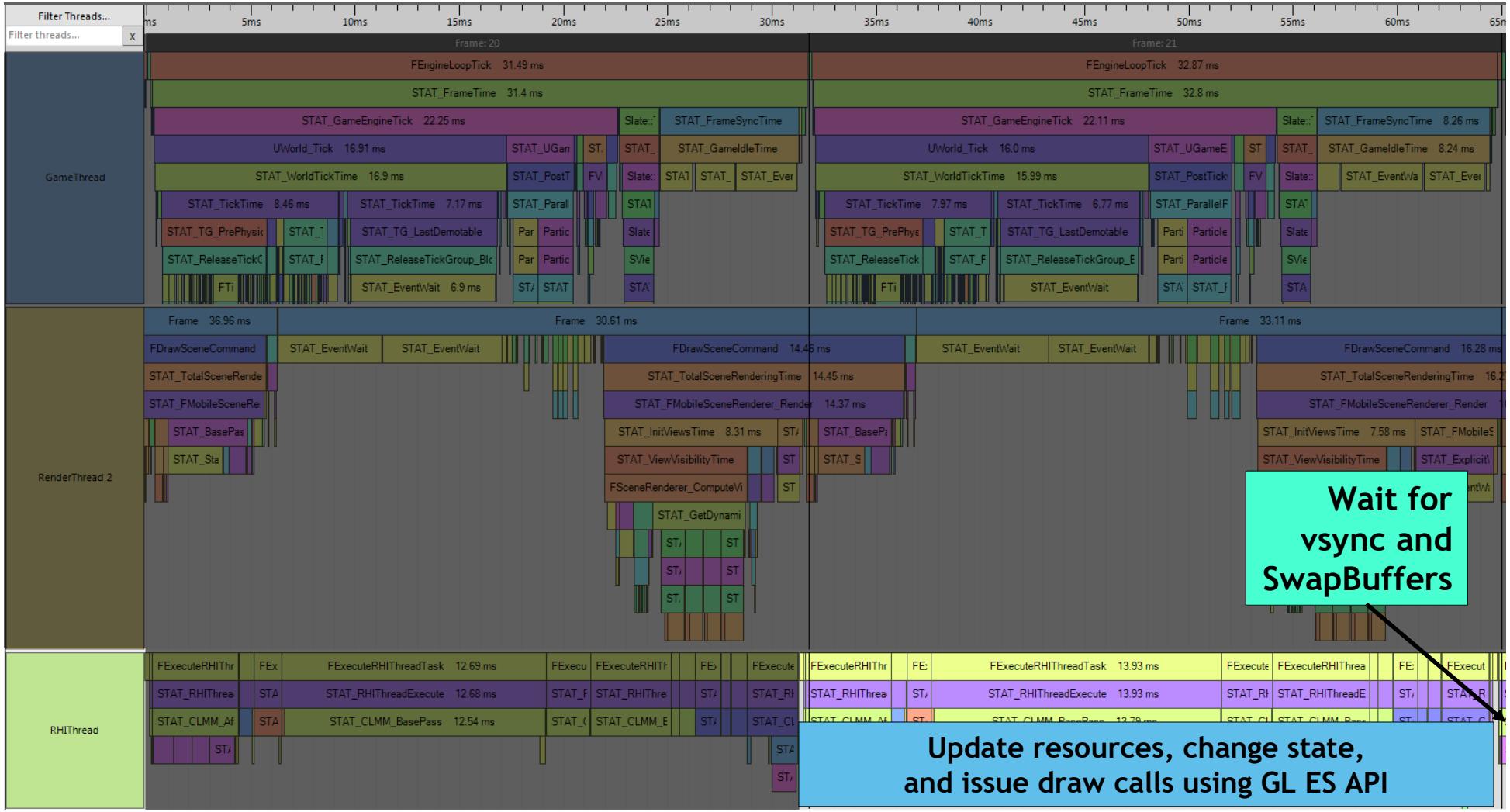
Significant part of Render thread time is spent inside GL API calls, especially when there has been a lot of state change.

- 25ms or more on low end devices
- Time mostly spent in glDrawElements
- Most of the benefits of instancing come from sorting better by state

Improvement was to add an "RHI Thread" that does nothing but issue GL API calls

- Rendering code never waits for GL API calls to return
  - ➢ Resource creation and update APIs return to Render thread immediately with a proxy handle

Game / Render / RHI Thread



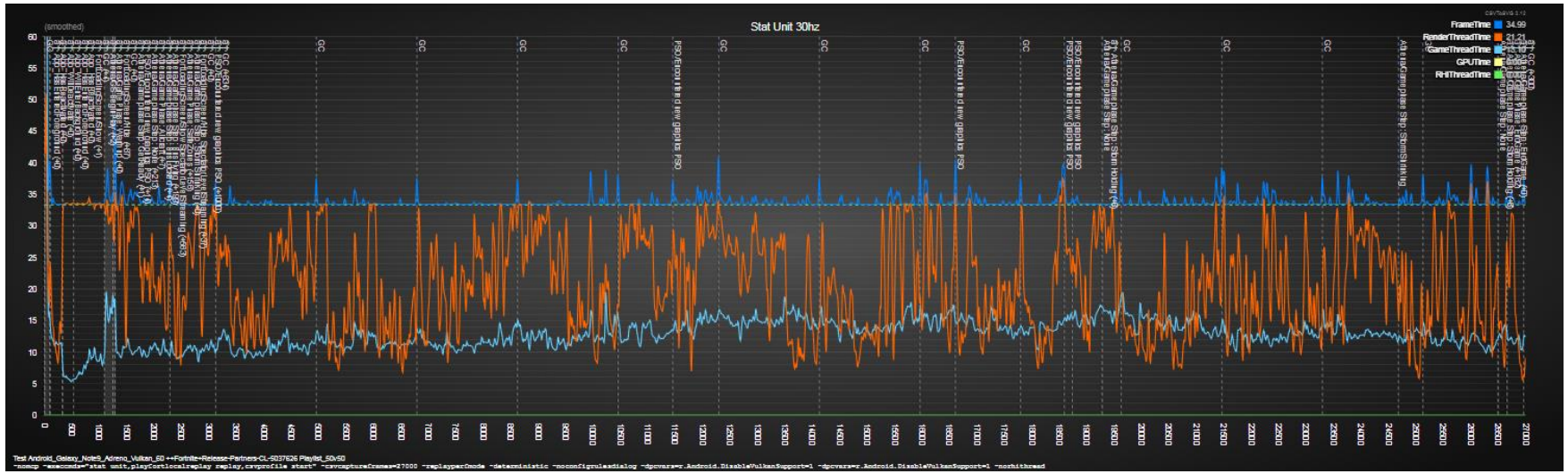
RHI Thread Synchronization

- **Render thread synchronizes with RHI thread**
  - Waiting on occlusion query results
    - ➢Using the RHI Thread adds an extra frame of latency for occlusion queries

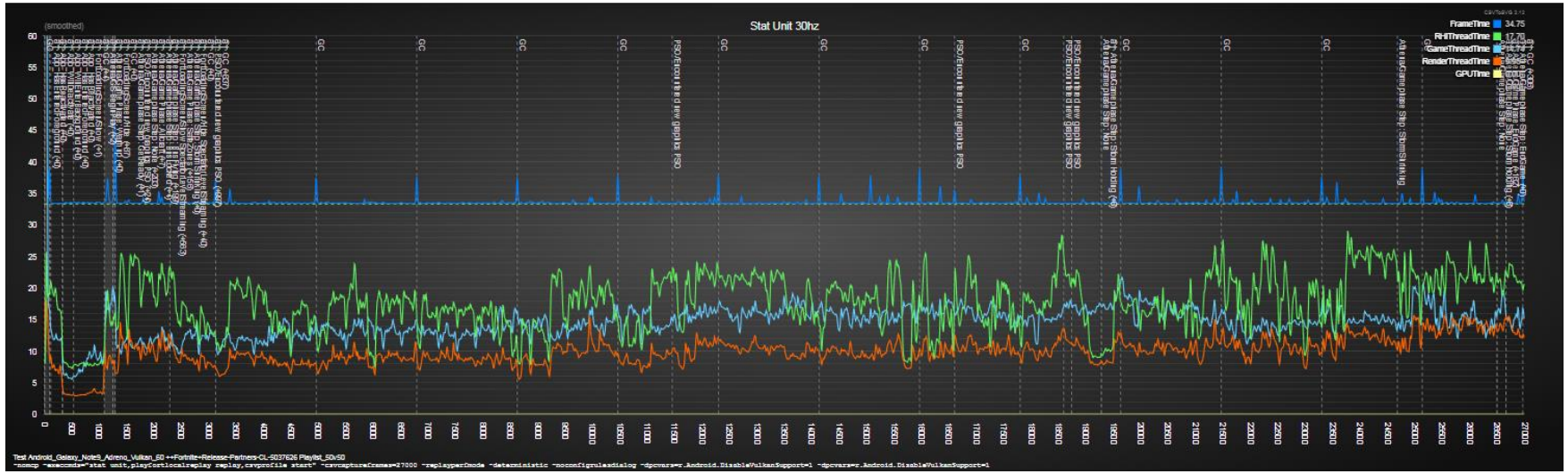
- **Game thread synchronizes with RHI thread**
  - Waits to ensure the RHI thread doesn't get more than 2 frames behind

RHI Thread comparison - OpenGL ES

RHI Thread disabled



RHI Thread enabled



Game Thread

Render Thread

RHI Thread

Overall Frame Time (ms)

Galaxy Note 9
Adreno
GL ES Mode

Draw Call Cost - Graphics API Selection

Fortnite for Android can run with either UE4's OpenGL ES or Vulkan Render Hardware Interface (RHI), chosen by the Device Profile at runtime.

- **OpenGL ES 3.1**
- **ASTC textures**
- **Android 6.0 or later**
- **Extensions**
  - EXT_color_buffer_half_float
  - EXT_copy_image (or ES 3.2)
  - OES_get_program_binary

- **Vulkan 1.0.1**
- **ASTC textures**
- **Android 8.0 or later**
- Whitelisted for specific devices based on improved measured performance

Graphics API Selection

Unfortunately Vulkan is not a clear win on many devices

- Lack driver maturity on older devices can lead to poor performance
- Modest CPU win on newer devices
- Extra GPU cost can negate any gains (working to reduce this)

Many devices where we'd most like to use Vulkan – ie devices with poor CPU performance limiting draw call counts - are unable to benefit from it.

Graphics API Selection

Currently Fortnite enables Vulkan only on:

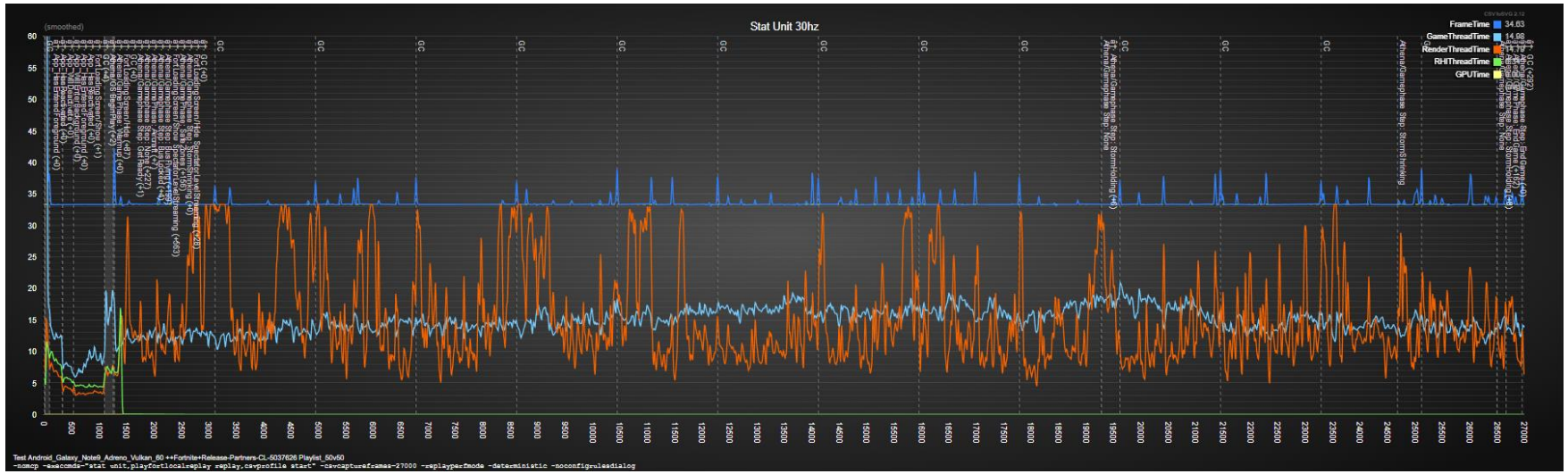
- Galaxy S9 Adreno
- Galaxy Note 9 Mali and Adreno
- Galaxy S10 Mali and Adreno

- Vulkan is also a win on modern devices such as Snapdragon 845 and Mali-G76 devices
  - Expect to ship it enabled by default for many of this year's flagship devices

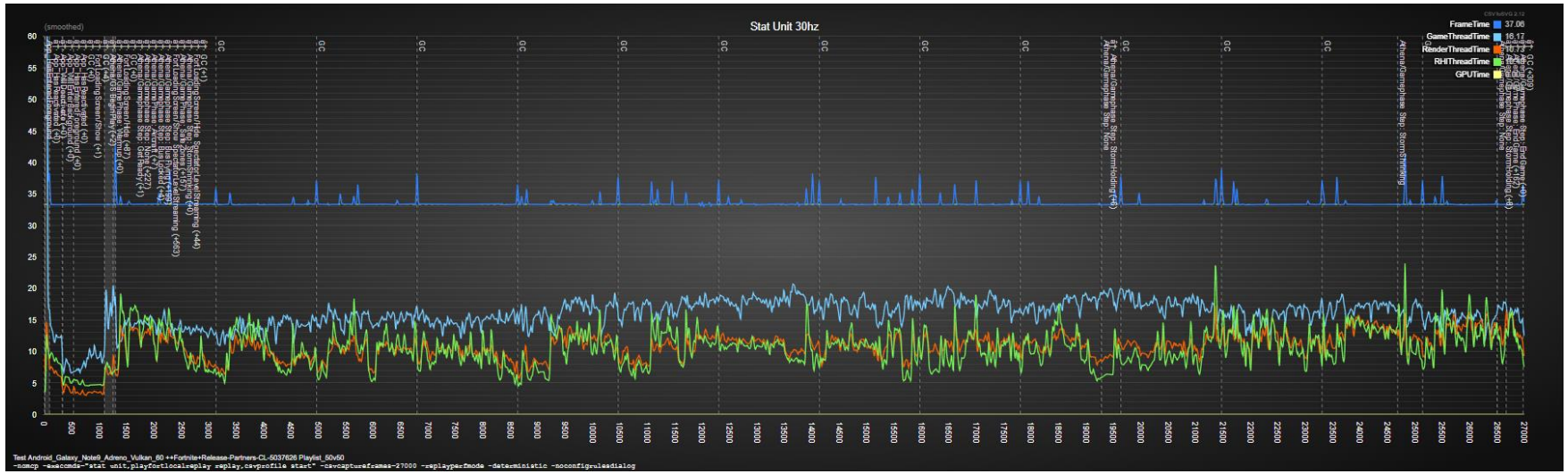
Vulkan is enabling us to push quality and performance at the high end

RHI Thread comparison - Vulkan

RHI Thread disabled



RHI Thread enabled



Game Thread

Render Thread

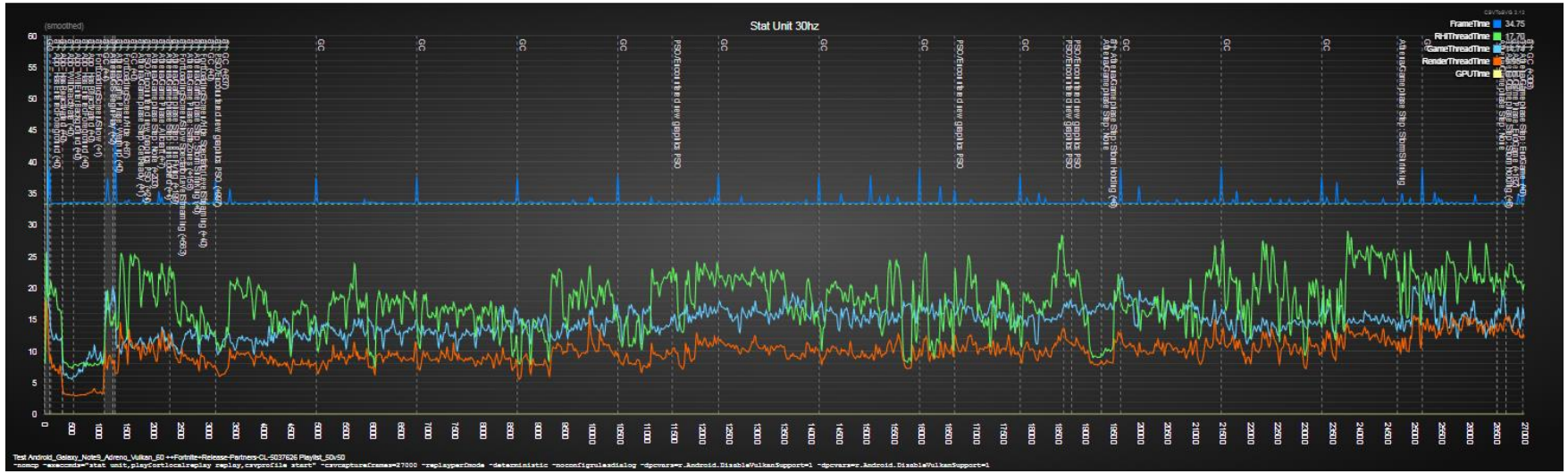
RHI Thread

Overall Frame Time (ms)

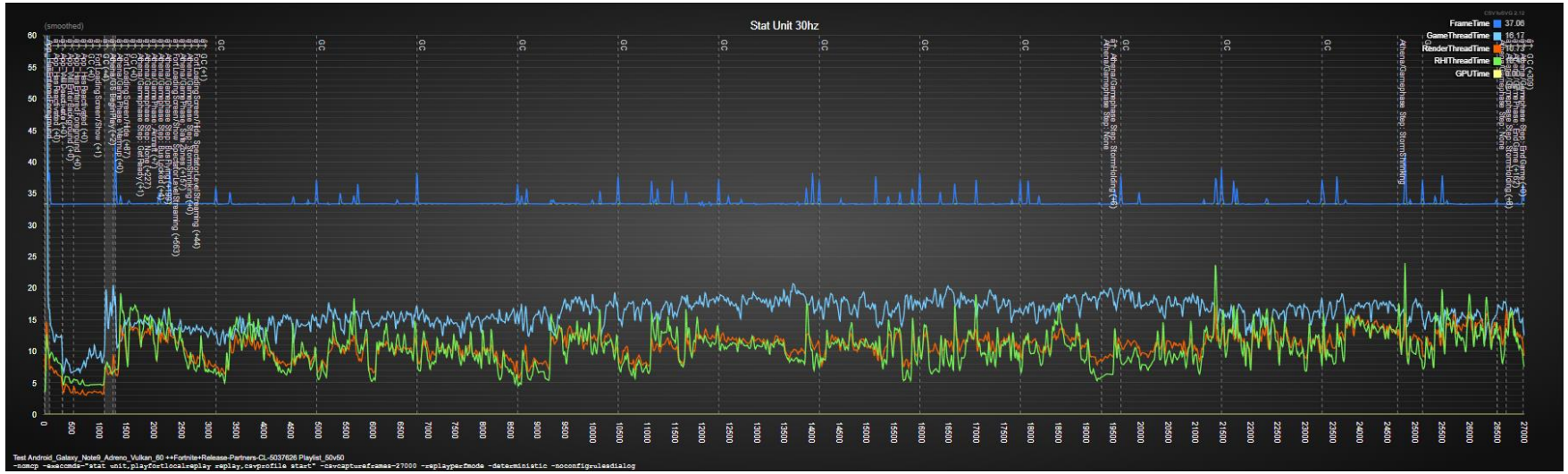
Galaxy Note 9
Adreno
Vulkan Mode

RHI Thread comparison - Vulkan vs OpenGL ES

OpenGL ES



Vulkan



Game Thread

Render Thread

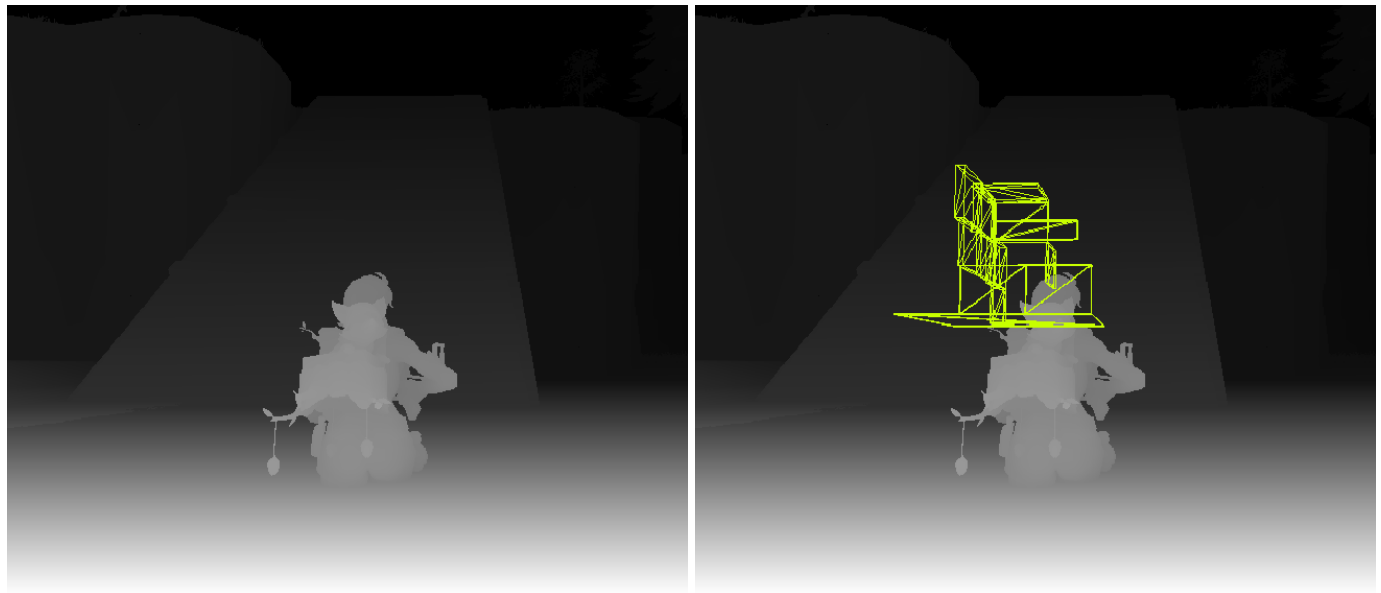
RHI Thread

Overall Frame Time (ms)

Galaxy Note 9
Adreno

Draw Call Cost - Occlusion Culling

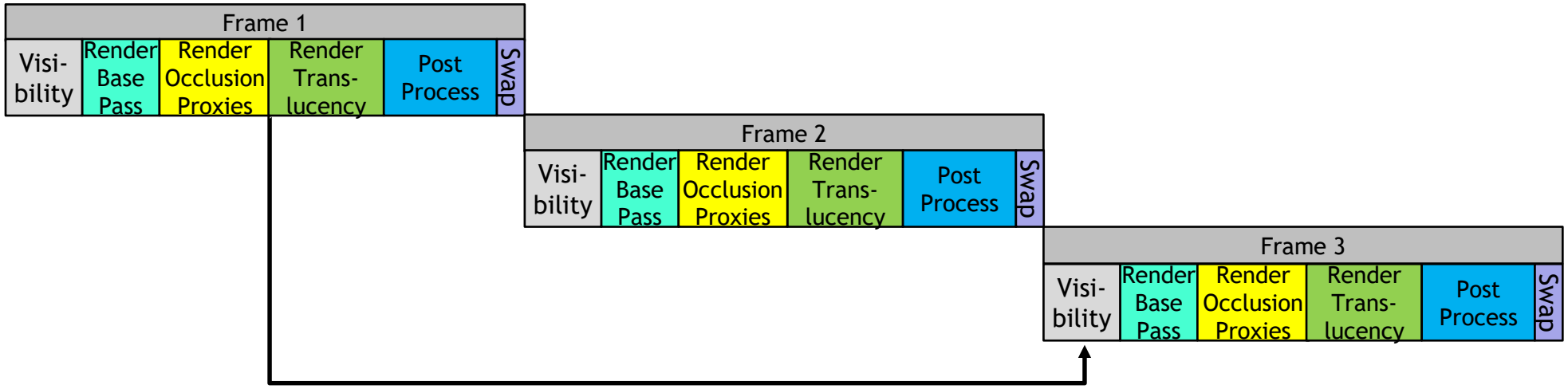
- **Render proxy geometry against the depth buffer wrapped with a query**
  - glBeginQuery, glEndQuery / vkCmdBeginQuery, vkCmdEndQuery
- **Check if any pixels of the proxy geometry was renderered**
  - glGetQueryObjectuiv(GL_QUERY_RESULT) / vkGetQueryPoolResults(VK_QUERY_RESULT_WAIT_BIT)
- **Use that information to decide whether to render the real geometry**



Occlusion Culling - Implementation

Latency

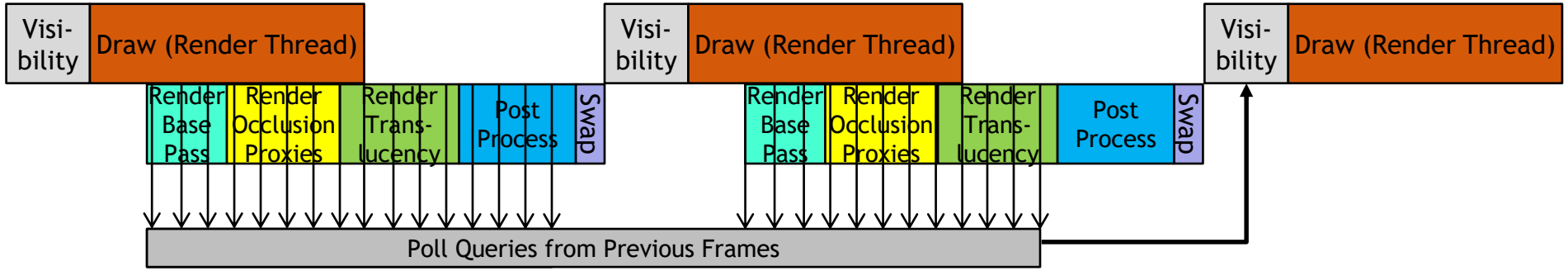
- **We need an existing depth buffer to test against**
  - On PC & console we do a depth prepass so we can render the queries early in the frame
  - On mobile we don't have the depth buffer until the end of the base pass
  - We need to adds one extra frame of latency to insure the results are available in time



Occlusion Culling - Implementation

Thread synchronization when reading results

- We can only wait for queries on RHI Thread
- Results are needed on the Render thread where we calculate visibility
  - Poll results using glGetQueryObjectuiv(GL_QUERY_RESULT_AVAILABLE) between RHIThread commands and update a thread-safe flag



  - Usually have results before Render thread asks for them and we do not need to block

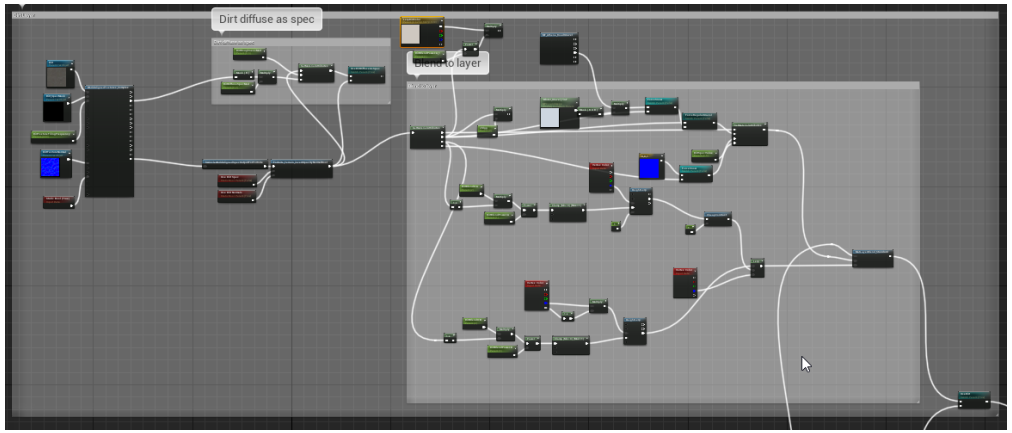
Occlusion Culling - Implementation

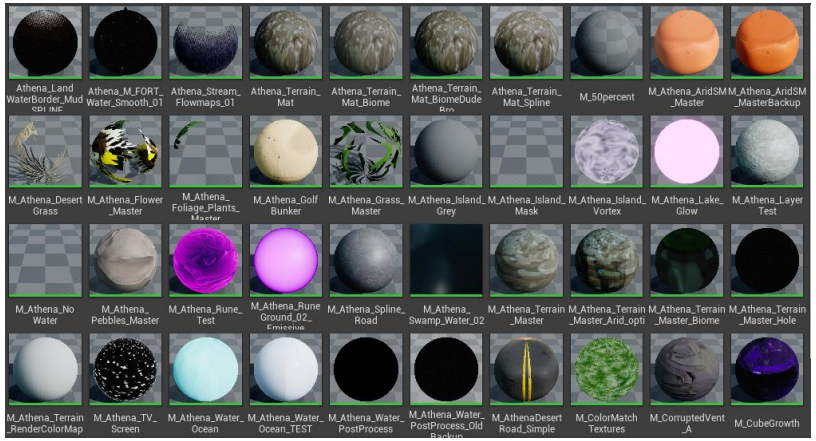
Limited number of queries

- **Ideally we would have one occlusion query per object**
- **Some mobile devices have internal limits for the number of outstanding queries**
  - OpenGL and Vulkan RHIs virtualize occlusion queries to abstract this away
  - Aggregate proxy geometry on some frames

Shader Program Memory

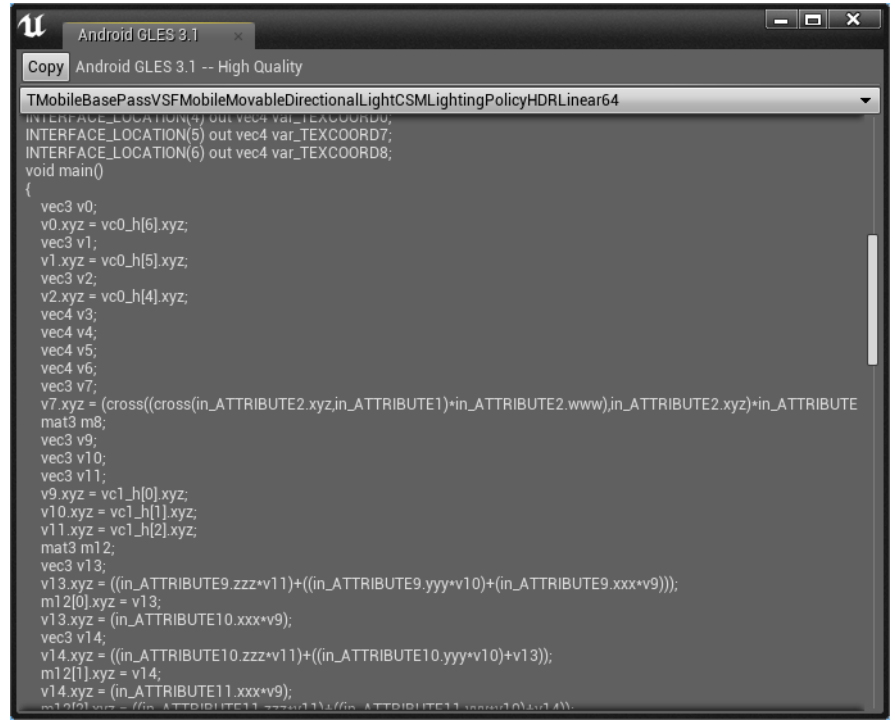
- In UE4 the majority of shaders are created from artist-generated materials





Artist-created material shader graphs

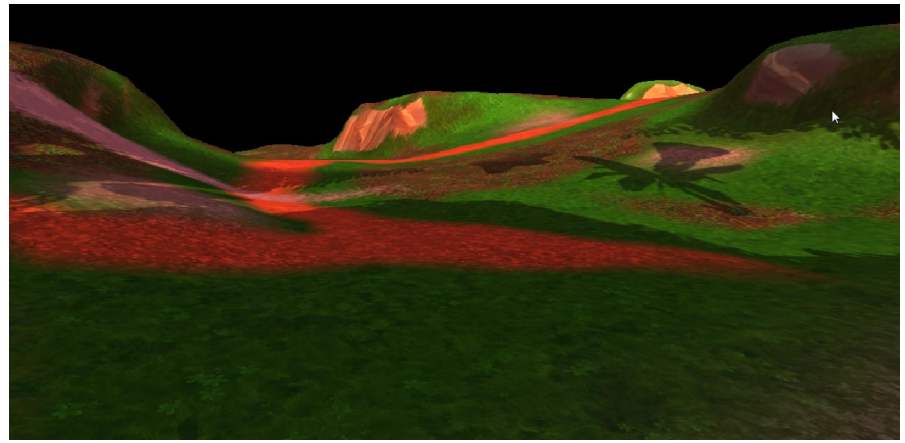
hlslcc

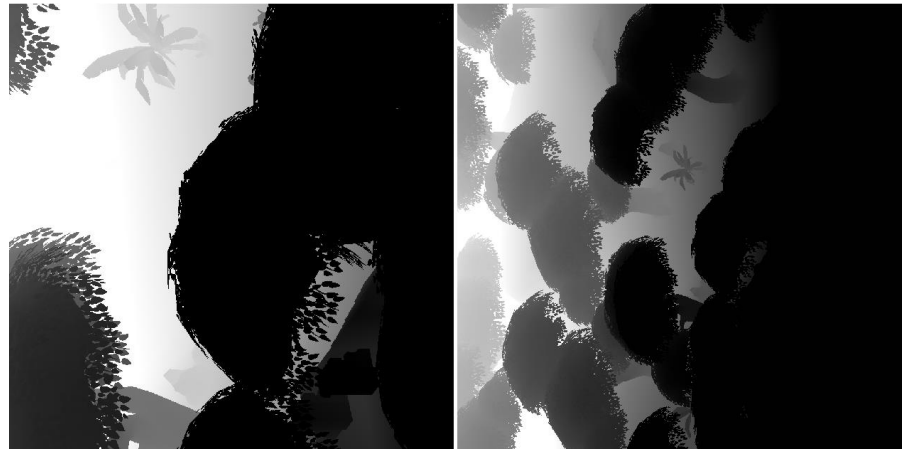


GLSL

Shader Permutations

- **From each material graph we generate fragment and vertex shader permutations**
  - "Vertex Factory" (mesh type)
    - Static mesh, skeletal mesh, particle, terrain, …
  - Forward lighting pass
    - Base forward pass with CSM shadow
    - Base forward pass, unshadowed
  - Shadow depths
    - Shared for opaque objects
    - Unique for alpha masked objects
  - Translucency & effects
- **Result is over 28,000 individual shaders**





Shaders – OpenGL ES

- Set of PSOs encountered while playing Fortnite gathered offline using automated and manual gameplay
  - 4,300 PSOs actually needed for rendering

On OpenGL ES we must compile from GLSL source code

- First launch of Fortnite
  - Compile all shader programs
  - Save the resulting shader program binary to the user's phone using GL_OES_get_program_binary

- Subsequent launches
  - Recreate shaders with glProgramBinaryOES

Shaders – OpenGL ES – LRU Cache

- **Ideally we would have all shader programs created before gameplay starts**
  - Shaders measured to expand to more than 10x their binary size in driver RAM allocations
  - Instead we use an LRU cache to keep only a limited number of shader programs resident
  - Saves over 400MB on some devices

Shaders – OpenGL ES – LRU Cache

- **Shader eviction strategies**
  - When resident shader program count exceeds some threshold
  - Estimation of resident shader memory
  - On object destruction
    - Not great for transient but frequent uses like particles, so we add an extra delay

- **Shader restoration strategies**
  - Stream shader binary from storage, creates hitches in practice
  - Recreate shader from compressed binary in RAM
    - On Adreno, shaders total about 20MB compressed so it's feasible to always keep them resident
    - On Mali we keep binaries in RAM for non-resident shader programs
      - Create create binary from shader program on eviction and store in RAM
      - Restore shader program from RAM binary and free RAM binary

Shaders - Vulkan

- We gather Vulkan PSOs using the same mechanism as for OpenGL GL
- On Vulkan we create pipelines on first launch and save vkPipelineCache to storage
- Vulkan mode also has a runtime PSO cache in memory with LRU
- Kostiantyn will provide some details including shader memory savings

Recent UE4 Renderer Improvements

Unreal Engine has been evolving to support new platforms and rendering APIs since its inception.

The Render Hardware Interface abstraction layer (RHI) has had some recent improvements made to better support modern graphics APIs like Vulkan.

1. Explicit render passes
2. Vulkan subpasses
3. High-level rendering refactor

1. Explicit Render Passes

Starting a new renderpass can be expensive on mobile tiled GPUs

- Save out the results of the previous render pass from the GPU core to RAM
- Load an existing render target from RAM back into the GPU core

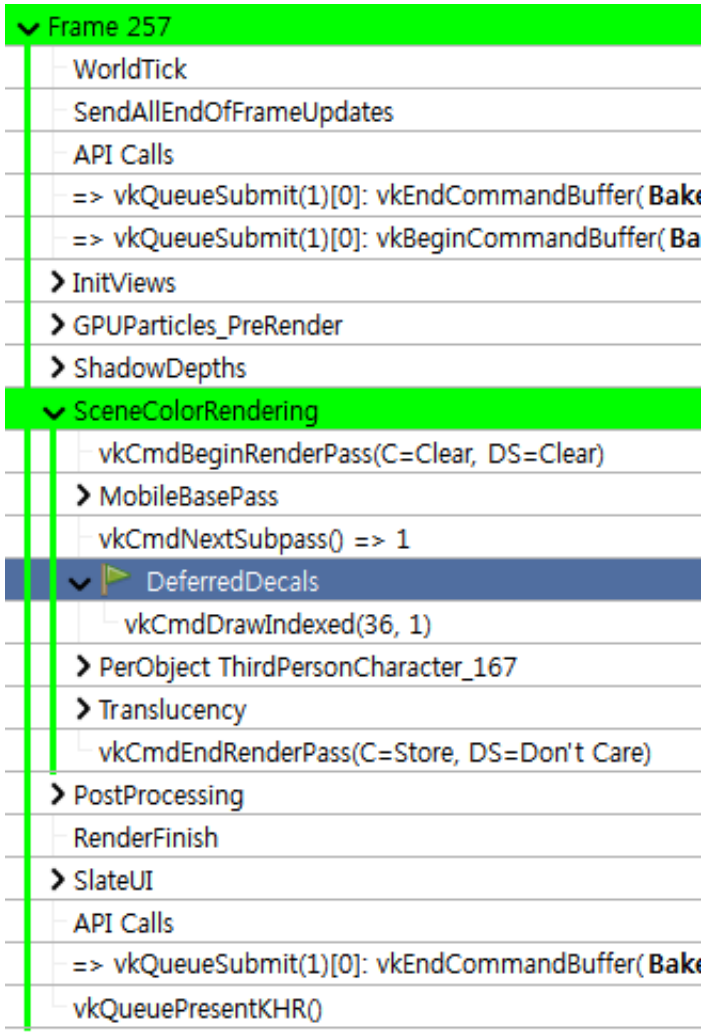
- **Render passes in the high level code were originally implict**
  - Engine code set render targets and then the RHI guessed if we were starting a new render pass
  - Each rendering operation (eg shadows, base color, translucency) called functions at the beginning and end of their operations to set render targets and resolve the results
- **New in UE 4.22**
  - RHI functions have been added to explictly begin and end render passes
  - UE4 mobile renderer now makes use of these to remove of unnecessary transitions
    - eg base pass → translucency → post processing

2. Vulkan Sub-passes

- **Use case: Soft Particle Translucency and Deferred Decals**
  - These rendering techniques require access to an existing fragment depth value
  - In GLES we use EXT_shader_framebuffer_fetch to get an existing depth value
    - Depth previously written to alpha in base pass, or we use ARM_shader_framebuffer_fetch_depth_stencil where available
    - ➢ Compare fragment depth against existing depth

2. Vulkan Sub-passes

- **Very Vulkan-specific feature, and only applicable to mobile GPUs**
  - ➢No general UE4 support for subpasses, instead:
- **RHIBeginRenderPass() call provides a hint that the following passes will use depth**
  - Vulkan RHI sets up 2 subpasses
  - RHINextSubpass()
    - VulkanRHI calls VkNextSubpass()
    - OpenGLRHI could use this to call FramebufferFetchBarrierQCOM() to support QCOM_shader_framebuffer_fetch_noncoherent



2. Vulkan Sub-passes

Unfortunately extra PSO permutations are necessary to support MSAA

- The depth is fetched using GLSL subpassLoad(input), but when using MSAA you must use subpassLoad(input, sampleindex)
  - ➢So toggling MSAA requires alternate shaders

```
UniformConstant DescSet=0 Bind=3 InputAttachmentIndex=0 ImageSubpassData<float>* GENERATED_SubpassDepthFetchAttachment;
UniformConstant DescSet=0 Bind=2 SampledImage2D<float>* Material_Texture2D_0;
Input float4* gl_FragCoord = FragCoord;
Uniform DescSet=0 Bind=1 HLSLCC_CBh* HLSLCC_CBh_34;
Output Location=0 float4* out_Target0;

void main_00000ff8_e02b3916() {

  float2* v0 = 0.0f.xx;
  v0 = HLSLCC_CBh_34.pu_h[4].xy;
  float4* v1 = 0.0f.xxxx;
  v1 = gl_FragCoord;
  v1.w = 1.0f / gl_FragCoord.w;
  float4* v2 = 0.0f.xxxx;
  v2 = v1;
  float* f3 = 0.0f;
  f3 = VulkanSubpassDepthFetch();
```

```
} // main_00000ff8_e02b3916

float VulkanSubpassDepthFetch() {
  return ImageRead(GENERATED_SubpassDepthFetchAttachment, 0.xx).x;
} // VulkanSubpassDepthFetch
```

- Targeting UE 4.23

3. High Level Rendering Refactor

- **Much more aggressive caching for static scene elements**
- **The full state of each drawcall is cached when mesh added to the scene**
  - Pipeline State ObjectBound resources, shader constants and uniform buffers
- **Much reduced Render thread cost**
  - ➢After calculating visibility it simply walks the drawlist and applies the cached state
- **Initial release in UE 4.22**

3. High Level Rendering Refactor

- **Automatic geometry instancing support**
  - Sort draw list by PSO, mesh and bound resources
  - Examine for sets of matching PSOs and bound resources
  - Requires all per-instance constants (eg transform matrices) to be stored in a single buffer
    - Look up per-instance parameters in the shader
    - Potentially bad for mobile performance.
      - Previously measured ~30% cost. Work in progress.

Part 2

Vulkan for Fortnite Mobile

Kostiantyn Drabeniuk,
Samsung Electronics

Galaxy GameDev

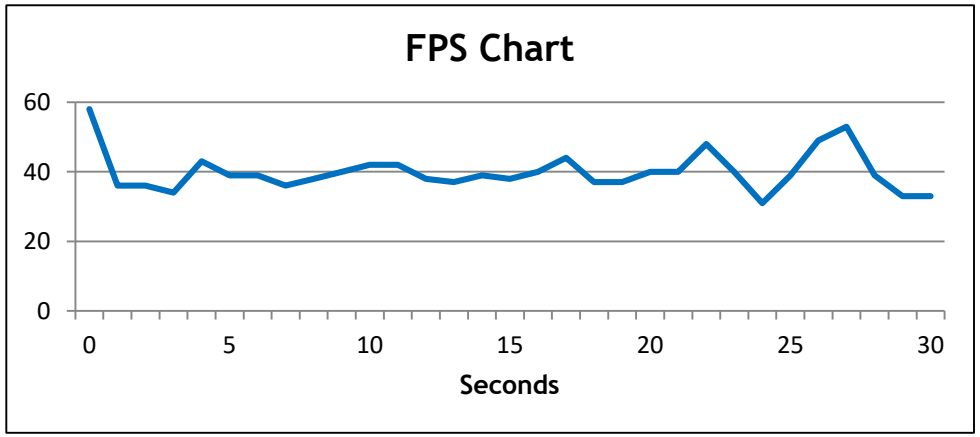
- Provide the best gaming experience to customers on Samsung devices
- Promote new technologies usage
- Contribute to the most popular game engines
- Support game developers all over the world

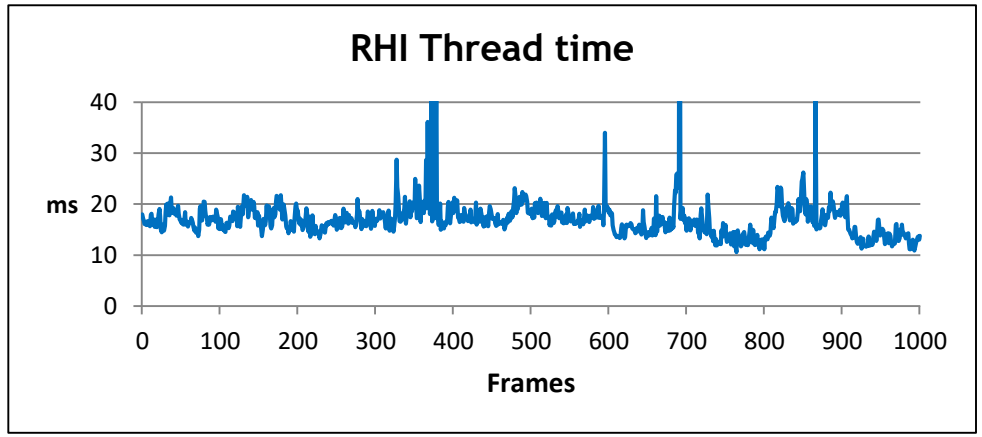
Agenda

- Advantages of using Vulkan in mobile games
- How to get more FPS - performance optimizations
- How to get stable FPS - hitching/memory optimizations

OpenGL ES

FPS	39
RHI Thread Time (ms)	16.94



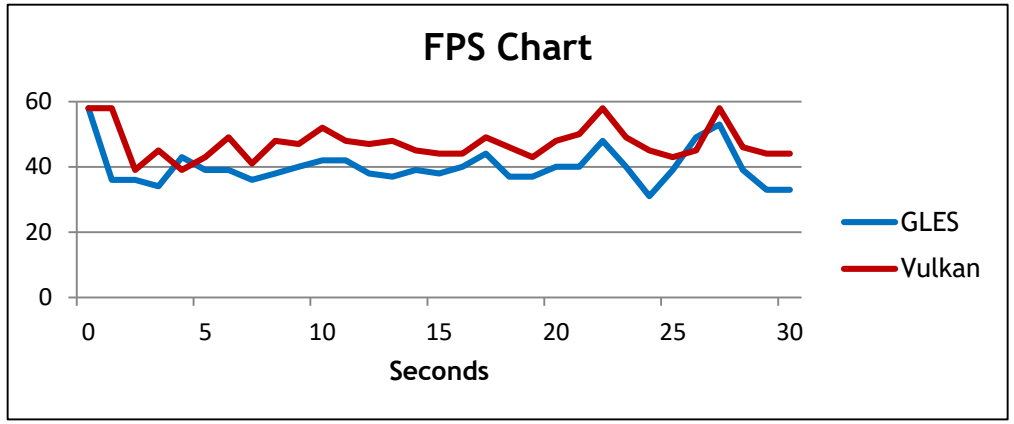


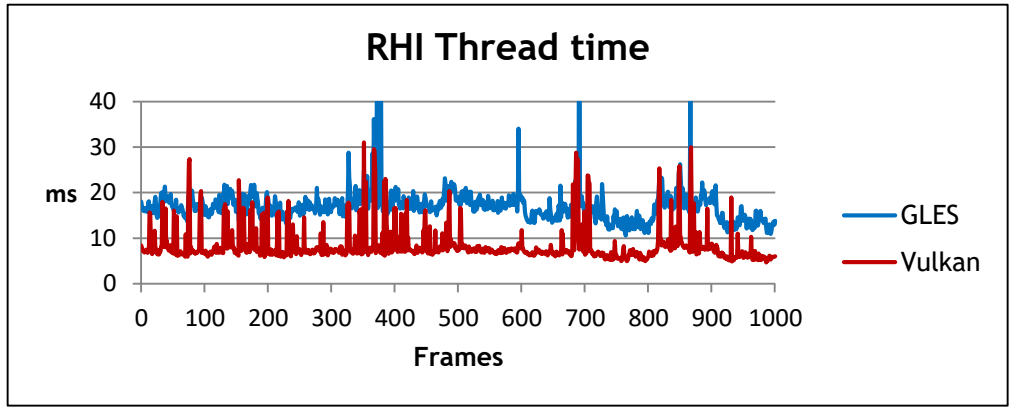
Vulkan

- Balanced CPU/GPU usage
- Lower CPU overhead
- Parallel tasking
- Explicit control
- No error checking at runtime

KHRONOS GROUP

Vulkan vs GLES





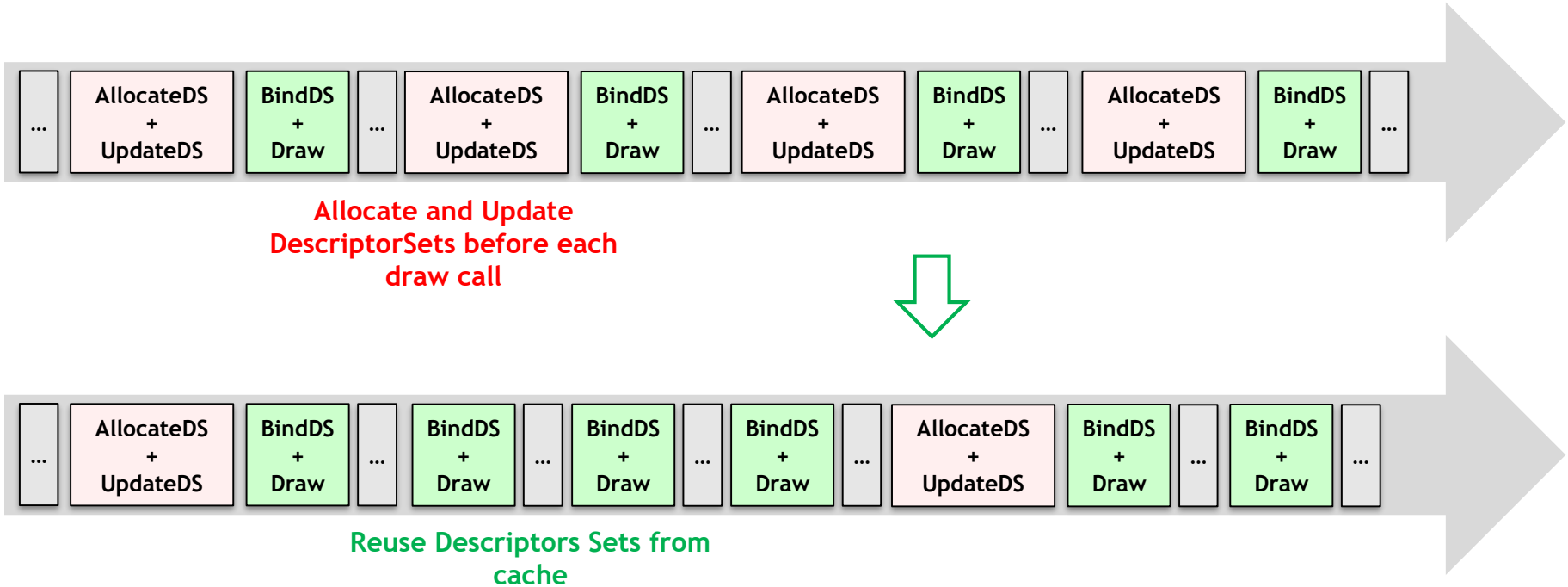
	GLES	Vulkan
FPS	39	47(+8)
RHI Thread Time (ms)	16.94	8.25(-51%)

Performance optimizations

- DescriptorSet cache
- Merge RenderPasses
- Remove useless barriers
- Remove extra depth copy
- Occlusion query
- Buffer upload

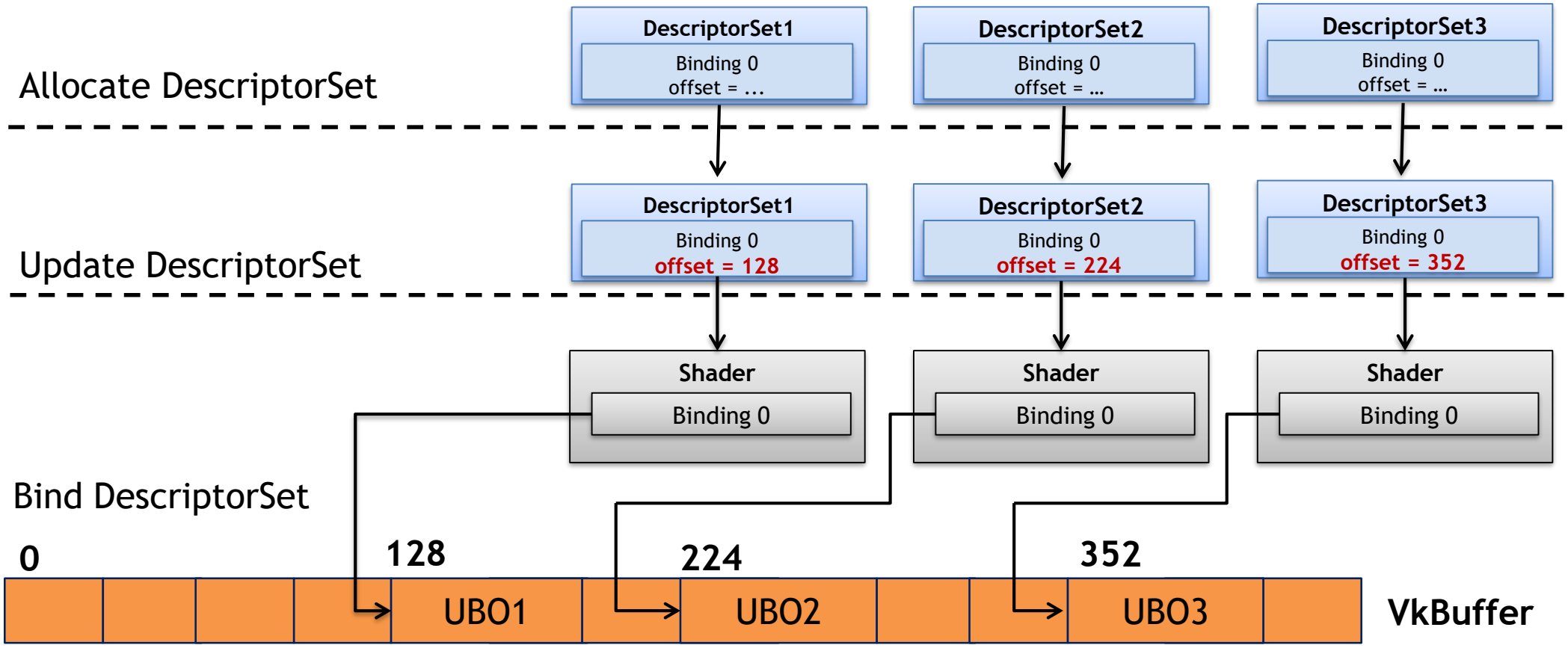
DescriptorSet cache

- Reuse already updated DescriptorSets



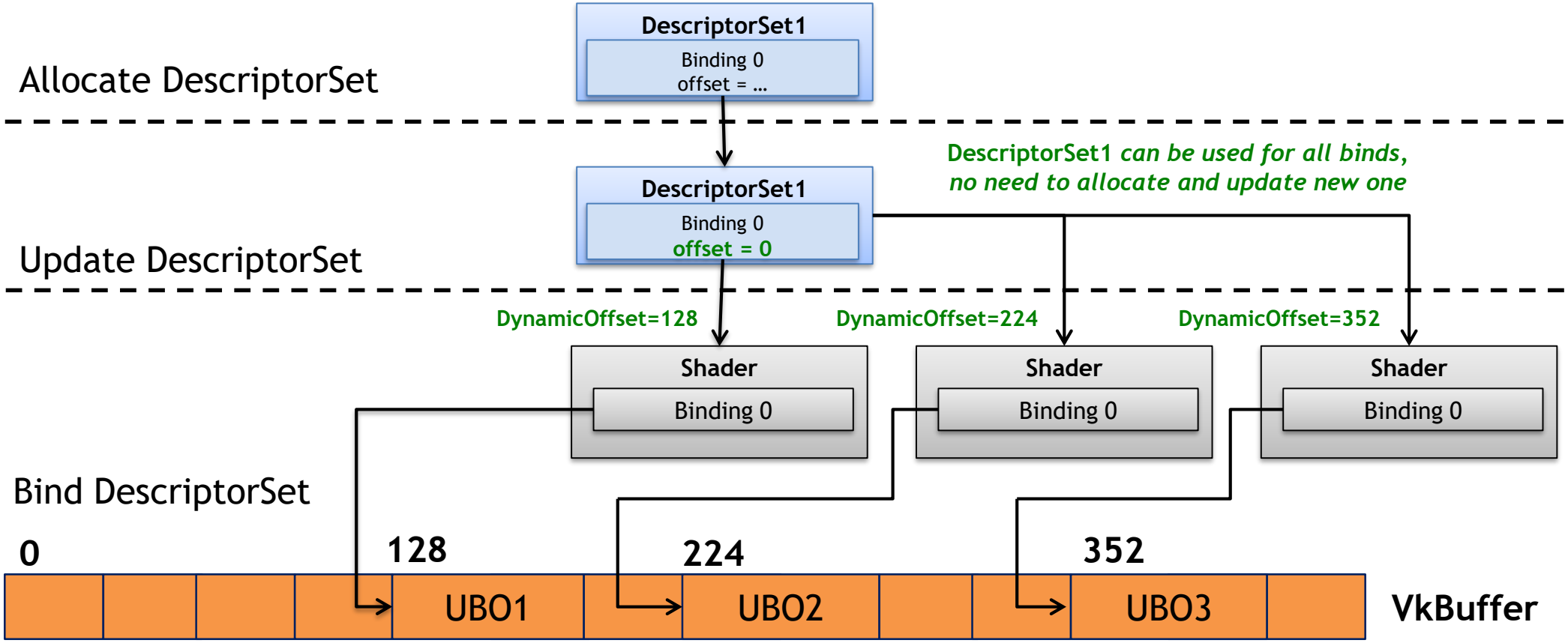
DescriptorSet cache

- There were a lot of cache misses due to storing buffer offset inside DescriptorSet



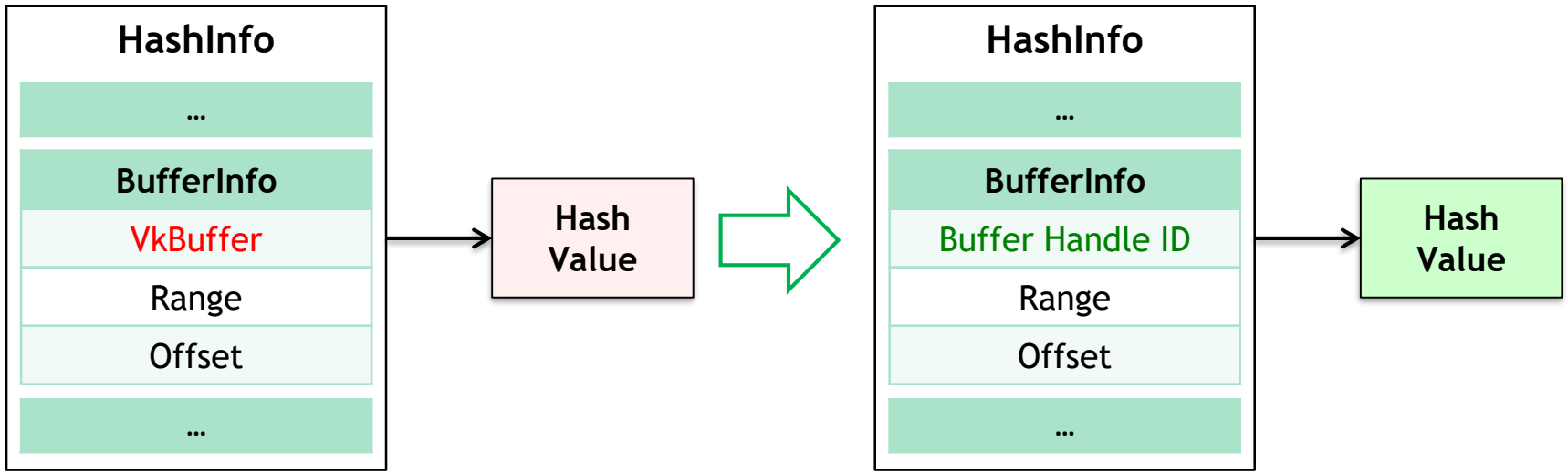
DescriptorSet cache

- Hit rate can be improved by using Dynamic Uniform Buffer

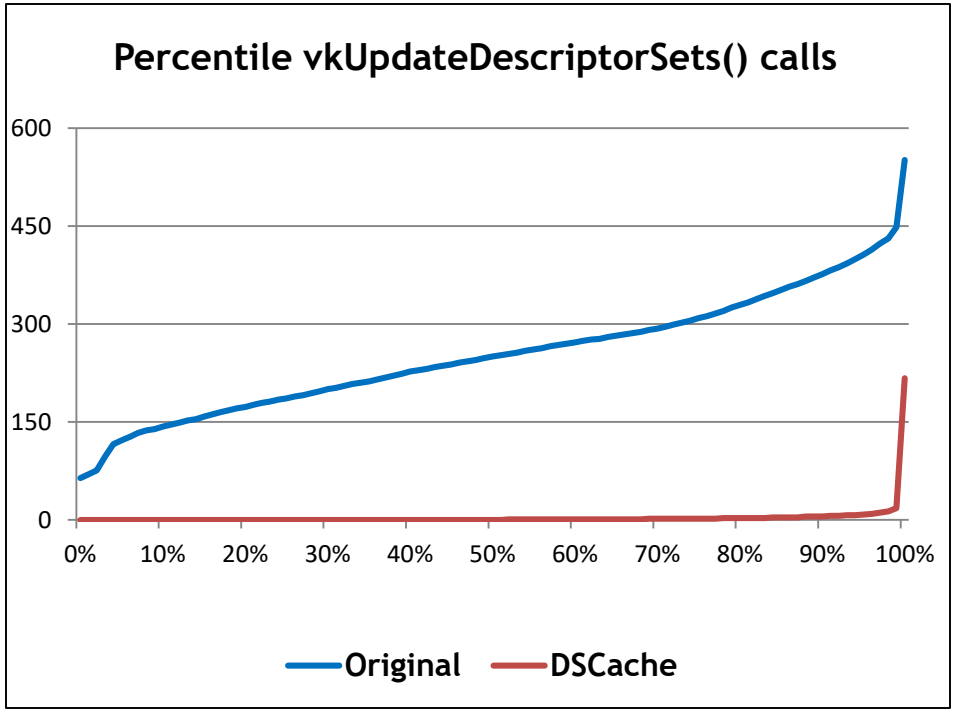


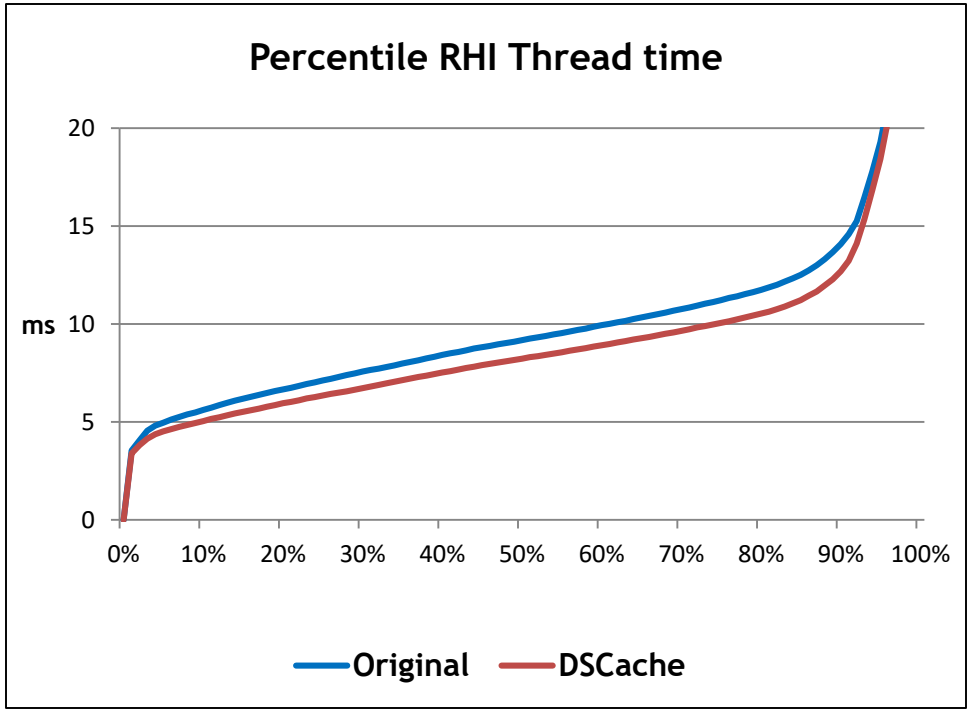
DescriptorSet cache

- **Do not use Vulkan Handle for hash calculation**
  - Vulkan can use same handles for different types
  - Vulkan can reuse handles from destroyed resources
- **Generate own Handle ID for all Vulkan resources and use it for hash calculation**



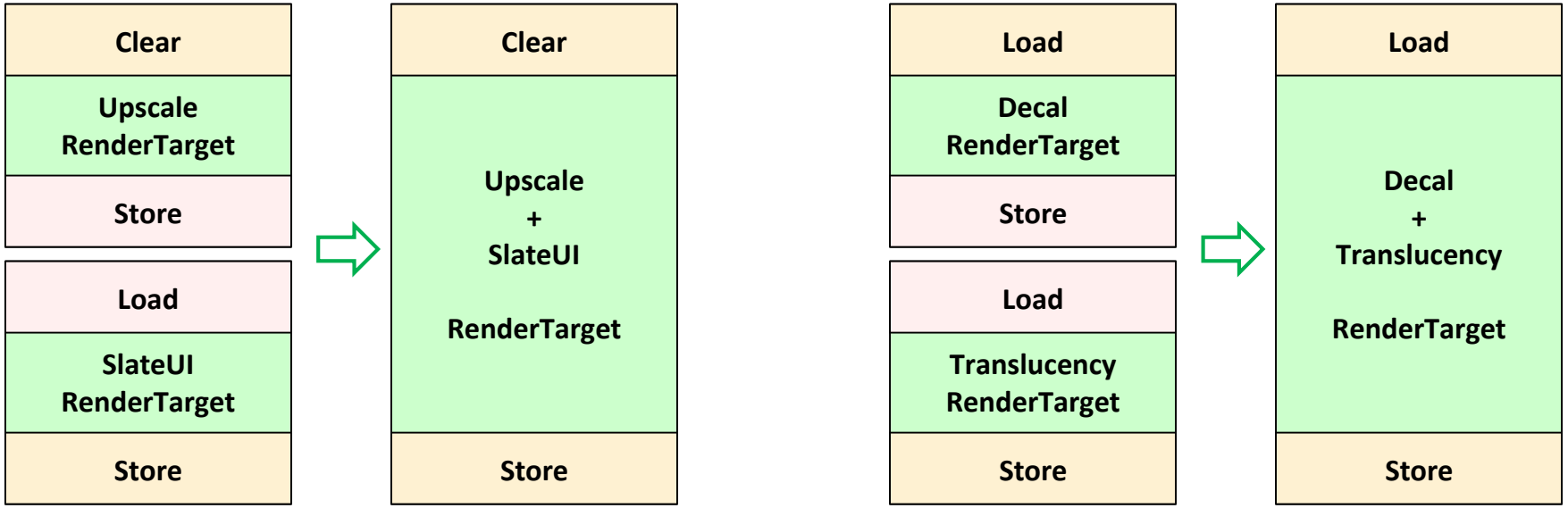
DescriptorSet cache



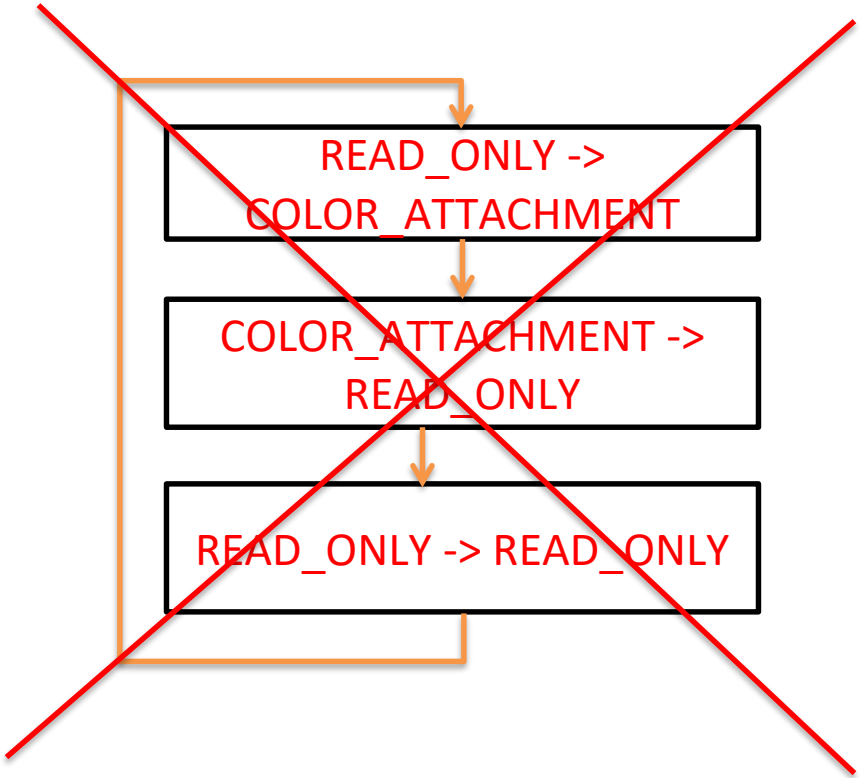


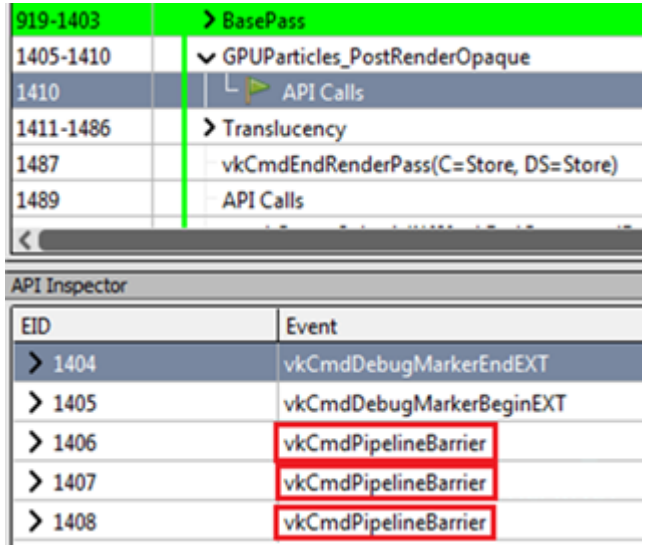
	Original	DSCache
Updates (avg calls per frame)	252	2(-99.2%)
RHI Thread Time Avg (ms)	10.12	9.15(-0.97)

Merge RenderPasses



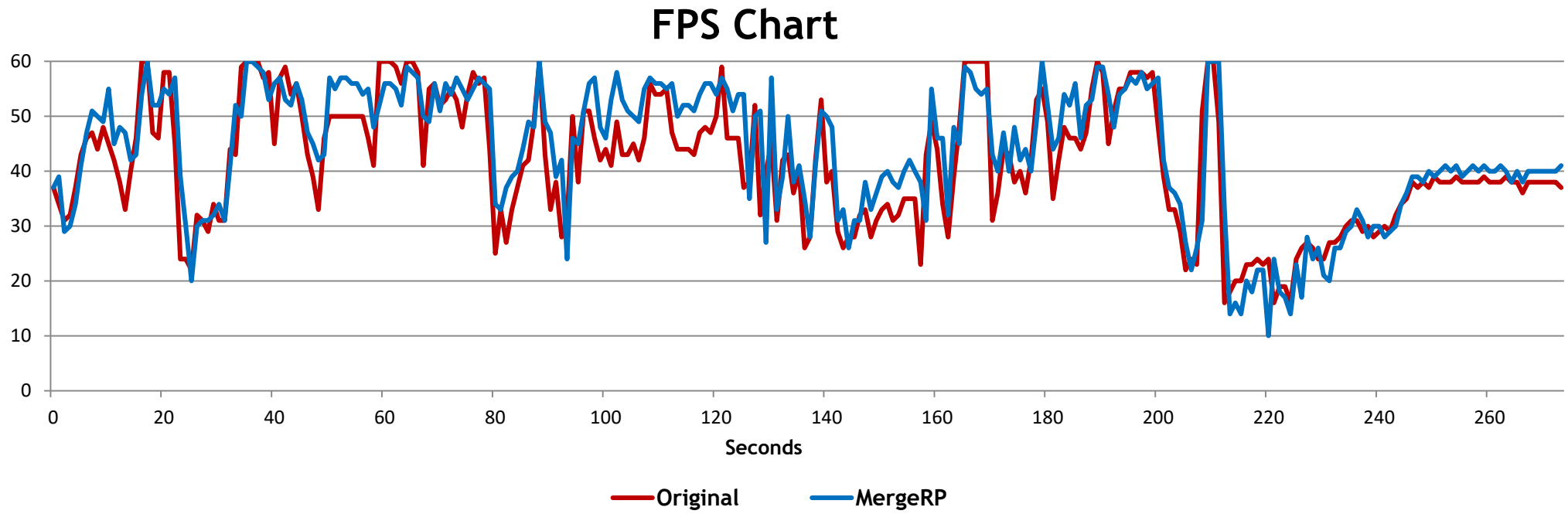
Remove useless barriers





RenderDoc capture

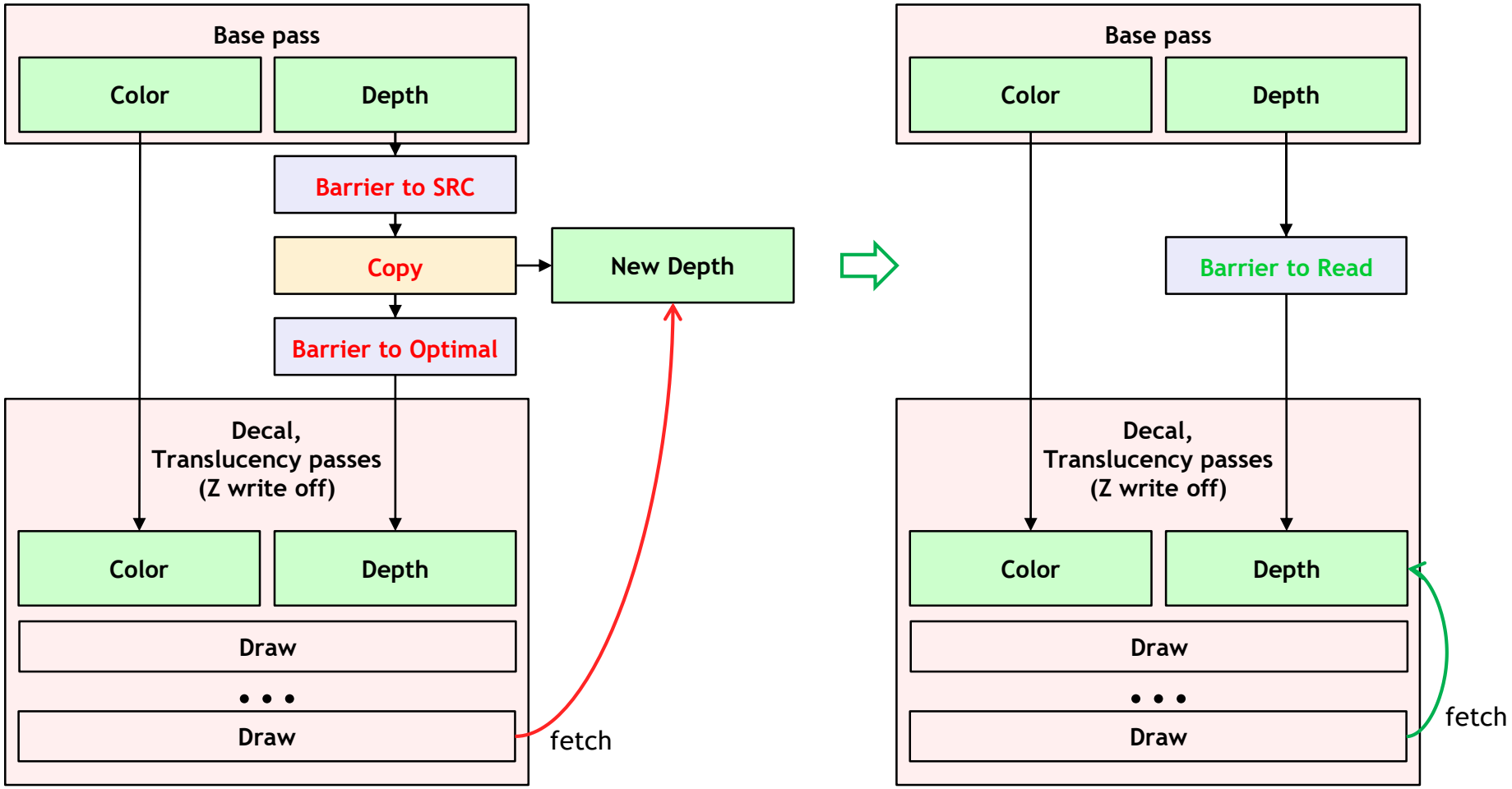
Merge RenderPasses/Remove extra barriers



Median FPS

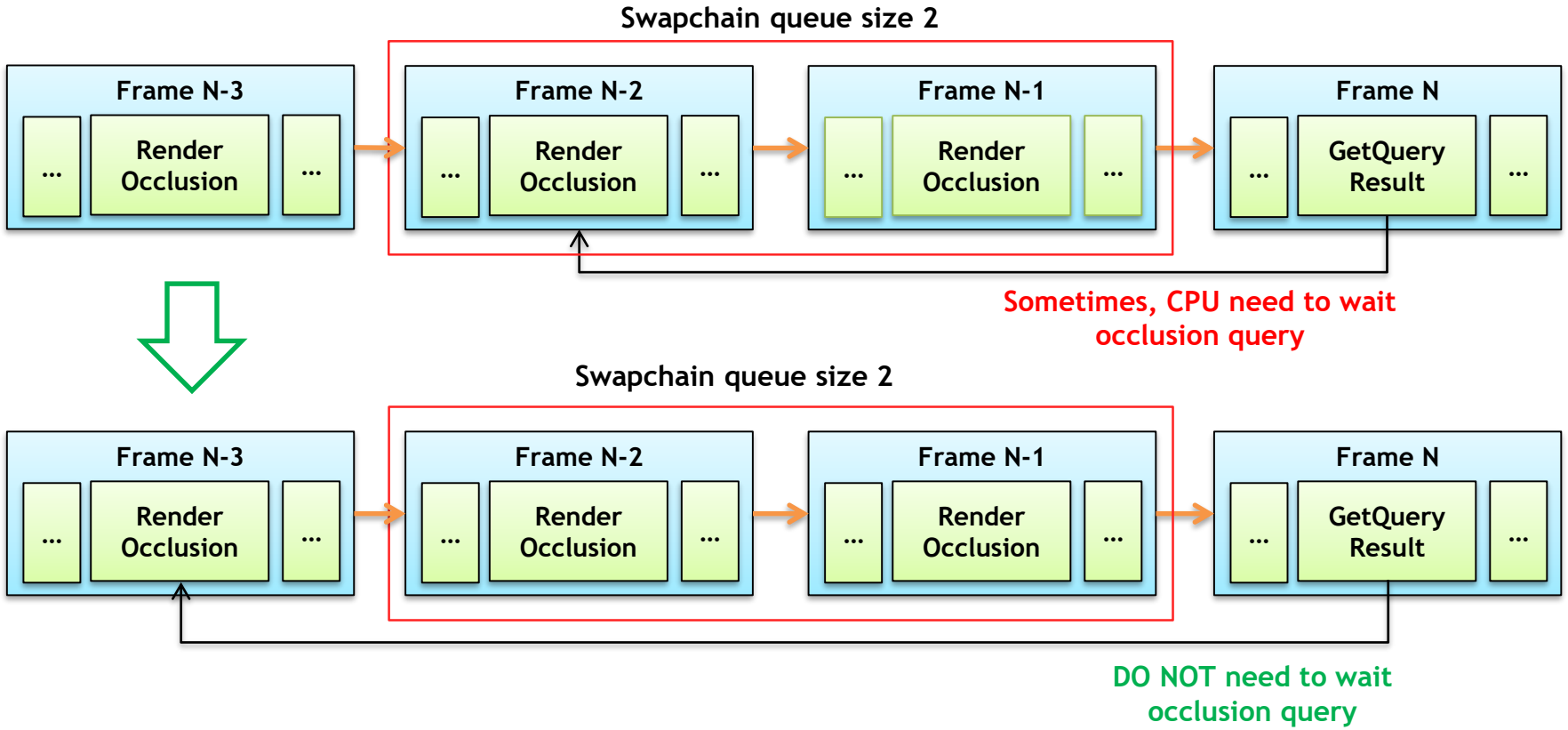
41 → 44

Remove extra depth copy



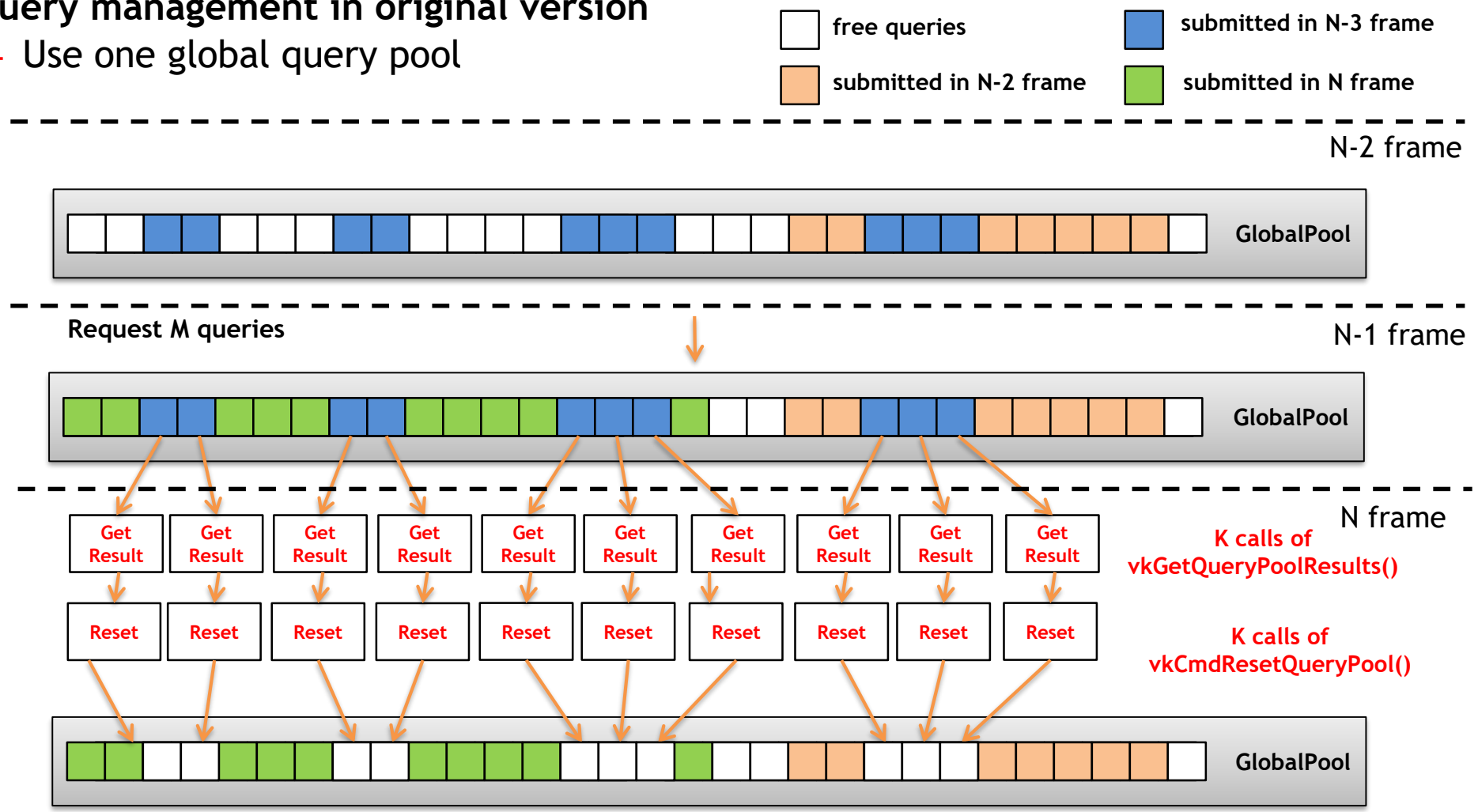
Occlusion query

- **Get occlusion query result for 3 frames back**
  - UE4 gets occlusion results for 2 frames back by default
  - 3 swapchain back buffers are used in Android, so sometimes waiting happens



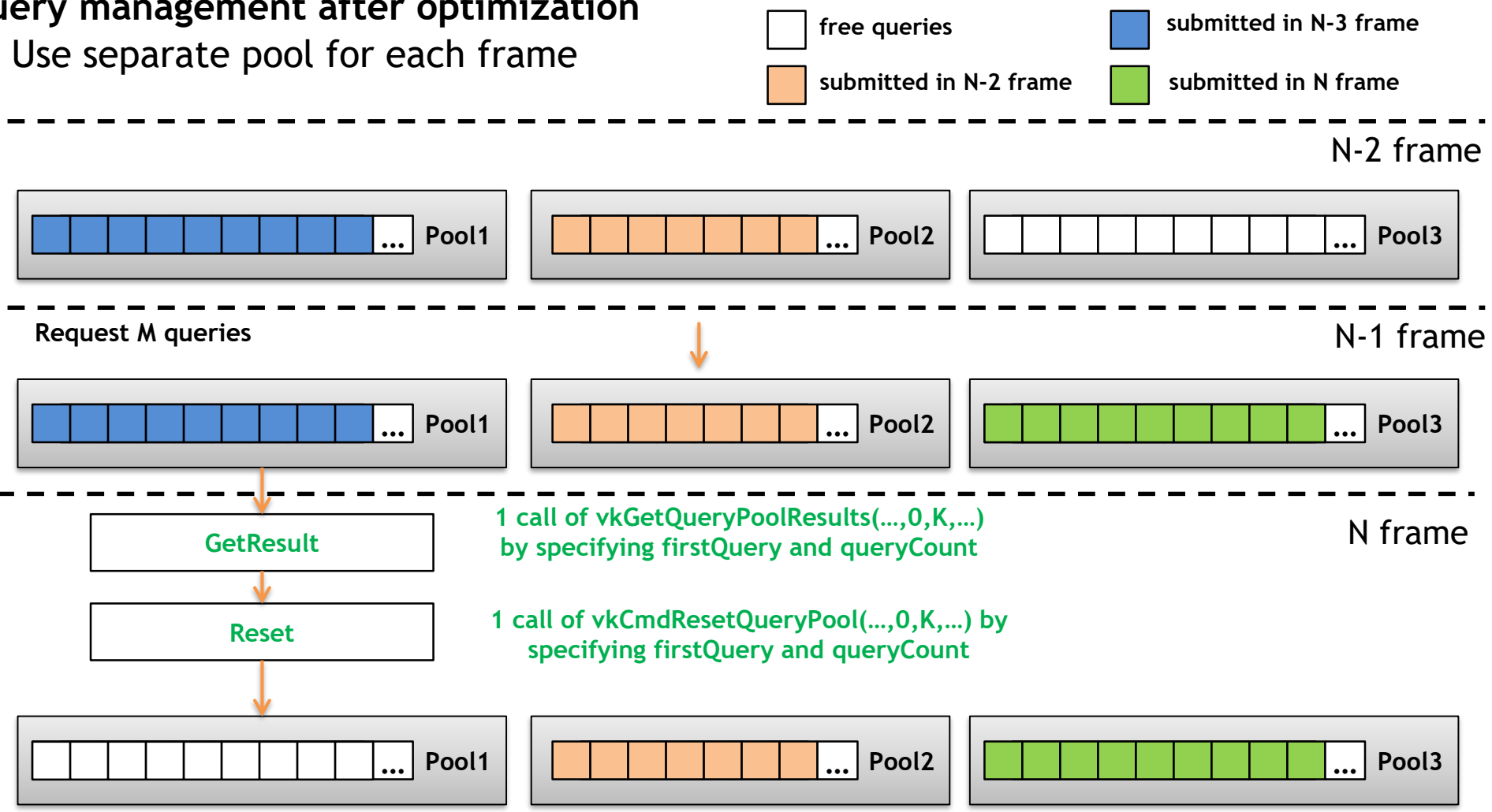
Occlusion query

- **Query management in original version**
  - Use one global query pool



Occlusion query

- **Query management after optimization**
  - Use separate pool for each frame



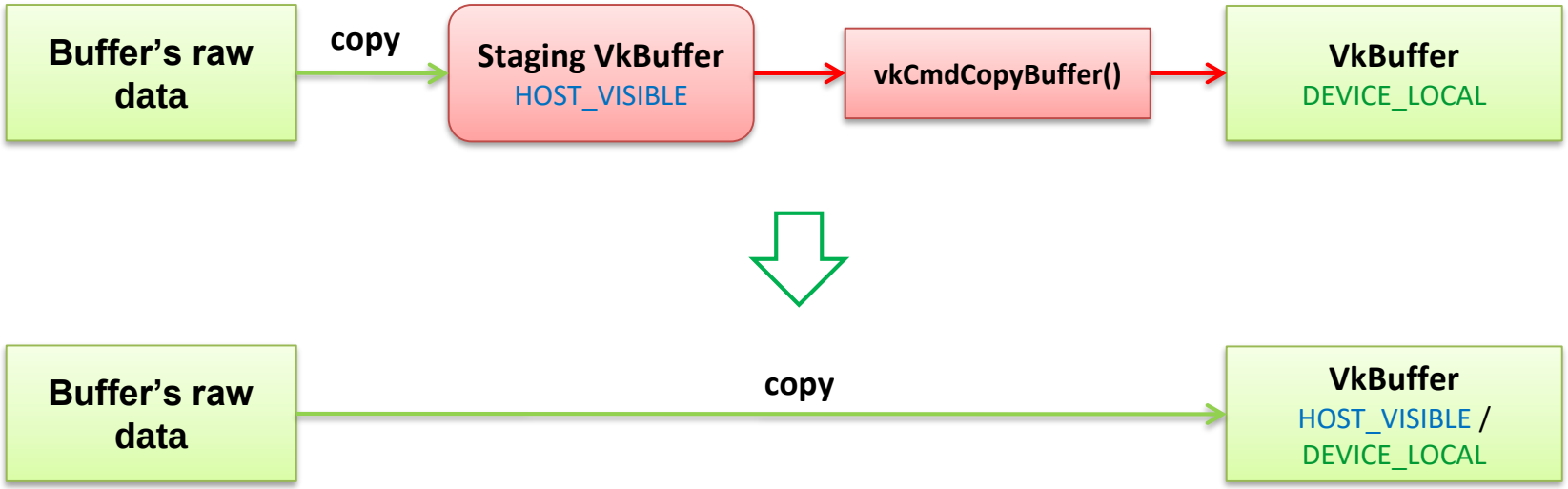
Occlusion query

- Performance measurement

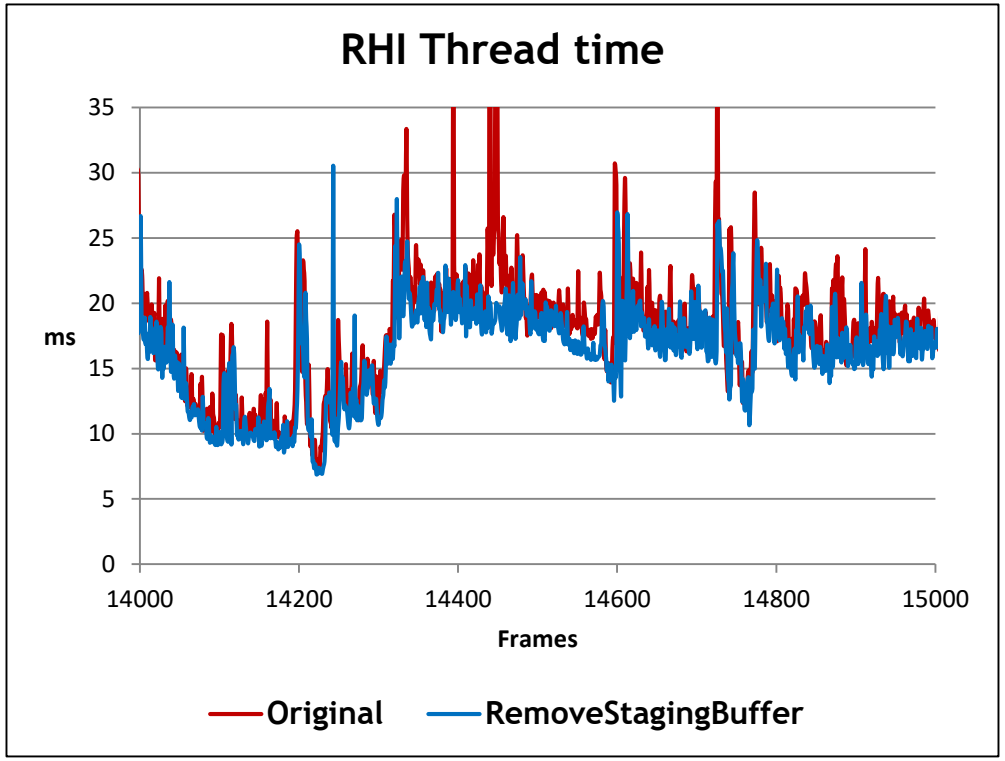
	Original	OcclusionQuery Optimization
Median FPS	27	29(+2)
FPS Stability	75%	90%(+15%)
CPU Usage	16.32%	15.55%
GPU Usage	70.90%	79.52%

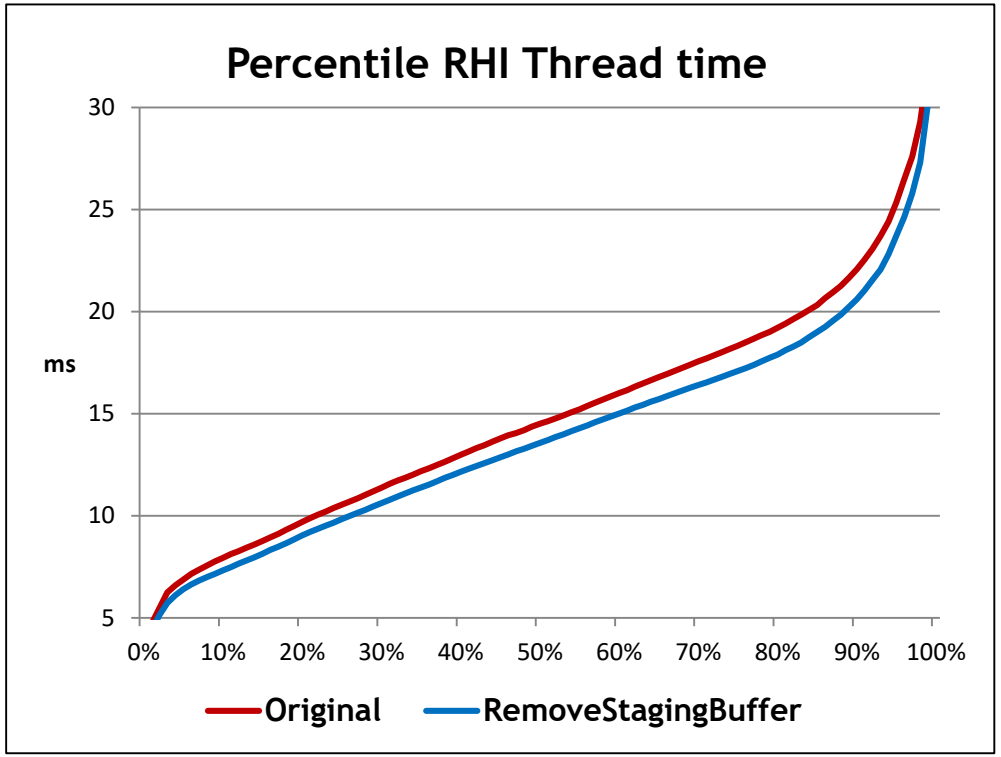
Buffer upload

- **Remove staging buffer usage**
  - Mobile GPUs usually have unified memory. Such memory allow direct host access
  - For mobile GPUs staging buffer is not needed and extra copying can be removed



Buffer upload





RHI Thread Time (avg)

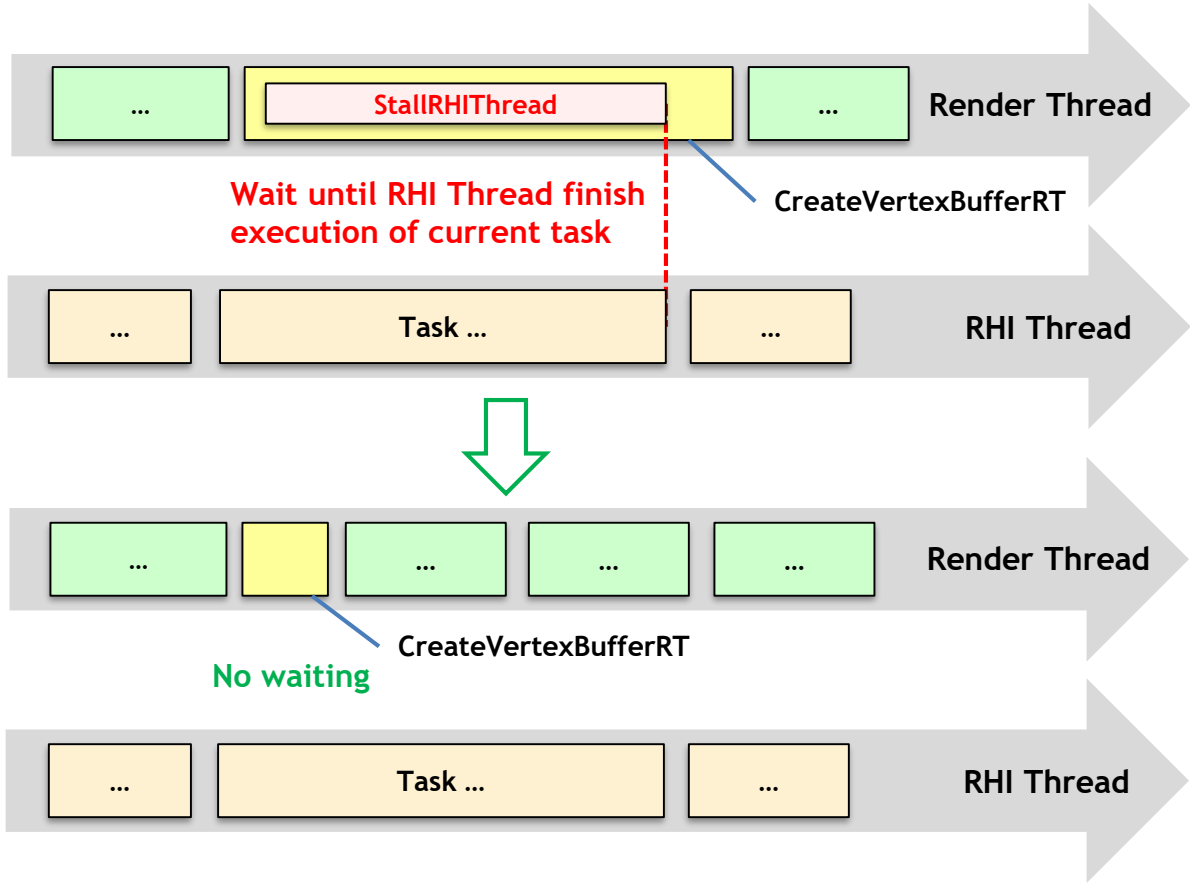
14.96 ms → 14.00 ms

Hitching/memory optimizations

- Asynchronous Vertex/Index buffer create
- Upload Texture
- DescriptorSetLayout cache miss
- Remove shader duplication
- Purge ShaderModules
- PSO cache miss

Asynchronous Vertex/Index buffer create

- **Allow asynchronous Vertex/Index buffer creation**
  - The basic versions of CreateVertex/IndexBuffer() use needless RHI Thread stall
  - Vulkan RHI allows asynchronous Vertex/Index buffer creation

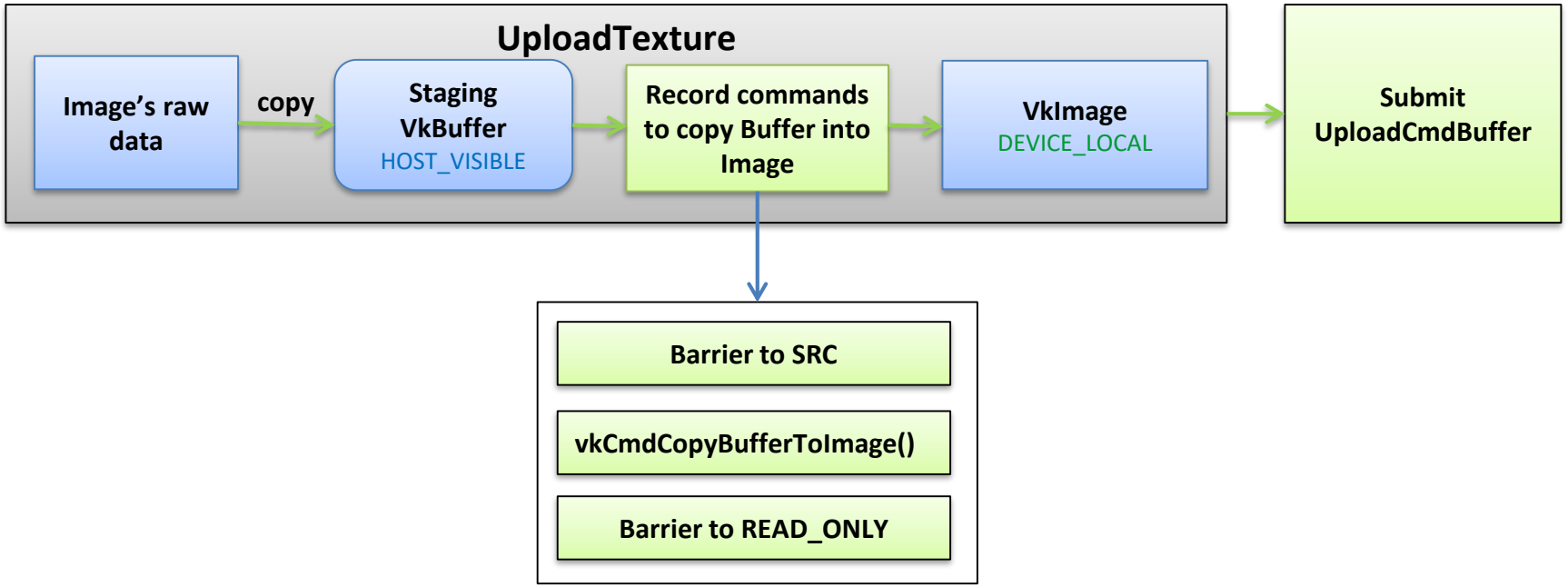


Number of Hitches

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

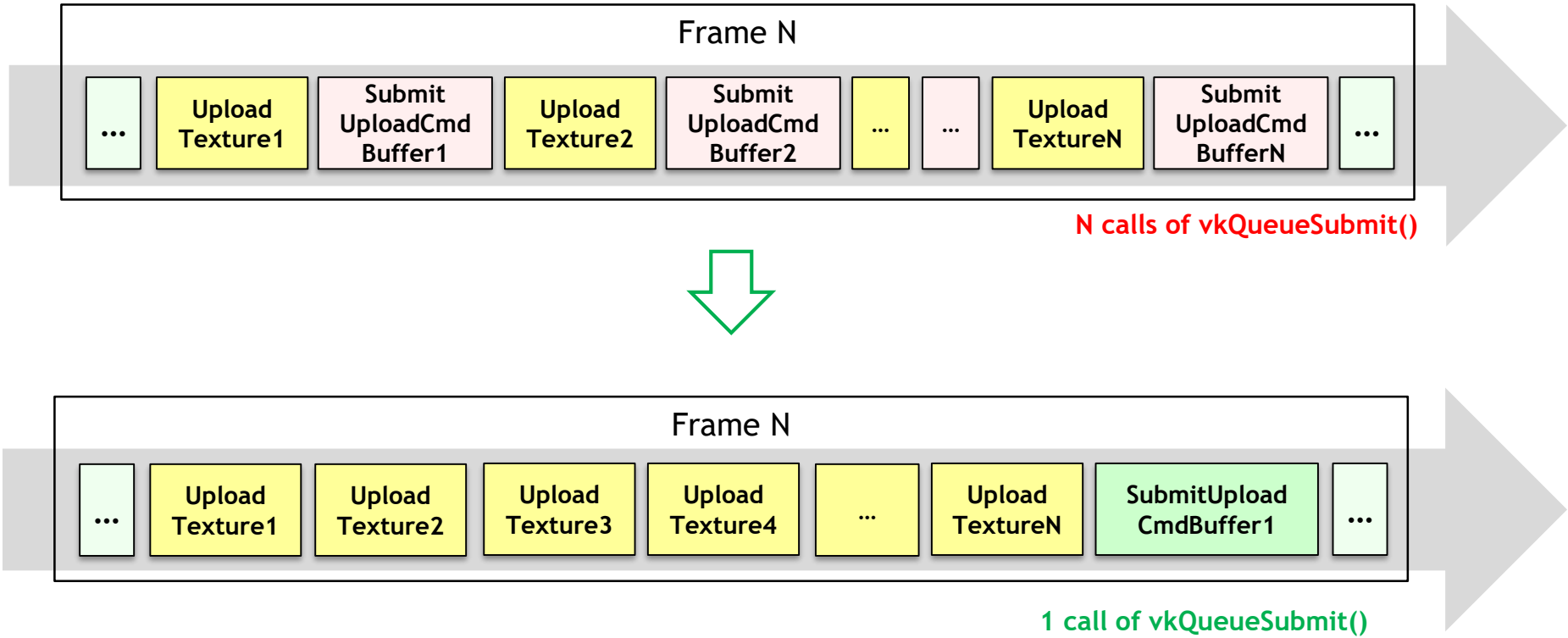
Upload Texture

- Texture uploading process:

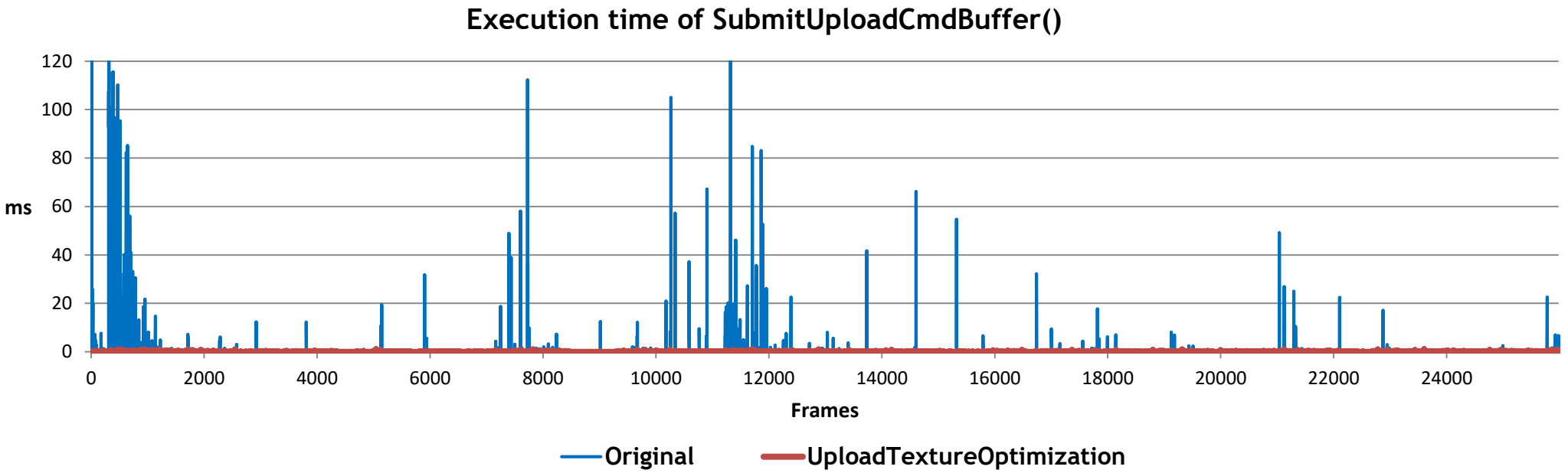


Upload Texture

- Record texture upload commands into one command buffer



Upload Texture

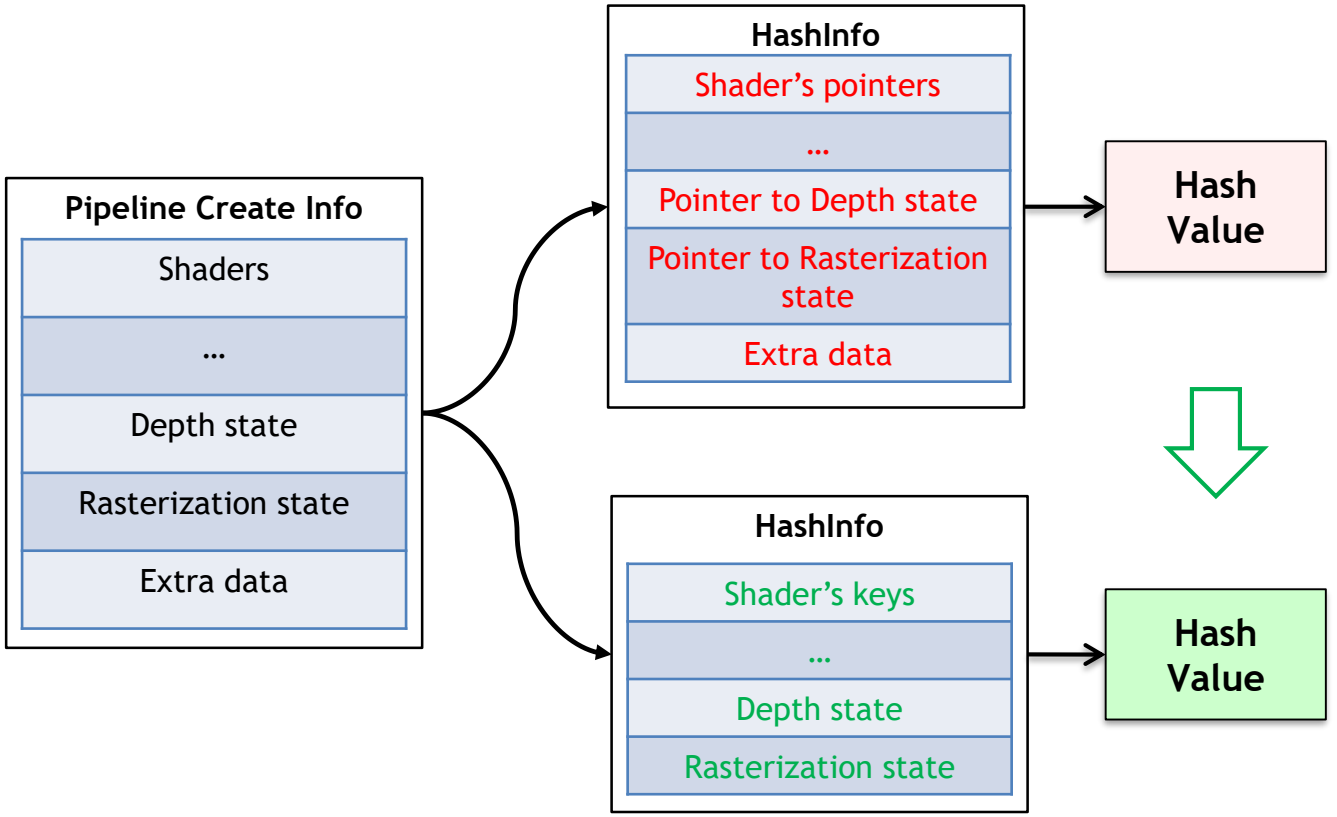


Number of Hitches

	Original	UploadTexture Optimization
Frame Time	89	40(-49)
Render Thread Time	16	8(-8)
RHI Thread Time	38	4(-34)

PSO cache miss

- Do not hash pointers
- Hash only info which is used for Pipeline creation
- Calculate shader's key from ShaderCode and use it for hashing



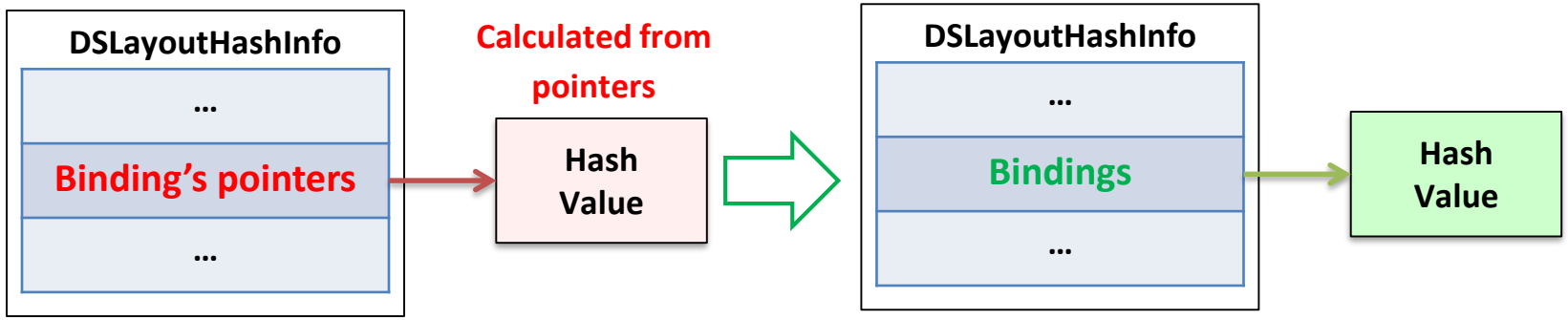
Calculated from pointers and extra data

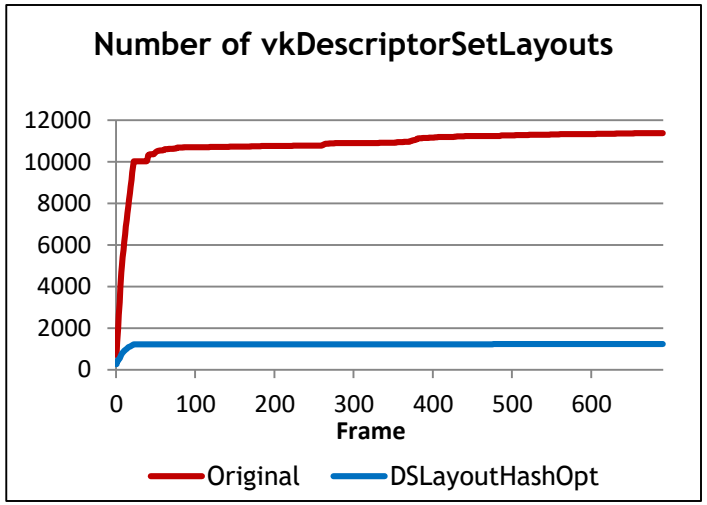
Cache hit rate

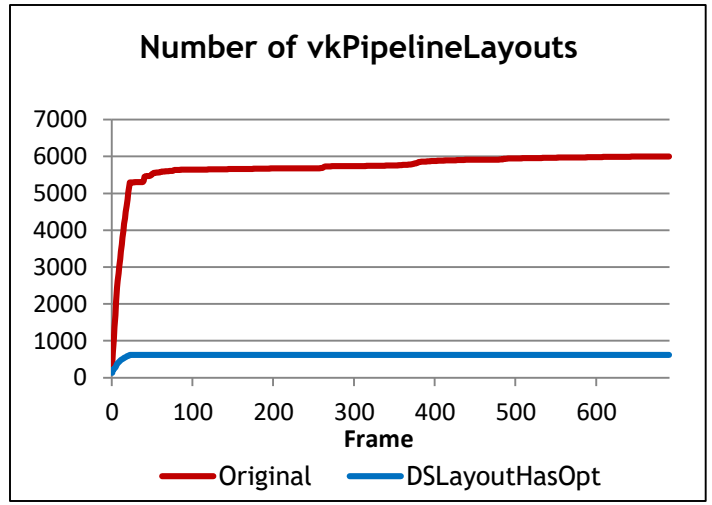
	Before	After
1 session	33.5%	34.9%
2 session	77.4%	95.4%
3 session	80.8%	99.6%

DescriptorSetLayout cache miss

- Hash data instead of pointers

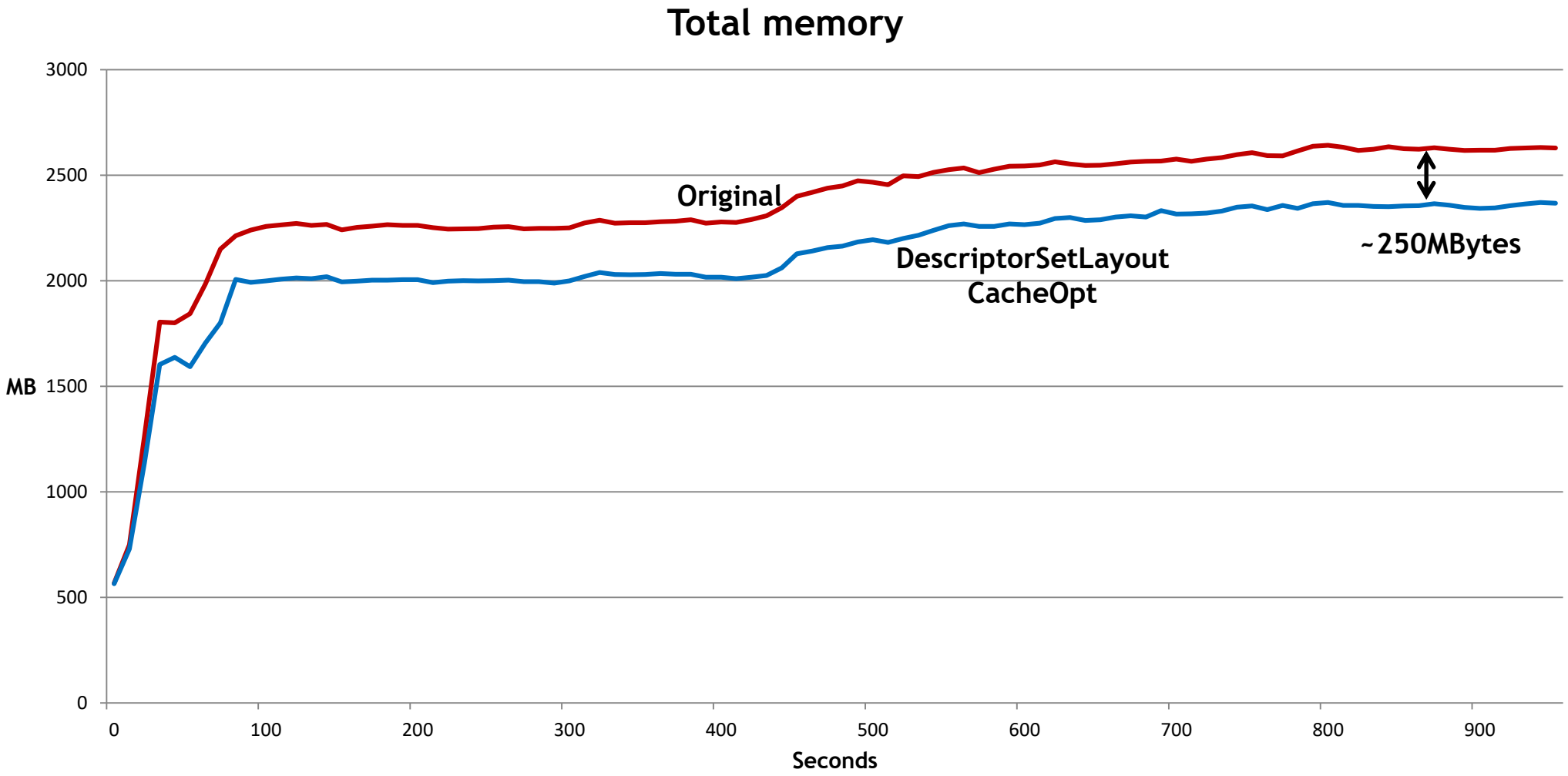






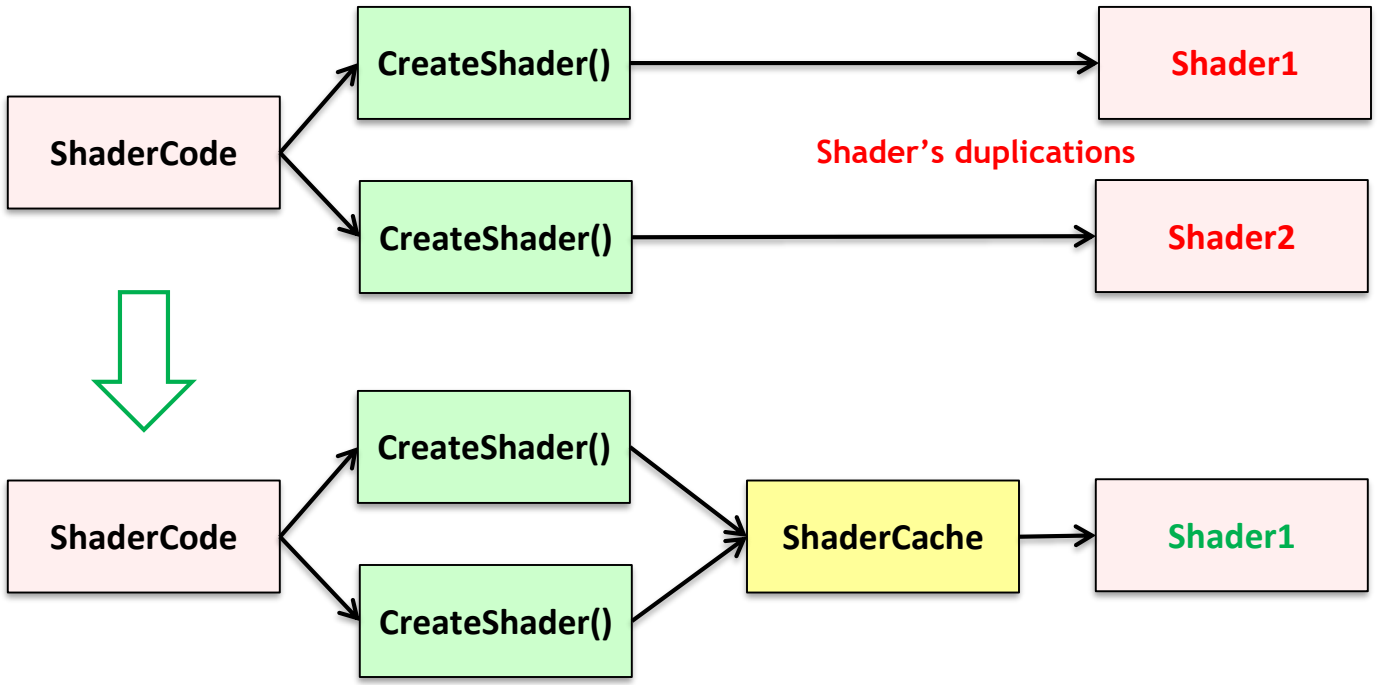
	Original	DSLayout HashOpt
vkDescriptor SetLayout	11379	1231 (-89%)
vkPipeline Layout	5999	621 (-90%)

DescriptorSetLayout cache miss

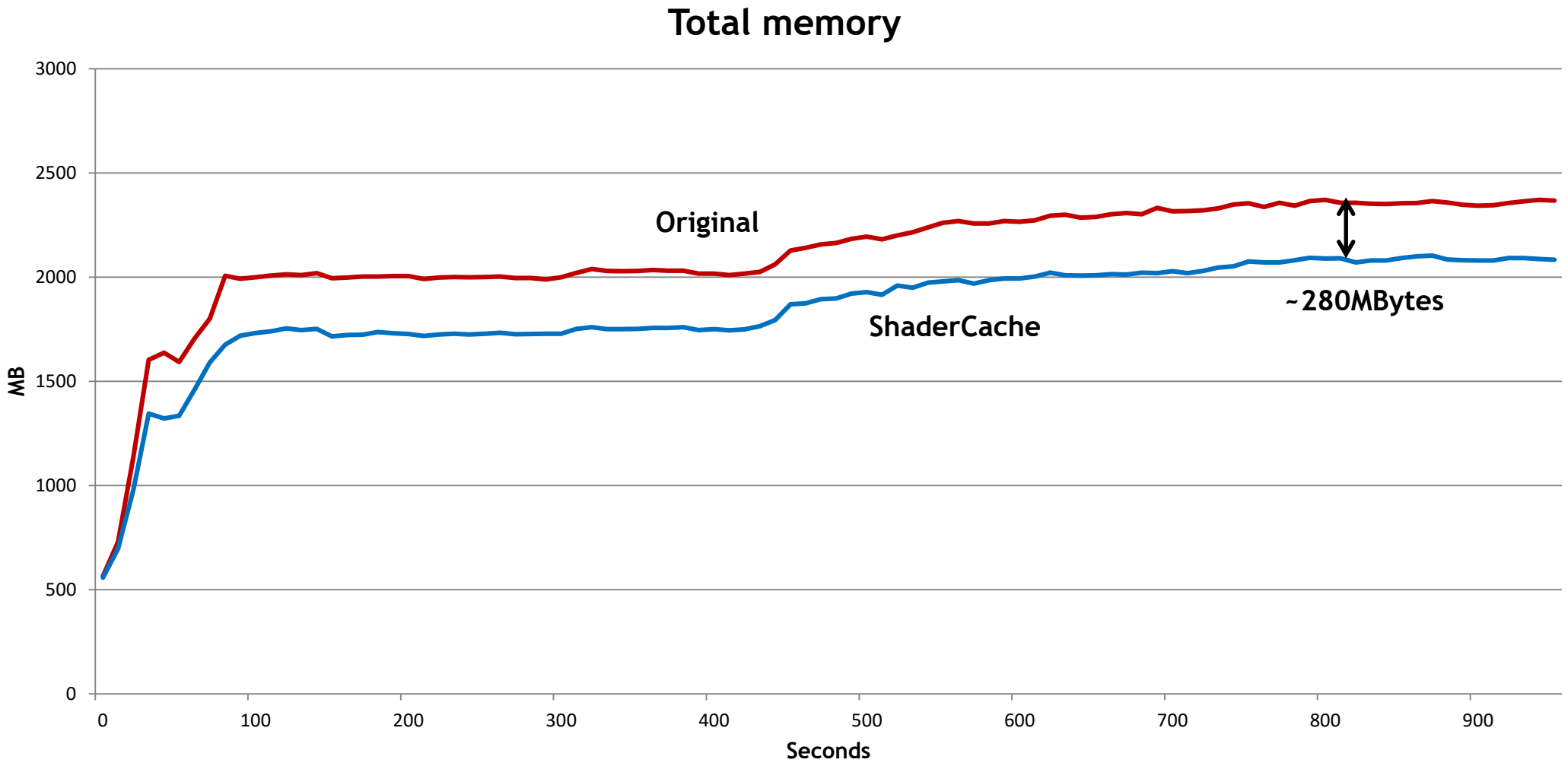


Remove shader duplication

- Add shader cache

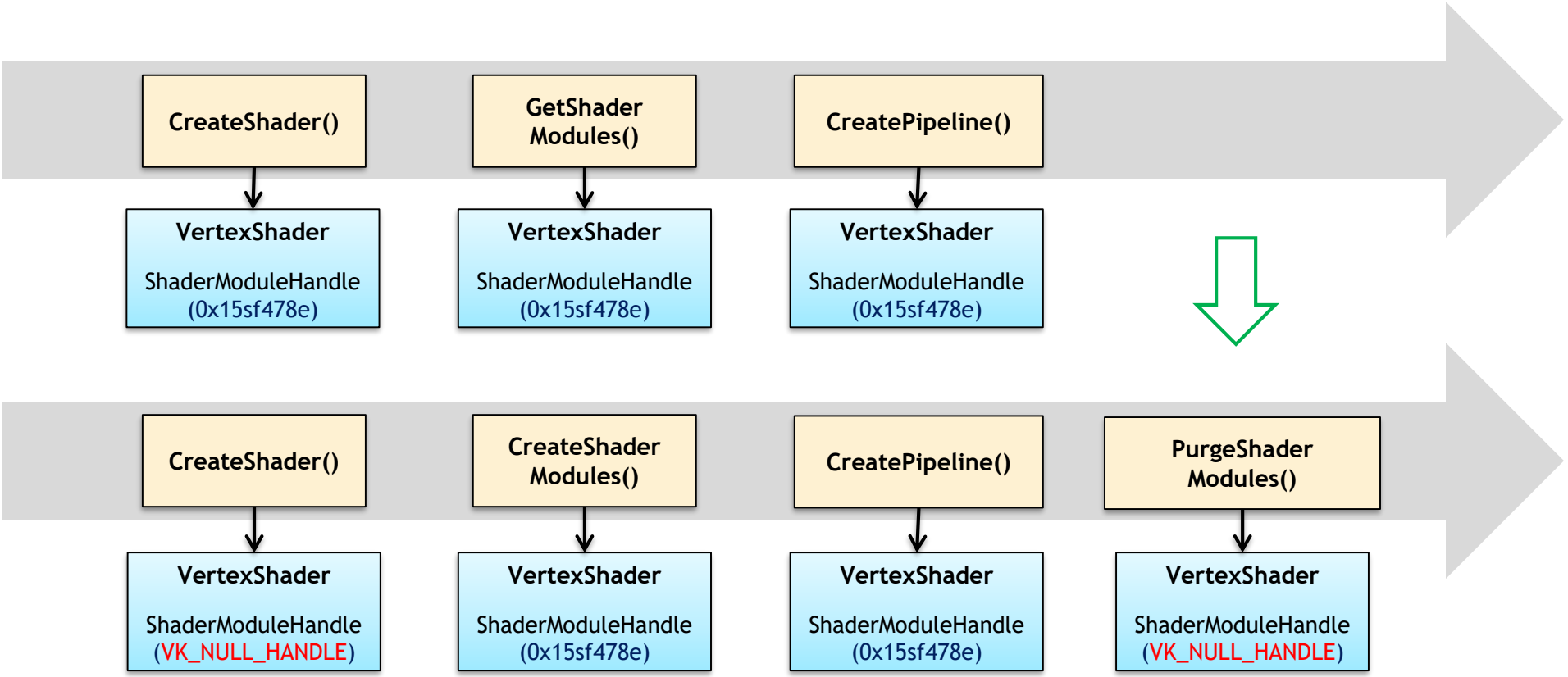


Remove shader duplication

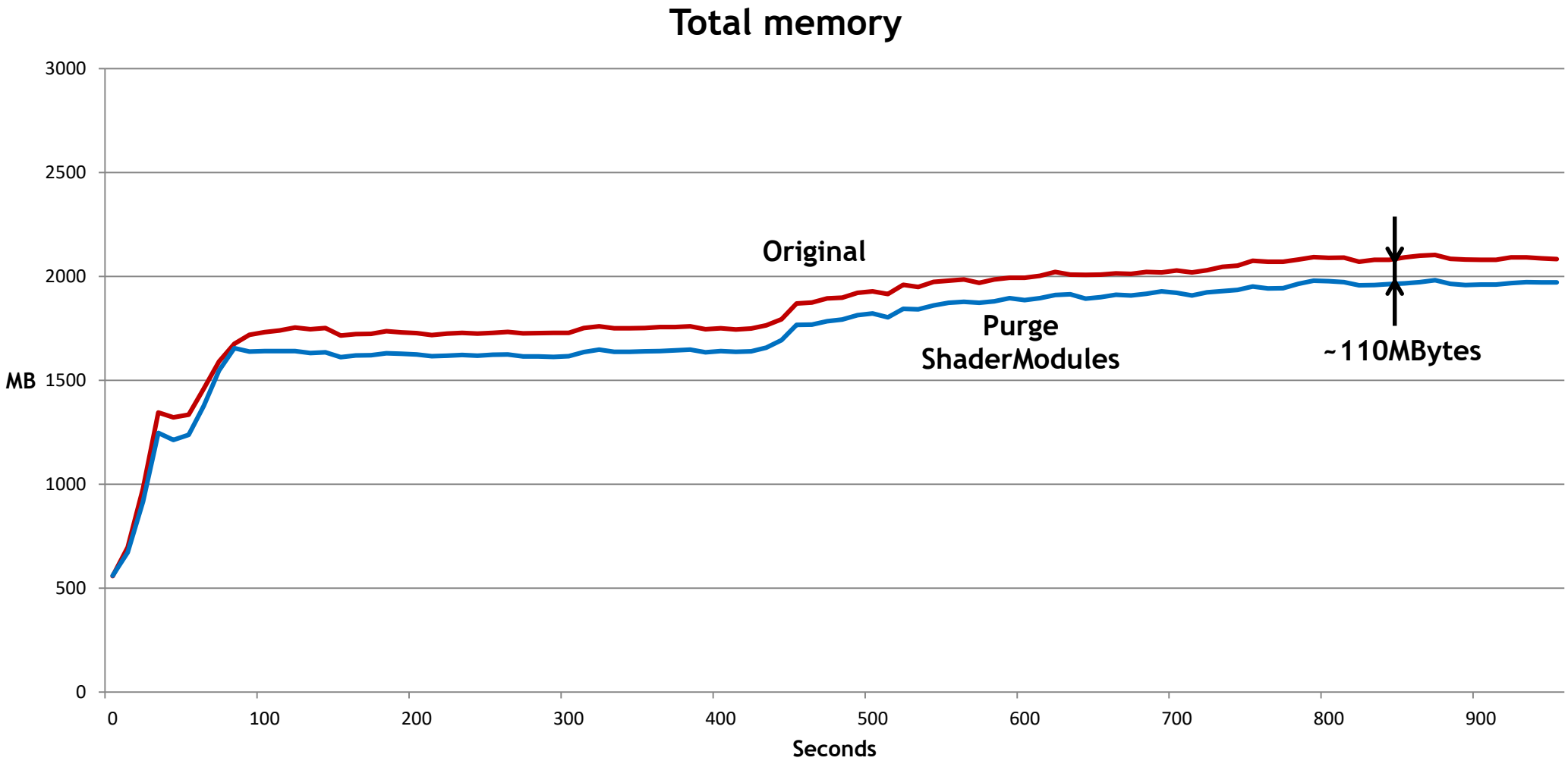


Purge ShaderModules

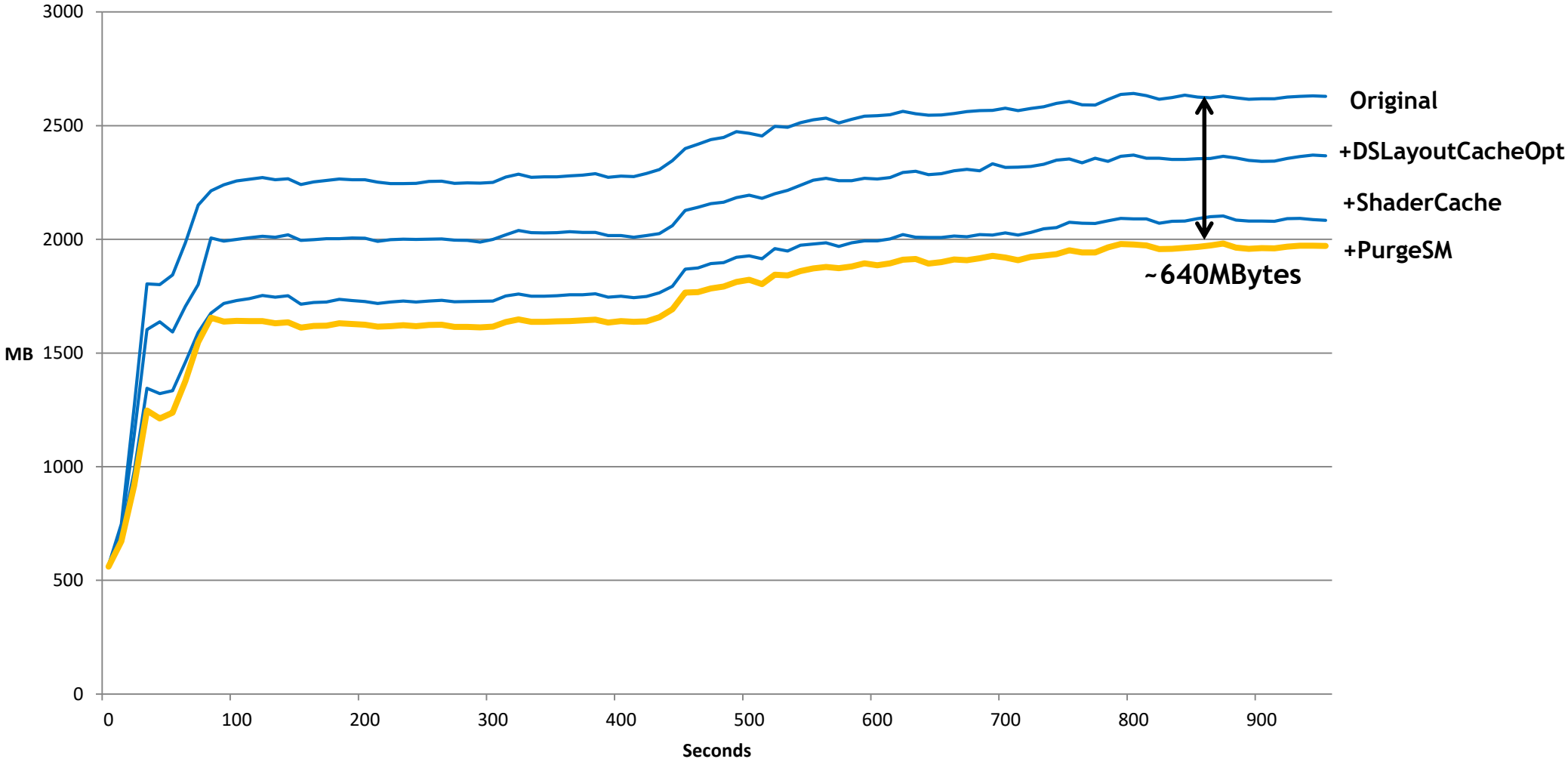
- Don’t create shader module at shader creation time
- Create shader modules before pipeline creation and destroy after



Purge ShaderModules



Total Memory Usage



Thank you!

Jack Porter
Epic Games

https://unrealengine.com
jack.porter@epicgames.com

Kostiantyn Drabeniuk
Samsung Galaxy GameDev

https://developer.samsung.com/game
k.drabeniuk@samsung.com
gamedev@samsung.com