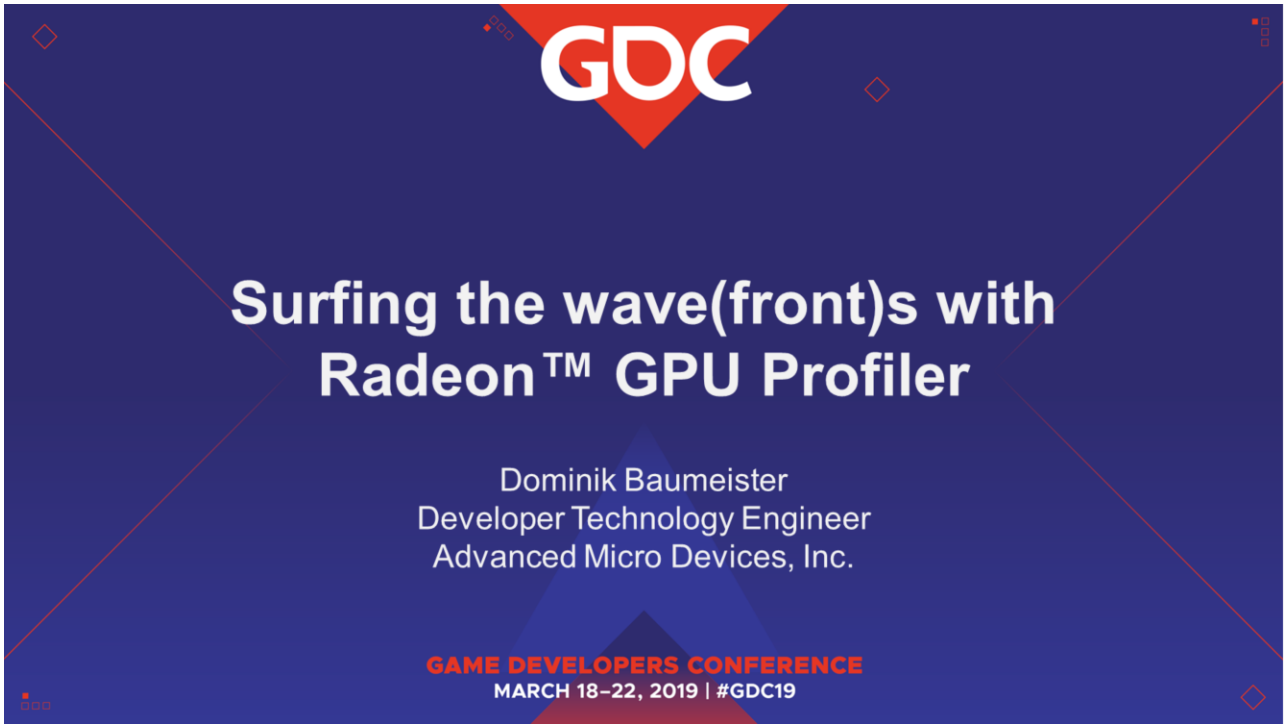

GDC
Surfing the wave(front)s with Radeon™ GPU Profiler
Dominik Baumeister
Developer Technology Engineer
Advanced Micro Devices, Inc.
GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

Overview

- Wavefronts & Barriers
- Cross queue synchronization
- Compression
- Wrap-up

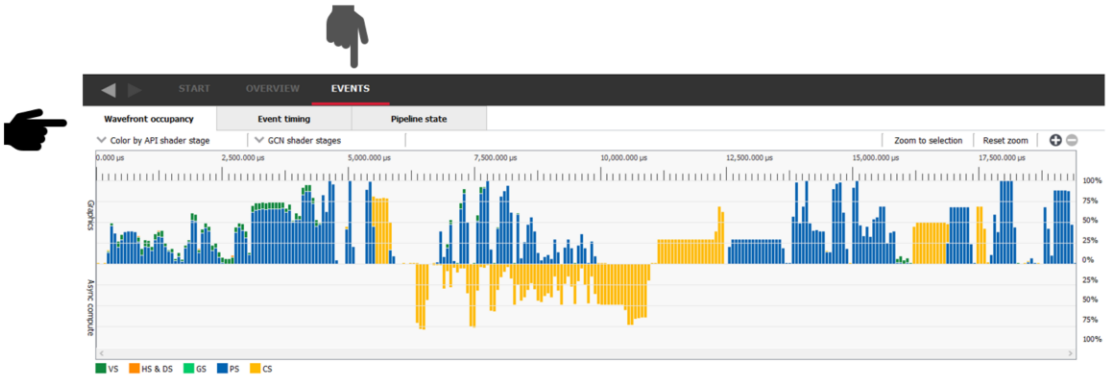
Now this is how the talk will look like:

- I present you with a problem that has been seen in a looooooot of AAA titles. Both shipped and upcoming ones.
  I'm not going to say which titles these are, don't want to offend anyone and sometimes you have to call a solution "good enough". People want to ship games at the end of the day.
- I'll show you how to use RGP to

identify the problem

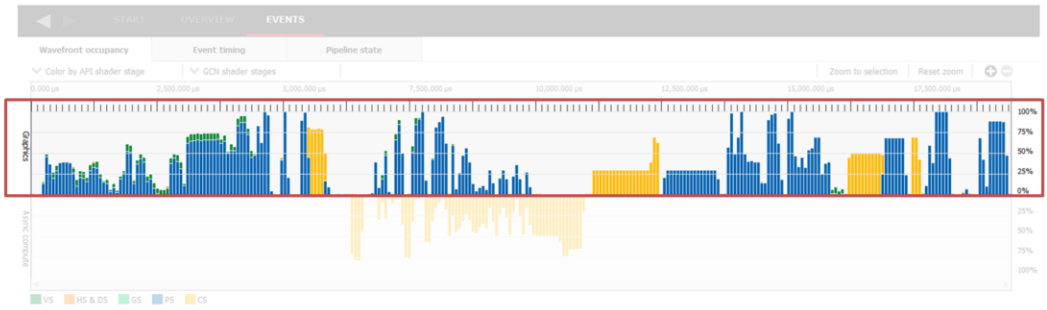
- I’ll explain the underlying issue
- I’ll tell you how to fix it
- For the following three topics: Barriers, Async, Compression

All of the information given in this talk is about Vega.



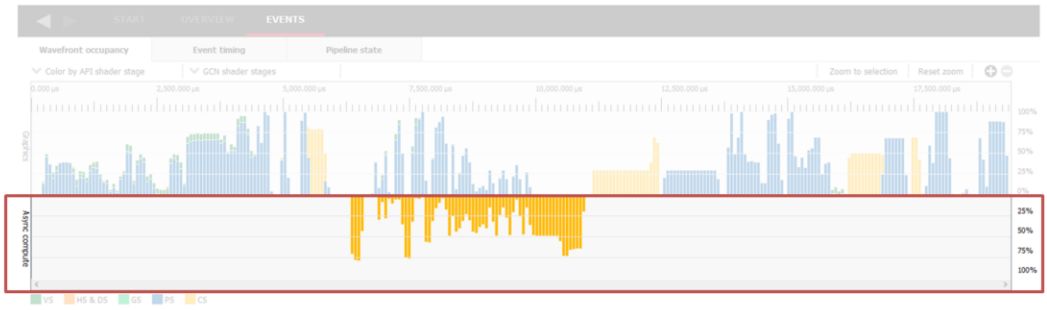
You can access this via Events->Wavefront occupancy.

This is probably the view you will spent your first few minutes on to figure out what’s going on in your frame.



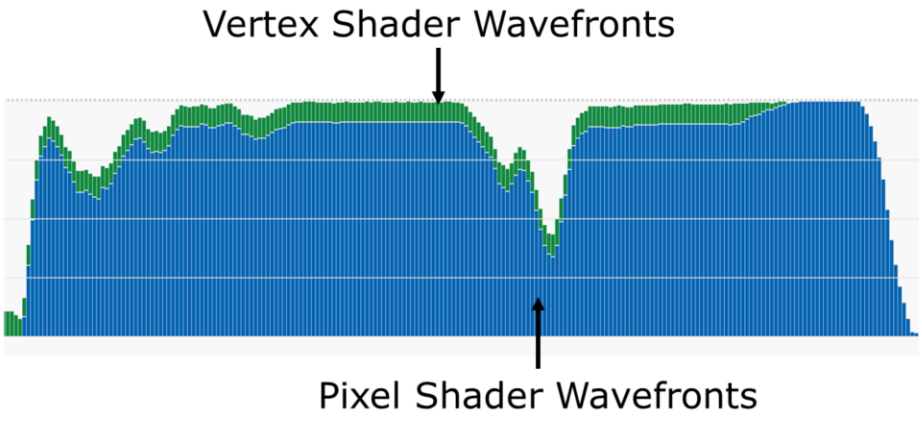
You can access this via Events->Wavefront occupancy.

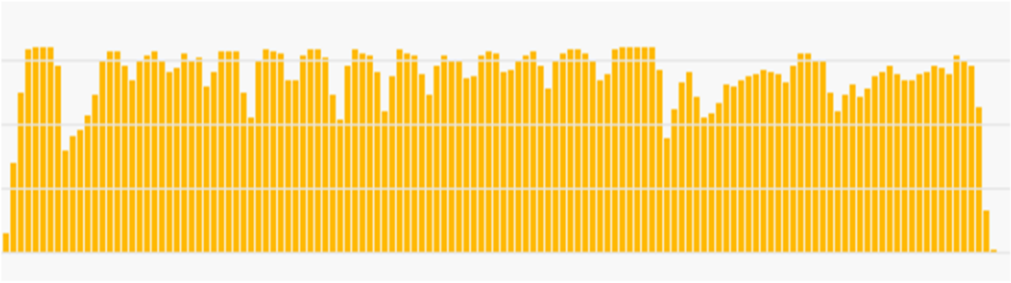
This is probably the view you will spent your first few minutes on to figure out what’s going on in your frame.



You can access this via Events->Wavefront occupancy.

This is probably the view you will spent your first few minutes on to figure out what's going on in your frame.


Vertex Shader Wavefronts
Pixel Shader Wavefronts



Compute Shader Wavefronts

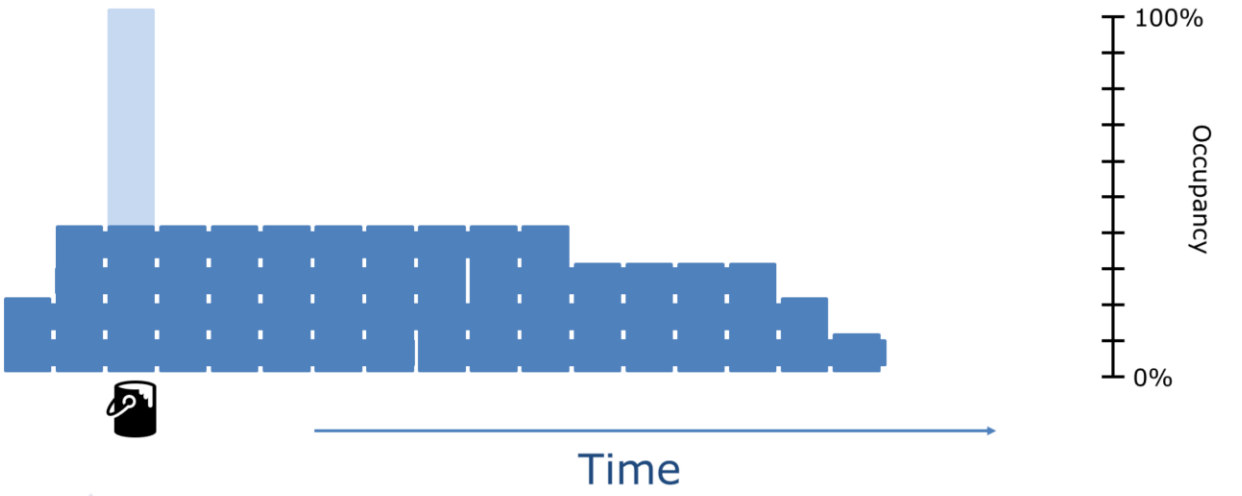
Wavefront

Time

May I introduce: a wavefront.

Your dispatch/draw is split up into multiple wavefronts. Each executing your shader program with multiple threads in lockstep.

Once a wavefront finishes executing the shader program its HW resources (registers etc) are freed and new wavefronts can spawn.



RGP puts them into buckets of a certain duration and counts the total number of wavefronts that run on the GPU during the time slice of each single bucket.

Since there are different wavefront counts on each GPU family, the filling is normalized to the maximum amount of wavefronts.

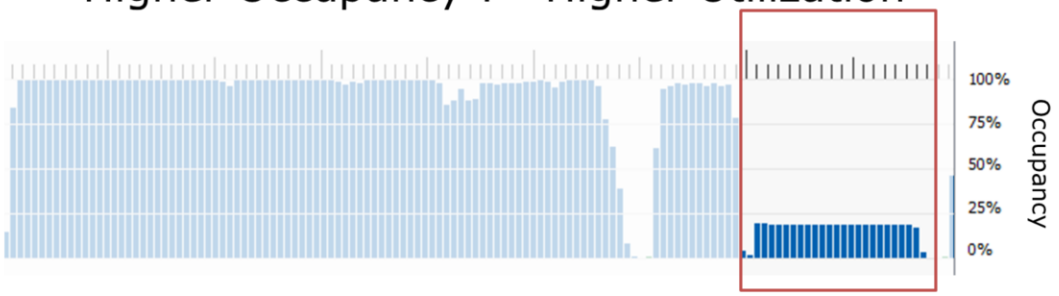
If our hypothetical GPU could run 10 wavefronts at once, RGP would state

40% occupancy here.

(real numbers are in the thousands)

And for completeness sake, this is how RGP would generate the trace for our hypothetical GPU.

Higher Occupancy != Higher Utilization

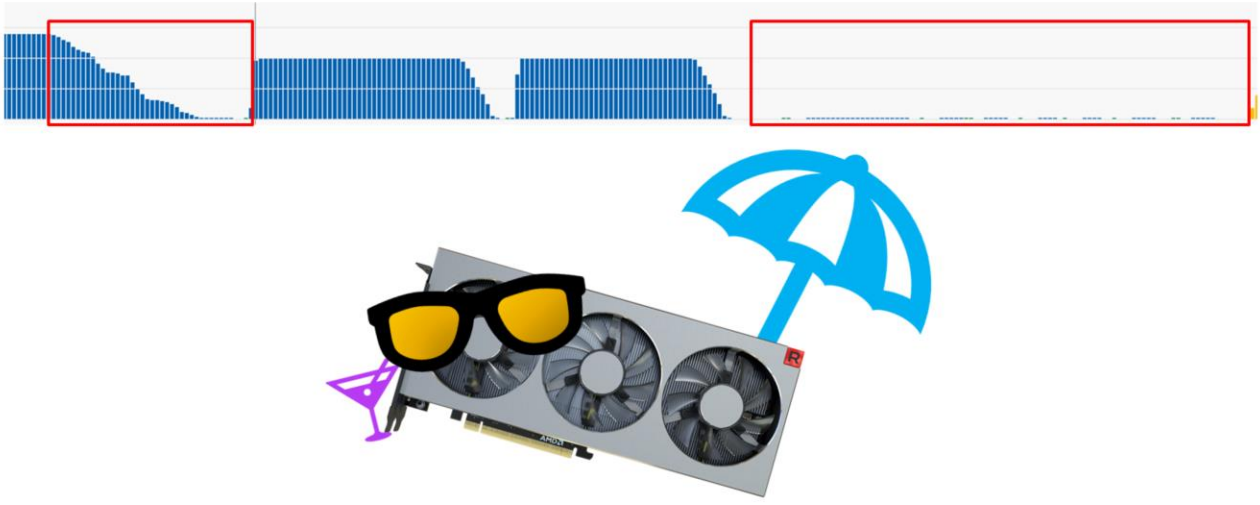


Remember: Higher occupancy is simply more waves in flight.

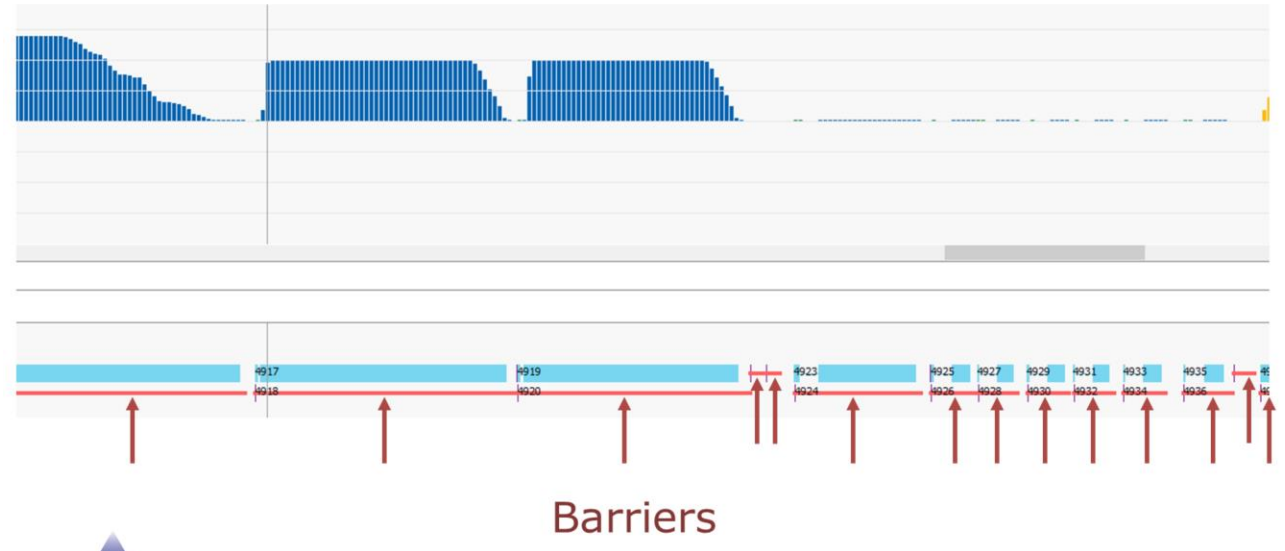
Important: It’s occupancy, not utilization! The ALUs may already be stressed to the max.

Imagine on a CPU: Dispatching a lot more virtual threads than HW threads doesn’t help you much.

→ Fewer waves is not necessarily bad! Not saturating the ALUs/Bandwidth is.

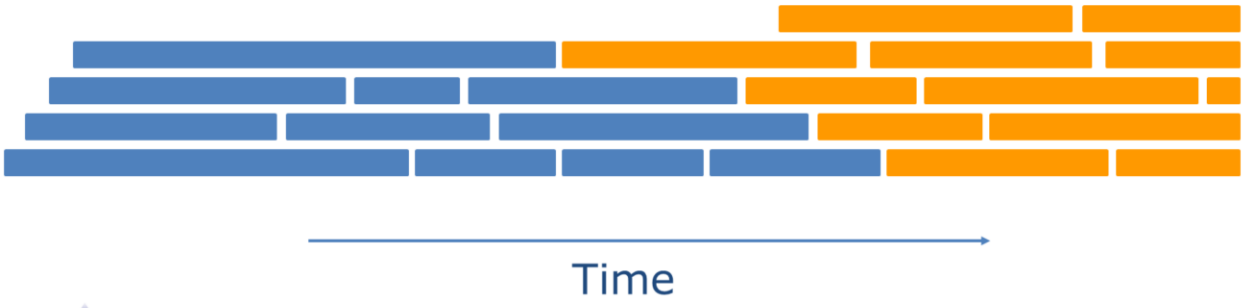


Obviously, if there are NO waves running, then we're not using the shader core to full extent (i.e. we don't get at least one wave onto each SIMD).
Also, usually not all waves are done at the same time. See the occupancy dropping at the end of the workloads.



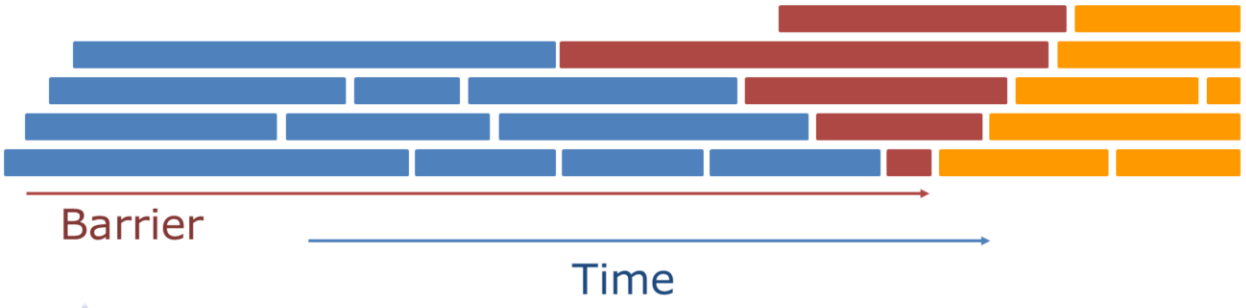
Barriers

But what is causing that? RGP knows the answer: Barriers!



Back to our “zoomed” in view.

Without any synchronization the next compute shader could start here.



Barriers cause the next bunch of work items to wait until the prior work all finished.

They also sometimes cause other work to happen (think of compression/decompression & making work visible to other units => flush caches).
Thus, they sometimes have some additional overhead aside from draining the shader core.

Now the important part, it’s not only that tiny bit of work being lost (1)

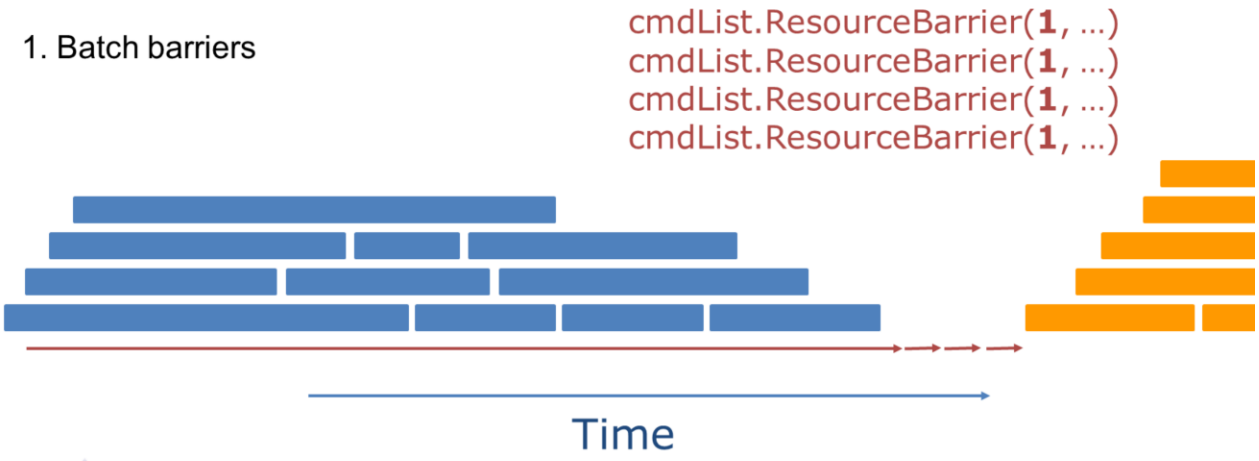
It’s actually that much (2)

So we’re up for issue number 1: Using a barrier after each draw.

Or even worse: multiple barriers after (almost) each draw.

Let’s concentrate on how we can solve issues like this.

Obviously we can’t just get rid of all barriers. Some synchronization will always be necessary.



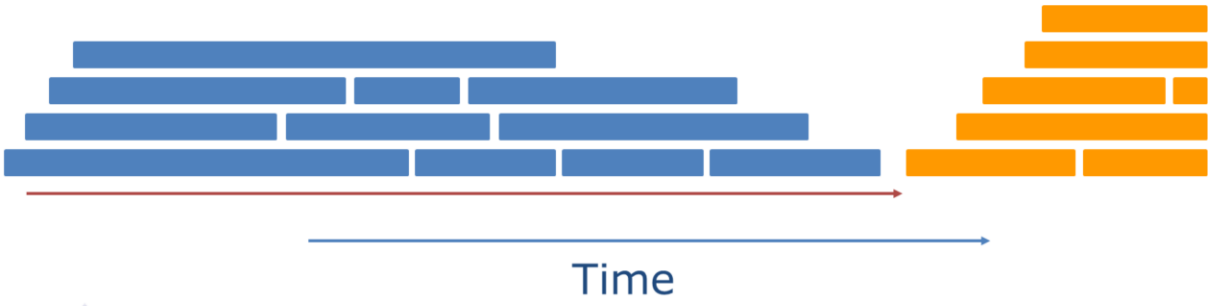
First, we can batch them, so the driver can manage the synchronization points more precisely.

It’s as easy as submitting them together in a single vkCmdPipelineBarrier / CommandList::ResourceBarrier.

The driver can then easily figure out the worst case synchronization or if only a single cache flush is necessary (instead of multiples) etc.

1. Batch barriers

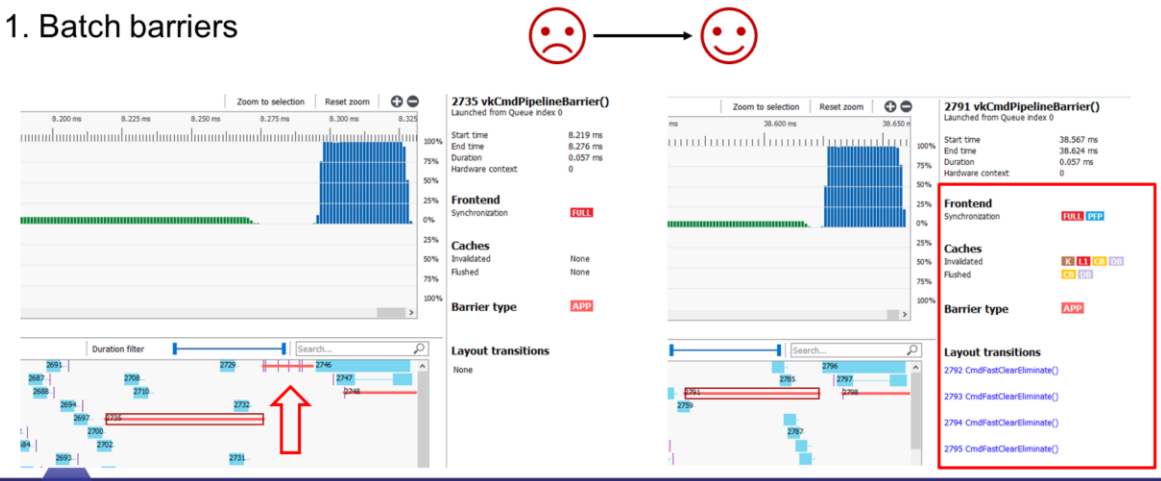
cmdList.ResourceBarrier(**4**, …)



Only a small impact, but the effects add up! Don’t neglect those.

Also, certain compressions may start earlier, that’s a net win!

1. Batch barriers

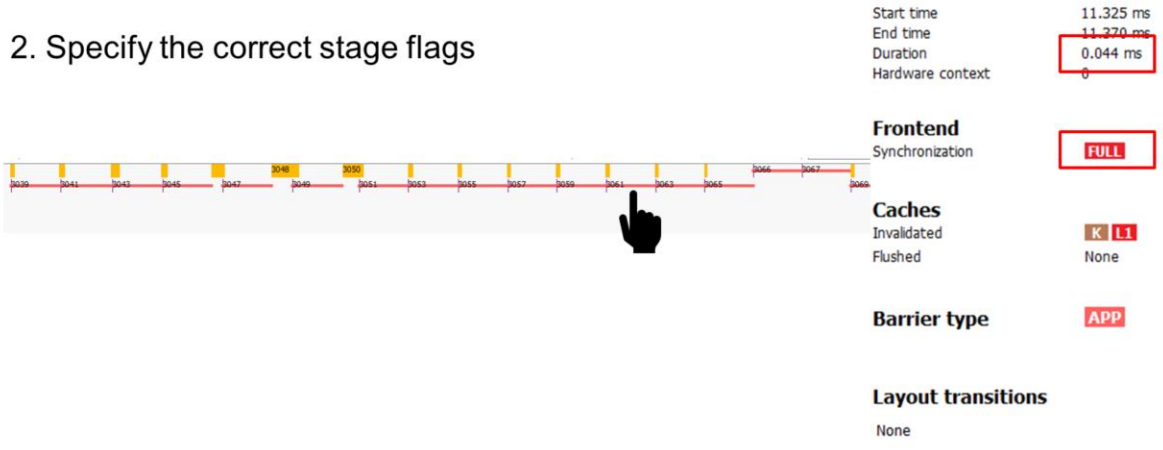


This is how you will see it in RGP.

Note that the first barrier is still exactly as long as before, but we were able to get rid of all the other barriers!

We successfully overlapped the FastClearEliminates (writing cleared values from meta data to every pixel -> decompressing the texture) as well.

2. Specify the correct stage flags

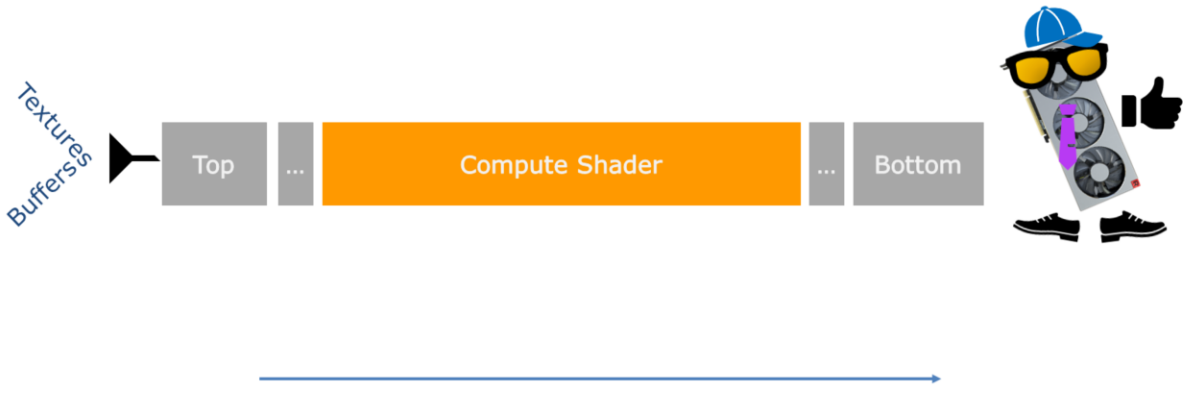


Be very descriptive in how you define the stage flags. In some cases you pay for being too general.

Vulkan specific. Found by a developer we work with.

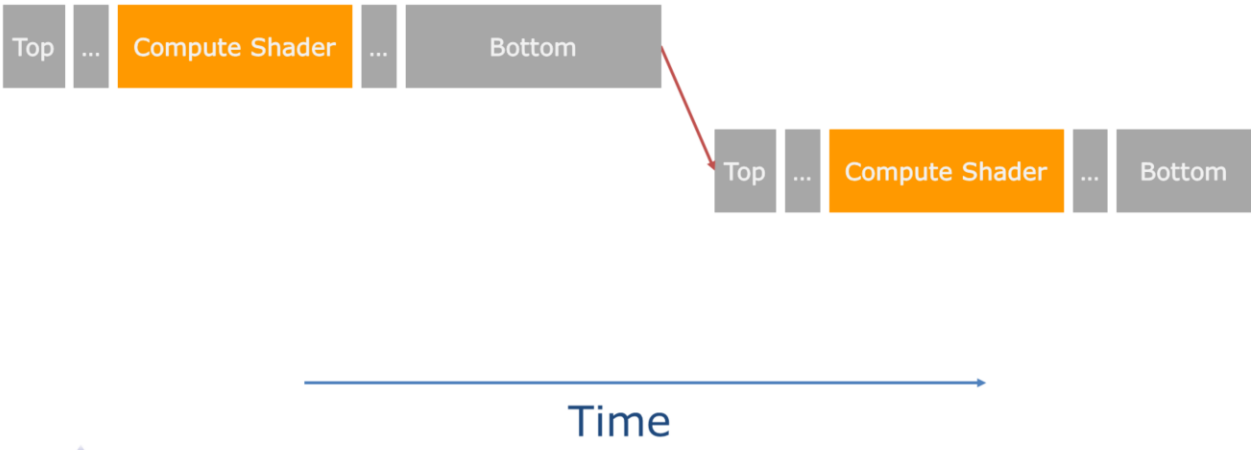
If you click on any of these barriers and look to the right side into the details again you see each barrier taking 0.044ms to execute.

Doing a FULL synchronization each.

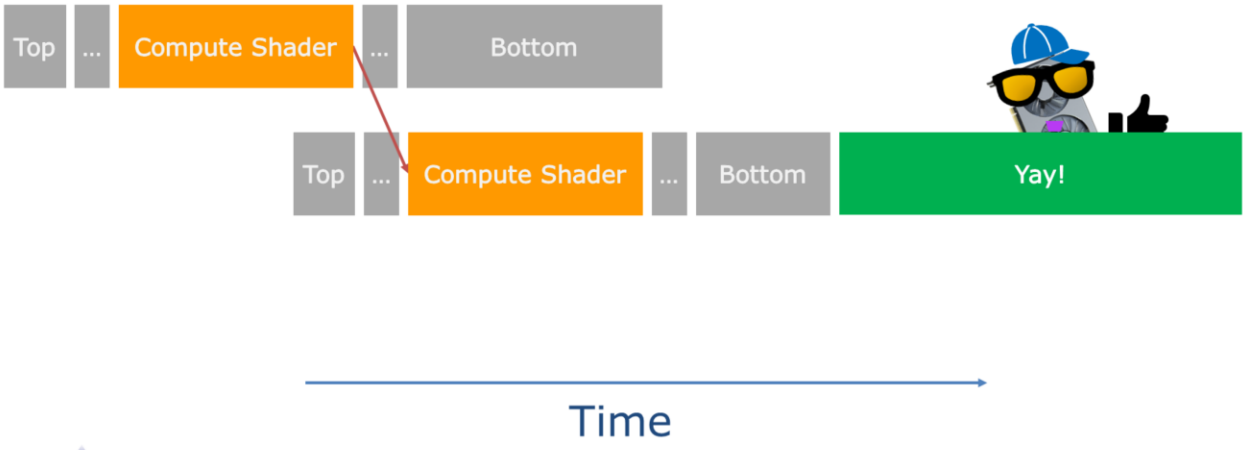


Pretend we would go through the GPU pipeline in a straight line. Compute only here.

We put stuff like textures and buffers in on top, pass a lot of stages until we execute the compute shader until our results drop out at the bottom.

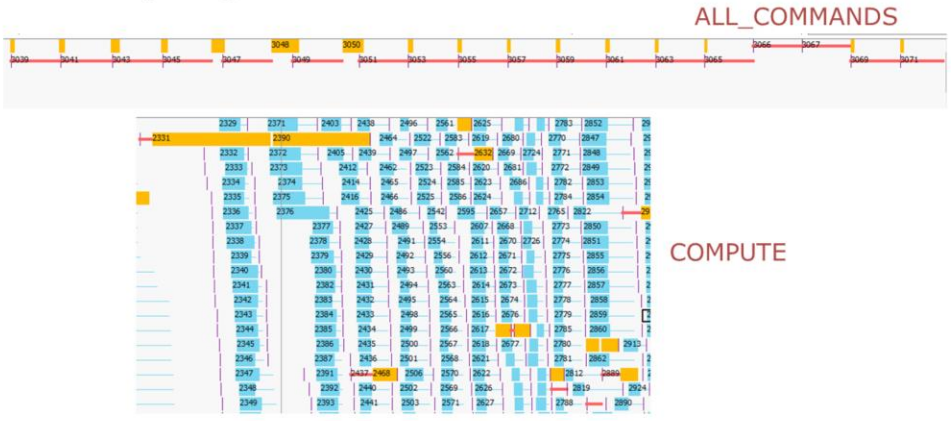


Now let’s say in some special corner cases BOTTOM_OF_PIPE takes a long time to execute.
But we’re only interested in dependencies between the CS!



If we would describe the accesses more precisely, we could overlap more.

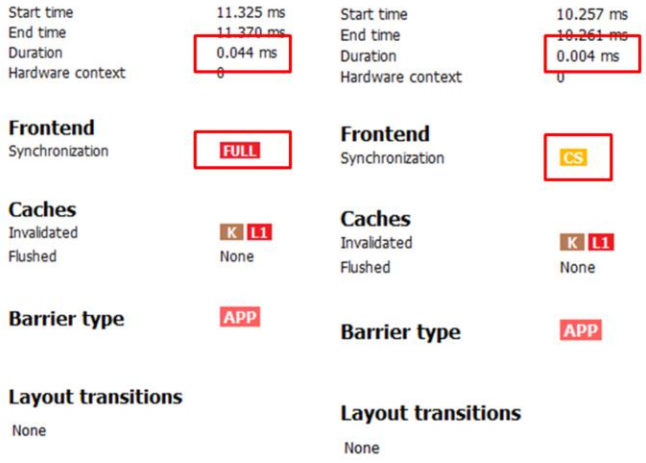
2. Specify the correct stage flags



Here it was set with ALL_COMMANDS (contains BOTTOM_OF_PIPE) on the async compute queue.

Replacing that with COMPUTE shows a big improvement, here enough to overlap with other draws on the graphics queue.

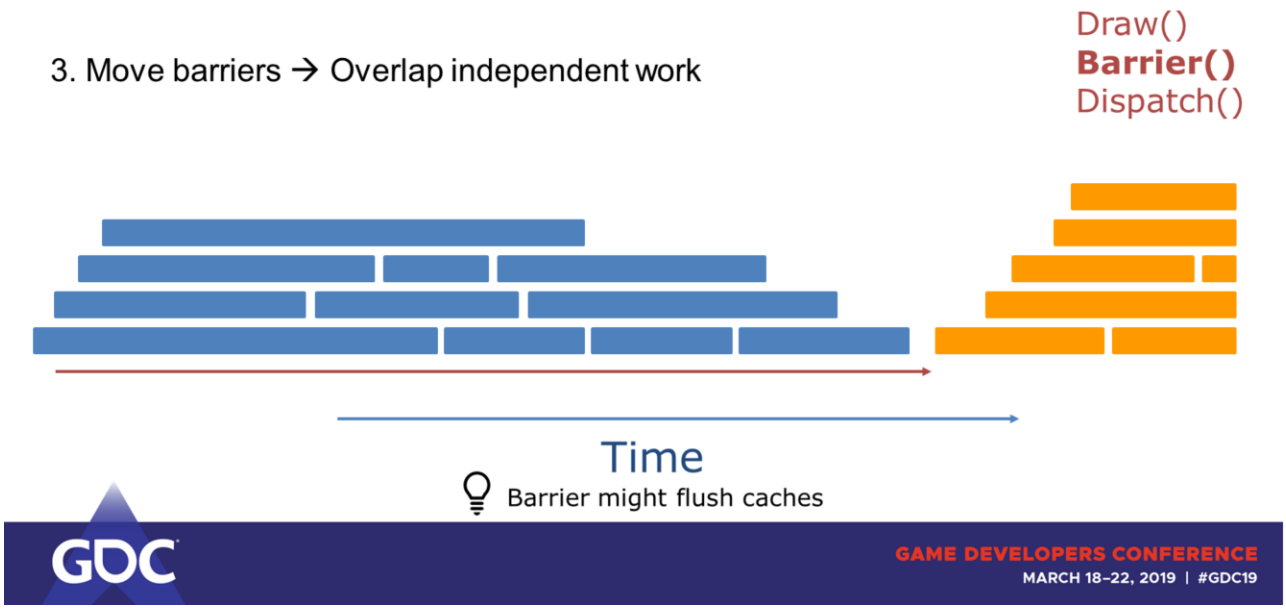
2. Specify the correct stage flags



To prove it, here's the detailed view when clicking on these barriers.

We replaced a full synchronization with the proper CS sync, which is a lot less expensive on the async queue.

Almost a magnitude improvement on each barrier.



Work that is independent does not necessarily have to wait.

→ Move the barrier after independent workloads.

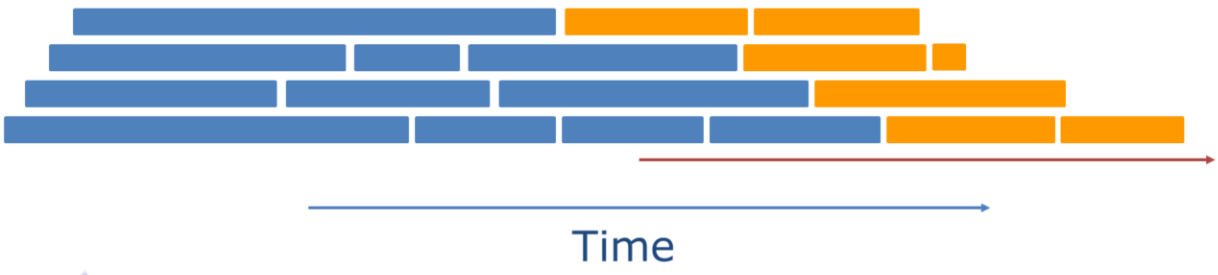
Added benefit: barriers often flush cashes as well.
If PS and CS share reads here, then the CS may run slower as it doesn't hit in L2 anymore.

Order of submissions here:

1. VS/PS
2. Barrier
3. CS

3. Move barriers → Overlap independent work

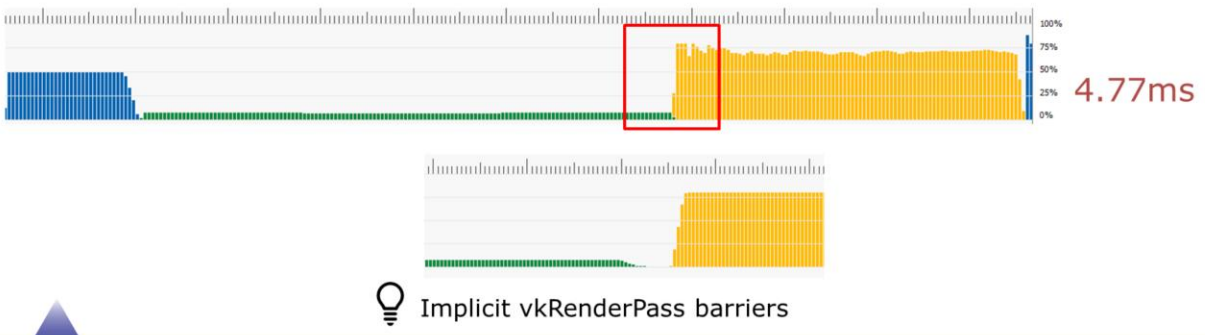
Draw()
Dispatch()
Barrier()



Order of submission would be:

1. PS
2. CS
3. Barriers

3. Move barriers → Overlap independent work



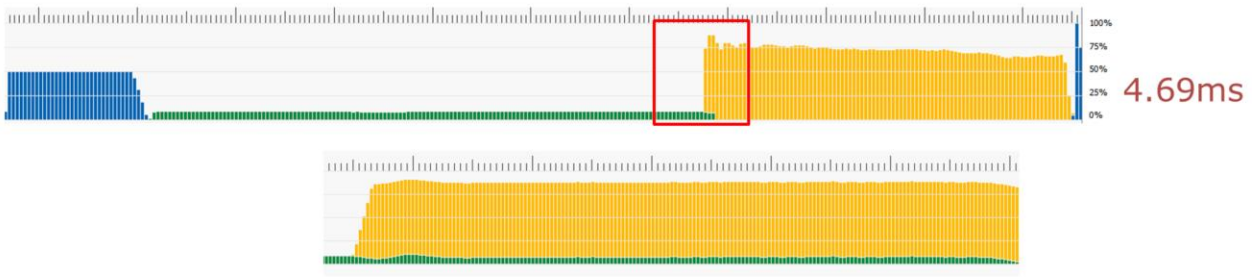
Usually what you want to do is to overlap long running CS waves with raster heavy workloads. In RGP these usually show up as short VS waves with low occupancy.

This is what we start out with. Notice the barrier that causes a gap between the Vertex and the Compute work.

Let’s start with moving that barrier behind the CS.

Pro tip these barriers may be due to start/end of RenderPasses: Add explicit TOP to BOTTOM barriers (basically no-op barriers) as subpass dependencies to EXTERNAL
to get rid of implicit barriers at the beginning/end of render passes in Vulkan.

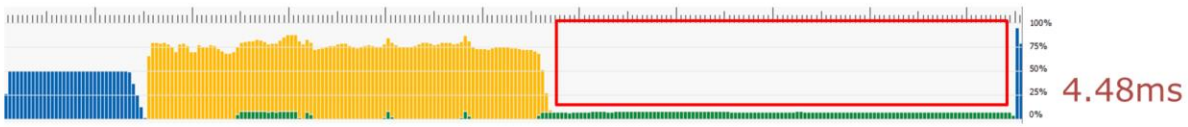
3. Move barriers → Overlap independent work



Some tiny overlap at least.

But we can do better: The CS wavefronts are running longer than the VS wavefronts → We should swap CS and VS submits.

3. Move barriers → Overlap independent work



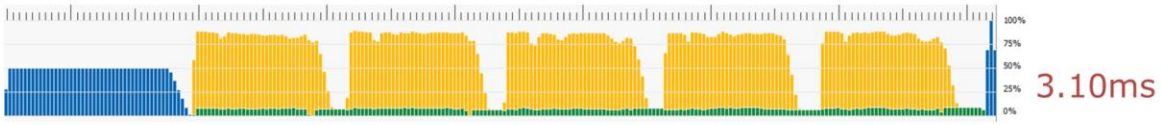
This already looks better than before. Still not ideal as at some point we only spawn new CS waves, no VS waves anymore.

→ We can split up the CS work into multiple dispatches and interleave it with VS work.

E.g. if the CS is fullscreen, split it into tiles and interleave it with draws.

Imagine shadow cascades with SSAO.

3. Move barriers → Overlap independent work

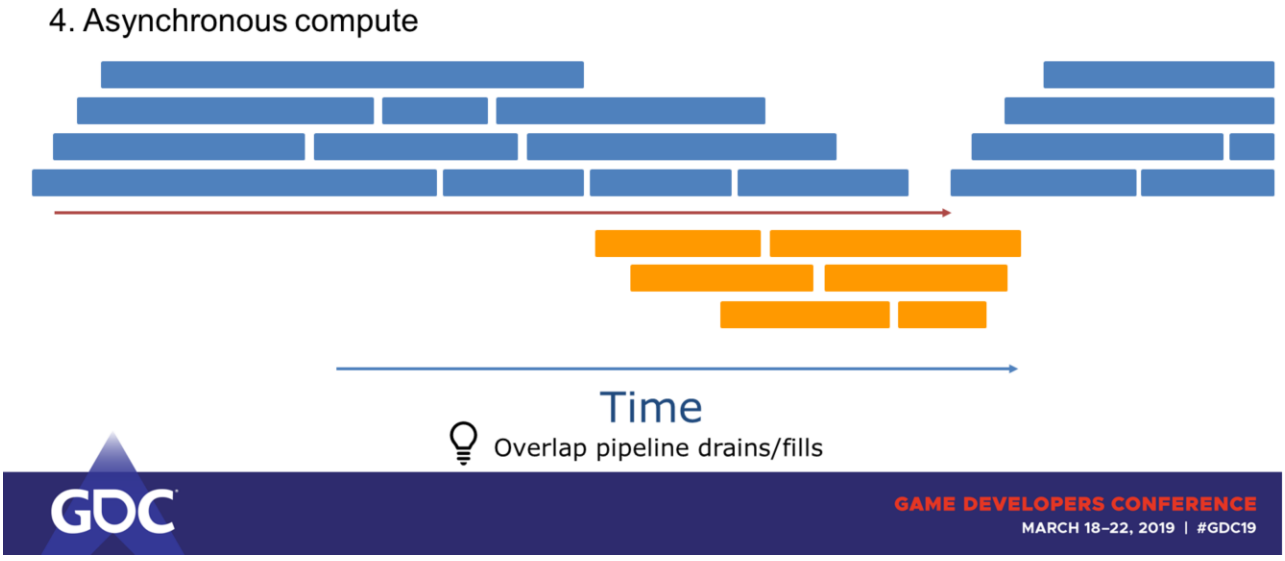


Hardware resources

This gives amazing results! And completely without going to the async compute queue.

Works particularly well if you find passes that have significantly different hardware utilization.

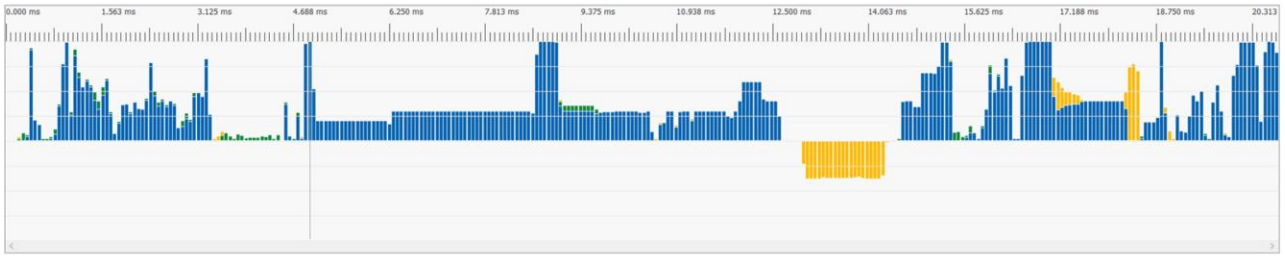
E.g. texture fetch heavy SSAO with vertex heavy shadow map rendering.



Async queue does not force waits on the graphics queue!
Useful for whole compute passes that require syncs to move to the async queue.

Keep attention to queues fighting for resources.
Move passes with a lot of work to the async queue, and keep the number of cross queue syncs to a minimum.

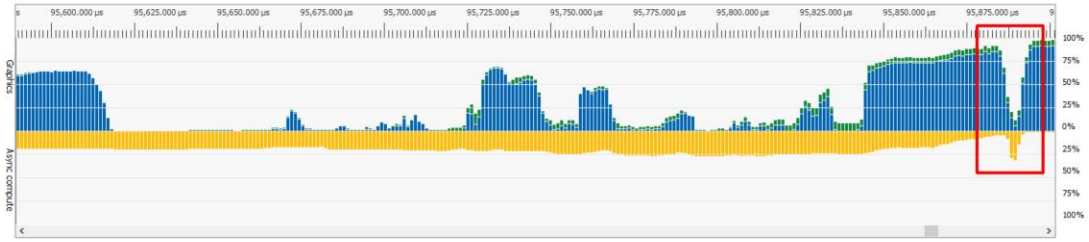
4. Asynchronous compute



¯_(ツ)_/¯

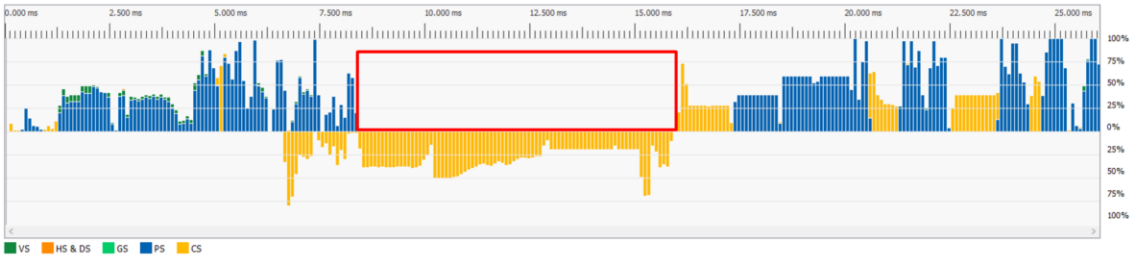
That probably didn’t turn out as expected.

4. Asynchronous compute



But if done correctly, this can work out great!

See how the compute queue can take over while the graphics queue is draining and filling.



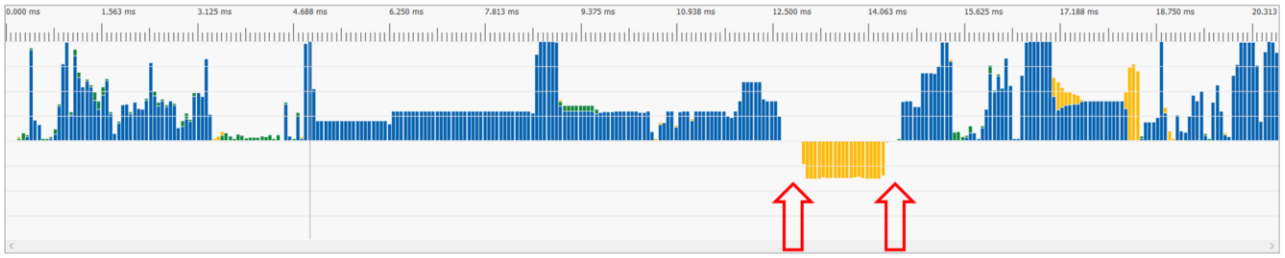
Varying workloads make this a hard problem. Sometimes you end up starving the graphics queue.

Rather aim to keep the graphics queue busy first.

Wavefronts & Barriers 

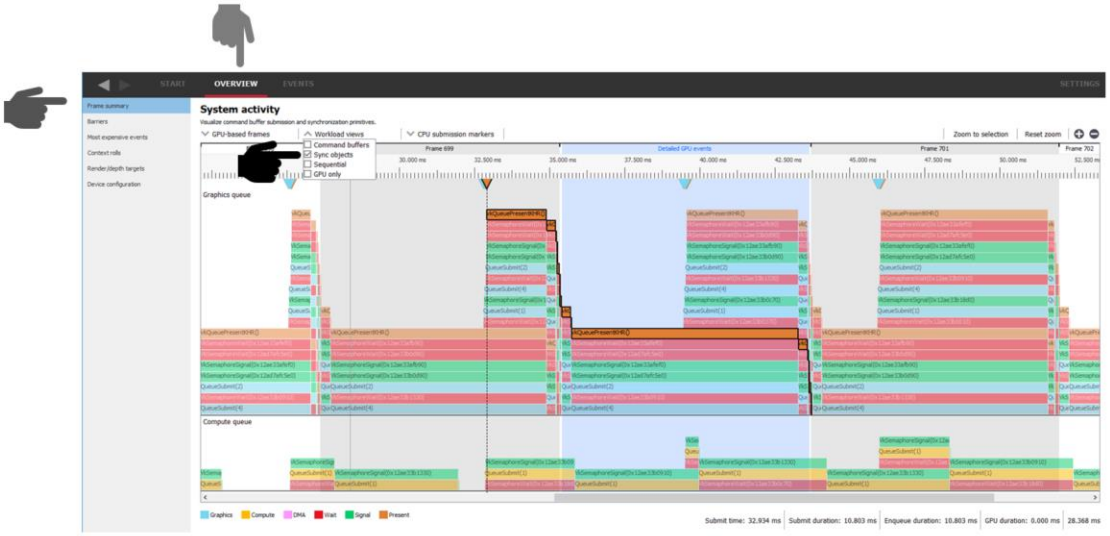
Cross queue synchronization

Compression

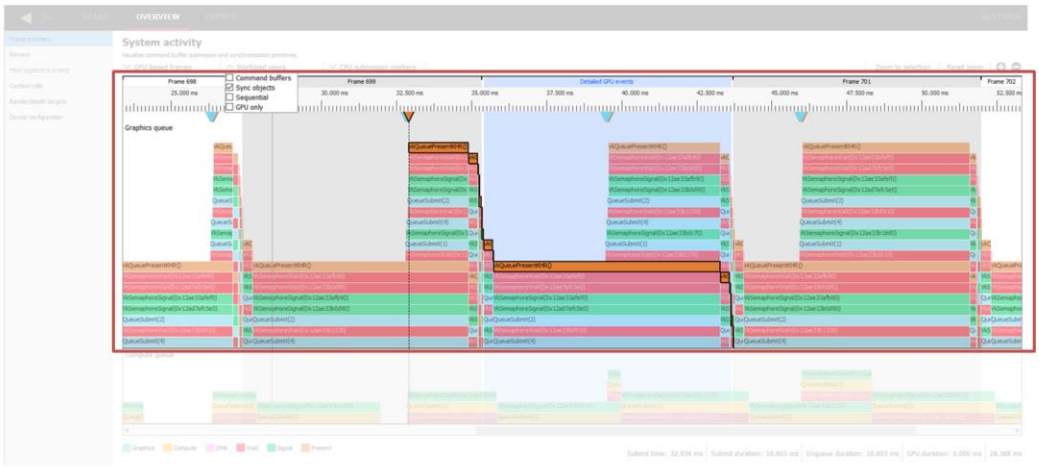


Gaps, but no barriers. Where do they come from?

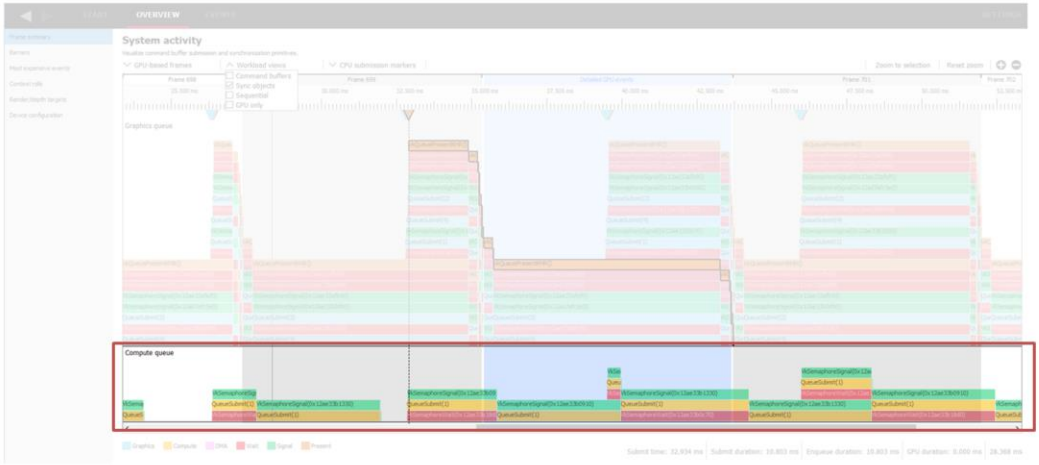
Let’s head over to a different view.



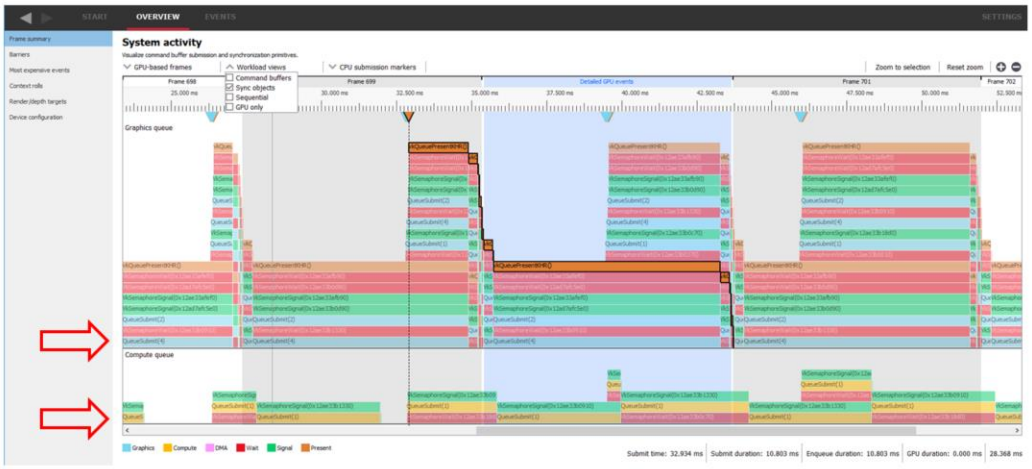
When you open up your capture you are greeted with the frame summary page.
Anyone working with GPUView might see similarities.



Top is the graphics queue.



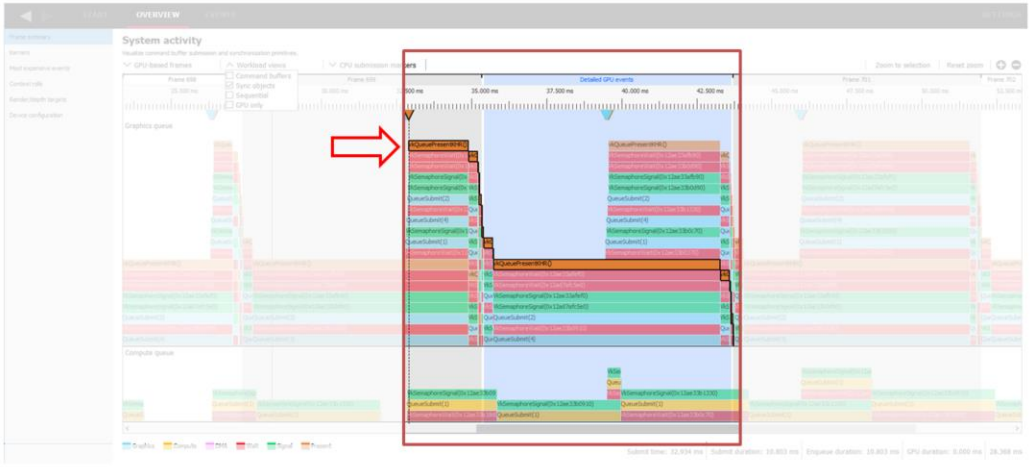
Bottom is the async compute queue (and/or copy queue if available)



New work item submissions (command buffers, waits, signals, presents) are added on top.

Each time one finishes, the others drop one level.

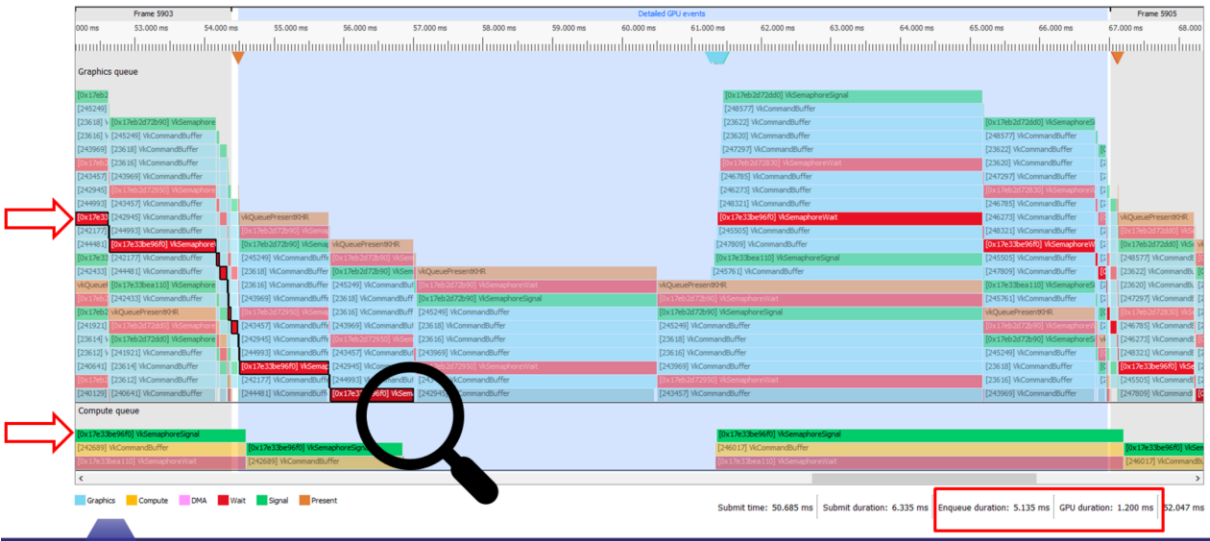
The lowest level are the work items the GPU queue worked on at that time.



RGP marks the frame it captured in light blue, but also shows you the frames prior and after.

The present packets are used to identify the frames.

This page will thus also show you on which queue the present ends up on.



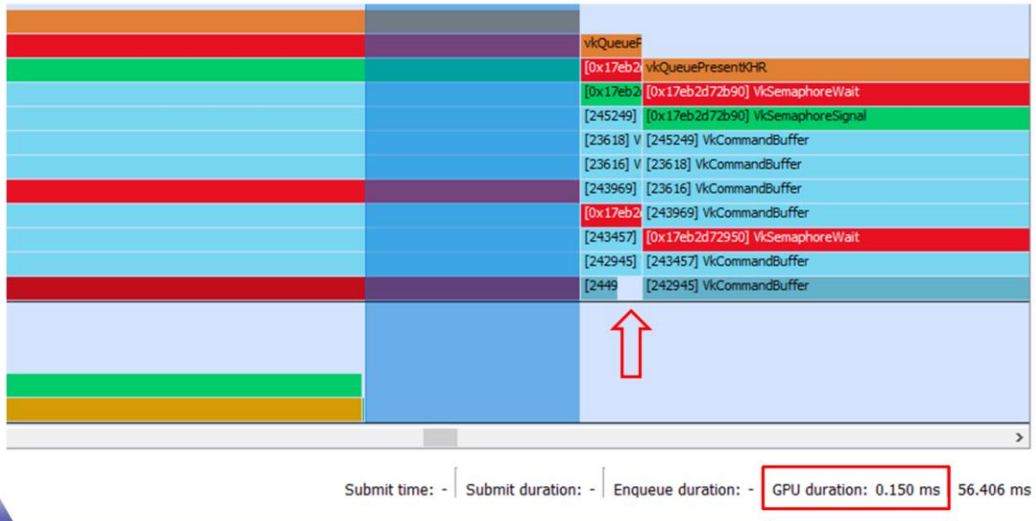
Let’s now concentrate on the sync primitives (different trace).

Clicking on one of them shows related signal/waits.

On the bottom right you see how long it took to submit the packet, how long it’s been queued up and how long it took the GPU to execute it.

Or in a case of the wait how long it prevented the GPU from progressing

further.



Marking a selection shows the timespan in the bottom right corner of the view.

You may have noticed these small gaps as well.

The GPU interrupts the CPU kernel to signal that a command buffer finished.

CPU side bookkeeping can sometimes cause delays.

Async compute recommendations:

- ❑ Synchronize seldomly, ideally only 1-2 times per frame
  Each synchronization point has significant overhead
- ❑ Move large continuous workloads to the async queue
  More opportunity to overlap pipeline drains / fills
- ❑ For adventurers: overlap with next frame
  Usually frames start with raster heavy work
  and end with compute heavy post processing
  May add latency!

Not much you can do about –> only sync seldomly. Only 1-2 times per frame ideally.

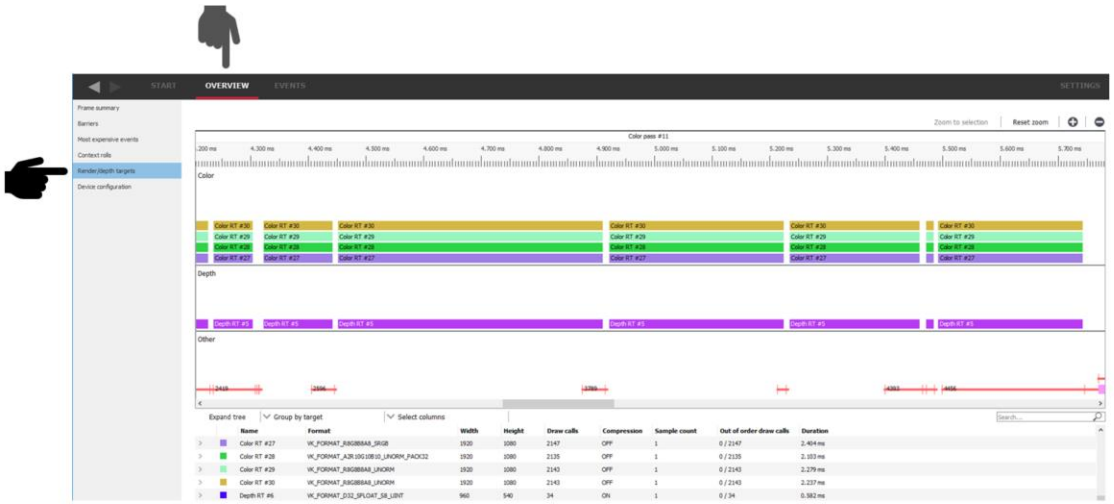
Let it run uninterrupted as long as possible to overlap pipeline drains and fills.

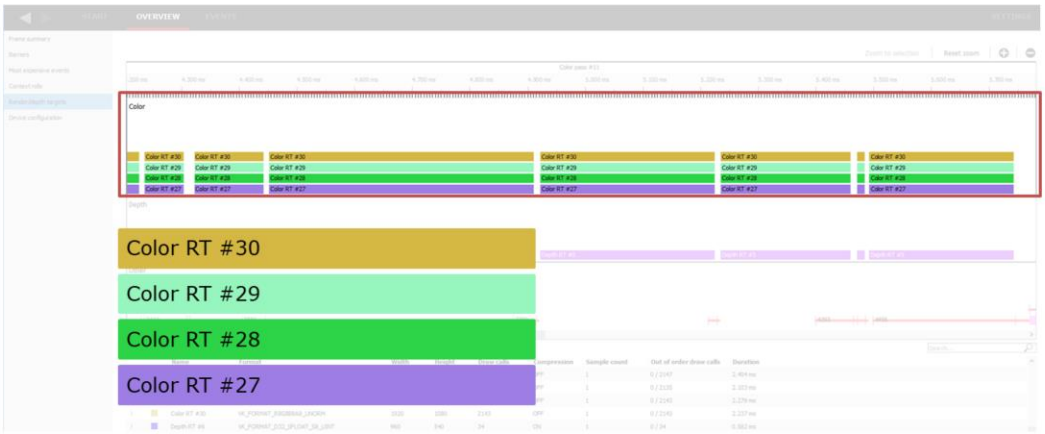
And can even overlap with the next frame if your game can take the added latency.

That works because frames usually start with raster heavy workloads (GBuffer, Shadow maps) and end

with compute heavy post processing.

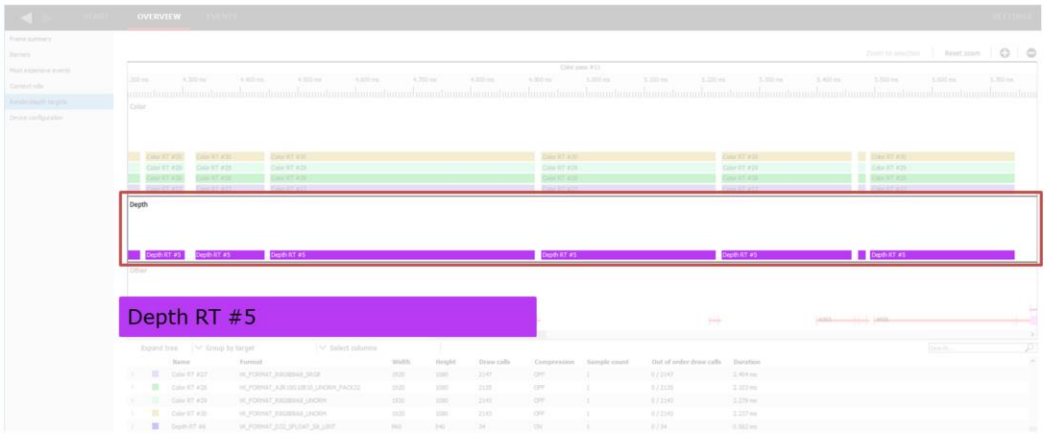
Wavefronts & Barriers ✓
Cross queue synchronization ✓
Compression


START
OVERVIEW
EVENTS
SETTINGS
Frame summary
Barriers
Most expensive events
Context rolls
Render/depth targets
Device configuration
Zoom to selection
Reset zoom
Color pass #11
Color
Depth
Other
Expand tree
Group by target
Select columns
Name
Format
Width
Height
Draw calls
Compression
Sample count
Out of order draw calls
Duration
Color RT #27
VK_FORMAT_R8G8B8A8_SRGB
1920
1080
2147
OFF
1
0 / 2147
2.404 ms
Color RT #28
VK_FORMAT_A2R10G10B10_UNORM_PACK32
1920
1080
2135
OFF
1
0 / 2135
2.103 ms
Color RT #29
VK_FORMAT_R8G8B8A8_UNORM
1920
1080
2143
OFF
1
0 / 2143
2.279 ms
Color RT #30
VK_FORMAT_R8G8B8A8_UNORM
1920
1080
2143
OFF
1
0 / 2143
2.237 ms
Depth RT #6
VK_FORMAT_D32_SFLOAT_S8_UINT
960
540
34
ON
1
0 / 34
0.582 ms



This view is not split into queues, but rather accesses to resources.

Color render targets on top, depth targets below and barriers & buffers on the bottom.

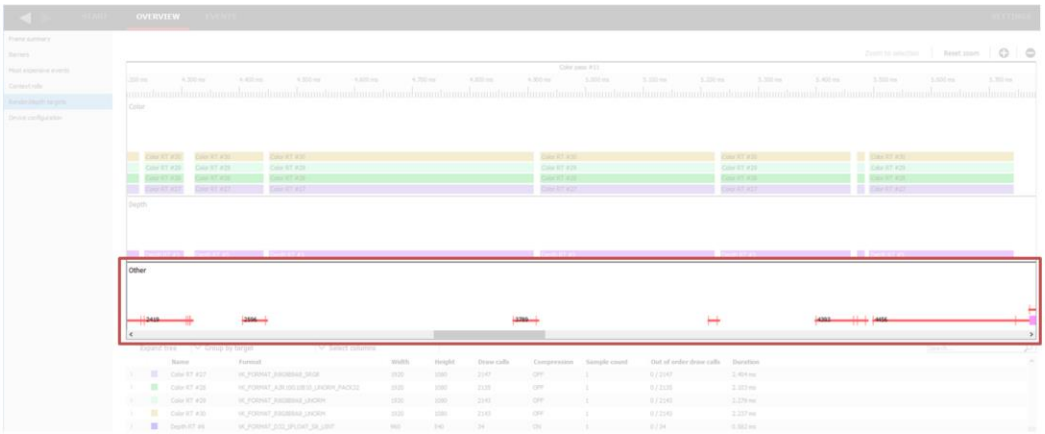


This view is not split into queues, but rather accesses to resources.

Color render targets on top, depth targets below and barriers & buffers on the bottom.

The part you see here is the Gbuffer pass (4 color targets + 1 depth target).

Side note: May want to take care of these barriers splitting up the Gbuffer pass.

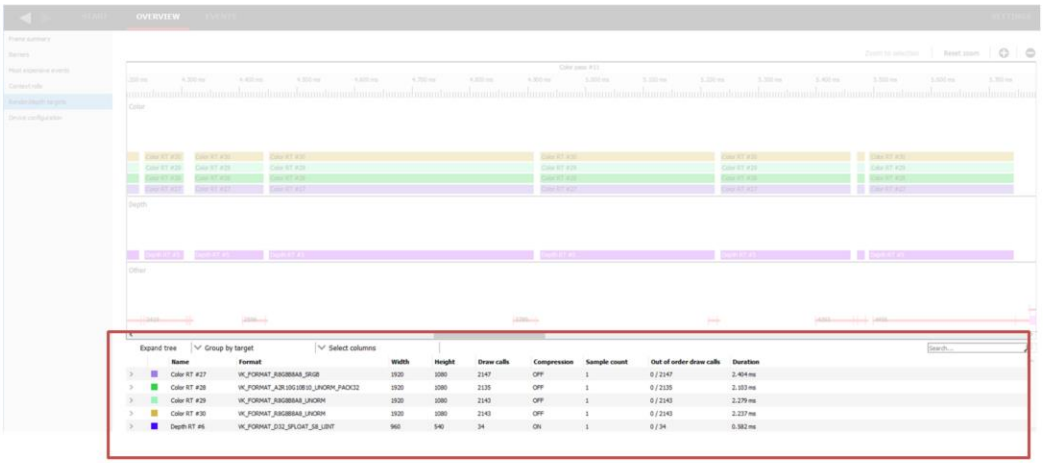


This view is not split into queues, but rather accesses to resources.

Color render targets on top, depth targets below and barriers & buffers on the bottom.

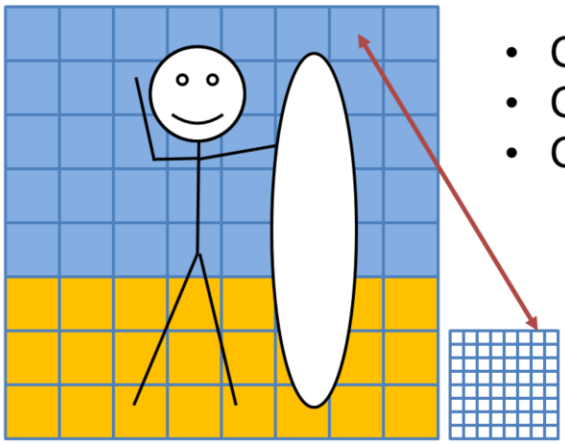
The part you see here is the Gbuffer pass (4 color targets + 1 depth target).

Side note: May want to take care of these barriers splitting up the Gbuffer pass.



Below it you can see details that unroll once you click on a RT.

	Name	Format	Width	Height	Draw calls	Compression	Sample count	Out of order draw calls	Duration
>	Color RT #27	VK_FORMAT_R8G8B8A8_SRGB	1920	1080	2777	OFF	1	0 / 2777	2.414 ms
>	Color RT #28	VK_FORMAT_A2R10G10B10_UNORM_PACK32	1920	1080	2766	OFF	1	0 / 2766	2.100 ms
>	Color RT #29	VK_FORMAT_R8G8B8A8_UNORM	1920	1080	2773	OFF	1	0 / 2773	2.170 ms
>	Color RT #30	VK_FORMAT_R8G8B8A8_UNORM	1920	1080	2774	OFF	1	0 / 2774	2.416 ms



- Cleared
- Clear color
- Compressed

Let’s make this a quick (barely scratching the surface) introduction to texture compression as it’s used for render targets.

Let’s start out with a texture and cut it into blocks.

We attach some meta data to the texture that describes attributes per block.

Like

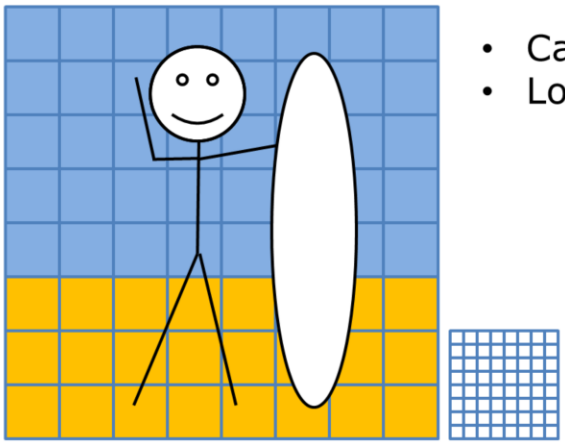
- Cleared
- Clear Color
- Compressed / Decompressed

In compressed state we can overwrite the contents of each original block.

Everything’s lossless compressed so we can restore the correct color per pixel.

Can be done during creation of that texture, meaning rendering into it.

Controlled by transitioning the texture, sometimes may need to decompress it.

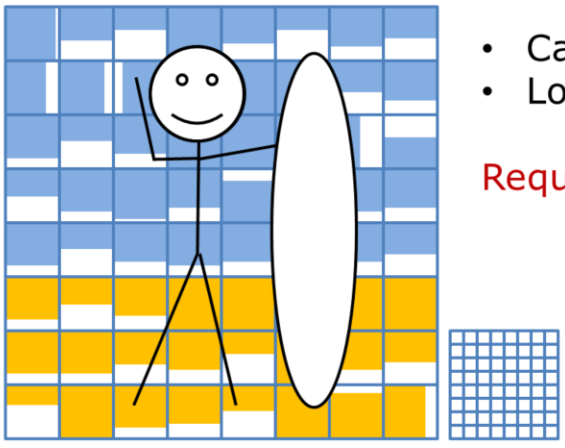


- Can only load meta data
- Load only parts of a block

The cool thing is:

- Can skip loading the actual pixels
- Or only load parts of a pixel block!

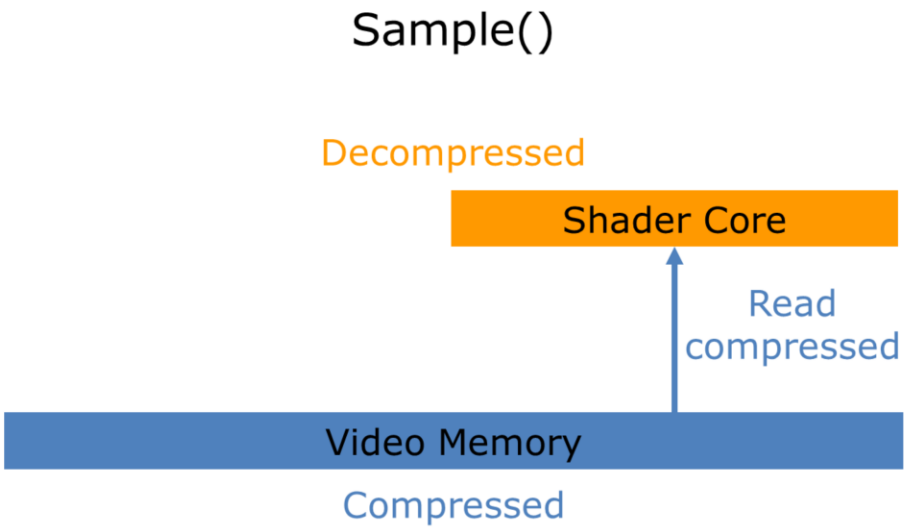
Make sure to transition correctly, or you may end up seeing the compressed blocks → corruptions.



- Can only load meta data
- Load only parts of a block

Requires correct transitions

Make sure to transition correctly, or you may end up seeing the compressed blocks → corruptions.

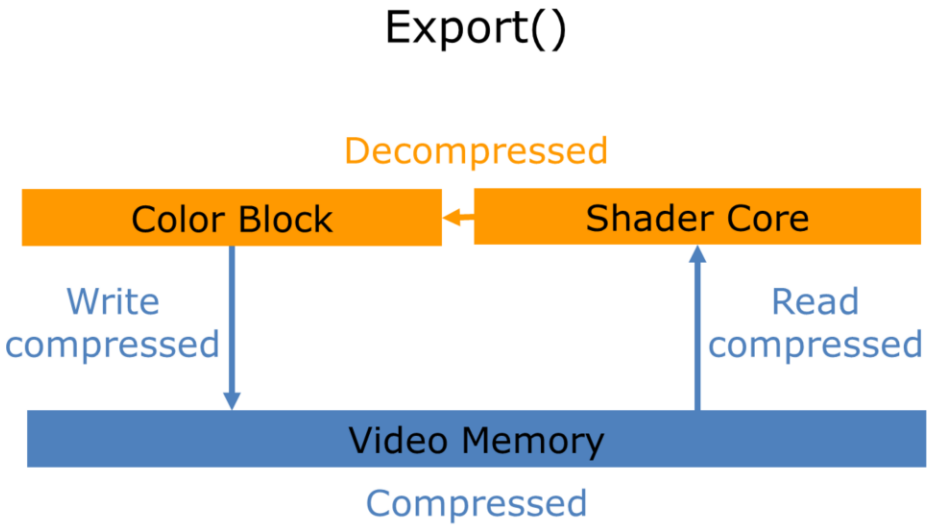


https://gpuopen.com/dcc-overview/

Cycle through sampling a texture and exporting color values to a render target.

→ We save on bandwidth both on read AND write to VMEM.

The texture itself isn't going to be stored more compactly – in fact we even need to attach more data to hold meta information!



https://gpuopen.com/dcc-overview/

Cycle through sampling a texture and exporting color values to a render target.

→ We save on bandwidth both on read AND write to VMEM.

The texture itself isn't going to be stored more compactly – in fact we even need to attach more data to hold meta information!

Compression checklist:

- ❑ Use exclusive queue ownership
  With shared ownership the driver must assume that it's used on hardware blocks that can't read or write compressed
- ❑ Explicitly state image format
  UNKNOWN / MUTABLE can prevent compression
  Can work around with VK_KHR_image_format_list
- ❑ Use only the required image usages
  Otherwise the resource can end up in less than optimal compression levels
- ❑ Clear render/depth targets
  Clearing resets meta data

- Rendering many triangles into a target can turn a block to decompressed state. Clearing resets meta data and thus improves compression ratio.
- Shared ownership disables compression (certain blocks can't handle compression)
- Quirks with UNKNOWN/MUTABLE prevent compression. Can work around it with VK_KHR_image_format_list

- Important if many draws render to that target.

Name	Format	Width	Height	Draw calls	Compression	Sample count	Out of order draw calls	Duration
Color RT #27	VK_FORMAT_R8G8B8A8_SRGB	1920	1080	2777	OFF	1	0 / 2777	2.414 ms
Color RT #28	VK_FORMAT_A2R10G10B10_UNORM_PACK32	1920	1080	2766	OFF	1	0 / 2766	2.100 ms
Color RT #29	VK_FORMAT_R8G8B8A8_UNORM	1920	1080	2773	OFF	1	0 / 2773	2.170 ms
Color RT #30	VK_FORMAT_R8G8B8A8_UNORM	1920	1080	2774	OFF	1	0 / 2774	2.416 ms

Name	Format	Width	Height	Draw calls	Compression	Sample count	Out of order draw calls	Duration
Color RT #27	VK_FORMAT_R8G8B8A8_SRGB	1920	1080	2779	ON	1	0 / 2779	2.207 ms
Color RT #28	VK_FORMAT_A2R10G10B10_UNORM_PACK32	1920	1080	2768	ON	1	0 / 2768	1.959 ms
Color RT #29	VK_FORMAT_R8G8B8A8_UNORM	1920	1080	2775	ON	1	0 / 2775	2.014 ms
Color RT #30	VK_FORMAT_R8G8B8A8_UNORM	1920	1080	2776	ON	1	0 / 2776	2.218 ms

5-10%

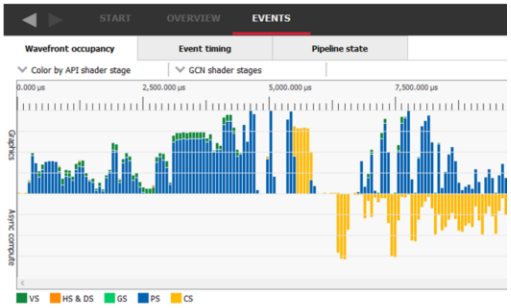
Generous 10% win in this case.

Wavefronts & Barriers ✓
Cross queue synchronization ✓
Compression ✓

Wrap-Up

Barriers

- ❑ Batch barriers
- ❑ Use the proper pipeline stage flags
- ❑ Overlap independent work
- ❑ Use async compute queue



Wrap-Up

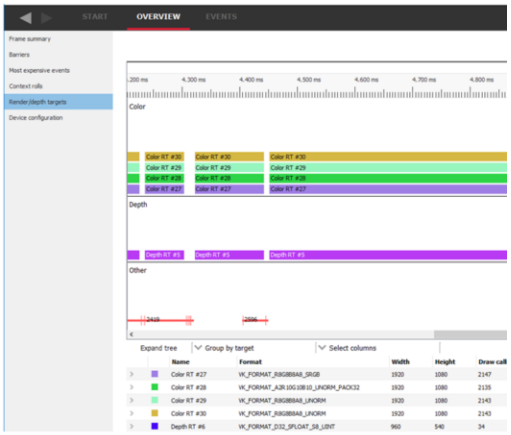
Cross queue synchronization

- ❑ Sync seldomly
- ❑ Prefer large workloads

Wrap-Up

Compression

- ❑ Use exclusive queue access
- ❑ Explicit image format
- ❑ Use only required image usage
- ❑ Clear render / depth targets



There is more

- More information on **barriers**
  https://gpuopen.com/vulkan-barriers-explained/
- More information on **DCC**
  https://gpuopen.com/dcc-overview/
- Check out RGPs **new features** on Wednesday 5pm!
- And a more detailed look at barriers by **Matt Pettineo** at 4pm today ☺

Thanks!

- To the AMD tools team
- To all reviewers

… and to you – Thank you for your attention!

Time for questions :)

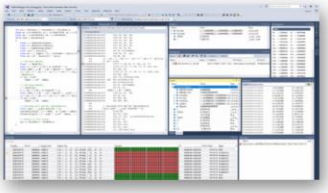
Find **RGP** on GPUOpen
https://gpuopen.com/gaming-product/radeon-gpu-profiler-rgp/


GDC
Dissecting Real-Time Rays:
Debugging & Profiling DXR/NVIDIA VKRay Applications
Aurelio Reis
SWE Director, Graphics Developer Tools, NVIDIA
GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

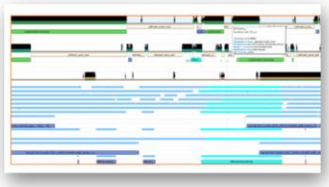
NVIDIA Developer Tools

Nsight Graphics

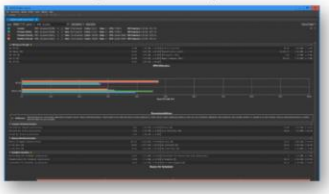
Nsight VSE (& Eclipse)



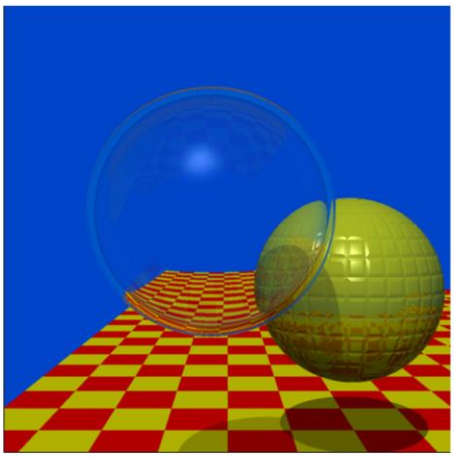
Nsight Systems



Nsight Compute



1979

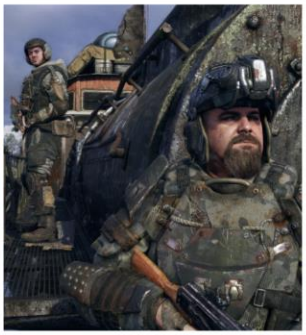


2019

Building Real-Time Ray Tracing Tools

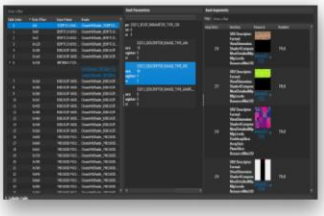
Shadow of the Tomb Raider, Battlefield V, Metro: Exodus...

- "Trial by fire"
- Made many assumptions and educated guesses
- Analyzed new titles utilizing **DXR/NVIDIA VKRay**
- Development experiences uncovered numerous problems to help solve
  - API usage
  - BVH/AS building & complexity
  - Shader complexity
  - Crashing

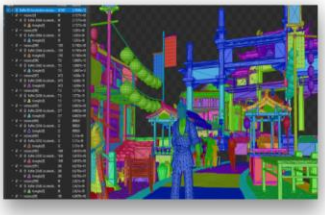


NVIDIA DXR Debugging & Profiling Tools

Shader Table & Resource Inspector

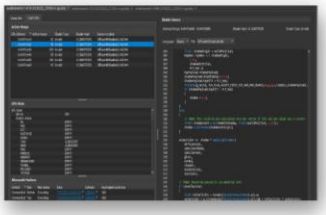


Acceleration Structure Viewer



Warp Occupancy & Metrics (GPU Trace)

Crash Debugging (Nsight Aftermath)

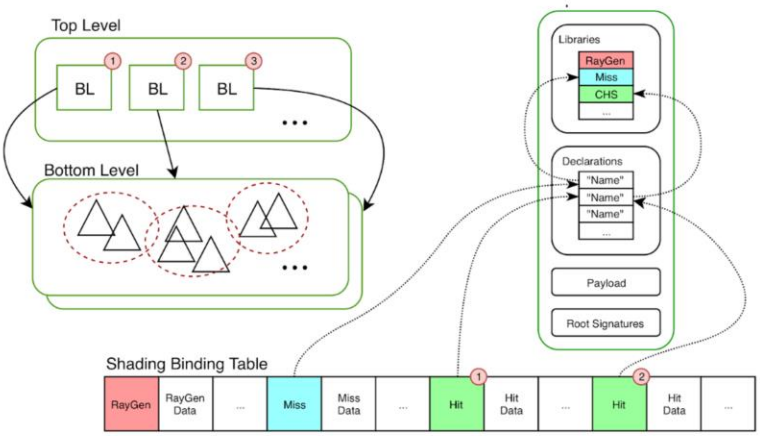


DEVELOPMENT PROCESS

...and common problems...

1. Implement API
2. Resolve crashes
3. Fix rendering issues
4. Profile & Optimize
5. Rinse, repeat, ship!

IMPLEMENT API



API USAGE ISSUES

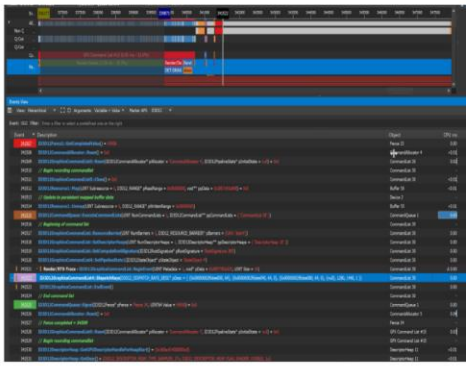
Shader Table/Root Signature misconfigured

Shader binding errors

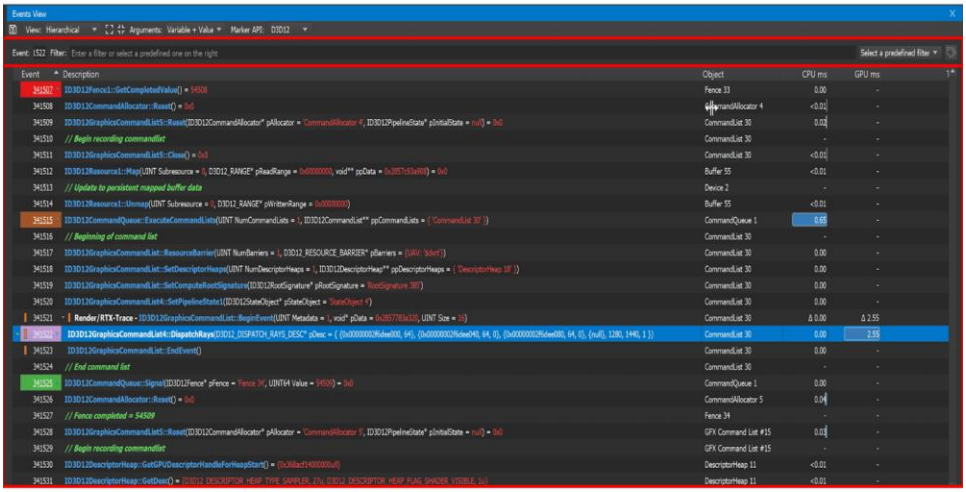
AS/BVH issues

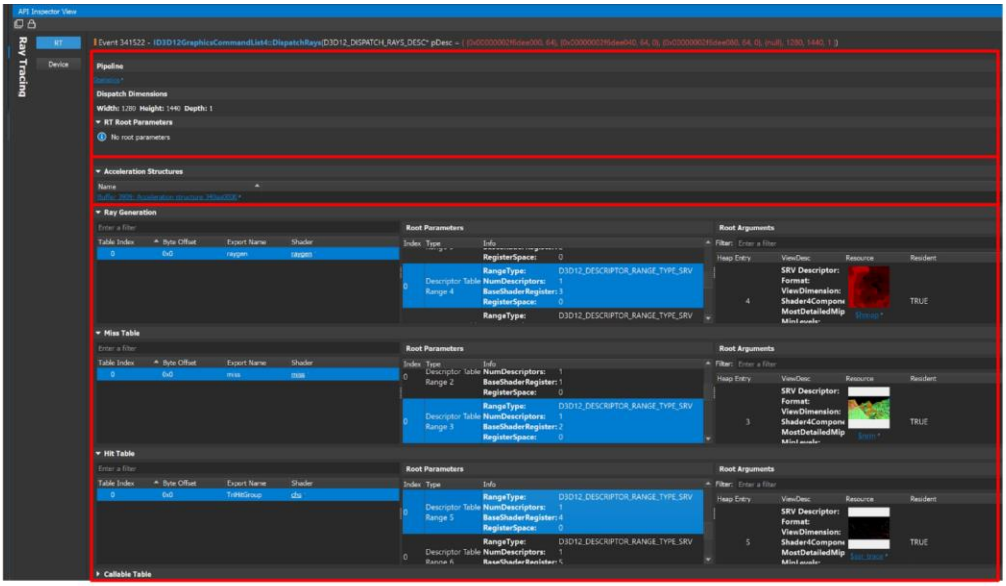
Incorrect root argument resources

Resource transitions



Nsight Graphics: Event Viewer





Nsight Graphics: API Inspector

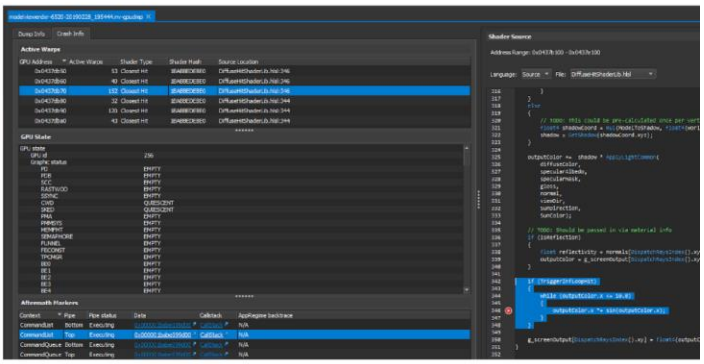
RESOLVE CRASHES

GPU CRASH

TDR (Timeout Detection & Recovery)

Device Removed

Exception Faults



modelviewerdxr-6520-20190228_195444.nv-gpudmp

Dump Info | Crash Info

Active Warps

GPU Address	Active Warps	Shader Type	Shader Hash	Source Location
0x0437db50	53	Closest Hit	1BABBEDEBE0	DiffuseHitShaderLib.hlsl:346
0x0437db60	40	Closest Hit	1BABBEDEBE0	DiffuseHitShaderLib.hlsl:346
0x0437db70	152	Closest Hit	1BABBEDEBE0	DiffuseHitShaderLib.hlsl:346
0x0437db80	32	Closest Hit	1BABBEDEBE0	DiffuseHitShaderLib.hlsl:344
0x0437db90	120	Closest Hit	1BABBEDEBE0	DiffuseHitShaderLib.hlsl:344
0x0437dba0	43	Closest Hit	1BABBEDEBE0	DiffuseHitShaderLib.hlsl:344

******

GPU State

GPU state
- GPU id: 256
- Graphic status
  - PD: EMPTY
  - PDB: EMPTY
  - SCC: EMPTY
  - RASTWOD: EMPTY
  - SSYNC: EMPTY
  - CWD: QUIESCENT
  - SKED: QUIESCENT
  - PMA: EMPTY
  - PMMSYS: EMPTY
  - MEMFMT: EMPTY
  - SEMAPHORE: EMPTY
  - FUNNEL: EMPTY
  - FECONST: EMPTY
  - TPCMGR: EMPTY
  - BE0: EMPTY
  - BE1: EMPTY
  - BE2: EMPTY
  - BE3: EMPTY
  - BE4: EMPTY

Aftermath Markers

Context	Pipe	Pipe status	Data	Callstack
CommandList	Bottom	Executing	0x00000 1baabe099d00	CallStack
CommandList	Top	Executing	0x00000 1baabe099d00	CallStack
CommandQueue	Bottom	Executing	0x00000 1baabe099d00	CallStack
CommandQueue	Top	Executing	0x00000 1baabe099d00	CallStack

Aftermath Marker Callstack

Module	Address	Name	Location
nvwgf2umx.dll	0x00007ffc7edd648e	Unknown symbol	
nvwgf2umx.dll	0x00007ffc7ef36055	Unknown symbol	
d3d12.dll	0x00007ffc96a52d11	Unknown symbol	
ModelViewerDXR.exe	0x00007ff66e37a7a1	NativeRaytracingCommandList::DispatchRays(D3D12_DISPATCH_RAYS_DESC const *)	c:\src\sw\sw\devrel\libdev\nsightaftermatl
ModelViewerDXR.exe	0x00007ff66e28d917	D3D12RaytracingMiniEngineSample::RaytraceDiffuse(GraphicsContext &,Math::Camera const &,ColorBuffer &)	c:\src\sw\sw\devrel\libdev\nsightaftermatl
ModelViewerDXR.exe	0x00007ff66e28c604	D3D12RaytracingMiniEngineSample::Raytrace(GraphicsContext &)	c:\src\sw\sw\devrel\libdev\nsightaftermatl
ModelViewerDXR.exe	0x00007ff66e28c250	D3D12RaytracingMiniEngineSample::RenderScene(void)	c:\src\sw\sw\devrel\libdev\nsightaftermatl
ModelViewerDXR.exe	0x00007ff66e31a48a	GameCore::UpdateApplication(GameCore::IGameApp &)	c:\src\sw\sw\devrel\libdev\nsightaftermatl
ModelViewerDXR.exe	0x00007ff66e31a300	GameCore::RunApplication(GameCore::IGameApp &,wchar_t const *)	c:\src\sw\sw\devrel\libdev\nsightaftermatl
ModelViewerDXR.exe	0x00007ff66e28f653	wmain	c:\src\sw\sw\devrel\libdev\nsightaftermatl
ModelViewerDXR.exe	0x00007ff66e453444	invoke_main	f:\dd\vctools\crt\vcstartup\src\startup\exe
ModelViewerDXR.exe	0x00007ff66e45336e	__scrt_common_main_seh	f:\dd\vctools\crt\vcstartup\src\startup\exe
ModelViewerDXR.exe	0x00007ff66e45322e	__scrt_common_main	f:\dd\vctools\crt\vcstartup\src\startup\exe
ModelViewerDXR.exe	0x00007ff66e4534b9	wmainCRTStartup	f:\dd\vctools\crt\vcstartup\src\startup\exe
KERNEL32.DLL	0x00007ffcce0a81f4	Unknown symbol	
ntdll.dll	0x00007ffcd0d3a251	Unknown symbol	

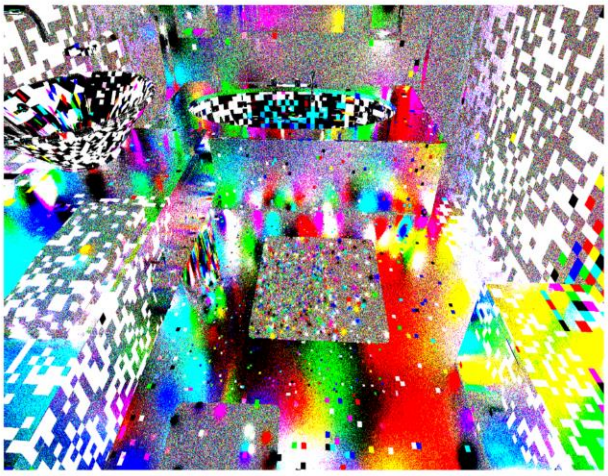
Shader Source

Address Range: 0x0437b100 - 0x0437e100 | Shader Hash: 1BABBEDEBE0 | Shader Type: Closest Hit

Language: Source | File: DiffuseHitShaderLib.hlsl

```
316         }
317     }
318     else
319     {
320         // TODO: This could be pre-calculated once per vertex if this mul per pixel was a concern
321         float4 shadowCoord = mul(ModelToShadow, float4(worldPosition, 1.0f));
322         shadow = GetShadow(shadowCoord.xyz);
323     }
324
325     outputColor +=  shadow * ApplyLightCommon(
326         diffuseColor,
327         specularAlbedo,
328         specularMask,
329         gloss,
330         normal,
331         viewDir,
332         SunDirection,
333         SunColor);
334
335     // TODO: Should be passed in via material info
336     if (IsReflection)
337     {
338         float reflectivity = normals[DispatchRaysIndex().xy].w;
339         outputColor = g_screenOutput[DispatchRaysIndex().xy].rgb + reflectivity * outputColor;
340     }
341
342     if (TriggerInfLoopHit)
343     {
344         while (outputColor.x <= 10.0)
345         {
346             outputColor.x *= sin(outputColor.x);
347         }
348     }
349
350     g_screenOutput[DispatchRaysIndex().xy] = float4(outputColor, 1);
351 }
352
```

FIX RENDERING ISSUES



PROBLEMS WITH RESOURCES

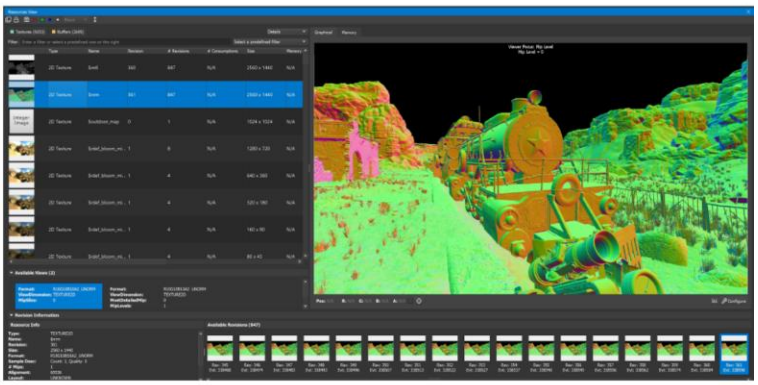
Incorrect textures

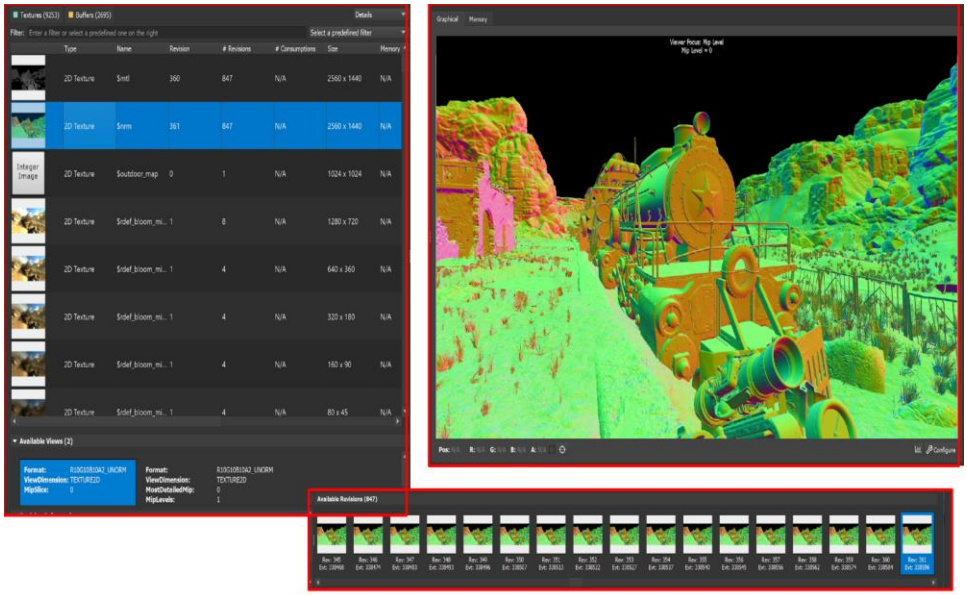
Bad buffer data

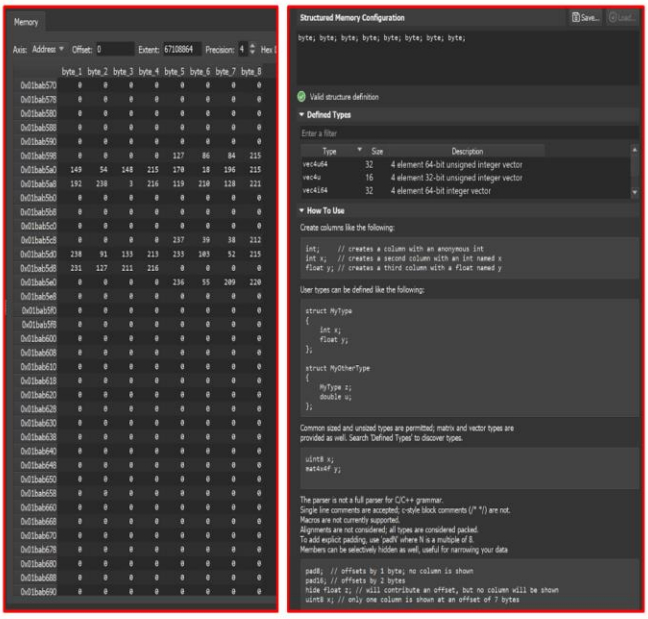
Wrong blending states

Resource memory utilization

Unnecessary updates







Nsight Graphics: Resource Inspector

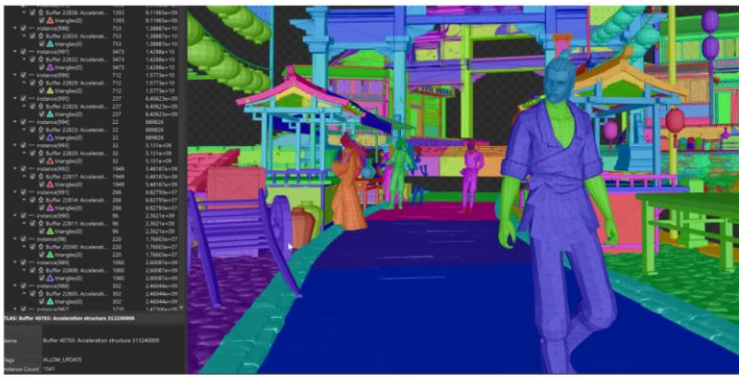
AS/BVH SETUP ISSUES

TLAS/BLAS flags

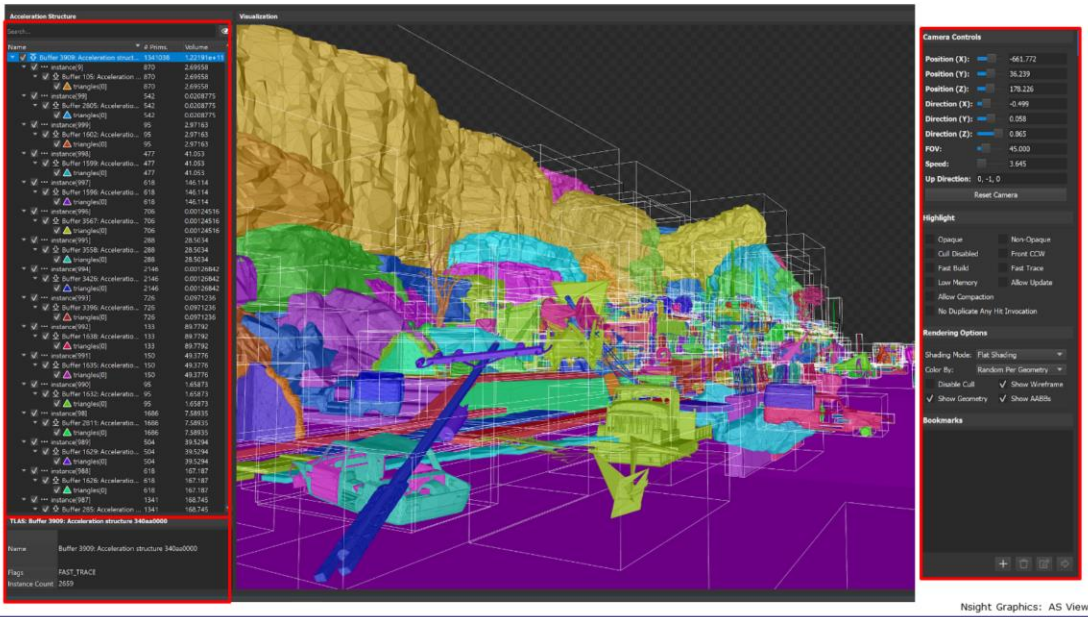
Large volume size

High geometry complexity

Too many BLAS instances



Nsight Graphics: AS Viewer



Nsight Graphics: AS Viewer

PROFILE & OPTIMIZE

STUTTERING/HITCHES

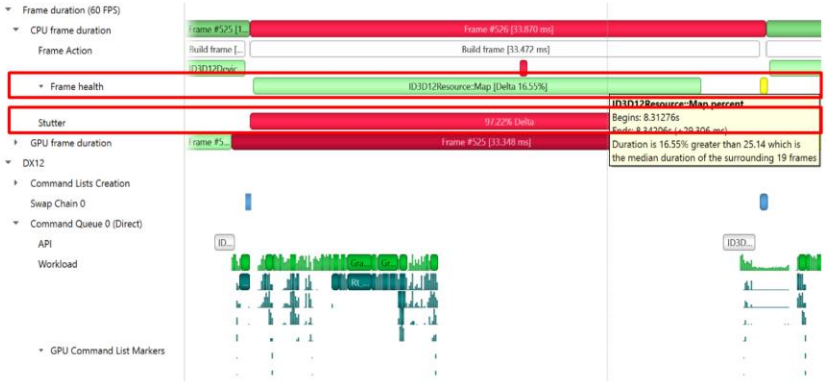
Shader compilation

Data copies

Redundant synchronizations

Redundant texture transitions

Resource loading



LOW GPU (WARP) OCCUPANCY

GPU Active timing

Copy engine in use

Shader type utilization

Synchronization issues

Async compute usage

Nsight Graphics: GPU Trace

SHADING & TRAVERSAL PERFORMANCE

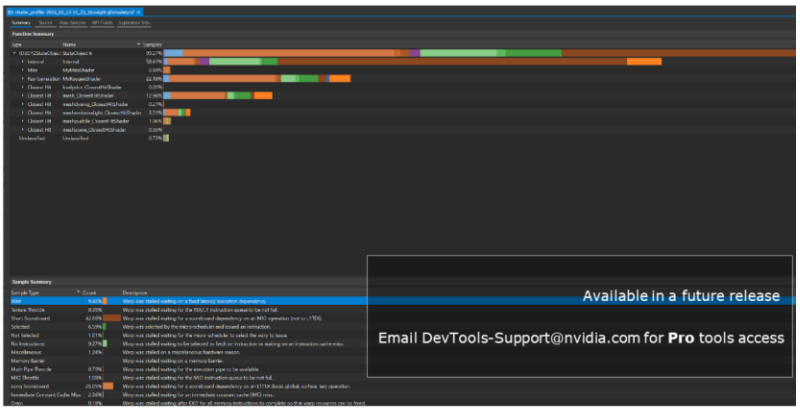
Memory barrier stalls

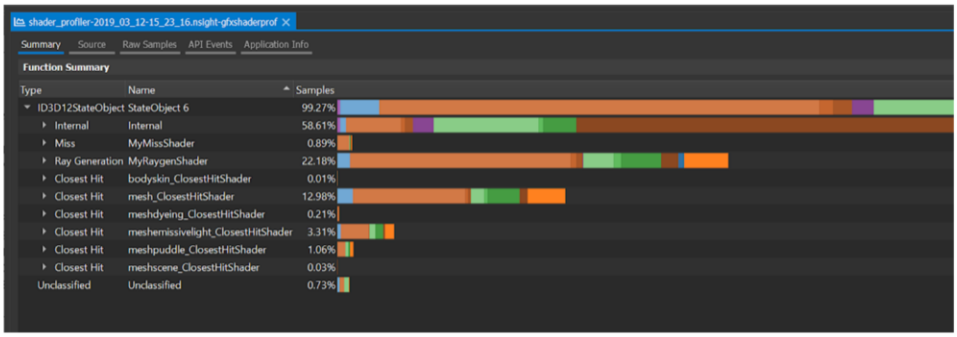
CTA barrier stalls

High instruction count

Cache misses

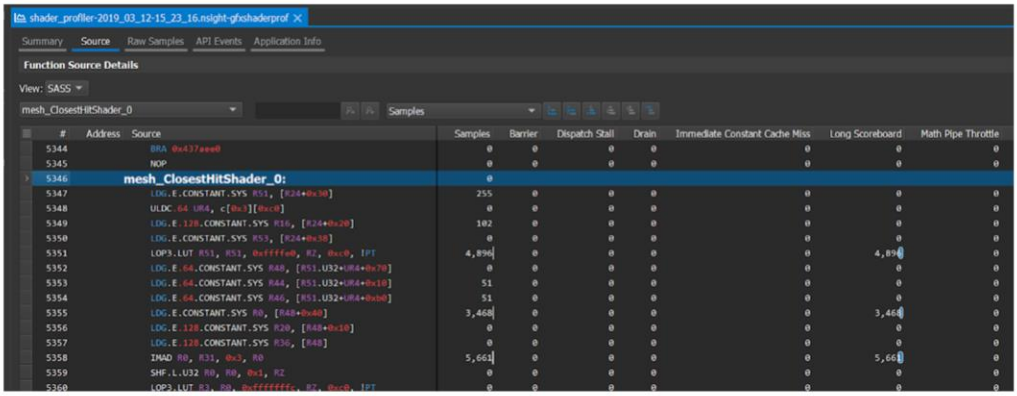
Sampling vs ALU





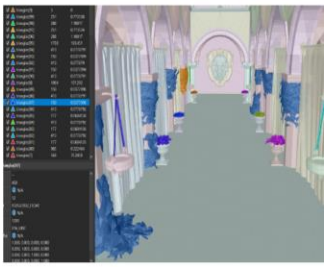
Sample Summary

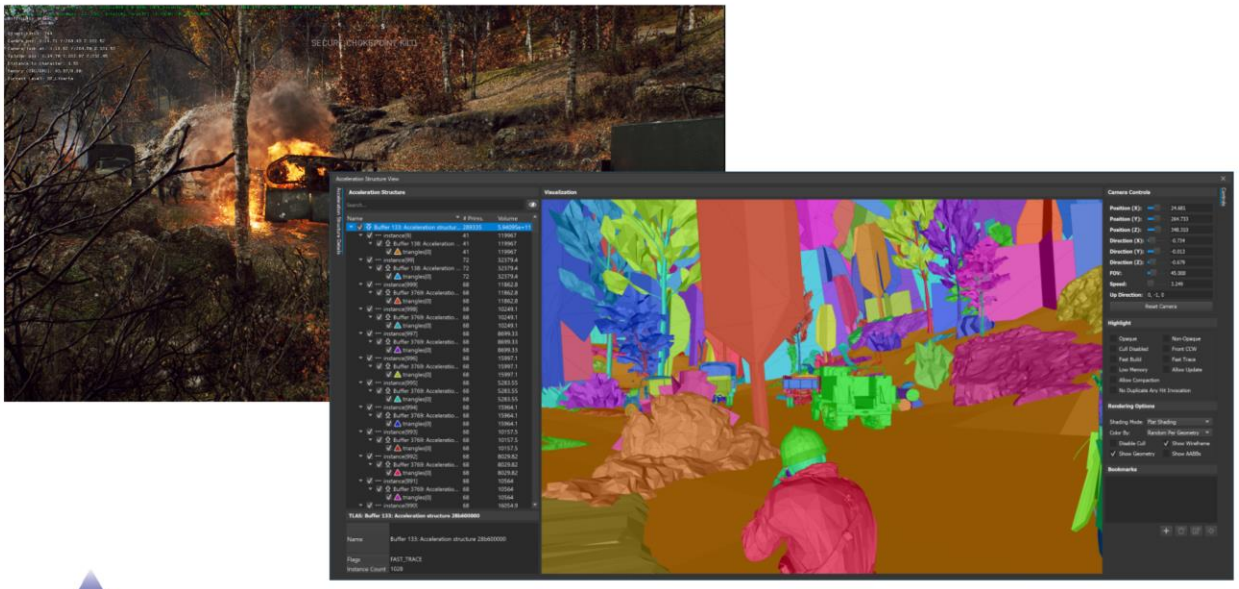
Sample Type	Count	Description
Wait	9.42%	Warp was stalled waiting on a fixed latency execution dependency.
Texture Throttle	0.35%	Warp was stalled waiting for the TEX/L1 instruction queue to be not full.
Short Scoreboard	42.66%	Warp was stalled waiting for a scoreboard dependency on an MIO operation (not to L1TEX).
Selected	6.59%	Warp was selected by the micro-scheduler and issued an instruction.
Not Selected	1.01%	Warp was stalled waiting for the micro-scheduler to select the warp to issue.
No Instructions	9.27%	Warp was stalled waiting to be selected to fetch an instruction or waiting on an instruction cache miss.
Miscellaneous	1.24%	Warp was stalled on a miscellaneous hardware reason.
Memory Barrier		Warp was stalled waiting on a memory barrier.
Math Pipe Throttle	0.79%	Warp was stalled waiting for the execution pipe to be available.
MIO Throttle	1.08%	Warp was stalled waiting for the MIO instruction queue to be not full.
Long Scoreboard	25.05%	Warp was stalled waiting for a scoreboard dependency on an L1TEX (local, global, surface, tex) operation.
Immediate Constant Cache Miss	2.36%	Warp was stalled waiting for an immediate constant cache (IMC) miss.
Drain	0.18%	Warp was stalled waiting after EXIT for all memory instructions to complete so that warp resources can be freed.
Dispatch Stall	< 0.01%	Warp was stalled waiting on a dispatch stall.
Barrier		Warp was stalled waiting for sibling warps at a CTA barrier.



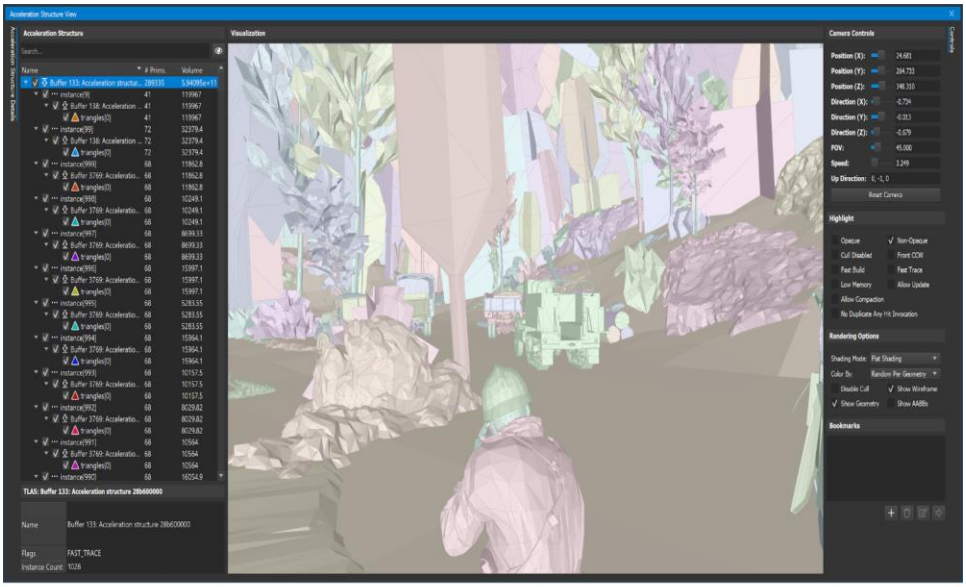
Case Study: Non-Opaque BLAS

- Individual BLAS geometry that are not see-through must use the 'Opaque' flag
- Non-opaque is the normal default for *everything*
- This can result in a performance penalty since rays that intersect with non-opaque geometry continuously restart in SMs
- **What do we do?**
  - AS Viewer filter can verify that BLAS geometry is properly configured



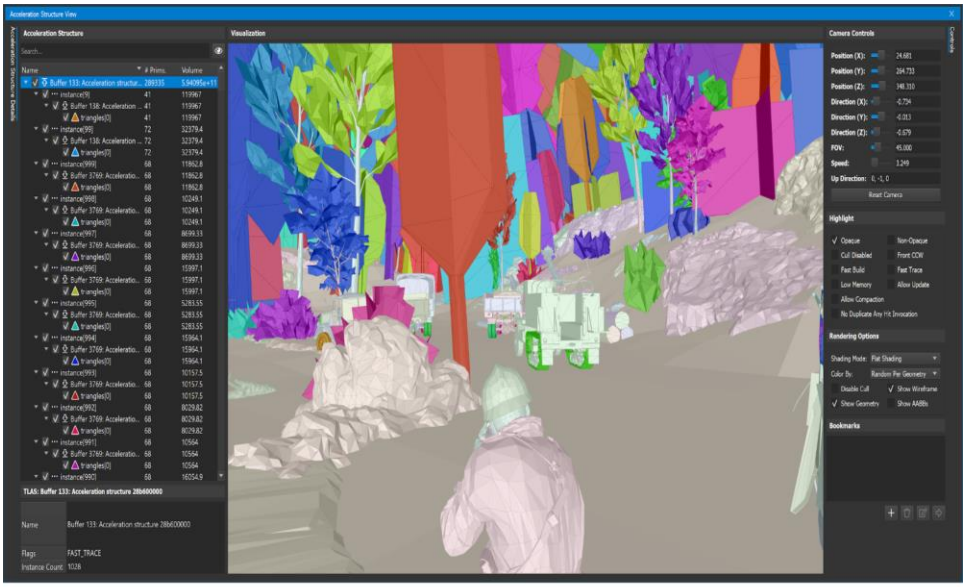


Nsight Graphics: AS Viewer


Acceleration Structure View
Acceleration Structure
Visualization
Camera Controls
Highlight
Rendering Options
Bookmarks
Nsight Graphics: AS Viewer

Case Study: Non-Opaque BLAS

```
1   D3D12_RAYTRACING_GEOMETRY_DESC descriptor = {};
2   descriptor.Type = D3D12_RAYTRACING_GEOMETRY_TYPE_TRIANGLES;
3   descriptor.Triangles.VertexBuffer.StartAddress = vertexBuffer->GetGPUVirtualAddress() + vertexOffsetInBytes;
4   descriptor.Triangles.VertexBuffer.StrideInBytes = vertexSizeInBytes;
5   descriptor.Triangles.VertexCount = vertexCount;
6   descriptor.Triangles.VertexFormat = DXGI_FORMAT_R32G32B32_FLOAT;
7   descriptor.Triangles.IndexBuffer = indexBuffer ? (indexBuffer->GetGPUVirtualAddress() + indexOffsetInBytes) : 0;
8   descriptor.Triangles.IndexFormat = indexBuffer ? DXGI_FORMAT_R32_UINT : DXGI_FORMAT_UNKNOWN;
9   descriptor.Triangles.IndexCount = indexCount;
10  descriptor.Triangles.Transform = transformBuffer ? (transformBuffer->GetGPUVirtualAddress() + transformOffsetInBytes) : 0;
11  descriptor.Flags = isOpaque ? D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE : D3D12_RAYTRACING_GEOMETRY_FLAG_NONE;
```



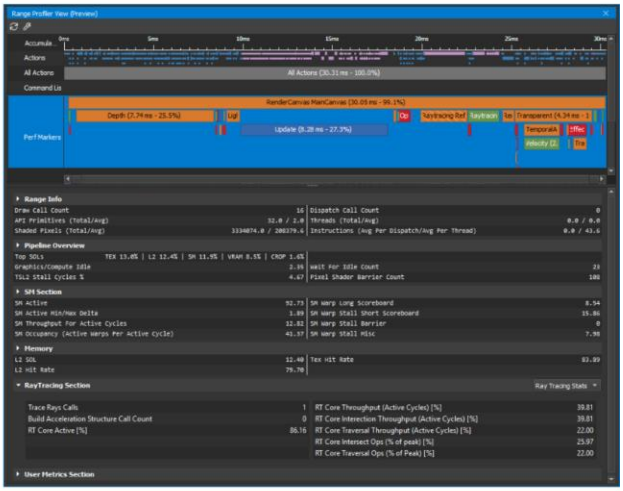
Nsight Graphics: AS Viewer

Case Study: AS Build Time

- Low 'Top SOL' metrics indicate poor utilization of GPU during AS Build

Tex	L2	SM	VRAM	CROP
13%	12.4%	11.9%	8.5%	1.6%

- Compute & Graphics work regimes can be executed simultaneously
- **What do we do?**
  - Move AS building to overlap with work happening on async compute queue

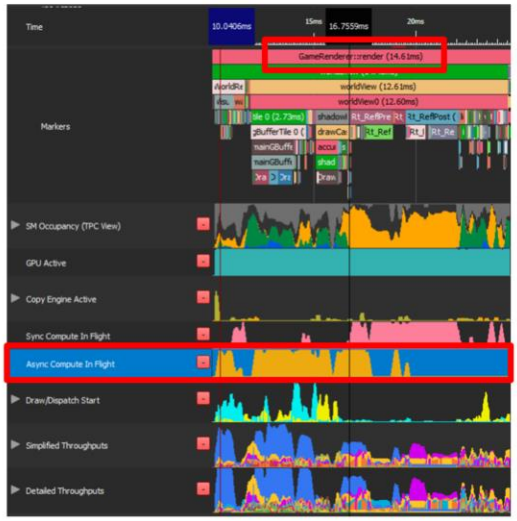


CASE STUDY: AS BUILD TIME

10% IMPROVEMENT!

AS BUILD *WITHOUT* ASYNC COMPUTE

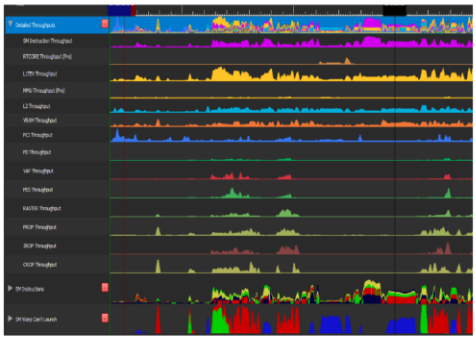
AS BUILD *WITH* ASYNC COMPUTE



GPU Trace: Metrics Graphs

NEXT STEP: ANALYZE THROUGHPUT…

AS BUILD *WITHOUT* ASYNC COMPUTE



AS BUILD *WITH* ASYNC COMPUTE

Top 5 Lessons

1. Merge BLAS to keep number of instances low
2. Reduce shading complexity as much as possible
3. Avoid shader compilation at runtime to avoid stutter
4. Prefer to Refit BLAS and Rebuild TLAS
5. Consider use of tracing for things like Water & Terrain

For more tips, check out the NVIDIA Dev Blog:
https://devblogs.nvidia.com/

NVIDIA

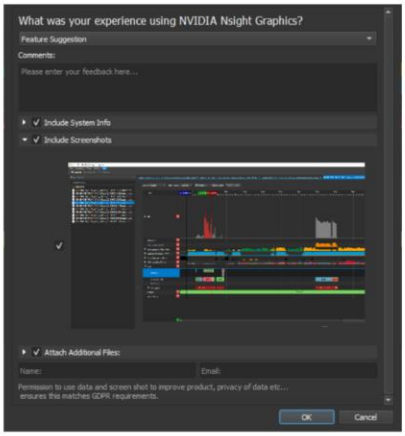
RAY TRACING SESSIONS

Tuesday, March 19 2:40pm	**Introduction to DirectX Raytracing**
Wednesday, March 20 12:30pm	**Graphics Reinvented: RTX Update**
Thursday, March 21 4:00pm	**Speed of Light DXR Ray Tracing with NVIDIA Nsight Graphics**
Thursday, March 21 5:30pm	**Getting The Most From Your Vulkan Applications with NVIDIA Nsight Graphics**
Friday, March 22 10:00am	**Optimizing DX12/DXR GPU Workloads Using Nsight Graphics: GPU Trace and the Peak-Performance-Percentage Method**

Full list at *https://developer.nvidia.com/gdc19-show-guide-nvidia*

THANK YOU!

Questions?



Get it at **developer.nvidia.com/nsight-graphics**

Contact us at **NsightGraphics@nvidia.com**

Visit our booth: **Booth #S466, South Hall**

More sessions: **developer.nvidia.com/gdc19-show-guide-nvidia**

Disclaimer and Attribution