

The GDC logo is positioned at the top center of the slide. It consists of the letters 'GDC' in a bold, white, sans-serif font, set against a red downward-pointing triangle. The background of the slide is dark blue with a red 'X' shape and small diamond icons at the corners.

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



GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18–22, 2019 | #GDC19

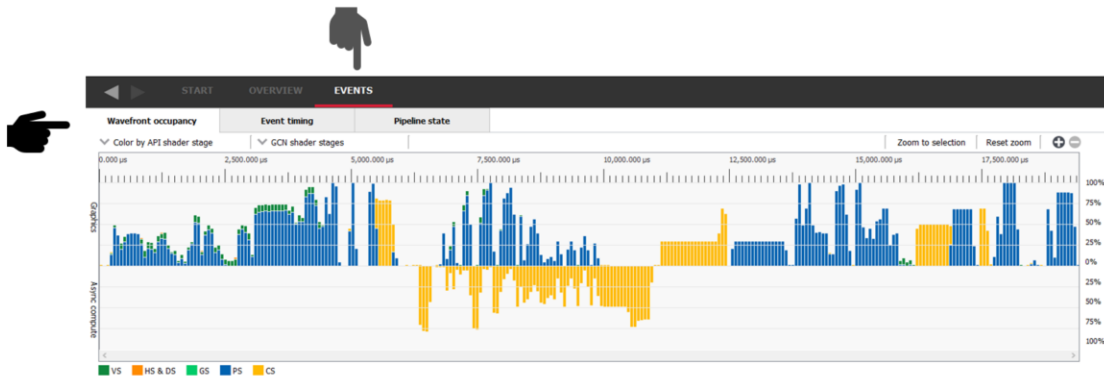
Now this is how the talk will look like:

- 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.

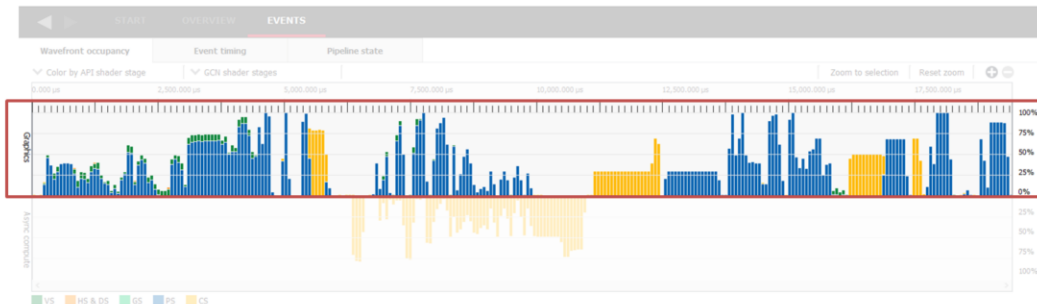


GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

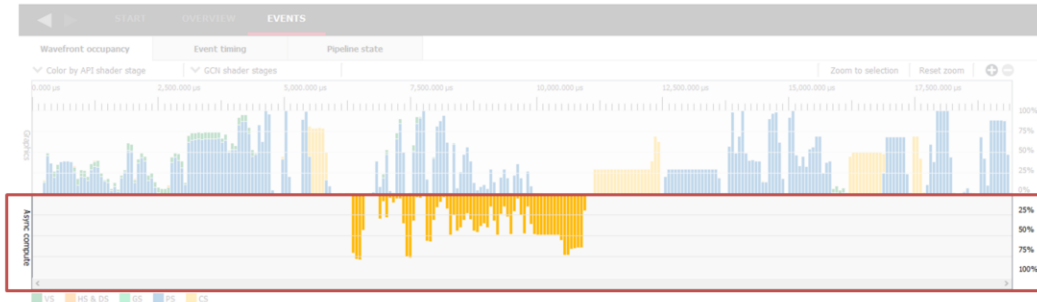
You can access this via Events-  
>Wavefront occupancy.

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.

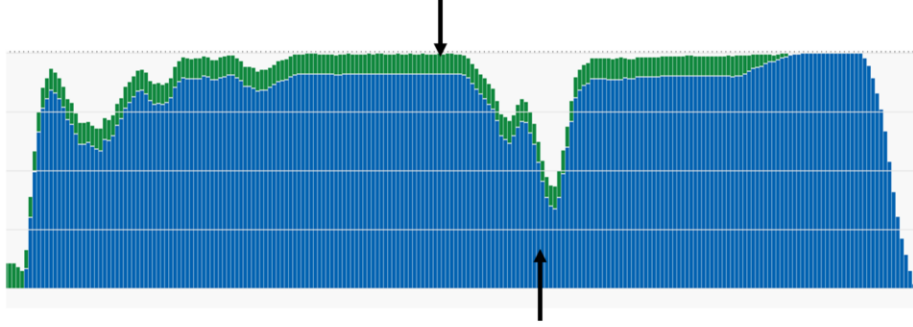
This is probably the view you will spend your first few minutes on to figure out what's going on in your frame.



You can access this via Events->Wavefront occupancy.

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

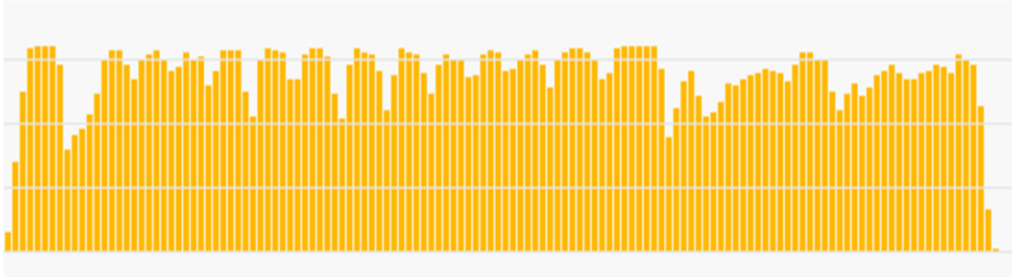
Vertex Shader Wavefronts



Pixel Shader Wavefronts

GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19



Compute Shader Wavefronts

**GDC**

**GAME DEVELOPERS CONFERENCE**  
MARCH 18-22, 2019 | #GDC19

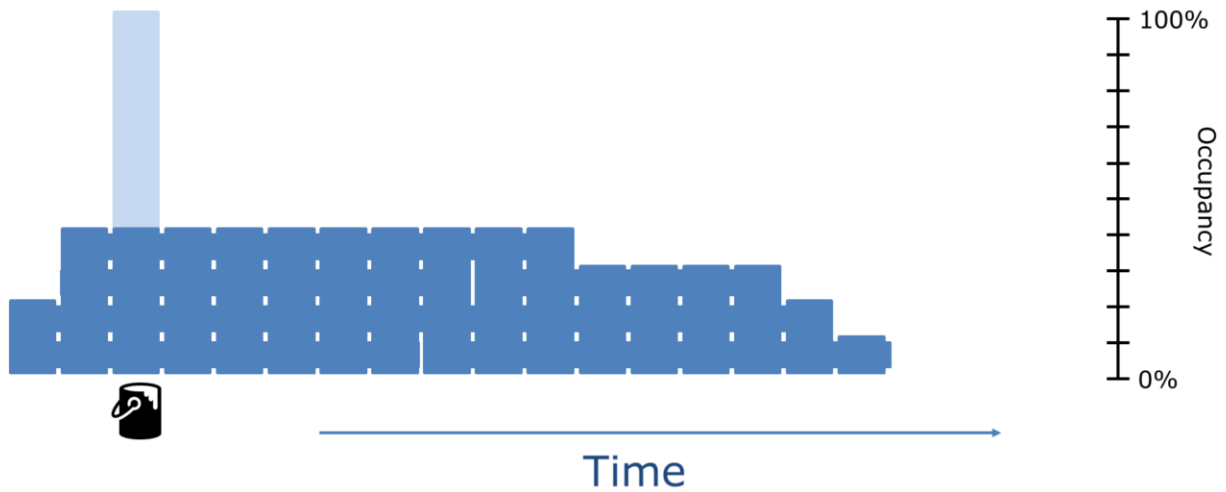




May I introduce: a wavefront.

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.

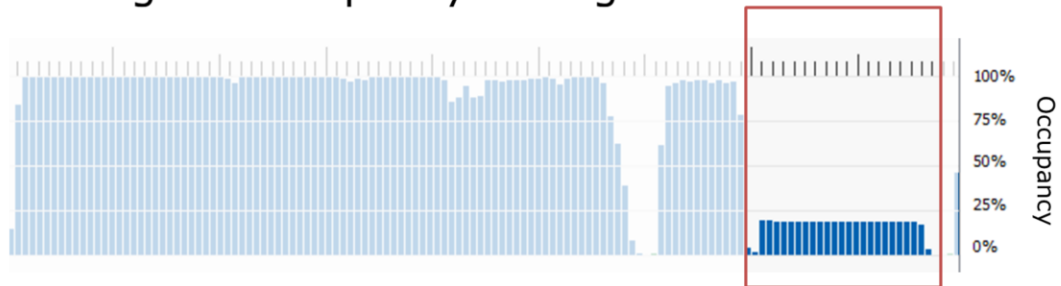
If our hypothetical GPU could run 10 wavefronts at once, RGP would state

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.



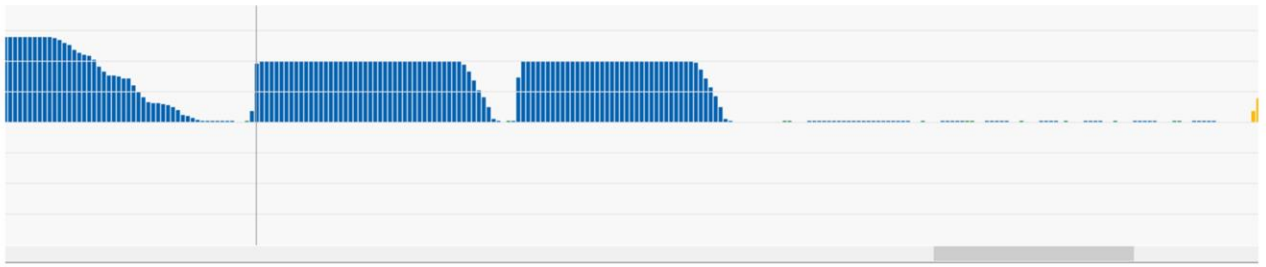
GDC

GAME DEVELOPERS CONFERENCE

MARCH 18-22, 2019 | #GDC19

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.

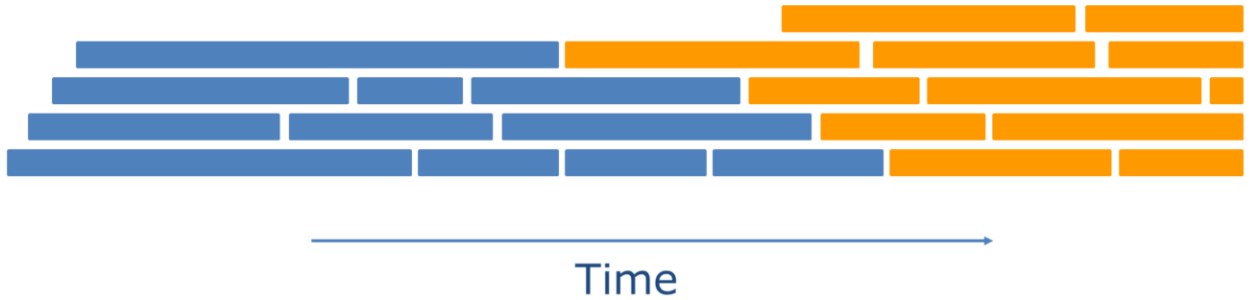


Barriers

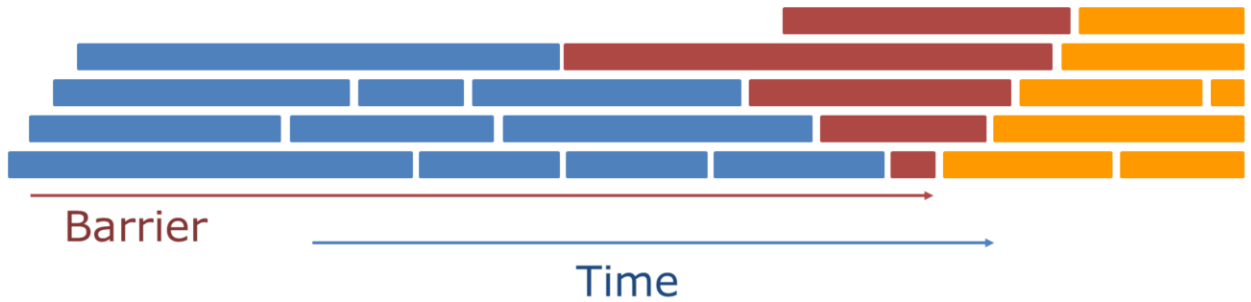
GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

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.

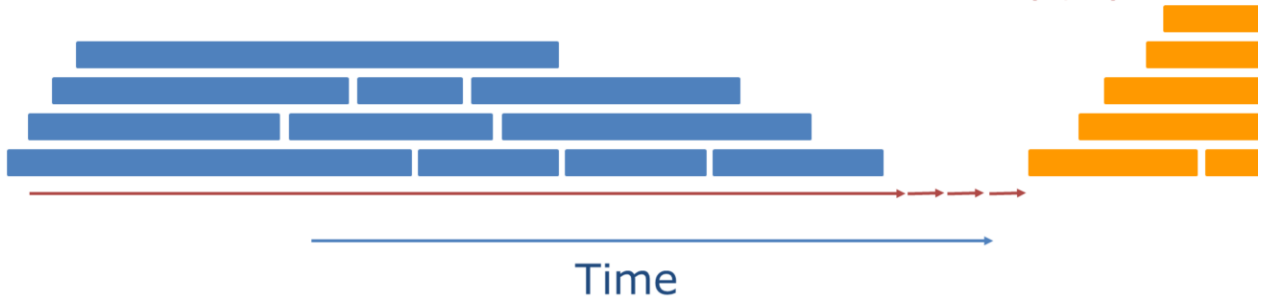


Let's concentrate on how we can solve issues like this.

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

## 1. Batch barriers

```
cmdList.ResourceBarrier(1, ...)  
cmdList.ResourceBarrier(1, ...)  
cmdList.ResourceBarrier(1, ...)  
cmdList.ResourceBarrier(1, ...)
```



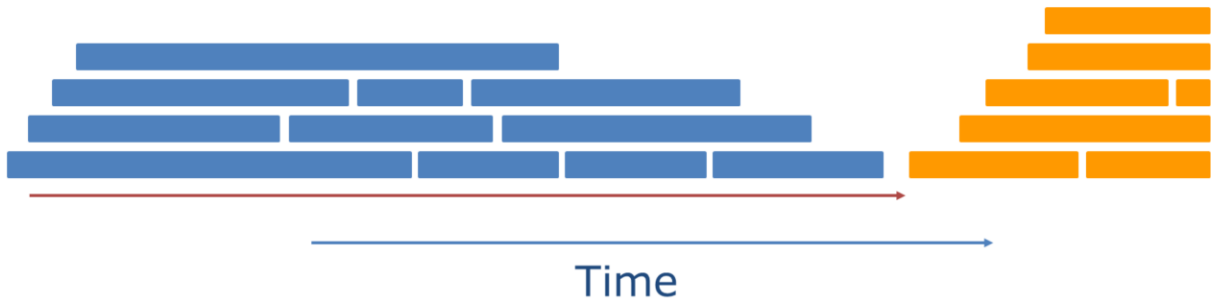
First, we can batch them, so the driver can manage the synchronization points more precisely.

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

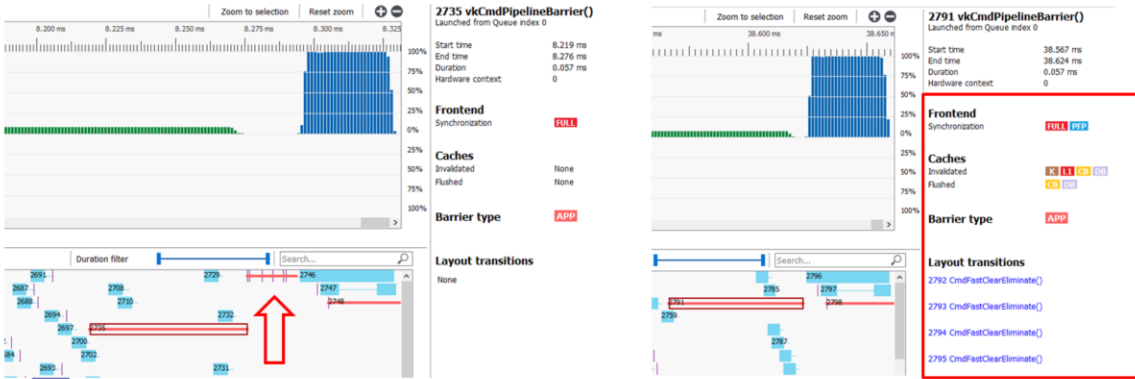
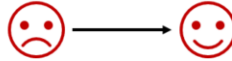
`cmdList.ResourceBarrier(4, ...)`



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

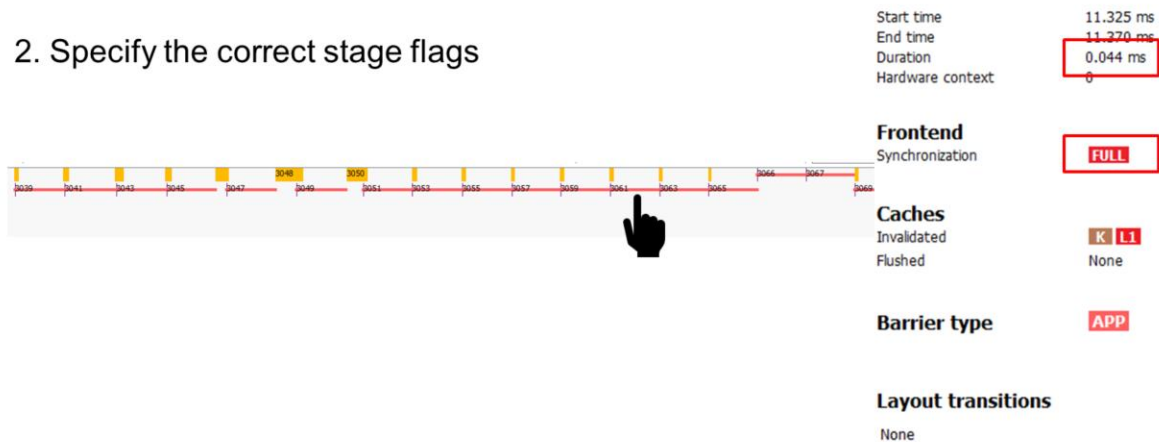
Also, certain compressions may start earlier, that's a net win!

# 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

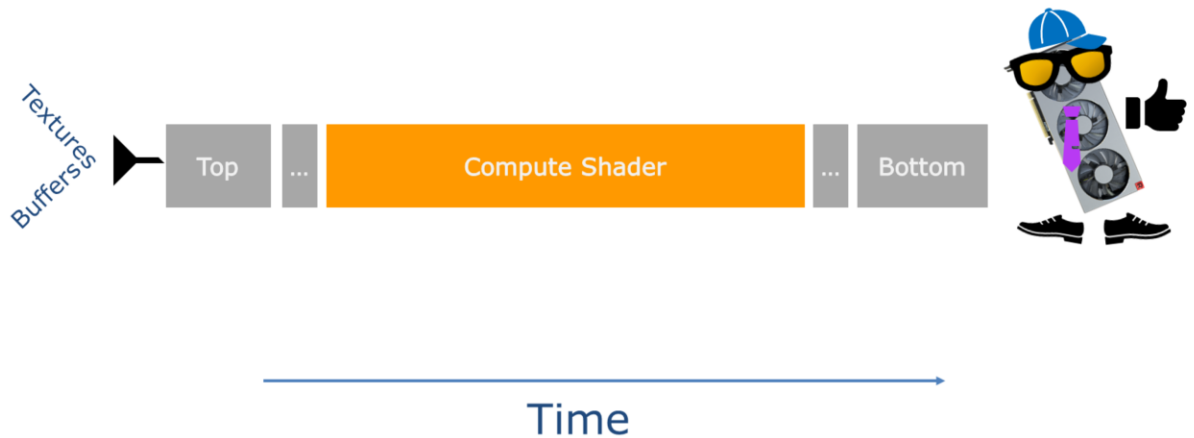


Be very descriptive in how you define the stage flags. In some cases you pay for being too general.

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.

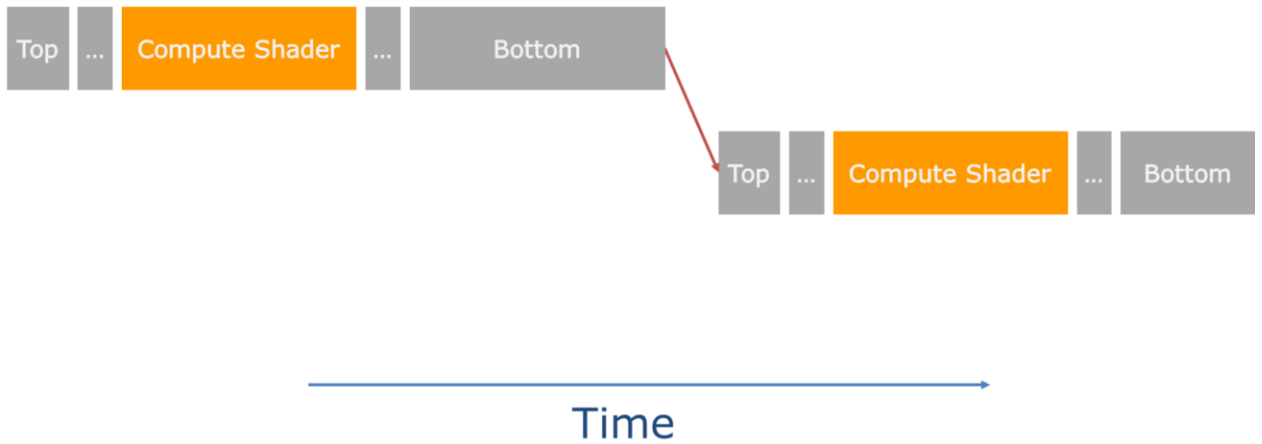
Doing a FULL synchronization each.



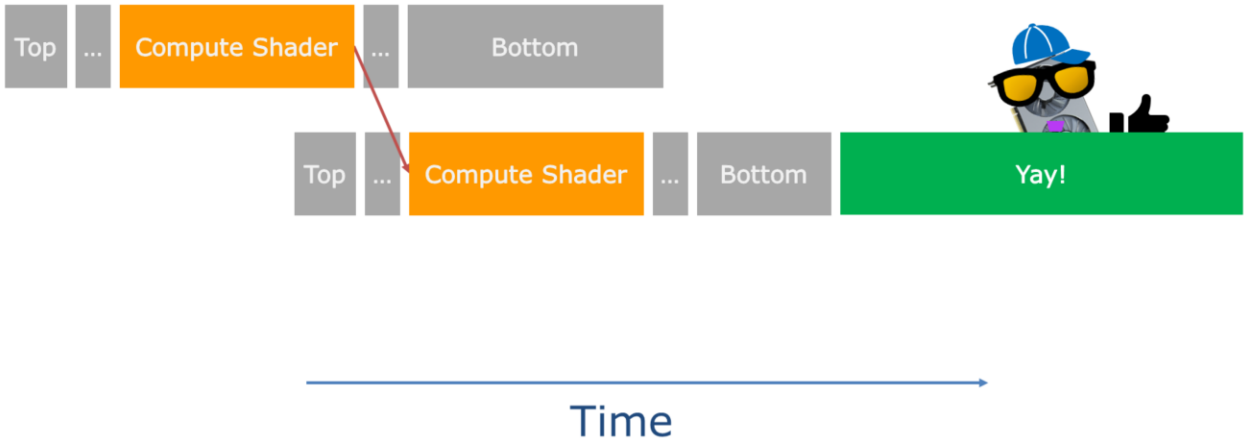
Pretend we would go through the GPU pipeline in a straight line. Compute only here.

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

Start time	11.325 ms	Start time	10.257 ms
End time	11.370 ms	End time	10.261 ms
Duration	0.044 ms	Duration	0.004 ms
Hardware context	0	Hardware context	0
<b>Frontend</b>		<b>Frontend</b>	
Synchronization	FULL	Synchronization	CS
<b>Caches</b>		<b>Caches</b>	
Invalidated	K L1	Invalidated	K L1
Flushed	None	Flushed	None
<b>Barrier type</b>	APP	<b>Barrier type</b>	APP
<b>Layout transitions</b>		<b>Layout transitions</b>	
None		None	

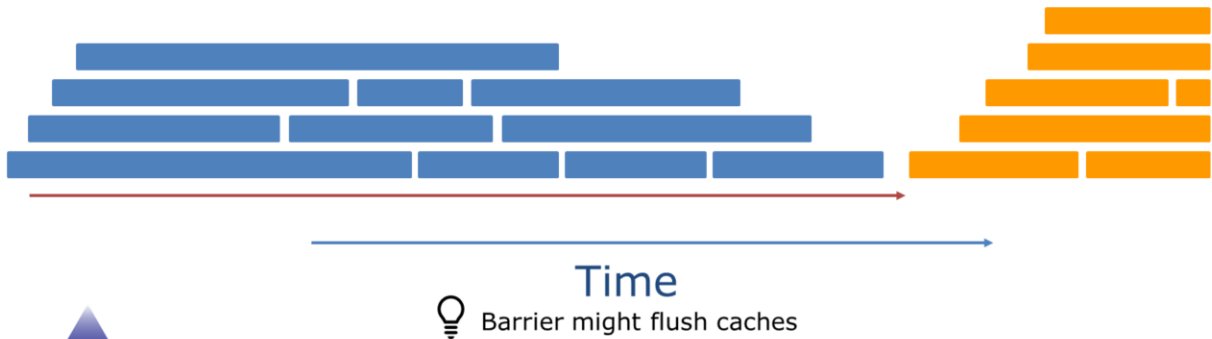
To prove it, here's the detailed view when clicking on these barriers.

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.

3. Move barriers → Overlap independent work

Draw()  
**Barrier()**  
Dispatch()



GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

Work that is independent does not necessarily have to wait.

→ Move the barrier after independent workloads.

Added benefit: barriers often flush caches as well.

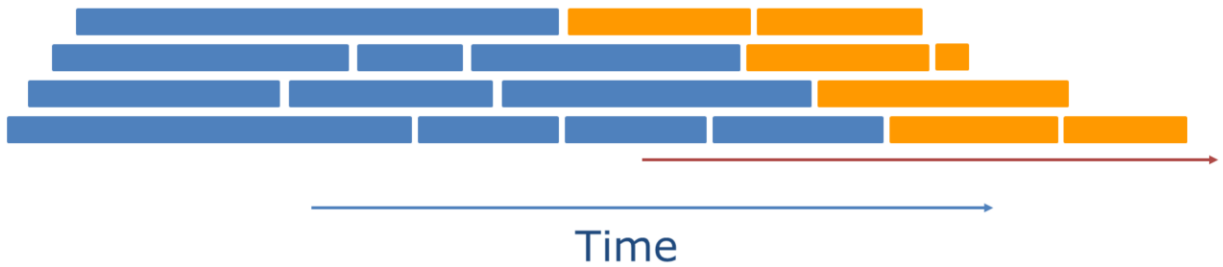
If PS and CS share reads here, then the CS may run slower as it doesn't hit in L2 anymore.

Order of submissions here:

1. VS/PS
2. Barrier
3. CS

3. Move barriers → Overlap independent work

Draw()  
Dispatch()  
**Barrier()**



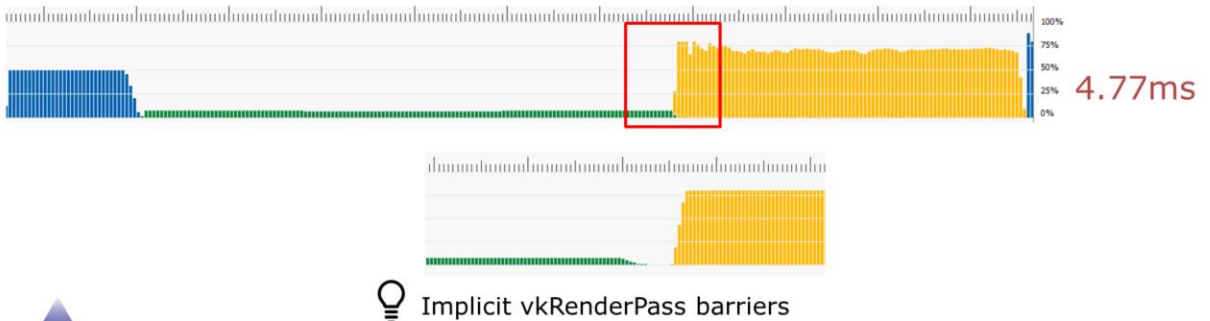
GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

Order of submission would be:

1. PS
2. CS
3. Barriers

### 3. Move barriers → Overlap independent work



GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18–22, 2019 | #GDC19

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.

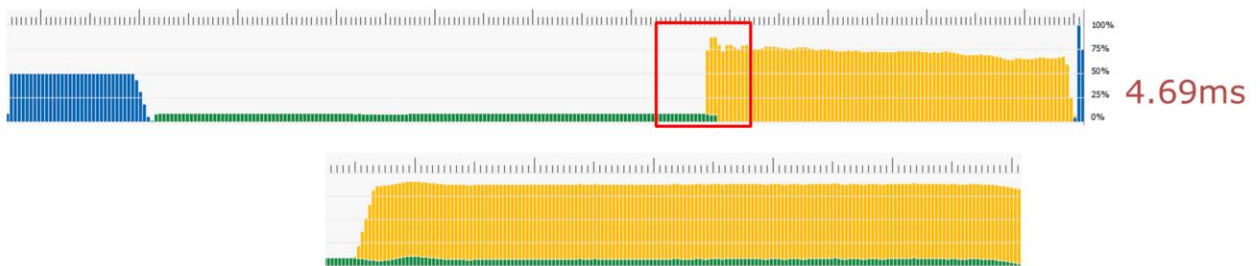
This is what we start out with. Notice the barrier that causes a gap between the Vertex and the Compute work.

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



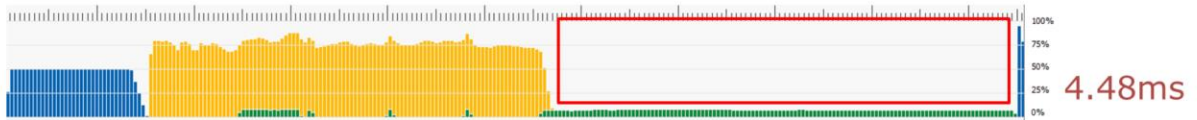
GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18–22, 2019 | #GDC19

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



GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18–22, 2019 | #GDC19

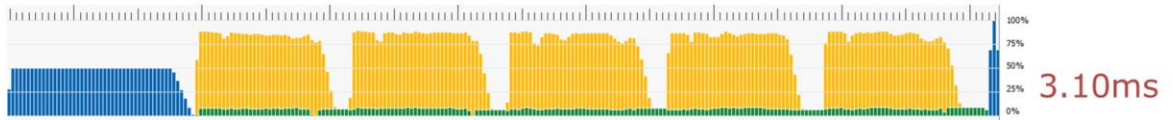
This already looks better than before. Still not ideal as at some point we only spawn new CS waves, no VS waves anymore.

→ 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

GDC

GAME DEVELOPERS CONFERENCE

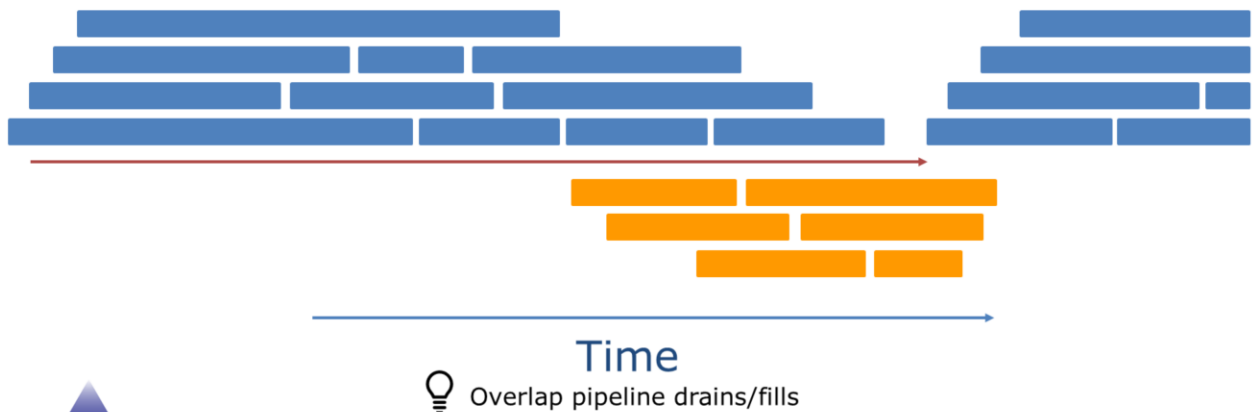
MARCH 18–22, 2019 | #GDC19

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.

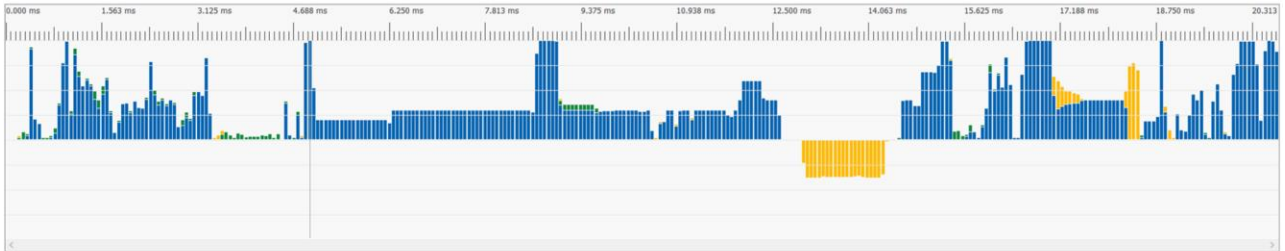
#### 4. Asynchronous compute



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



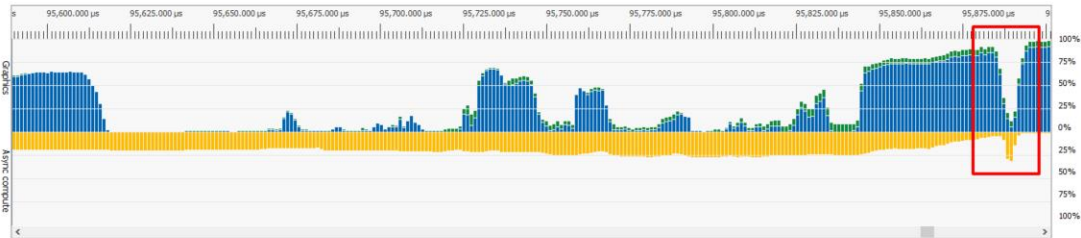
~\\_(ツ)\_/~

GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

That probably didn't turn out as expected.

#### 4. Asynchronous compute

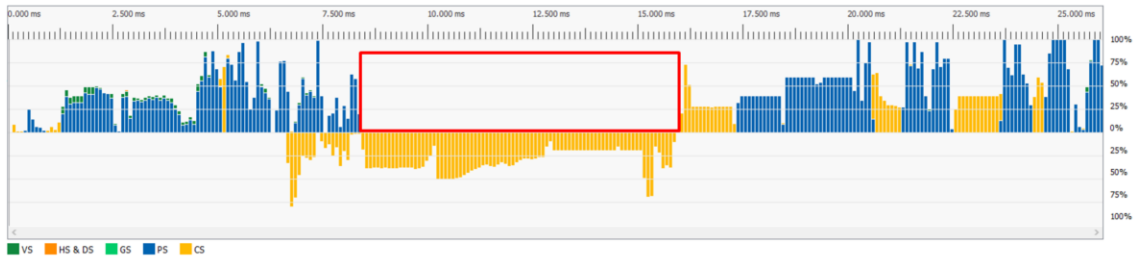


GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

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.

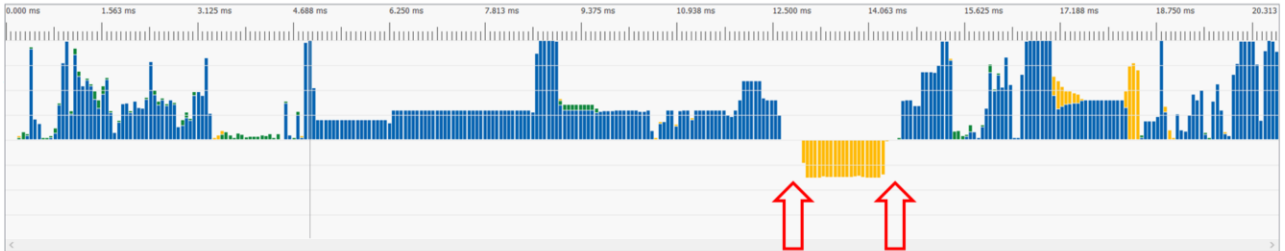
Rather aim to keep the graphics queue busy first.



Wavefronts & Barriers   
**Cross queue synchronization**  
Compression

GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

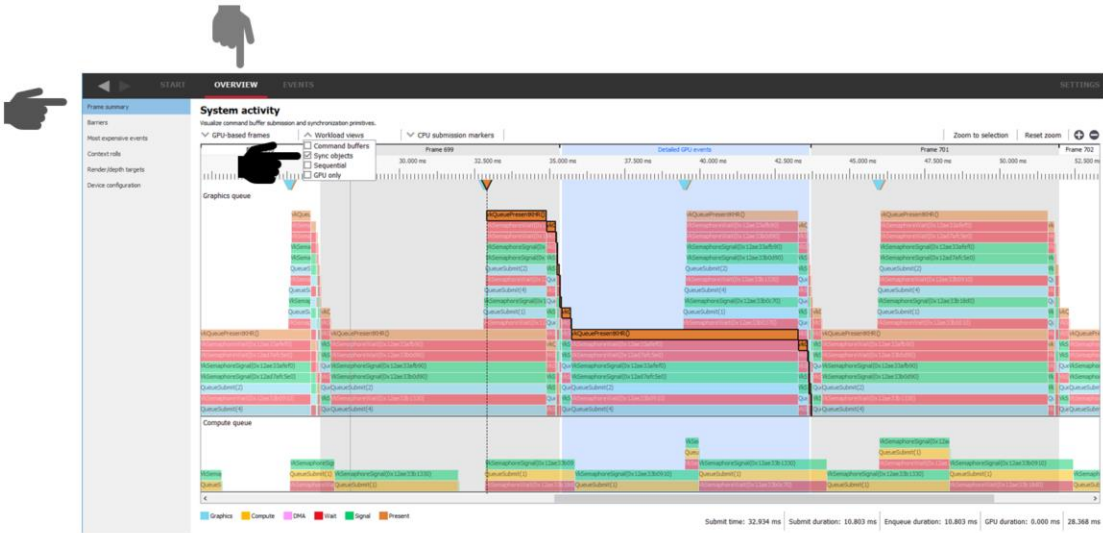


GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

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

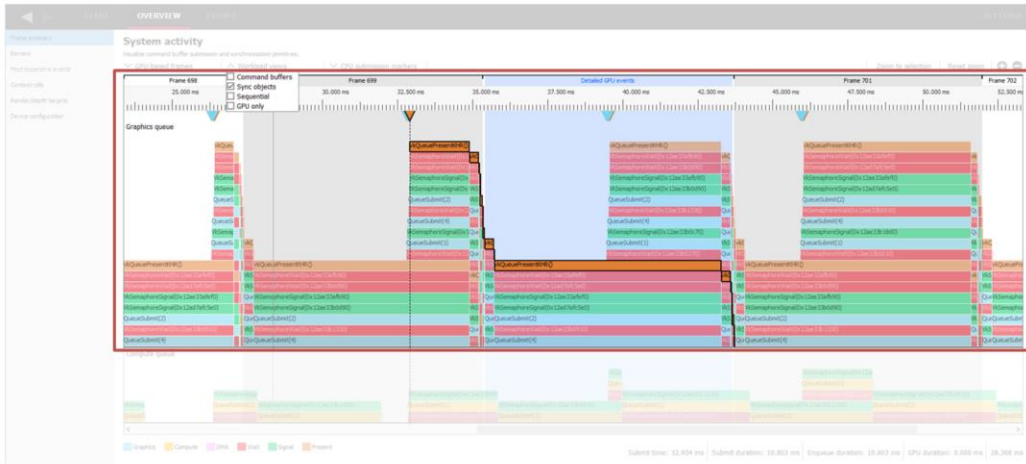
Let's head over to a different view.



GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18–22, 2019 | #GDC19

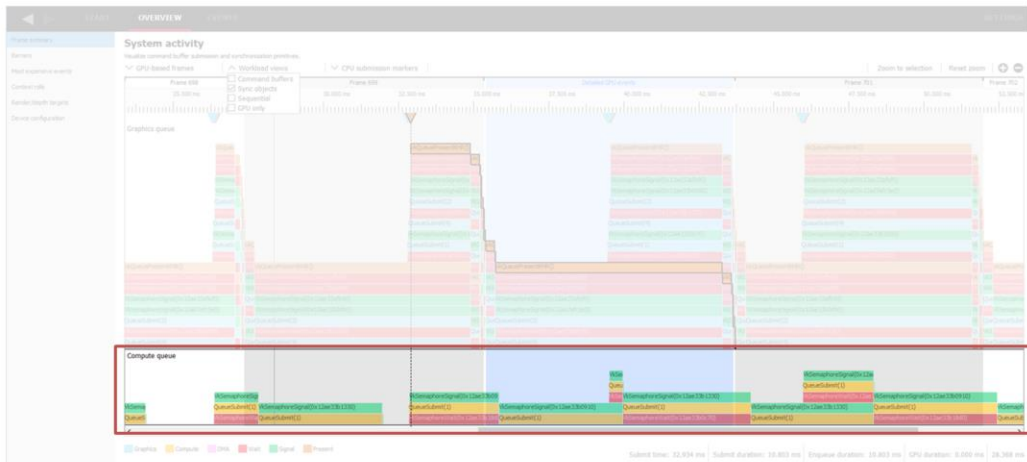
When you open up your capture you are greeted with the frame summary page. Anyone working with GPUView might see similarities.



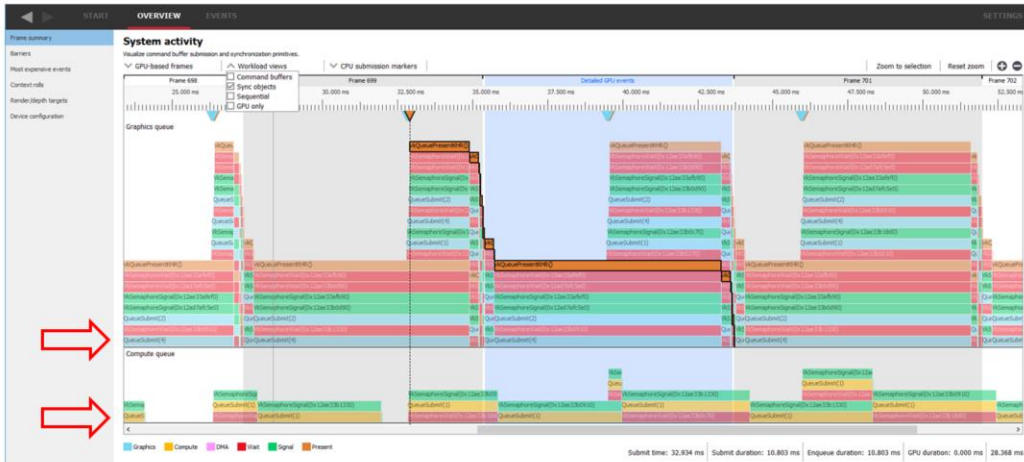
GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18–22, 2019 | #GDC19

Top is the graphics queue.



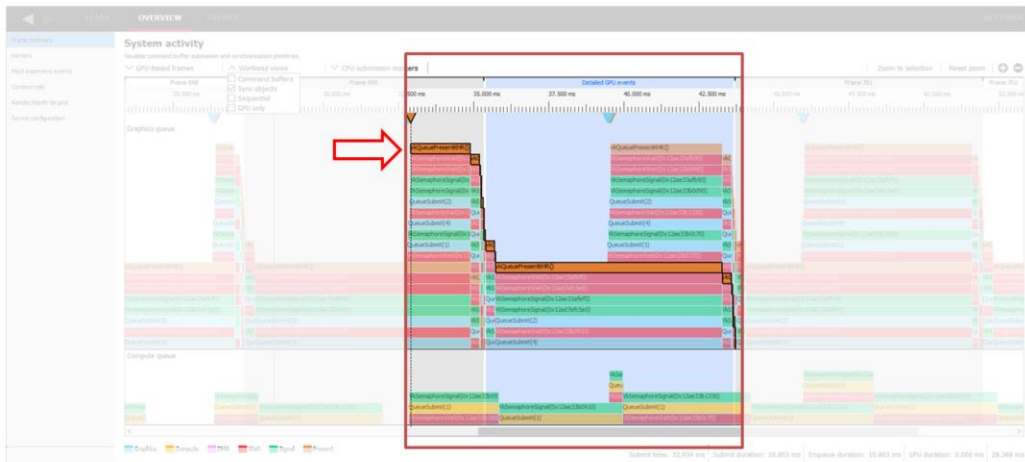
Bottom is the async compute queue  
(and/or copy queue if available)



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

Each time one finishes, the others drop one level.

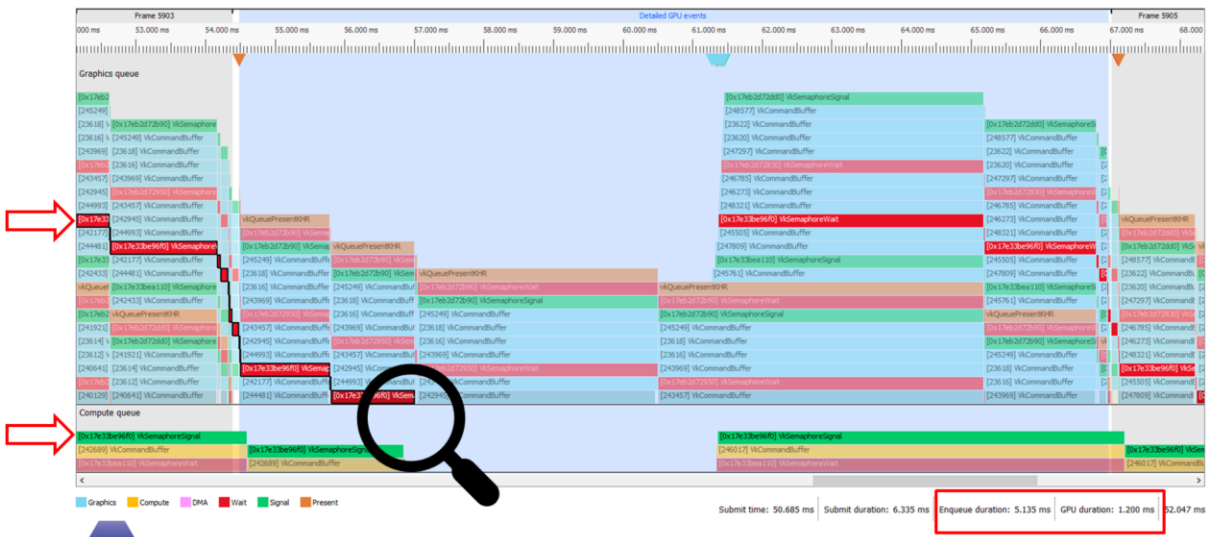
The lowest level are the work items the GPU queue worked on at that time.



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).

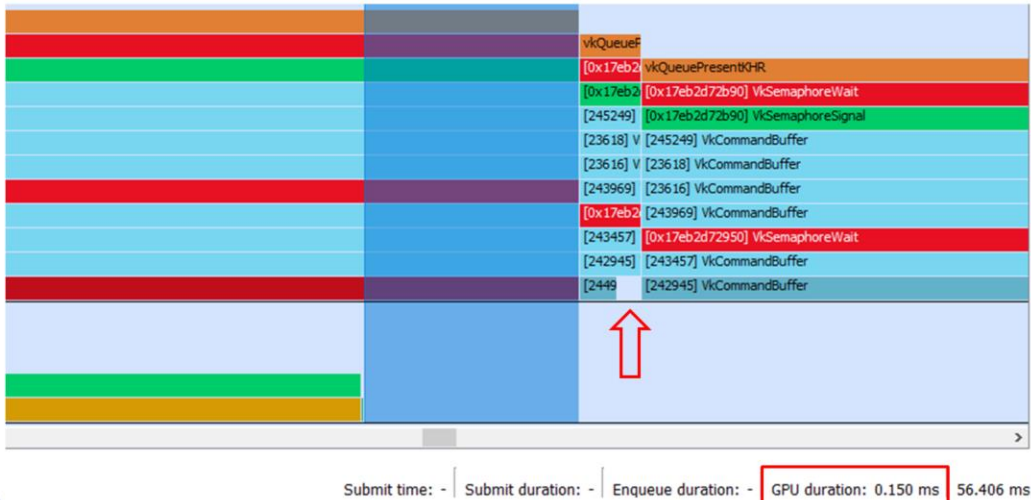
Clicking on one of them shows related signal/waits.

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

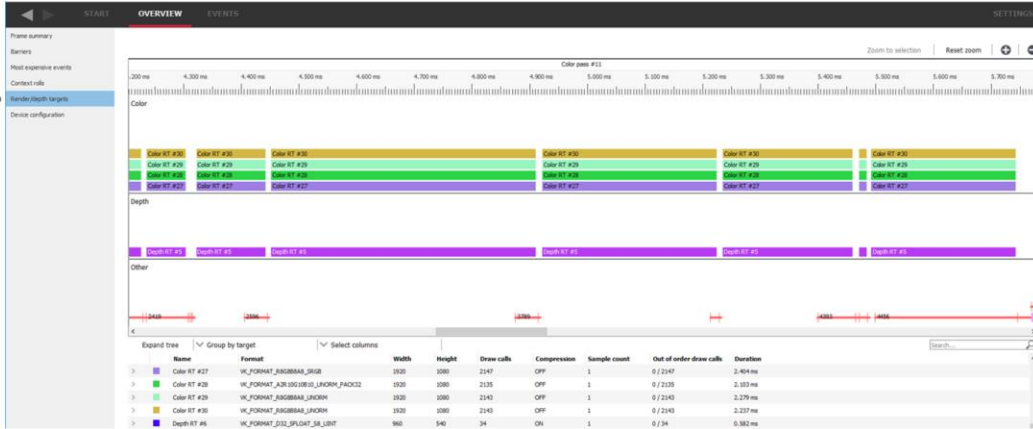
with compute heavy post processing.

Wavefronts & Barriers ✓  
Cross queue synchronization ✓  
**Compression**

GDC

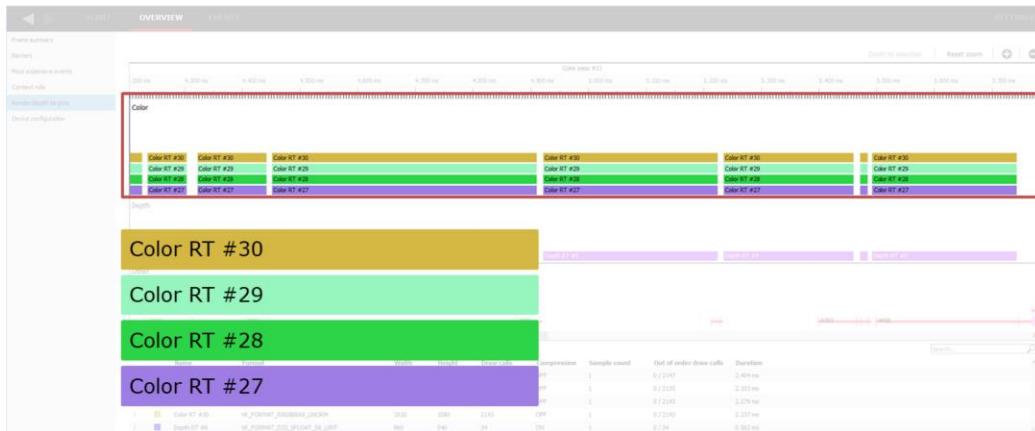
GAME DEVELOPERS CONFERENCE

MARCH 18-22, 2019 | #GDC19



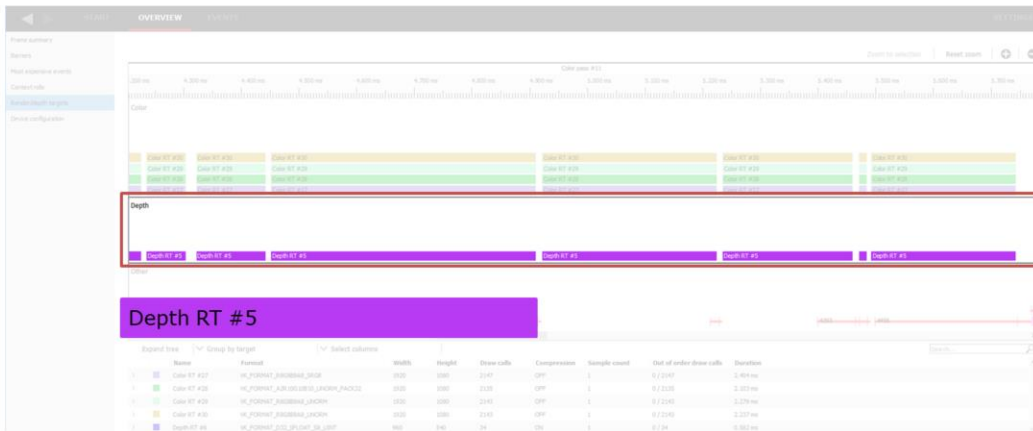
GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19



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.



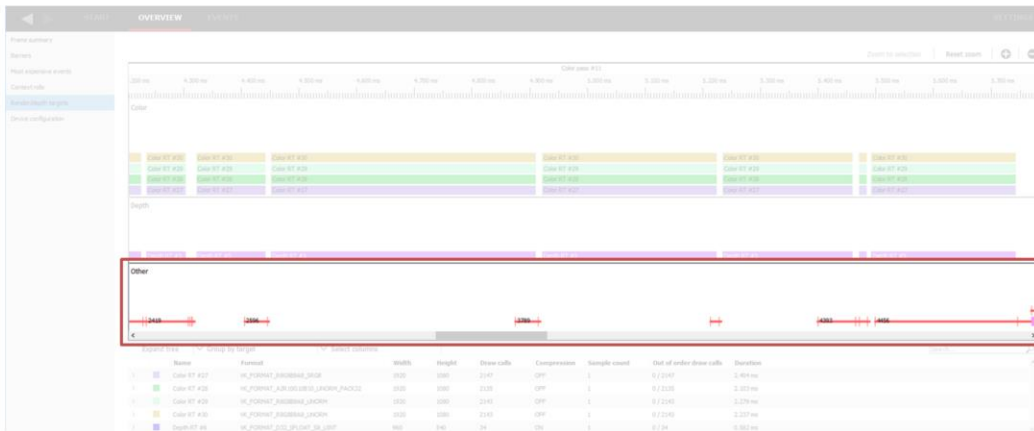
This view is not split into queues, but rather accesses to resources.

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.



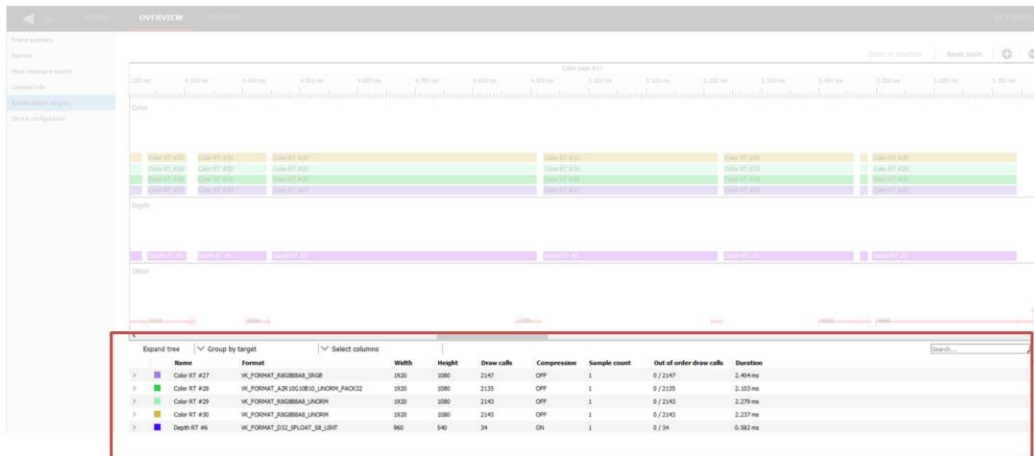


This view is not split into queues, but rather accesses to resources.

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).

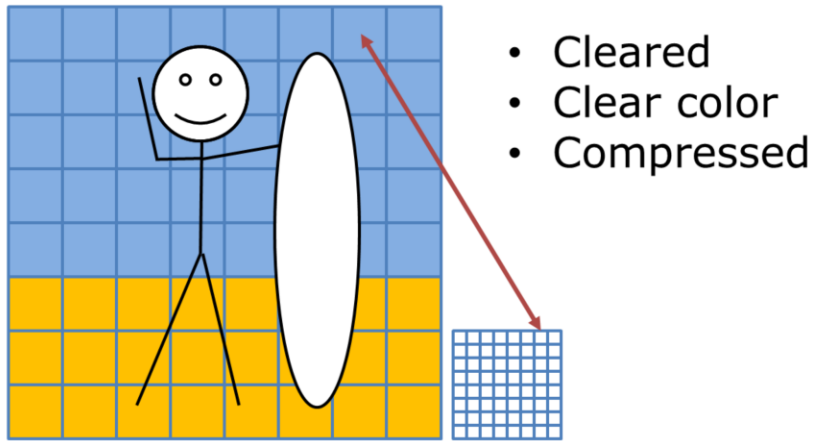
Side note: May want to take care of these barriers splitting up the Gbuffer pass.



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



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

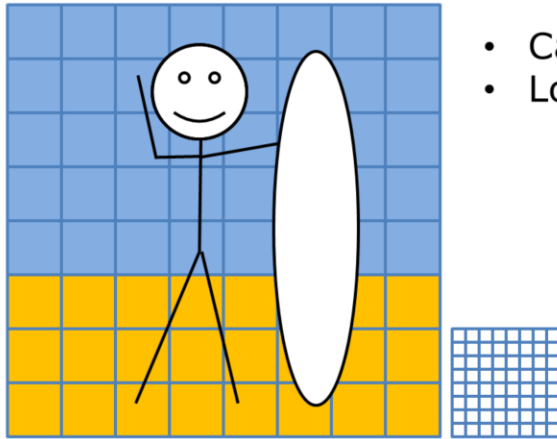
- Cleared
- Clear Color
- Compressed / Decompressed

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.

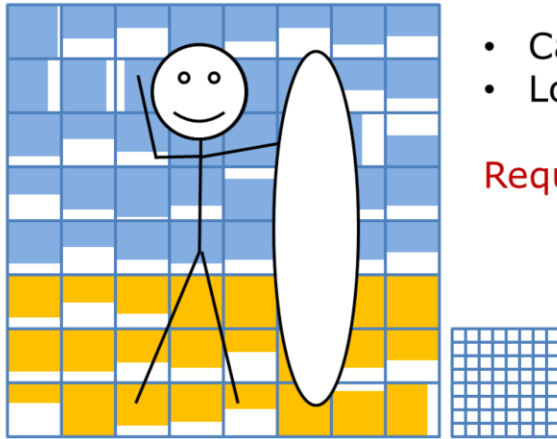


- Can only load meta data
- Load only parts of a block

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

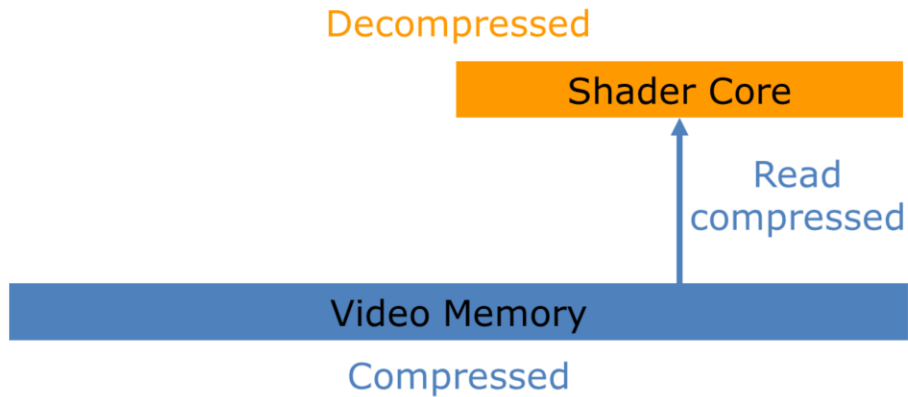
GDC

GAME DEVELOPERS CONFERENCE

MARCH 18-22, 2019 | #GDC19

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

Sample()



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

GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

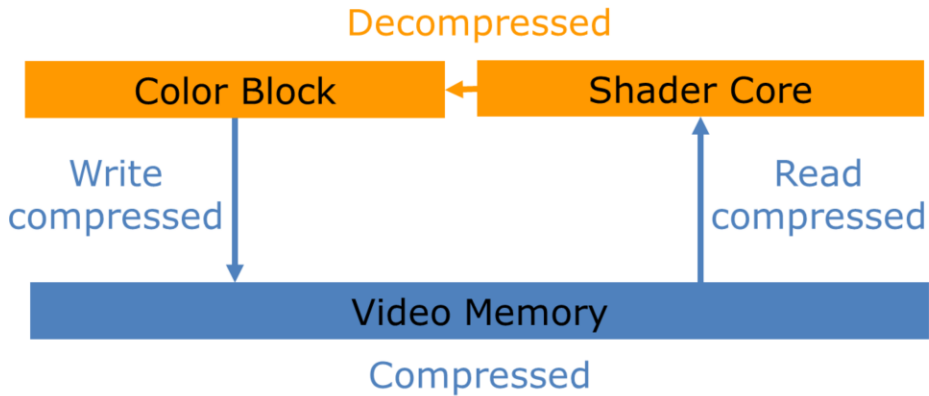
Cycle through sampling a texture and exporting color values to a render target.

→ 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!



Export()



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

GDC

GAME DEVELOPERS CONFERENCE  
MARCH 18-22, 2019 | #GDC19

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 ✓

GDC

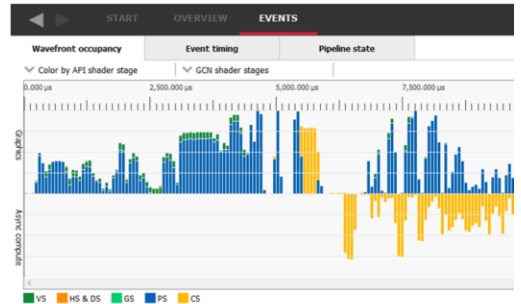
GAME DEVELOPERS CONFERENCE

MARCH 18-22, 2019 | #GDC19

# Wrap-Up

## Barriers

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



# Wrap-Up

## Cross queue synchronization

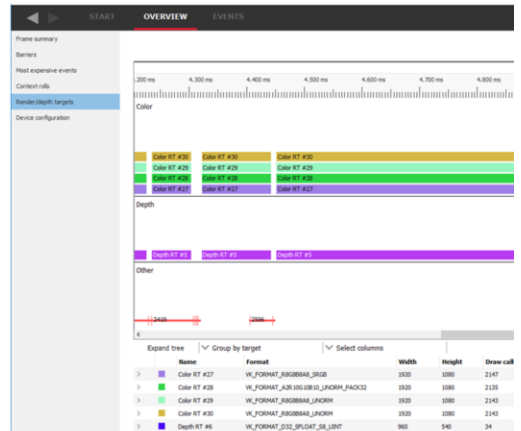
- Sync seldomly
- Prefer large workloads



# 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 RGP's **new features** on Wednesday 5pm!
- And a more detailed look at barriers by **Matt Pettineo** at 4pm today 😊

The logo for the Game Developers Conference (GDC) features the letters "GDC" in a bold, white, sans-serif font. The letters are positioned on a dark blue background that is part of a horizontal bar. Above the "G" and "D" are two overlapping, light blue triangles pointing upwards, creating a stylized mountain or peak shape.

GAME DEVELOPERS CONFERENCE

MARCH 18-22, 2019 | #GDC19

# 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/rgp/>

**GDC**

**GAME DEVELOPERS CONFERENCE**

MARCH 18–22, 2019 | #GDC19