DirectX 12 Optimization Techniques in Capcom's RE ENGINE

Ojiro Tanaka
Rendering Engineer
Capcom

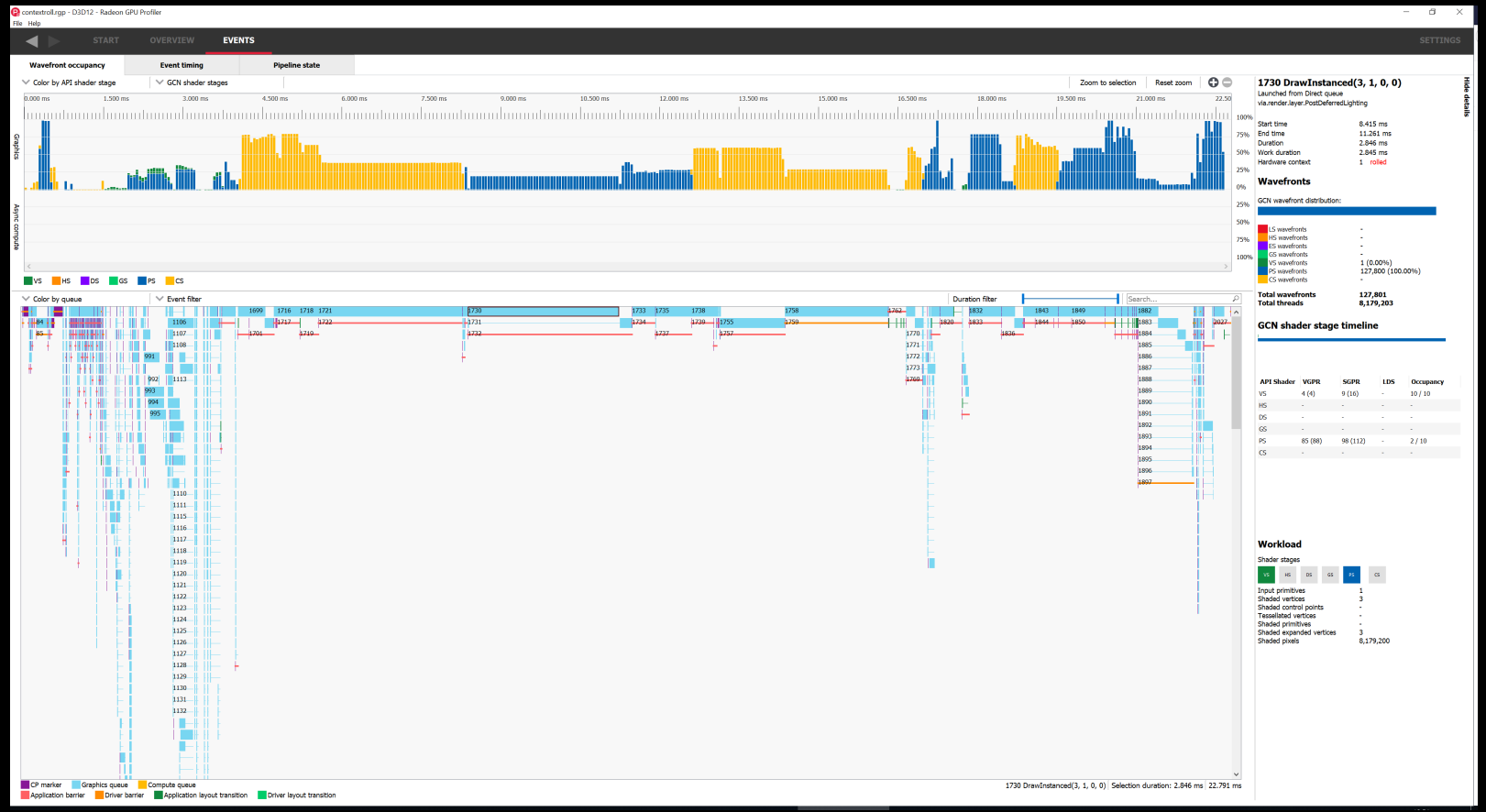
Ashley Smith
Developer Technology Engineer
AMD

Agenda

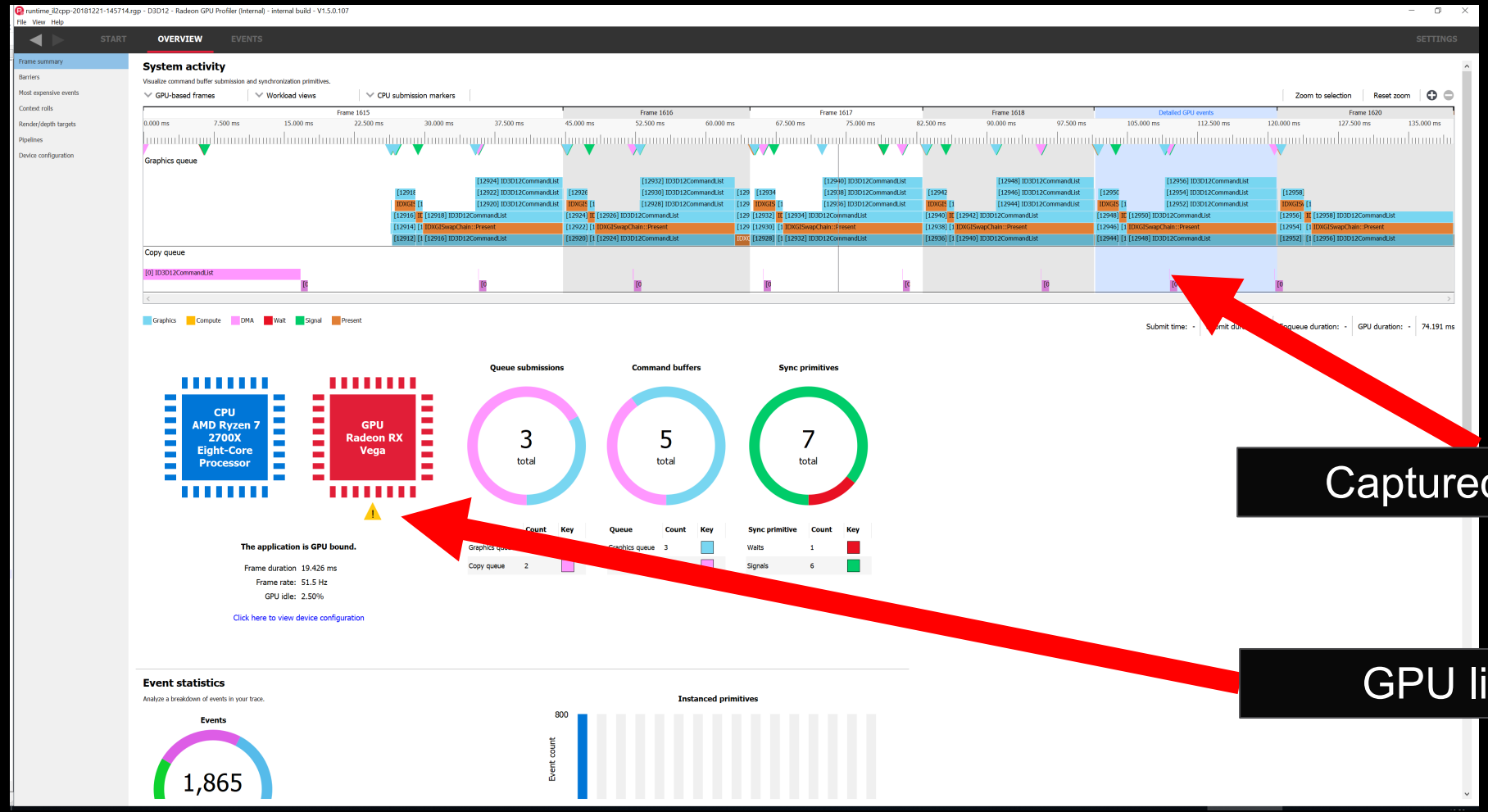
- Tools
  - RGP
  - RGA
  - Tips
- Optimizations
  - Optimization methods
  - Optimizations for DirectX 12
- Tips
  - Pre-bake PSO
  - QA

RGP

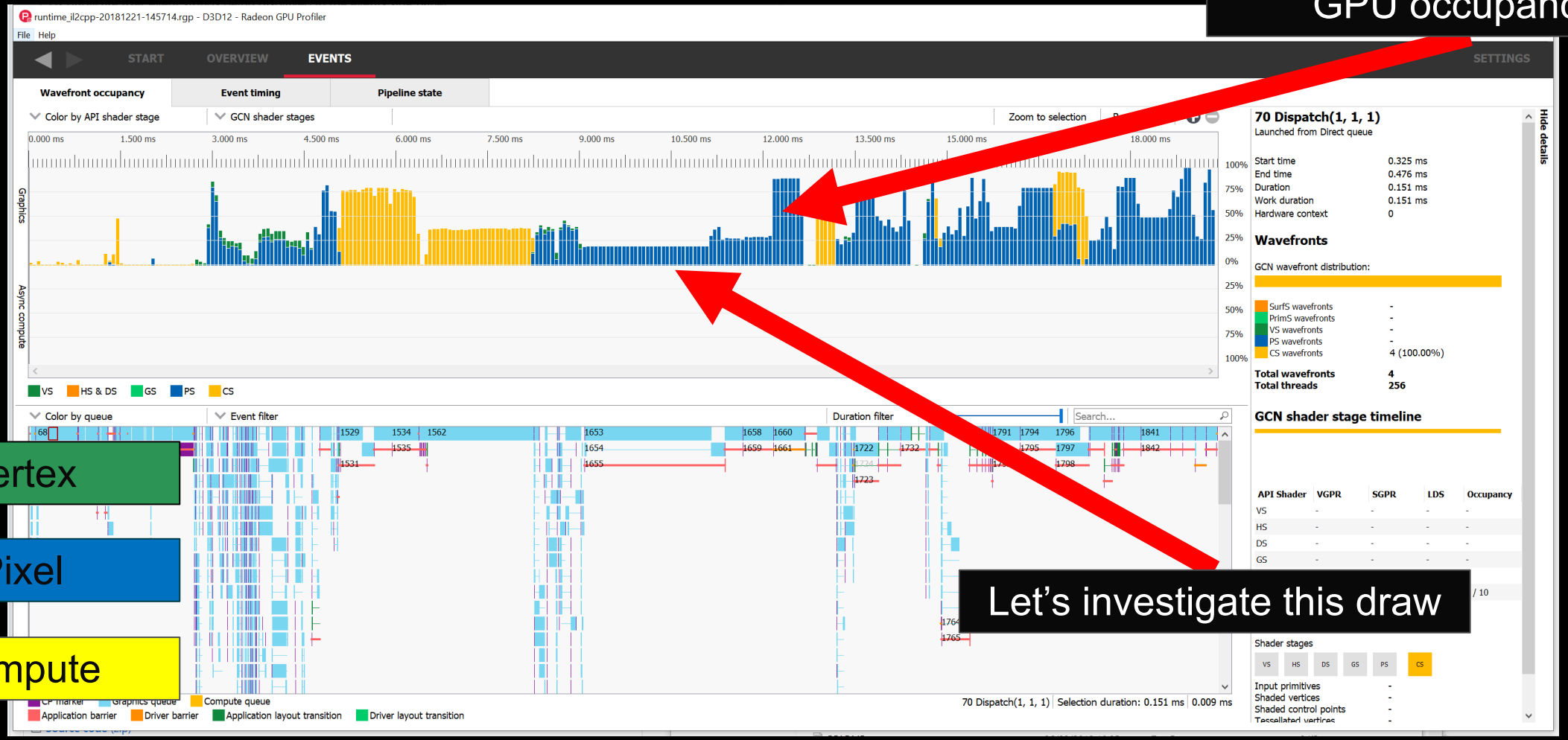
- Overview pages
  - Pipeline state
  - Context rolls
  - Barriers



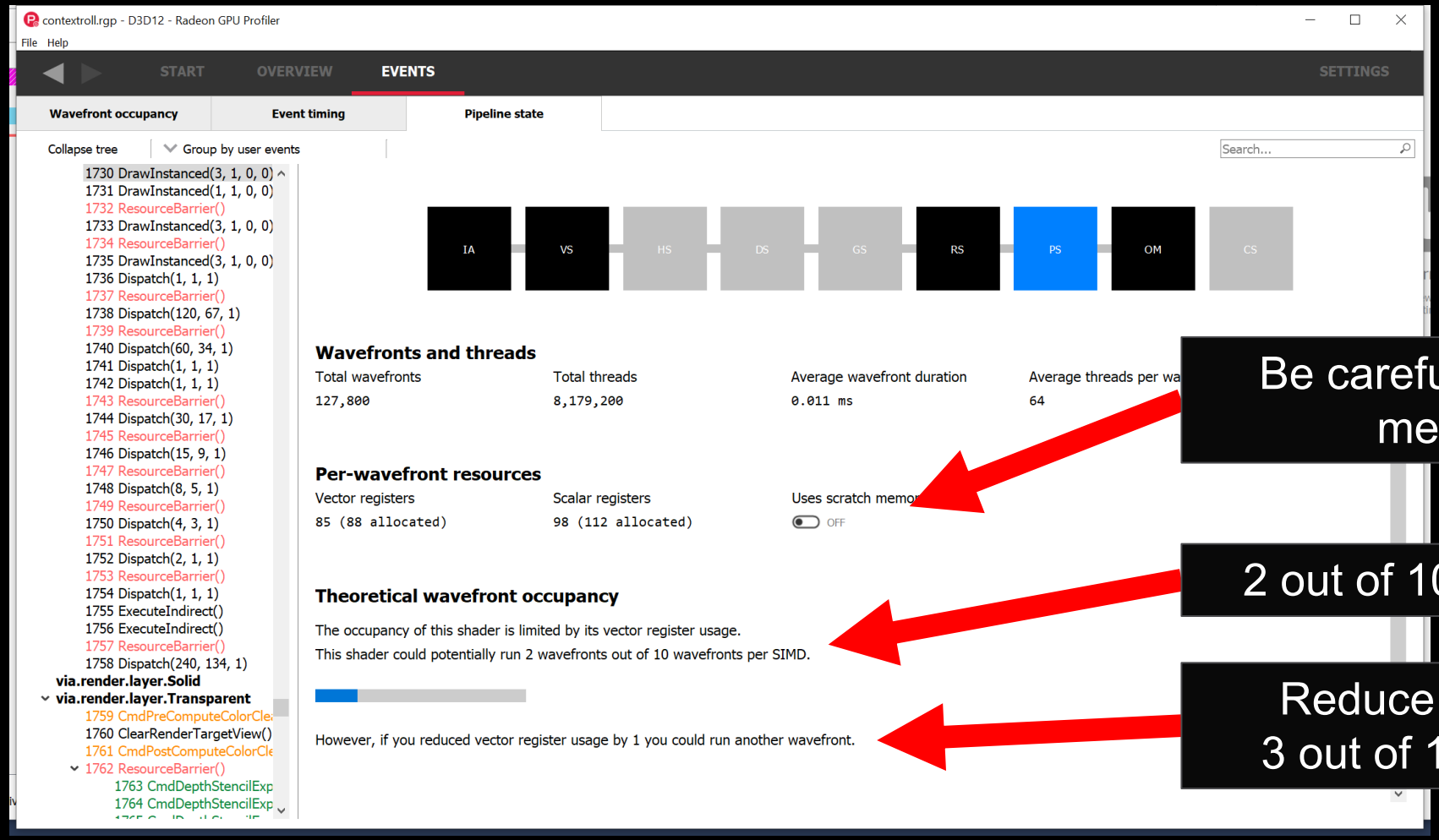
RGP


System activity
Graphics queue
Copy queue
CPU
AMD Ryzen 7
2700X
Eight-Core
Processor
GPU
Radeon RX
Vega
Queue submissions
3
total
Command buffers
5
total
Sync primitives
7
total
The application is GPU bound.
Frame duration 19.426 ms
Frame rate: 51.5 Hz
GPU idle: 2.50%
Click here to view device configuration
Event statistics
Events
1,865
Instanced primitives
Captured frame
GPU limited

RGP


GPU occupancy
runtime_il2cpp-20181221-145714.rgp - D3D12 - Radeon GPU Profiler
File Help
START OVERVIEW EVENTS SETTINGS
Wavefront occupancy
Event timing
Pipeline state
Color by API shader stage
GCN shader stages
Zoom to selection
Graphics
Async compute
VS HS & DS GS PS CS
Color by queue
Event filter
Duration filter
Search...
Compute queue
Application barrier
Driver barrier
Application layout transition
Driver layout transition
70 Dispatch(1, 1, 1)
Launched from Direct queue
Start time 0.325 ms
End time 0.476 ms
Duration 0.151 ms
Work duration 0.151 ms
Hardware context 0
Wavefronts
GCN wavefront distribution:
SurfS wavefronts -
PrimS wavefronts -
VS wavefronts -
PS wavefronts -
CS wavefronts 4 (100.00%)
Total wavefronts 4
Total threads 256
GCN shader stage timeline
API Shader VGPR SGPR LDS Occupancy
VS
HS
DS
GS
Shader stages
Input primitives
Shaded vertices
Shaded control points
Hide details
70 Dispatch(1, 1, 1) | Selection duration: 0.151 ms | 0.009 ms
Vertex
Pixel
Compute
Let's investigate this draw

RGP – Pipeline State



Be careful of scratch memory!

2 out of 10 wavefronts

Reduce VGPR by 1:
3 out of 10 wavefronts

RGP – Pipeline State

- Reducing register usage
  - `min16float`
  - `min16int`
  - `min16uint`
- No need to check for support
- Will default to lowest precision
- How do we investigate?
  - RGA

RGA

```
struct PSInput {
	float4 color : COLOR;
};
float4 PSMain(PSInput input) : SV_TARGET {
	return float4(pow(abs(input.color.rgb), 2.2), input.color.a);
}
```

RGA

```
Line | Ra   | Reg State | Instruction
-------------------------------------------------------------------
    1 |    2 | ::        | label_basic_block_1: s_mov_b32 m0, s2
    2 |    2 | ::        | s_nop 0x0000
    3 |    2 | v:^       | v_interp_p1_f32 v2, v0, attr0.x
    4 |    3 | :v^       | v_interp_p2_f32 v2, v1, attr0.x
    5 |    3 | v::^      | v_interp_p1_f32 v3, v0, attr0.y
    6 |    4 | :v:^      | v_interp_p2_f32 v3, v1, attr0.y
    7 |    4 | v:::^     | v_interp_p1_f32 v4, v0, attr0.z
    8 |    5 | :v::^     | v_interp_p2_f32 v4, v1, attr0.z
    9 |    5 | ::x::     | v_log_f32 v2, abs(v2)
   10 |    5 | :::x:     | v_log_f32 v3, abs(v3)
   11 |    5 | ::::x     | v_log_f32 v4, abs(v4)
   12 |    5 | ::x::     | v_mul_f32 v2, 0x400ccccd, v2
   13 |    5 | :::x:     | v_mul_f32 v3, 0x400ccccd, v3
   14 |    5 | ::::x     | v_mul_f32 v4, 0x400ccccd, v4
   15 |    5 | ::x::     | v_exp_f32 v2, v2
   16 |    5 | :::x:     | v_exp_f32 v3, v3
   17 |    5 | ::::x     | v_exp_f32 v4, v4
   18 |    5 | x::::     | v_interp_p1_f32 v0, v0, attr0.w
   19 |    5 | ^v:::     | v_interp_p2_f32 v0, v1, attr0.w
   20 |    5 | :^vv:     | v_cvt_pkrtz_f16_f32 v1, v2, v3
   21 |    3 | x:  v     | v_cvt_pkrtz_f16_f32 v0, v4, v0
   22 |    2 | vv        | exp mrt0, v1, v1, v0, v0
   23 |    0 |           | s_endpgm
Maximum # VGPR used   5, # VGPR allocated:   5
```

RGA

```
struct PSInput {
	float4 color : COLOR;
};
float4 PSMain(PSInput input) : SV_TARGET {
	return float4(pow(abs(input.color.rgb), 2.2), input.color.a);
}
```

```
 Line | Ra   | Reg State | Instruction
-------------------------------------------------------------
    1 |   2 | ::        | label_basic_block_1: s_mov_b32 m0, s2
// ...
   23 |   0 |           | s_endpgm

Maximum # VGPR used   5, # VGPR allocated:   5
```

RGA

```
struct PSInput {
	min16float4 color : COLOR;
};
float4 PSMain(PSInput input) : SV_TARGET {
	return float4(pow(abs(input.color.rgb), 2.2), input.color.a);
}
```

```
 Line | Ra   | Reg State | Instruction
-------------------------------------------------------------
    1 |   2 | ::        | label_basic_block_1: s_mov_b32 m0, s2
// ...
   24 |   4 | :^v:      | v_cvt_f32_f16 v1, v2
   25 |   4 | ::^v      | v_cvt_f32_f16 v2, v3
   26 |   4 | :::x      | v_cvt_f32_f16 v3, v3
   27 |   4 | x:::      | v_cvt_f32_f16 v0, v0
// ...
   33 |   0 |           | s_endpgm

Maximum # VGPR used   4, # VGPR allocated:   4 5
```

RGA

```
struct PSInput {
	min16float4 color : COLOR;
};
float4 PSMain(PSInput input) : SV_TARGET {
	return float4(pow(abs(input.color.rgb), 2.2), input.color.a);
}
```

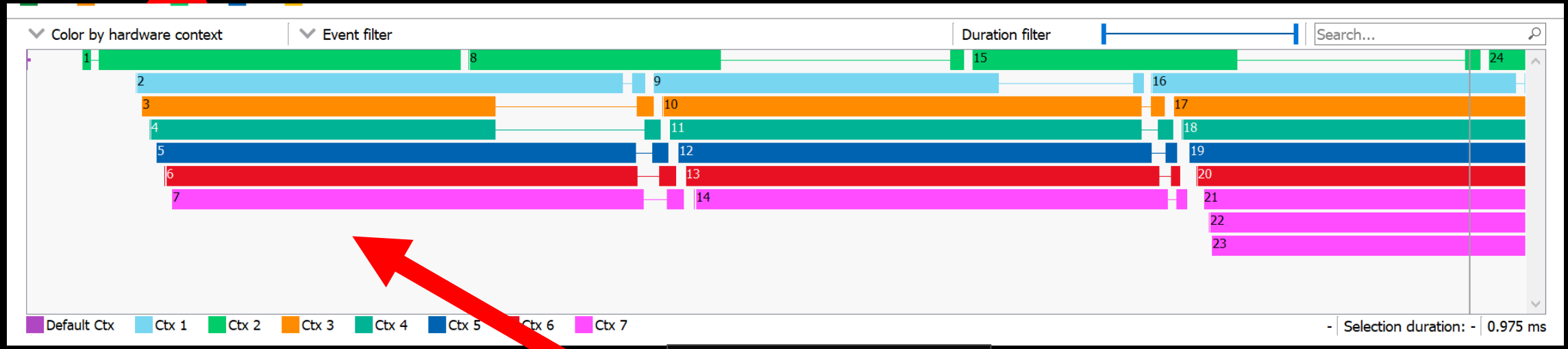
```
 Line | Ra   | Reg State | Instruction
-------------------------------------------------------------
    1 |   2 | ::        | label_basic_block_1: s_mov_b32 m0, s2
// ...
   24 |   4 | :^v:      | v_cvt_f32_f16 v1, v2
   25 |   4 | ::^v      | v_cvt_f32_f16 v2, v3
   26 |   4 | :::x      | v_cvt_f32_f16 v3, v3
   27 |   4 | x:::      | v_cvt_f32_f16 v0, v0
// ...
   33 |   0 |           | s_endpgm

Maximum # VGPR used   4, # VGPR allocated:   4 5
```

RGP – Context Rolls

```
cmdBuf->RSSetViewports(a);
cmdBuf->Draw(1);
cmdBuf->RSSetViewports(b);
cmdBuf->Draw(2);
cmdBuf->RSSetViewports(a);
cmdBuf->Draw(3);
```

RGP Profiler – Context Rolls

Color by hardware context



Could be running more draws here

RGP – Context Rolls

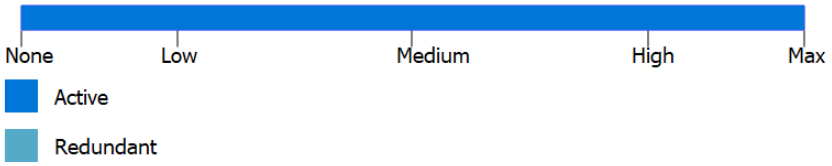
How do we check context rolls?

Every draw caused a context roll ☹

Can see what state change caused context roll

There were 21 context rolls from 21 draw calls. That's an average of 1.00 draw calls per context.

None of these context rolls were completely redundant.



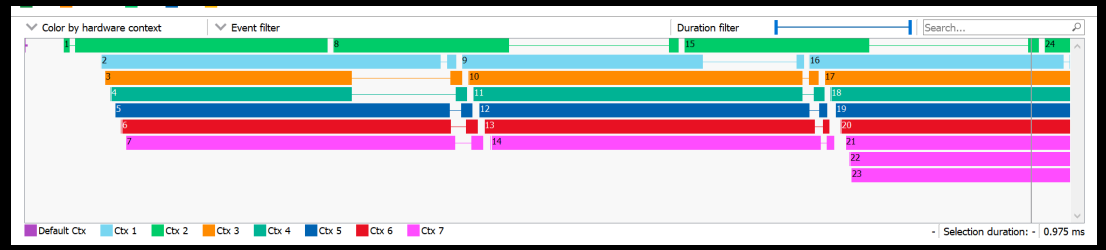
State which causes context rolls

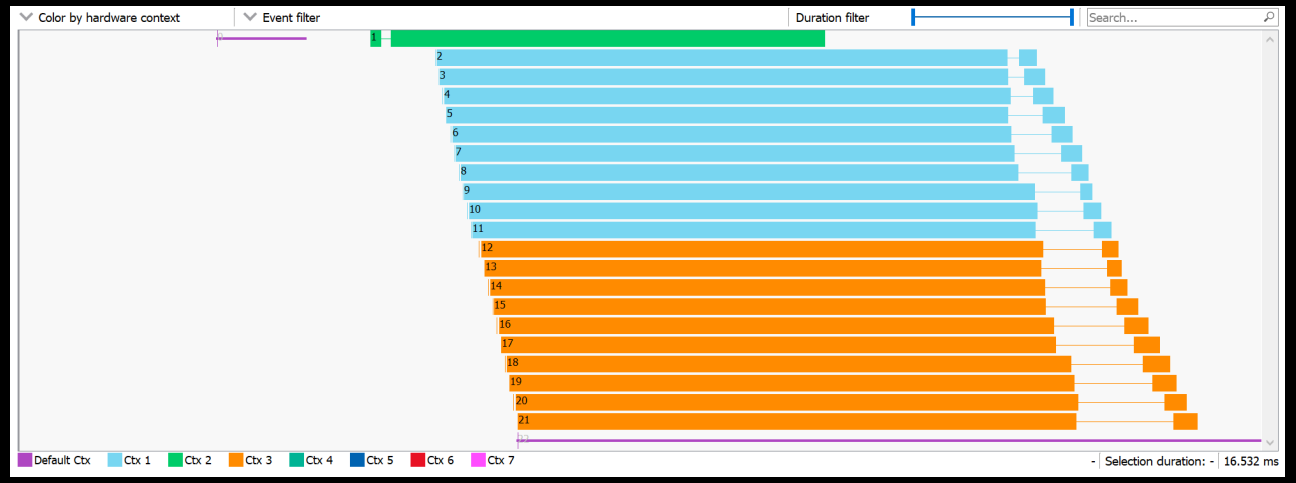
State	Context rolls	Redundant	Completely redundant	Exclusive
D3D12_GRAPHICS_PIPELINE_STATE.VS				
Export Count				
Half Pack				
D3D12_GRAPHICS_PIPELINE_STATE.PS				
TexKill	1	1 (100%)	0 (0%)	0 (0%)
Input Count	1	0 (0%)	0 (0%)	0 (0%)
D3D12_GRAPHICS_PIPELINE_STATE.BlendState				
AlphaToCoverageEnable	1	1 (100%)	0 (0%)	0 (0%)
RenderTarget[0].BlendEnable	1	1 (100%)	0 (0%)	0 (0%)
RenderTarget[0].SrcBlend	1	1 (100%)	0 (0%)	0 (0%)
RenderTarget[0].DestBlend	1	1 (100%)	0 (0%)	0 (0%)
RenderTarget[0].BlendOp	1	1 (100%)	0 (0%)	0 (0%)

RGP – Context Rolls

```
cmdBuf->RSSetViewports(a);
cmdBuf->Draw(1);
cmdBuf->RSSetViewports(b);
cmdBuf->Draw(2);
cmdBuf->RSSetViewports(a);
cmdBuf->Draw(3);
```

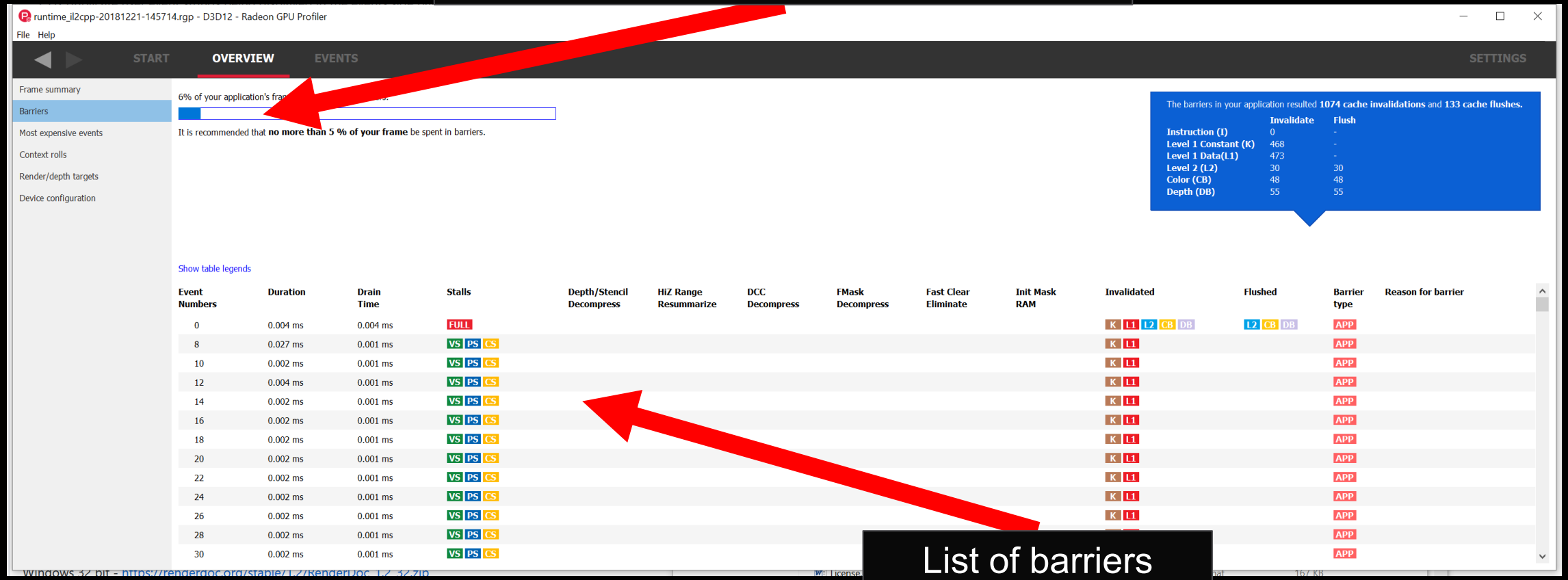
```
cmdBuf->RSSetViewports(a);
cmdBuf->Draw(1);
cmdBuf->Draw(3);
cmdBuf->RSSetViewports(b);
cmdBuf->Draw(2);
```





RGP – Barriers

6% of time spent in barriers



List of barriers

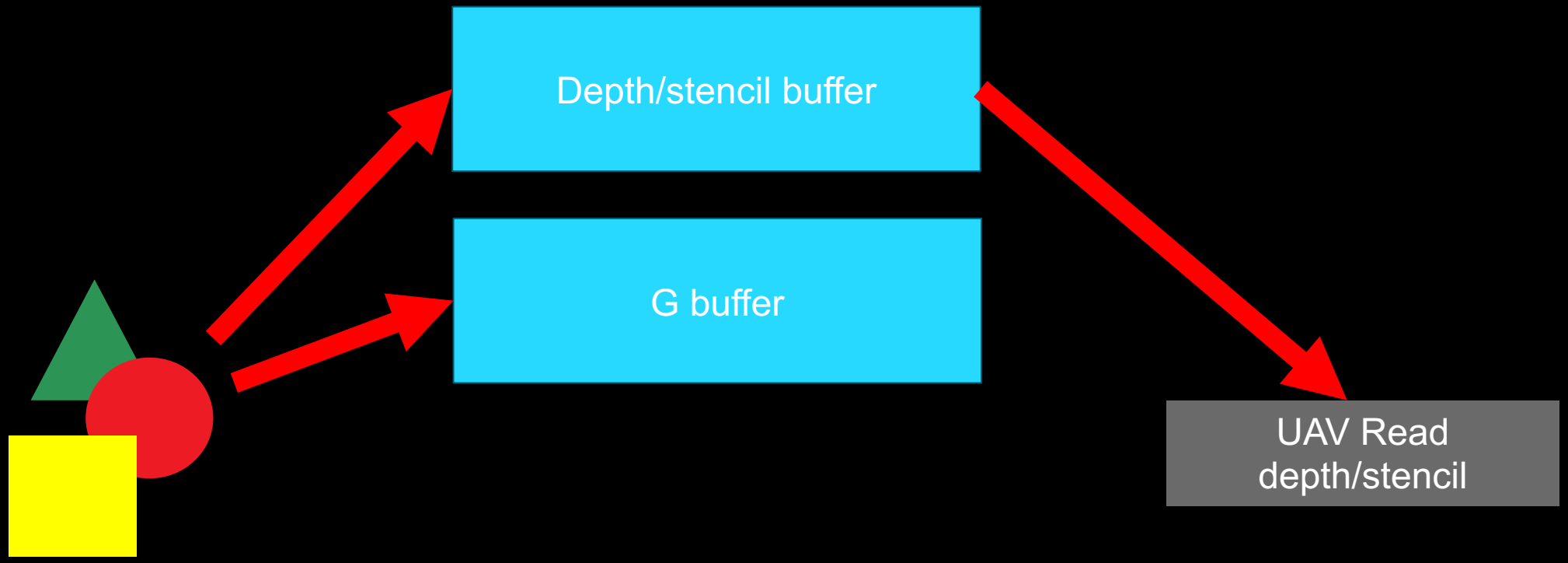
RGP – Barriers

Show table legends

Event Numbers	Duration	Drain Time	Stalls	Depth/Stencil Decompress	HiZ Range Resummarize	DCC Decompress	FMask Decompress	Fast Clear Eliminate	Init Mask RAM	Invalidated	Flushed	Barrier type	Reason f
0	0.001 ms	0.001 ms	FULL							K L1 CB DB	CB DB	DRIVER	Before CS
2	0.002 ms	0.001 ms	CS							K L1		DRIVER	After CS c
3	0.002 ms	0.002 ms	DMA							K L1		APP	
> 5..6	0.011 ms	0.002 ms	FULL					✓		K L1 CB DB	CB DB	APP	
7	0.001 ms	0.001 ms								K L1		DRIVER	Blit synch
8	0.001 ms	0.001 ms	FULL							K L1 CB DB	CB DB	DRIVER	Before CS
10	0.002 ms	0.001 ms	CS							K L1		DRIVER	After CS c
v 19..20	0.009 ms	0.008 ms	FULL			✓				K L1 CB DB	CB DB	APP	
20	0.002 ms	0.008 ms				✓						APP	
21	0.001 ms	0.001 ms								K L1		APP	
> 22..23	0.008 ms	0.008 ms	FULL					✓		K L1 L2 CB DB	L2 CB DB	APP	
26	0.035 ms	0.001 ms	FULL							K L1 CB DB	CB DB	APP	

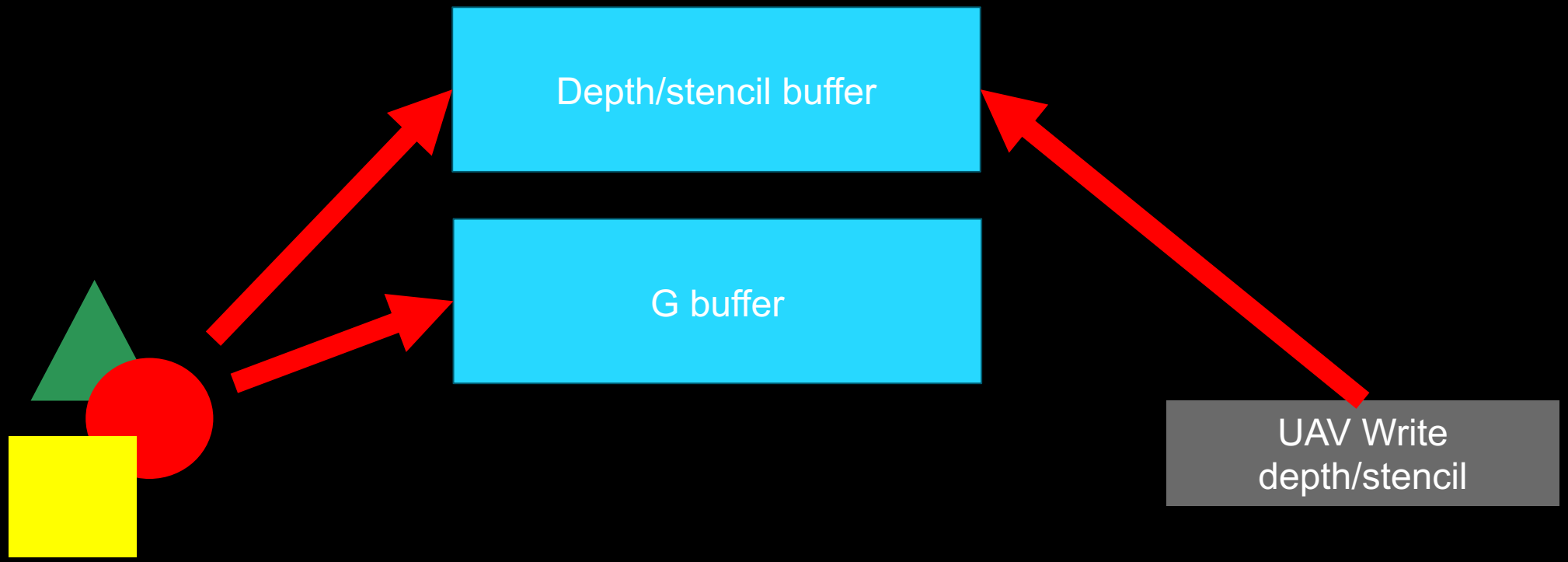
RGP – Barriers

- Depth/stencil decompress



RGP – Barriers

- HiZ range resummarize



RGP – Barriers

- DCC decompress

Format, flags, usage

Compression is enabled or disabled by many factors

	Name	Format	Width	Height	Size in memory	Draw calls	Compression
>	Color RT #0	DXGI_FORMAT_R8G8B8A8_UNORM	512	512	1 MB	6	OFF
>	Color RT #1	DXGI_FORMAT_R8G8B8A8_UNORM	256	256	256 KB	3	OFF
>	Color RT #2	DXGI_FORMAT_R8G8B8A8_UNORM	256	256	256 KB	2	OFF
>	Depth RT #0	DXGI_FORMAT_D32_FLOAT	256	128	128 KB	31	ON
>	Depth RT #1	DXGI_FORMAT_D32_FLOAT	3840	2160	32 MB	608	ON
>	Color RT #3	DXGI_FORMAT_R11G11B10_FLOAT	3840	2160	32 MB	539	OFF
>	Color RT #4	DXGI_FORMAT_R8G8B8A8_UNORM_SRGB	3840	2160	32 MB	375	ON

RGP – Barriers

- DCC decompress
- Example:

```
ResourceBarrier(D3D12_RESOURCE_STATE_RENDER_TARGET,
     D3D12_RESOURCE_STATE_COPY_DEST);

ResourceBarrier(D3D12_RESOURCE_STATE_COPY_DEST,
     D3D12_RESOURCE_STATE_RENDER_TARGET);
```

Tips

- Fast clears
- Debugging

Tips – Fast clears

`ClearRenderTargetView()`
`ClearDepthStencilView()`
`pOptimizedClearValue`

Stick to `1.0f` or `0.0f` for depth

Black or **white** for color

Tips – Debugging

Breadcrumbs / WriteBufferImmediate()

```
WriteMarker(TopOfPipe, 1)
Draw(x)
WriteMarker(BottomOfPipe, 2)
WriteMarker(TopOfPipe, 3)
Draw(y)
WriteMarker(BottomOfPipe, 4)
```

Tips – Debugging

Breadcrumbs / WriteBufferImmediate()

```
WriteMarker(TopOfPipe, 1)
Draw(x) < TDR happens here
WriteMarker(BottomOfPipe, 2)
WriteMarker(TopOfPipe, 3)
Draw(y)
WriteMarker(BottomOfPipe, 4)
```

Tips – Debugging

Breadcrumbs / WriteBufferImmediate()

```
WriteMarker(TopOfPipe, 1)
Draw(x) < TDR happens here
WriteMarker(BottomOfPipe, 2)
WriteMarker(TopOfPipe, 3)
Draw(y)
WriteMarker(BottomOfPipe, 4)
// ...
< Crash reported afterwards
```

Tips – Debugging

Breadcrumbs / WriteBufferImmediate()

```
WriteMarker(TopOfPipe, 1)
Draw(x) < TDR happens here
WriteMarker(BottomOfPipe, 2)
WriteMarker(TopOfPipe, 3)
Draw(y)
WriteMarker(BottomOfPipe, 4)
// ...
< Crash reported afterwards
```

Tips – Debugging

Breadcrumbs / WriteBufferImmediate()

```
WriteMarker(TopOfPipe, 1)       // 1
Draw(x) < TDR happens here
WriteMarker(BottomOfPipe, 2)    // 0
WriteMarker(TopOfPipe, 3)       // 0
Draw(y)
WriteMarker(BottomOfPipe, 4)    // 0
// ...
< Crash reported afterwards
```

We know what caused the TDR now

Tips – Debugging

Breadcrumbs / WriteBufferImmediate()

DX11: AGS on github

DX12: WriteBufferImmediate()

Only for debugging (May cause stalls!)

RE ENGINE Optimization

Agenda

- Optimization
  - Adaptation of console optimizations to PC
  - Optimization for DirectX 12
- Tips

Background of in-house engine

- RE ENGINE
  - Capcom’s in-house engine
  - Targets consoles and PC

- Shipped
  - Resident Evil 7:Biohazard (RE7)
  - Resident Evil 2 (RE2)
  - Devil May Cry 5 (DMC5)

Background of in-house engine

- RE ENGINE uses “Intermediate drawing command”
  - Platform independent commands
  - Allows programmers to write drawing commands without platform knowledge
  - Useful for multi-platform development
  - Able to create drawing commands on multiple threads
  - These “Intermediate drawing commands” are sorted after creation then translated to API commands
  - Drawing order is controlled using priority variable (uint 64 bit value)
  - Allows batch process at the discretion of the user
  - Useful for controlling sync timing of UAVOverlap and AsyncDispatch

Implementation of DirectX 12 in RE ENGINE

- Trials started during RE7 production, but was not implemented
- RE2 and DMC5 implements DirectX 12

Optimization

- Adaptation of console optimizations to PC
  - OcclusionCulling using MultiDraw
  - UAVOverlap
  - Wave Intrinsics
  - Depth Bounds Test
- Optimization for DirectX 12
  - Reduction of resource barrier
  - Buffer update
  - RootSignature
  - Memory management

Comparison of before and after

- 24% frame time saving!

Adaptation of console optimizations to PC

Testing environment

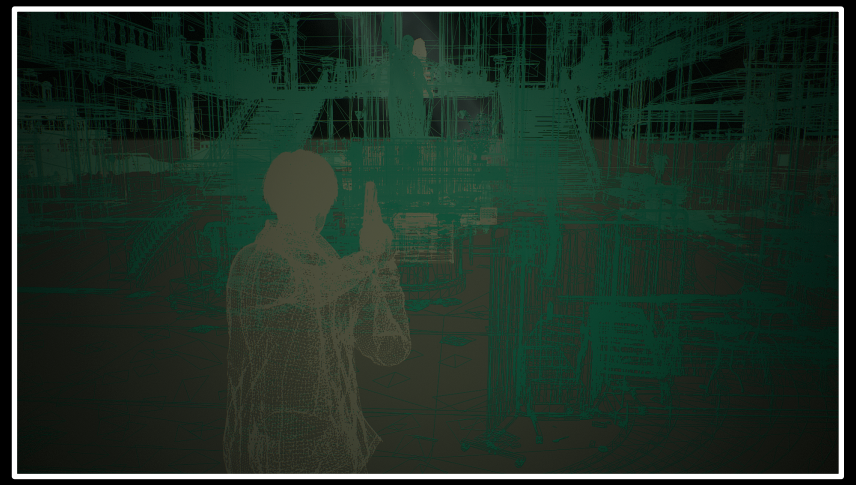
- RE2 (2/15 patch)
- 1080p
- Mainly Radeon RX480, partially Radeon R9 Fury X
- Radeon GPU Profiler 1.3.1.70, OCAT, PIX for Windows

MultiDraw

- In DirectX 12 we use ExecuteIndirect
  - Allows execution of multiple drawing commands at once
  - Aim to reduce the overhead of drawing meshes
- In DirectX 11 MultiDraw is supported by AGS or NVAPI

Any improvements?

- Overhead-wise there was not as much improvement as we had hoped
- ExecuteIndirect was useful for implementation of GPU-based occlusion culling



GPU-based occlusion culling OFF

GPU-based occlusion culling ON

FYI

- 2 possible solution; ExecuteIndirect and Predication command
- ExecuteIndirect
  - 4 byte Alignment
  - Controls the number of IndirectArgument executions with CountBuffer
- Predication command
  - 8 byte Alignment - Incompatible with consoles

Data structures - VisibleBuffer

- Visibility managed using “VisibleBuffer”
  - Practically, it is a CountBuffer in RE ENGINE
  - ByteAddressBuffer
  - Number of elements is equal to maximum number of meshes in scene
  - Each element contains per mesh visibility
    - 0xffff for visible, 0x0000 for invisible

Data structures – Mesh data

- StructuredBuffer
  - AABB - CPU made or GPU made
  - VisibleBuffer’s byte offset
  - IndirectArgument’s byte offset

Visibility test

- Draw with EarlyZ
  - [earlydepthstencil] attribute!
  - Store 0xffff into VisibleBuffer
- Minimize writing to same address in units of Wave[dorobot16]

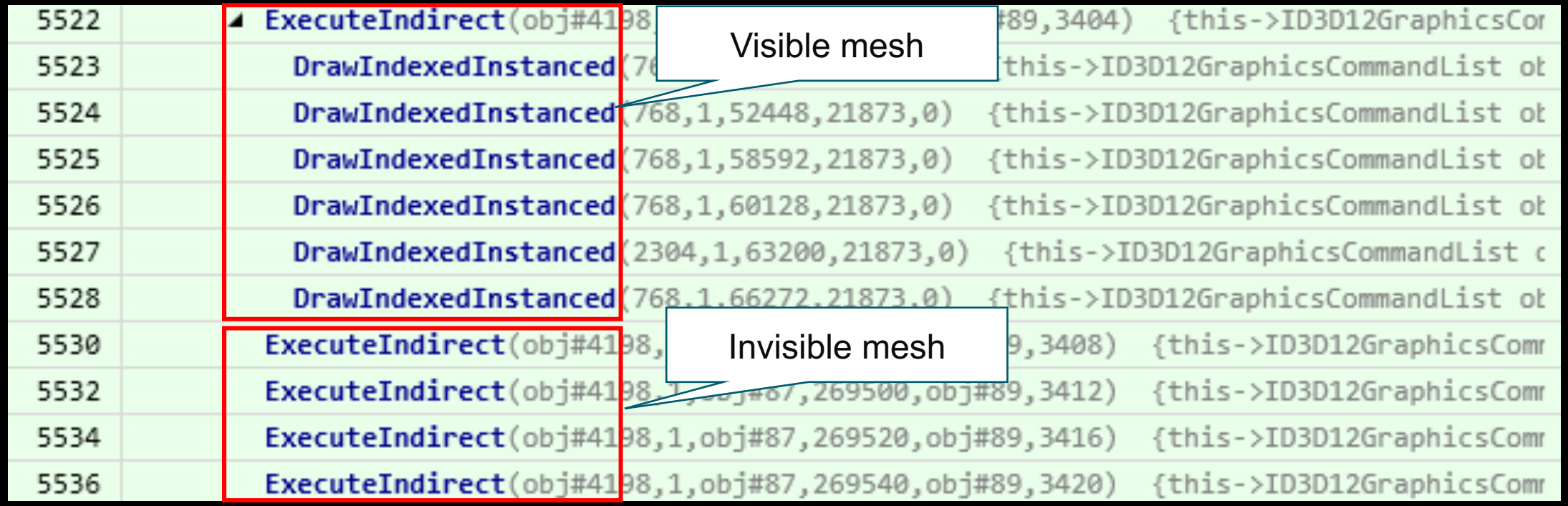
```
[earlydepthstencil]
void PS_Culltest(OccludeeOutput I){
        uint hash = WaveCompactValue(I.outputAddress);
        [branch]
        if (hash == 0){
                RWCountBuffer.Store(I.outputAddress, 0xffff);
        }
}
```

Apply visibility test result

- Apply drawing per mesh
- Specify number of draws using MaxCommandCount
  - VisibleBuffer as CountBuffer
    - CountBuffer 0xffff : Enable draw (count is MaxCommandCount)
    - CountBuffer 0: Disable draw

```
void ExecuteIndirect(
  ID3D12CommandSignature       *pCommandSignature,
  UINT                  MaxCommandCount,
  ID3D12Resource               *pArgumentBuffer,
  UINT64                       ArgumentBufferOffset,
  ID3D12Resource               *pCountBuffer,
  UINT64                       CountBufferOffset);
```

Result on PIX



Per mesh occlusion culling OFF

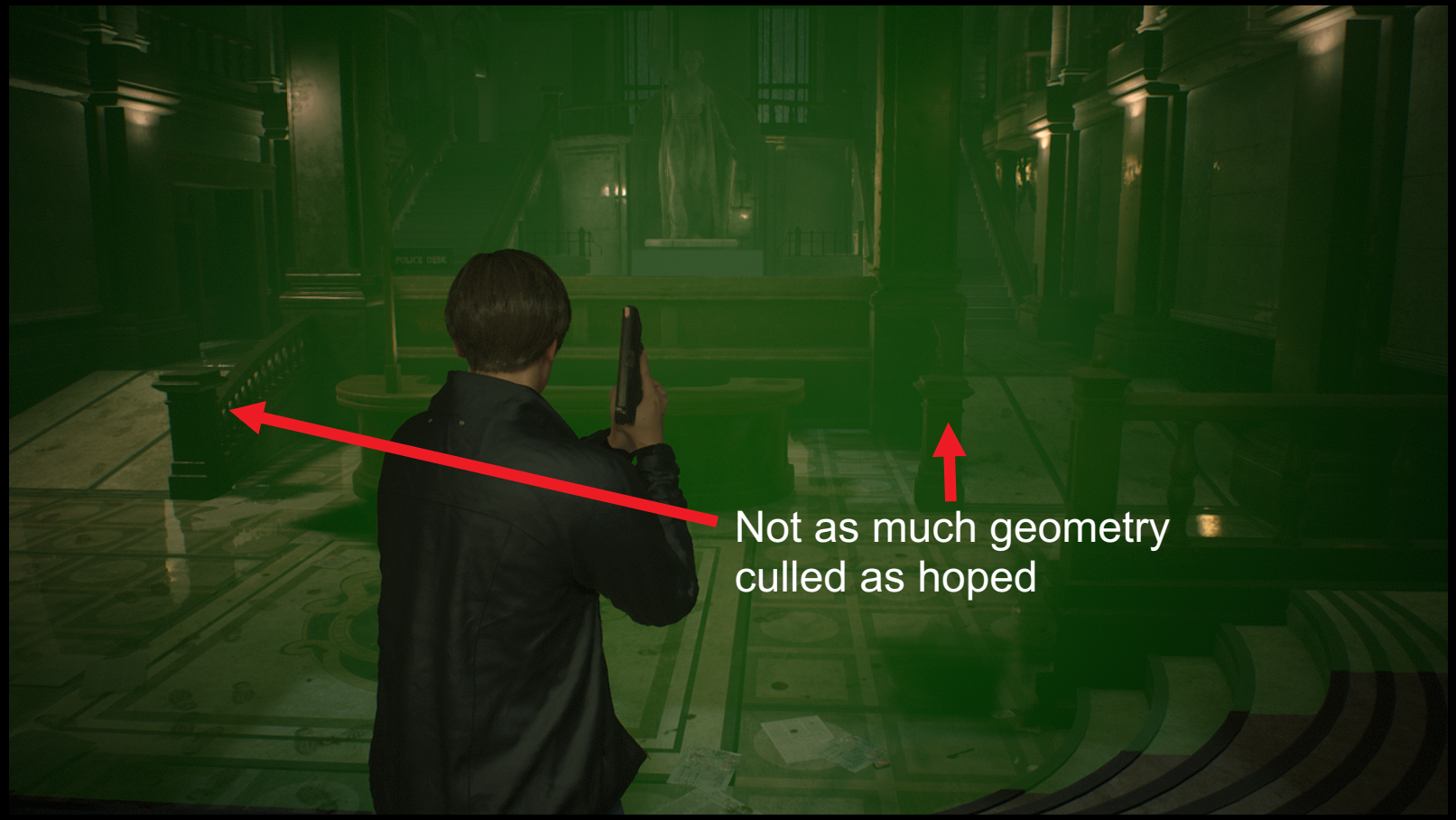
Per mesh occlusion culling ON

Per mesh occlusion culling ON

Per mesh occlusion culling OFF

Per mesh occlusion culling ON

Per mesh occlusion culling ON

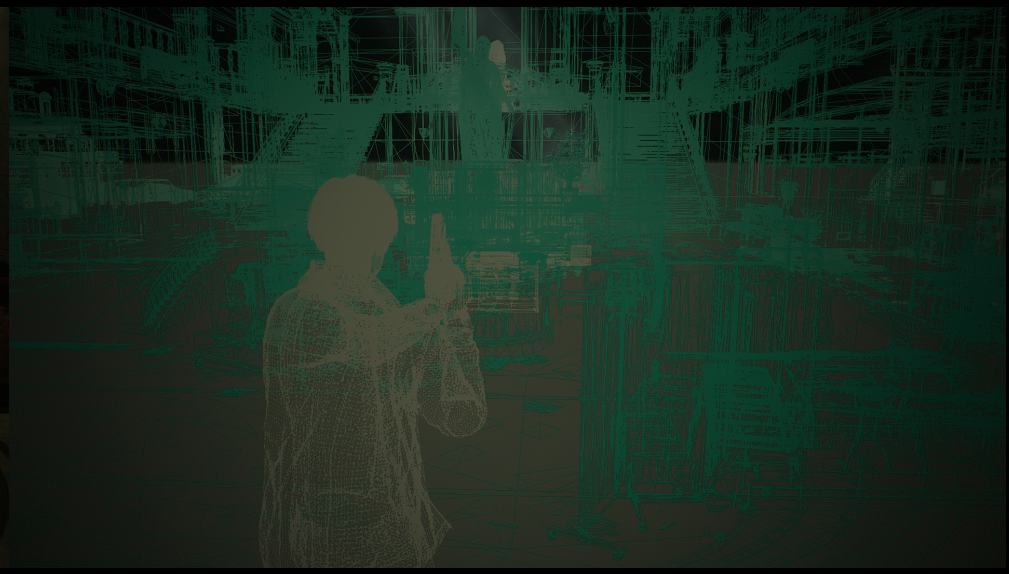


Room for improvement?

- Effective against props and character mesh
  - Culling methods are effective against smaller AABB units
- Ineffective against large mesh
  - Large meshes are always visible
  - Need to split the mesh finely for better results

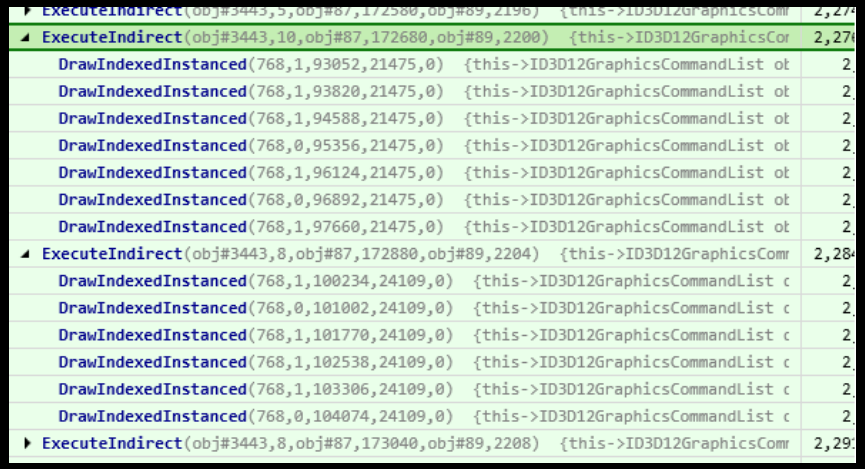
Automatic division of large mesh

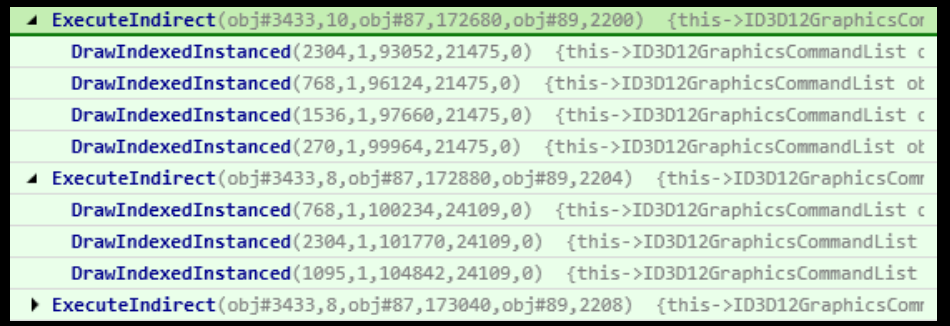
- Cut out 256 triangles as one batch
- Each batch consists of consecutive Indirect Argument
  - Create AABB per batch



Issues with many micro drawing command

- Almost all draws fall below 768 indices
- Large amounts of batches cause bad performance
  - Depend on the hardware
- Merge commands if adjacent IndirectArguments are continuous





Mesh division OFF

Mesh division ON

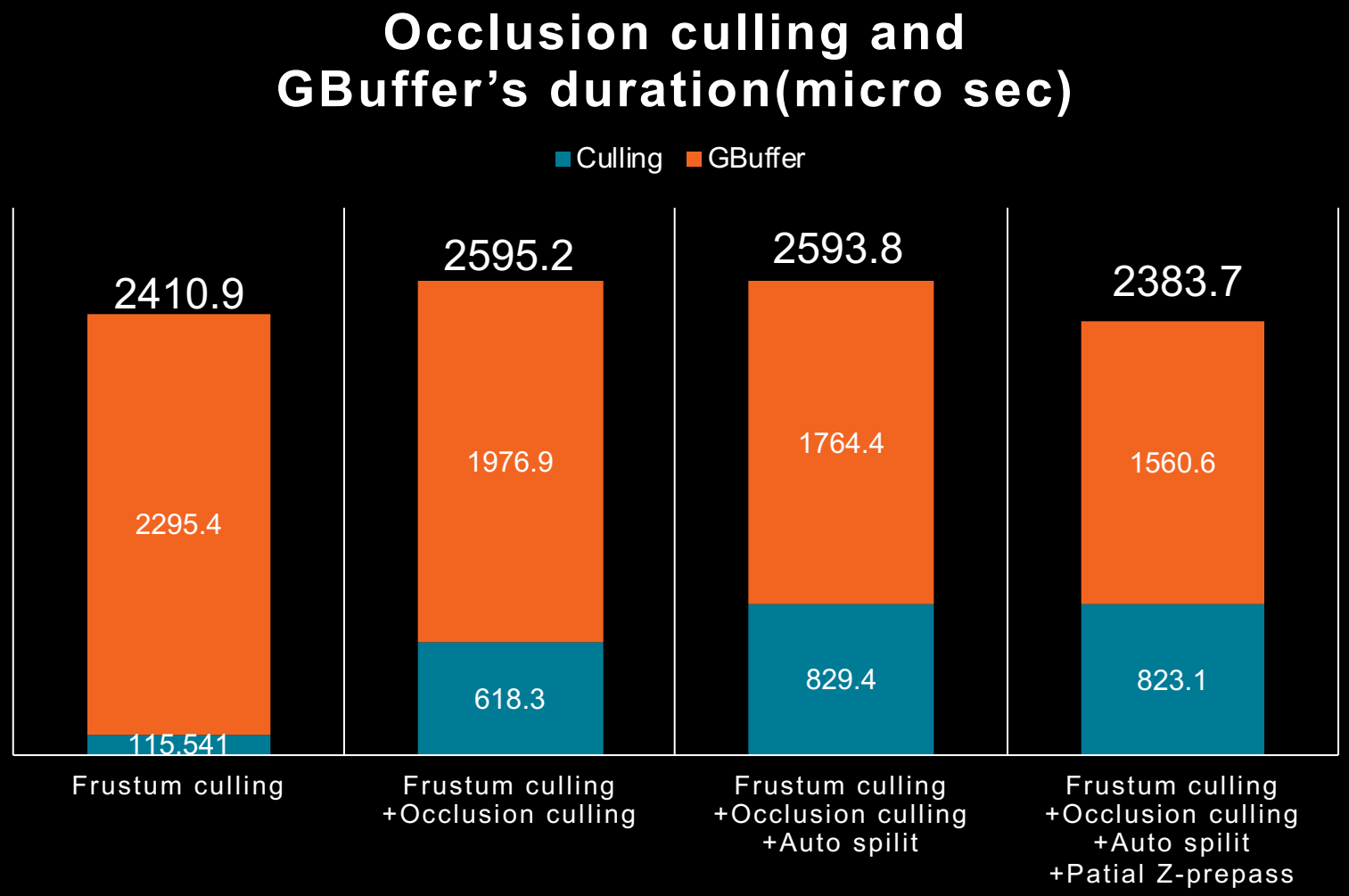
Divide mesh OFF

Divide mesh ON

Partial Z-prepass

- To run as few fragment shaders as possible
- Z-prepass with every mesh is expensive
  - Cost can surpass the benefit
- Limiting Z-prepass to meshes close to the camera
  - Reuse auto-division models

Comparison of each method



API DirectX 12, GPU AMD Radeon RX480 , Radeon Profiler

Comparison of GPU-based occlusion culling

- At this point not gain performance

UAVOverlap

- In DirectX 12 shaders without dependency can execute in parallel

- UAV barrier has ambiguous dependency
  - Unclear whether read or write
  - If each batch writes to a separate location, it can be executed in parallel
    - If WAW(write-after-write) hazard is avoidable

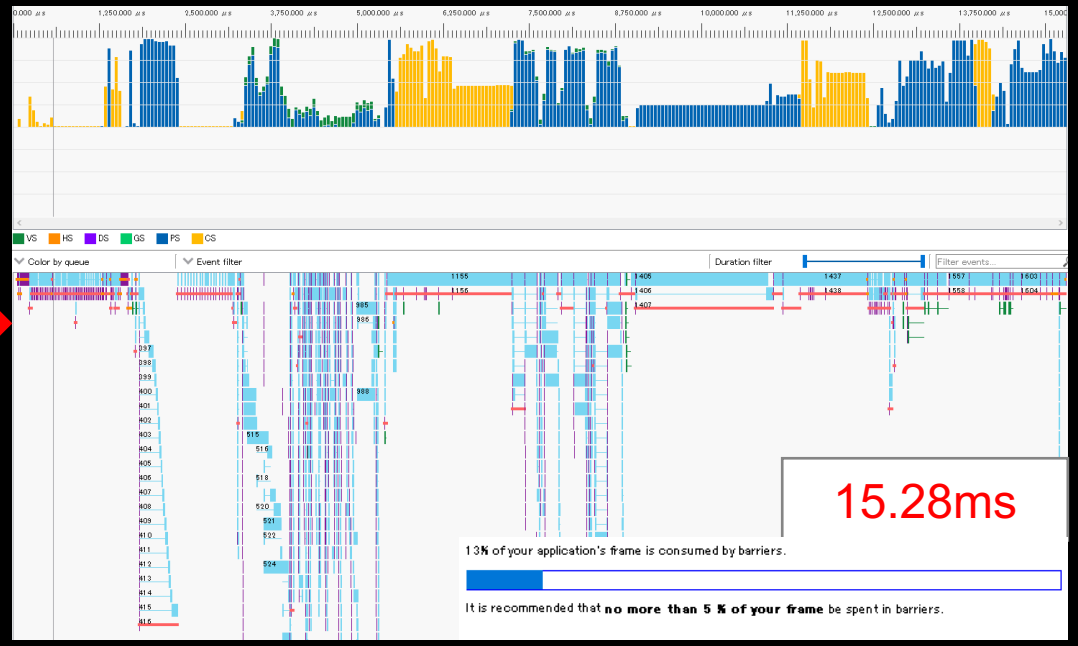
UAVOverlap

- Controllable UAV Synchronization for each compute shader dispatch
  - Parallel execution made possible by disabling synchronization of UAV
  - In DirectX 11, it is possible to introduce equivalent functions using AGS and NVAPI.

- void dispatch(u32 threadGroupX,u32 threadGroupY,u32 threadGroupZ, bool uavResourceSyncDisable = false);
- void dispatchIndirect(Buffer& buffer,u32 alignedOffsetForArgs, bool uavResourceSyncDisable = false);

Comparison : UAV Overlap

- Overall performance improvement



Wave Intrinsics

- Shader scalarization can improve the rate the threads work in parallel.
- Used for Lighting, GPU-based occlusion culling , SSR…
- For scalarization, refer to [Sousa16]

- Wave Intrinsics improves efficiency of scalarization by removing unnecessary synchronizations.

- Supported in DirectX 11 and DirectX 12
  - Using AGS Intrinsic with Shader Model 5.1
  - Can also be used with Shader Model 6.0

Comparison : Wave Intrinsics

- Overall performance improvement

Depth Bounds Test

- Clamp depth to a specific depth range
  - Mainly used to eliminate extraneous pixel shaders
  - Available with DirectX 12 (Creators Update) and DirectX 11.3
    - DirectX 11 With AGS and NVAPI

- In RE ENGINE, it is used for decals and light shafts

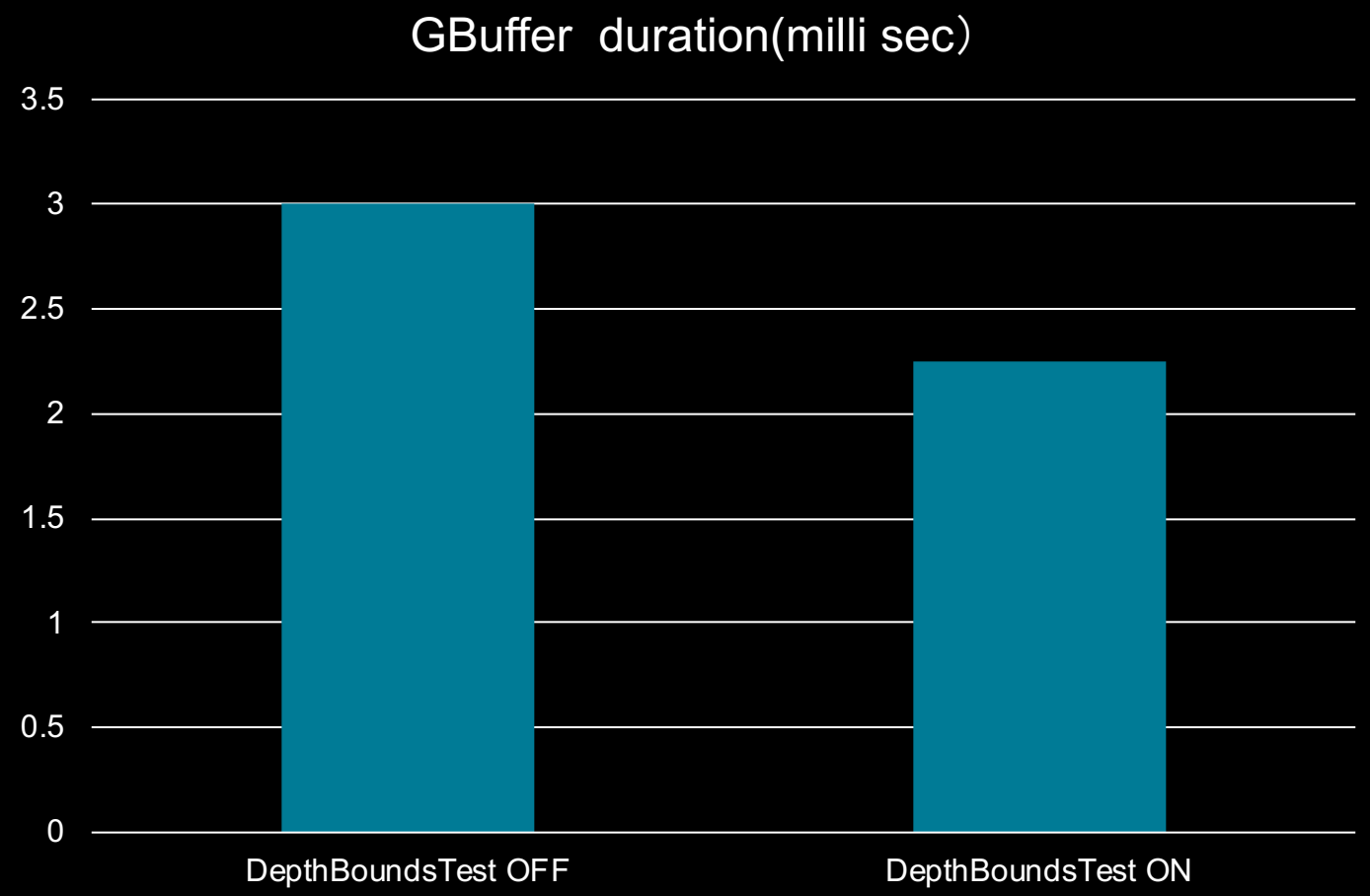
Decals

- Runs on pixels that failed the depth test
- Preferably omit processing when completely occluded
  - Resolved using Depth Bounds Test

Depth Bounds Test OFF

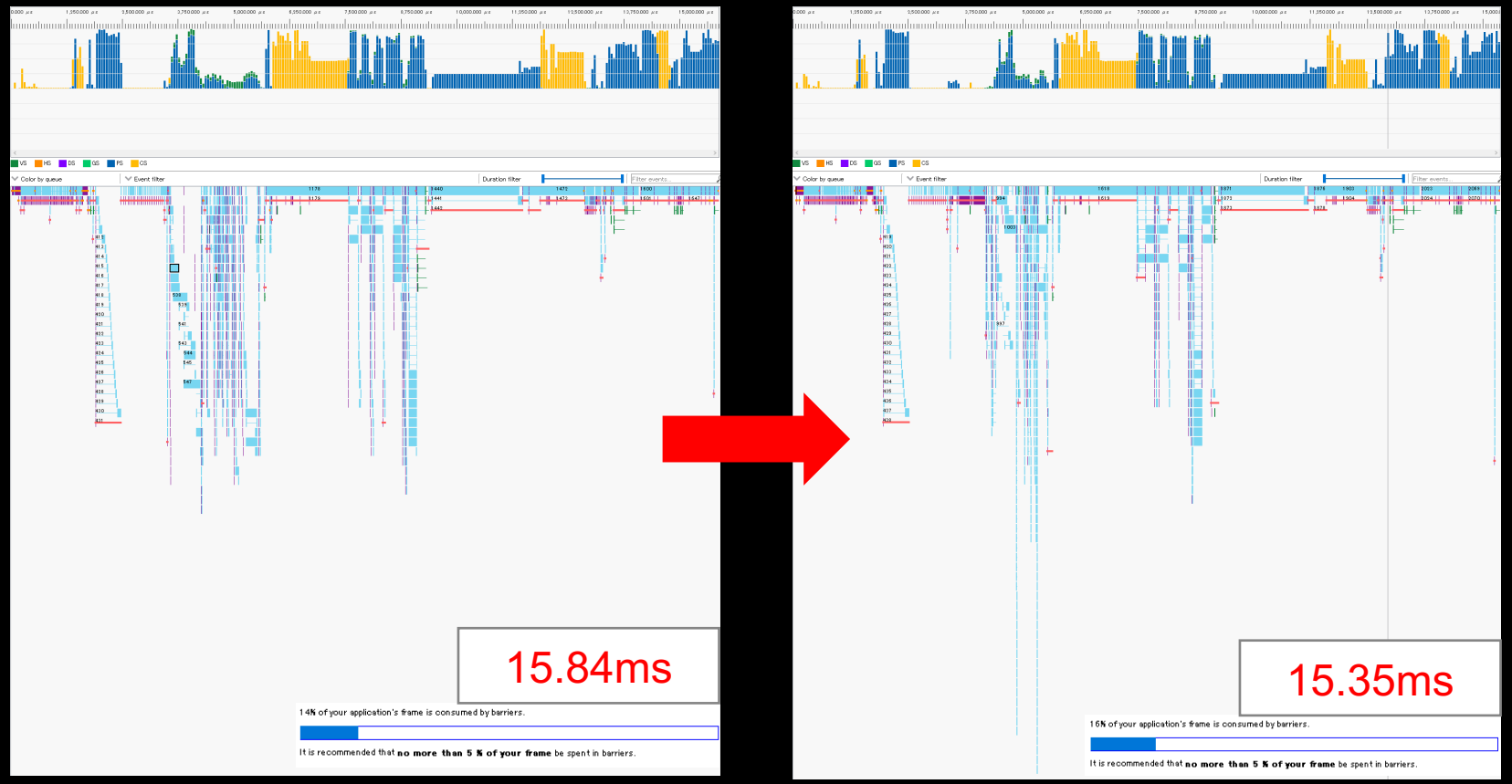
Depth Bounds Test ON

Comparison of Depth Bounds Test for decals



API DirectX 12, GPU AMD **Radeon R9 Fury X**, RadeonProfiler

Console optimization comparison



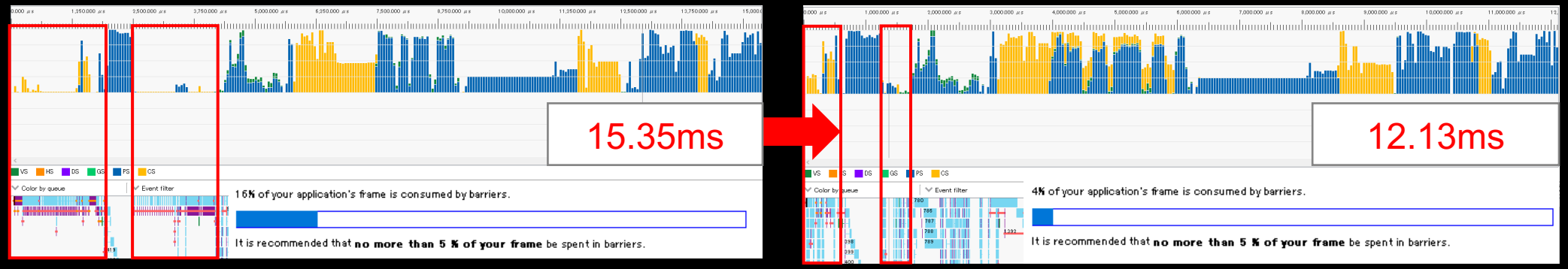
Optimization for DirectX 12

Optimization

- Feedback console optimizations method to PC
  - MultiDraw
  - UAVOverlap
  - Wave Intrinsics
  - Depth Bounds Test

- Optimization for DirectX 12
  - Reduction of resource barriers
  - Buffer update
  - RootSignature
  - Memory management

Reduction of resource barriers

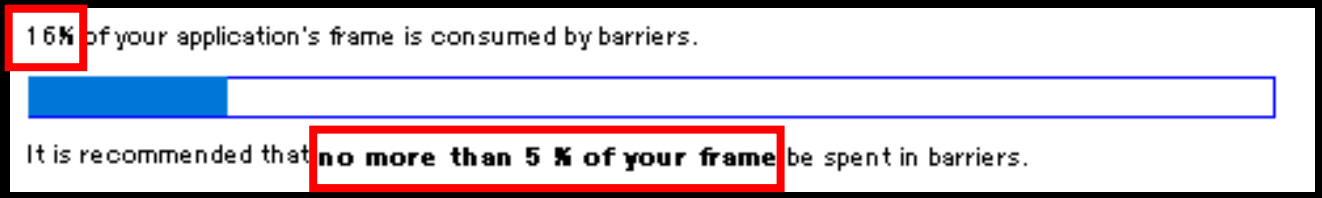


Resource barrier without optimzation

- In our original build without optimization, we inserted resource barrier in batches
- Immediately before executing drawing command, transition the resource barrier required for the current batch

Resource barriers

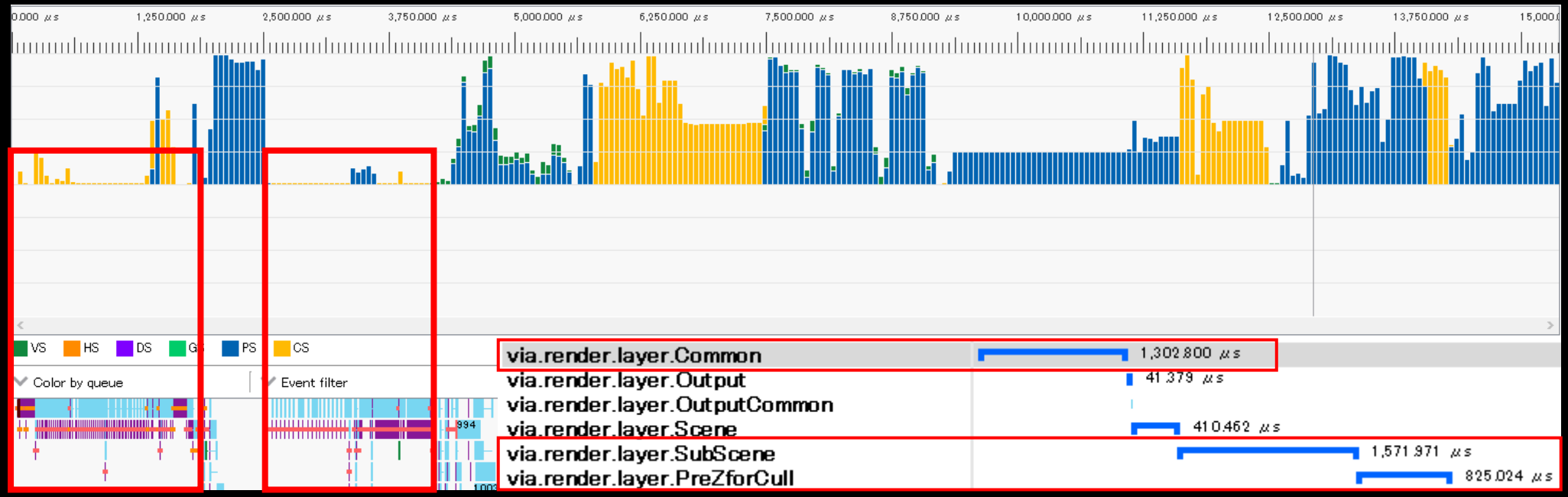
- Large number of resource barriers
  - One of the reasons GPU-based occlusion culling did not improve performance as much




```
540 ResourceBarrier()
541 ExecuteIndirect()
542 ResourceBarrier()
543 Dispatch(1, 1, 1)
544 ResourceBarrier()
545 Dispatch(1, 1, 1)
546 ResourceBarrier()
547 Dispatch(1, 1, 1)
548 ResourceBarrier()
549 Dispatch(1, 1, 1)
550 ResourceBarrier()
551 Dispatch(1, 1, 1)
552 ResourceBarrier()
553 Dispatch(1, 1, 1)
554 ResourceBarrier()
555 Dispatch(1, 1, 1)
```

Resource barriers

- Sections with many resource barriers are not operating efficiently



Reducing resource barriers

- Optimize by considering the sub resource for each resource
- It is difficult to manually create the best resource barrier from all intermediate drawing commands
- difficulty
  - Getting maximum GPU performance
  - Keeping it Bug free ☺

Add pre-pass for command analysis

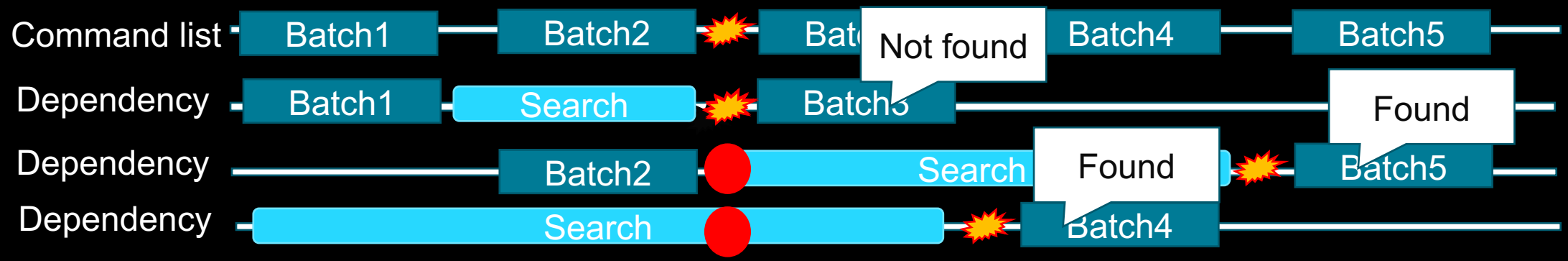
- Calculate the position of resource barrier automaticaly
  - Analyze intermediate drawing command
- Intermediate drawing commands are sorted by priority
  - Able to track the usage of drawing command chronologically for each resource
- Analysing batches with dependency can easily improve efficiency of GPU by shifting the priority order

Resource barrier compaction

- Search for precursor resource barrier

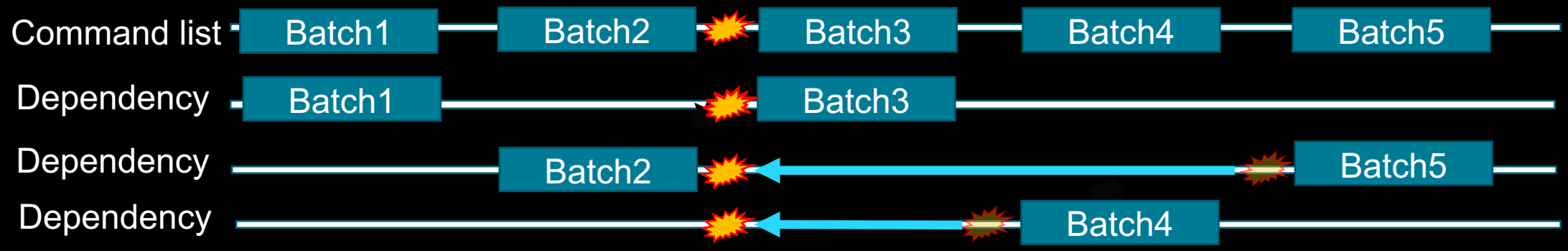
Resource barrier compaction

- Search for precursor resource barrier



Resource barrier compaction

- Search for precursor resource barrier
  - Bundle if possible

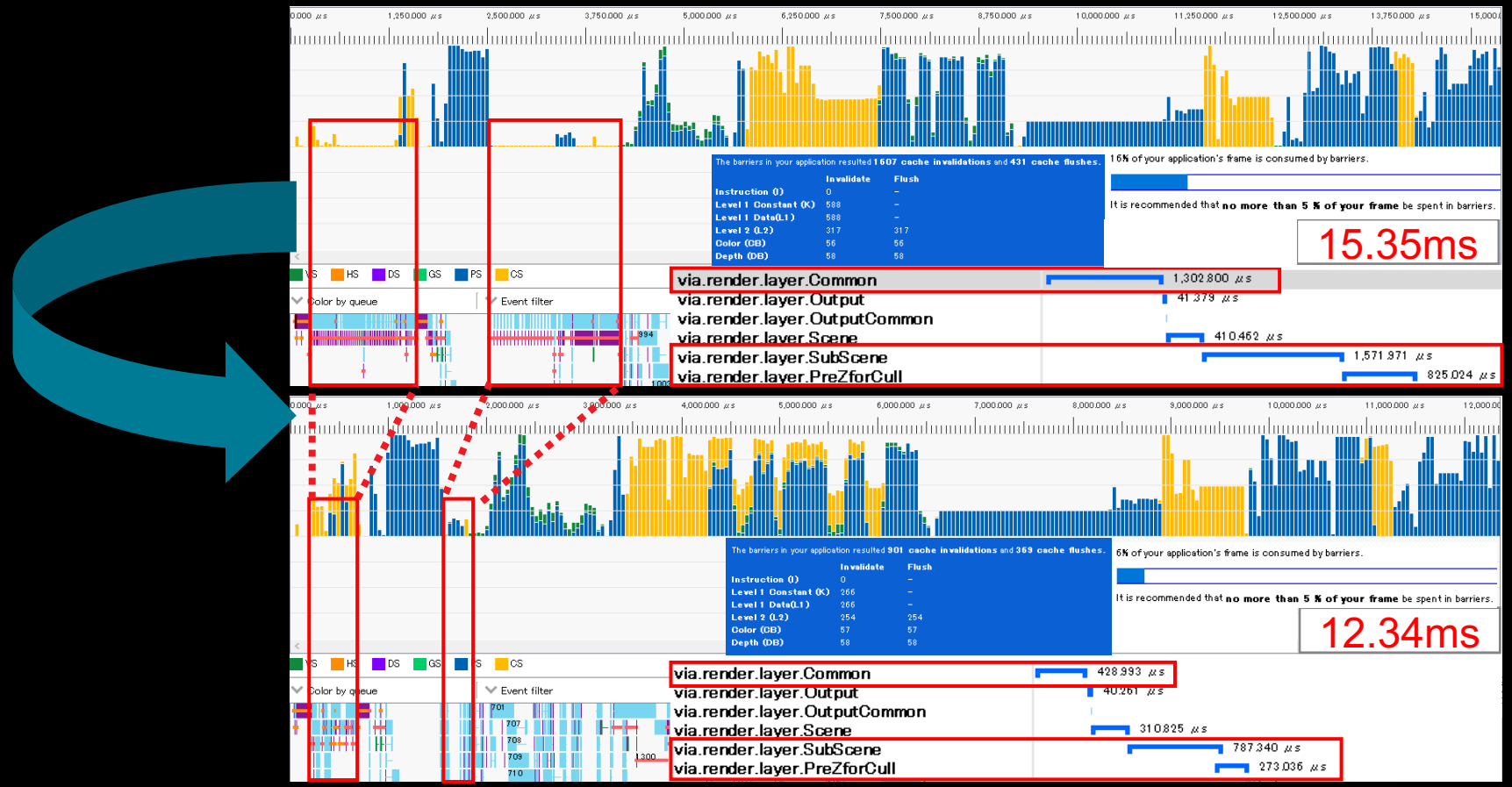


Advantage / Disadvantage

- Advantage
  - Need not be as conscious of internal implementation and caching
  - Reduce unnecessary resource barriers

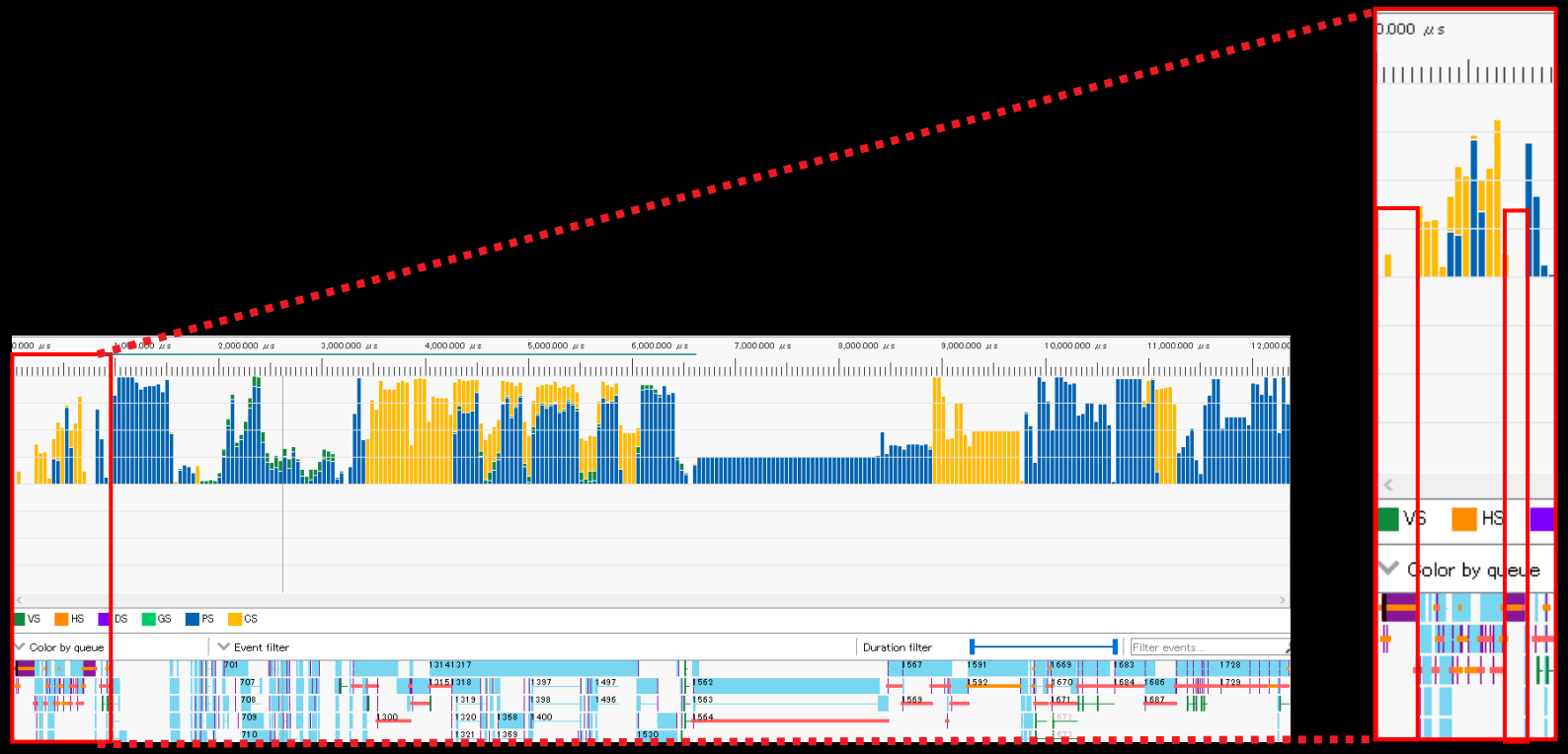
- Disadvantage
  - Requires command parsing time
    - PC is super fast!

Comparison : Resource barrier reduction



Still not enough?

- There are still inefficient sections in updating the buffer



Still not enough?

- A large amount of resource barriers caused by the driver in DMA transfer

9	1.839 μs	1.839 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
10	1.924 μs	1.924 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
11	1.858 μs	1.858 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
12	1.877 μs	1.877 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
13	2.067 μs	2.067 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
14	1.806 μs	1.806 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
15	1.580 μs	1.580 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
16	1.703 μs	1.703 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
17	1.684 μs	1.684 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
18	2.274 μs	2.274 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
19	2.214 μs	2.214 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
20	1.798 μs	1.798 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
21	1.798 μs	1.798 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
22	2.224 μs	2.224 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
23	1.826 μs	1.826 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
24	2.190 μs	2.190 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.
25	1.937 μs	1.937 μs	DMA	K L1 L2	L2	DRIVER	Blit synchronization.

What was going on?

- Buffer updates on graphics queue
- CopyBufferRegion
  - GPU particle buffer update
  - Updating skinning matrix
- CopyBufferRegion is executed as DMA transfer

What was going on?

- Strong cache flush was operating when DMA transfer was performed
- L1-Cache,L2-Cache,K-Cache
  - Batching resource barrier has no effect

Invalidated	Flushed	Barrier type	Reason for barrier
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.
K L1 L2	L2	DRIVER	Blit synchronization.

- Possible solutions
  - Update with CopyQueue if only one update per frame
  - Update using compute shader

- We used compute shader

Compute shader based update

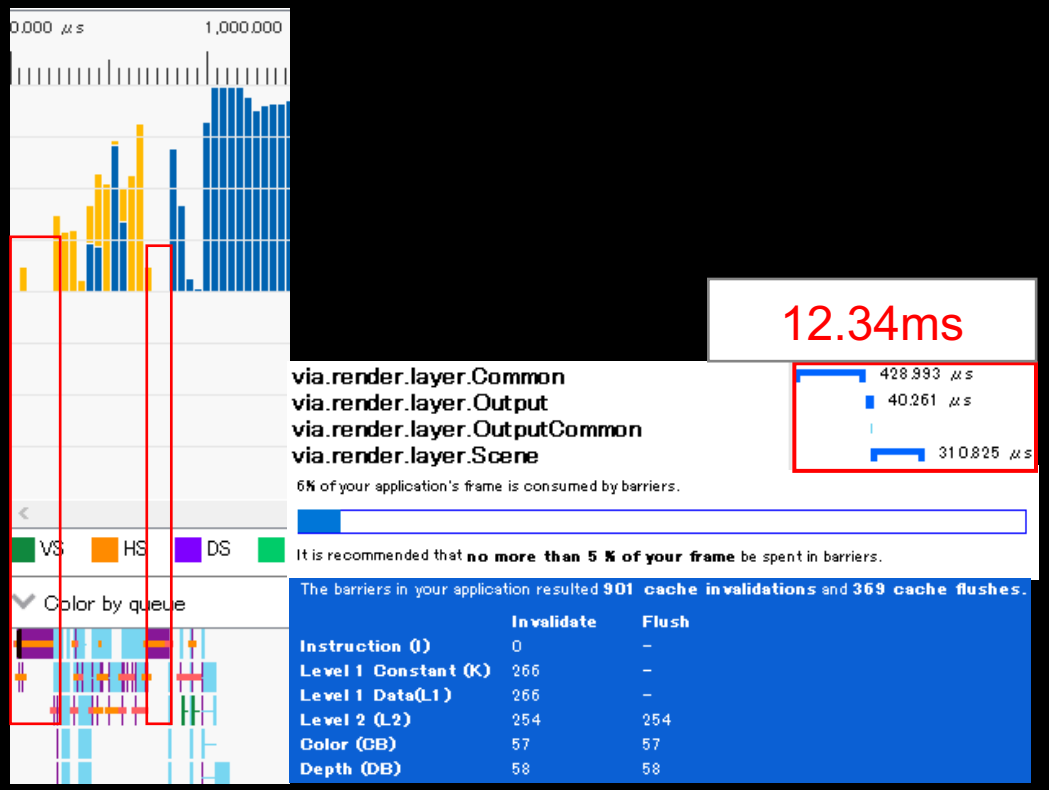
```
StructuredBuffer<uint> fastCopySource;
RWStructuredBuffer<uint> fastCopyTarget;

[numthreads(256,1,1)]
void CS_FastCopy( uint groupID : SV_GroupID, uint threadID : SV_GroupThreadID )
{
        fastCopyTarget[(groupID.x * 2 + 0)*256 + threadID.x]  = fastCopySource[(groupID.x * 2 + 0)*256 + threadID.x];
        fastCopyTarget[(groupID.x * 2 + 1)*256 + threadID.x]  = fastCopySource[(groupID.x * 2 + 1)*256 + threadID.x];
}
```

Optimization of constant buffer update

- Update all constant buffer via upload heap
  - Updates to the same Constant Buffer needs resource-barrier and CopyBufferRegion(DMA transfer)
  - Store new value into upload heap and get upload heap offset address
- Shaders that use ConstantBuffer only needs reference offset address
  - Resource barrier and CopyBufferRegion are no longer needed

CopyBufferRegion reduction comparison

- Successfully removed inefficiency!



12.34ms

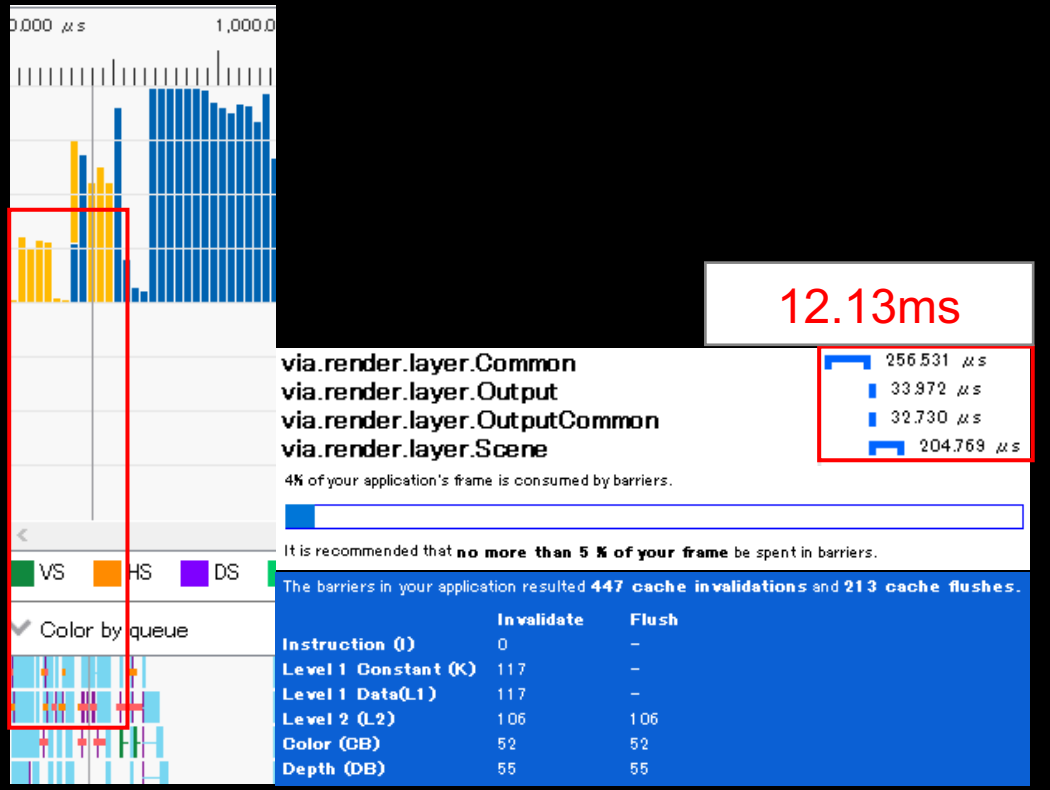
via.render.layer.Common 428.993 µs
via.render.layer.Output 40.261 µs
via.render.layer.OutputCommon
via.render.layer.Scene 310.825 µs

6% of your application's frame is consumed by barriers.

It is recommended that **no more than 5 % of your frame** be spent in barriers.

The barriers in your application resulted **901 cache invalidations** and **369 cache flushes.**

	Invalidate	Flush
Instruction (I)	0	–
Level 1 Constant (K)	266	–
Level 1 Data(L1)	266	–
Level 2 (L2)	254	254
Color (CB)	57	57
Depth (DB)	58	58



12.13ms

via.render.layer.Common 256.531 µs
via.render.layer.Output 33.972 µs
via.render.layer.OutputCommon 32.730 µs
via.render.layer.Scene 204.769 µs

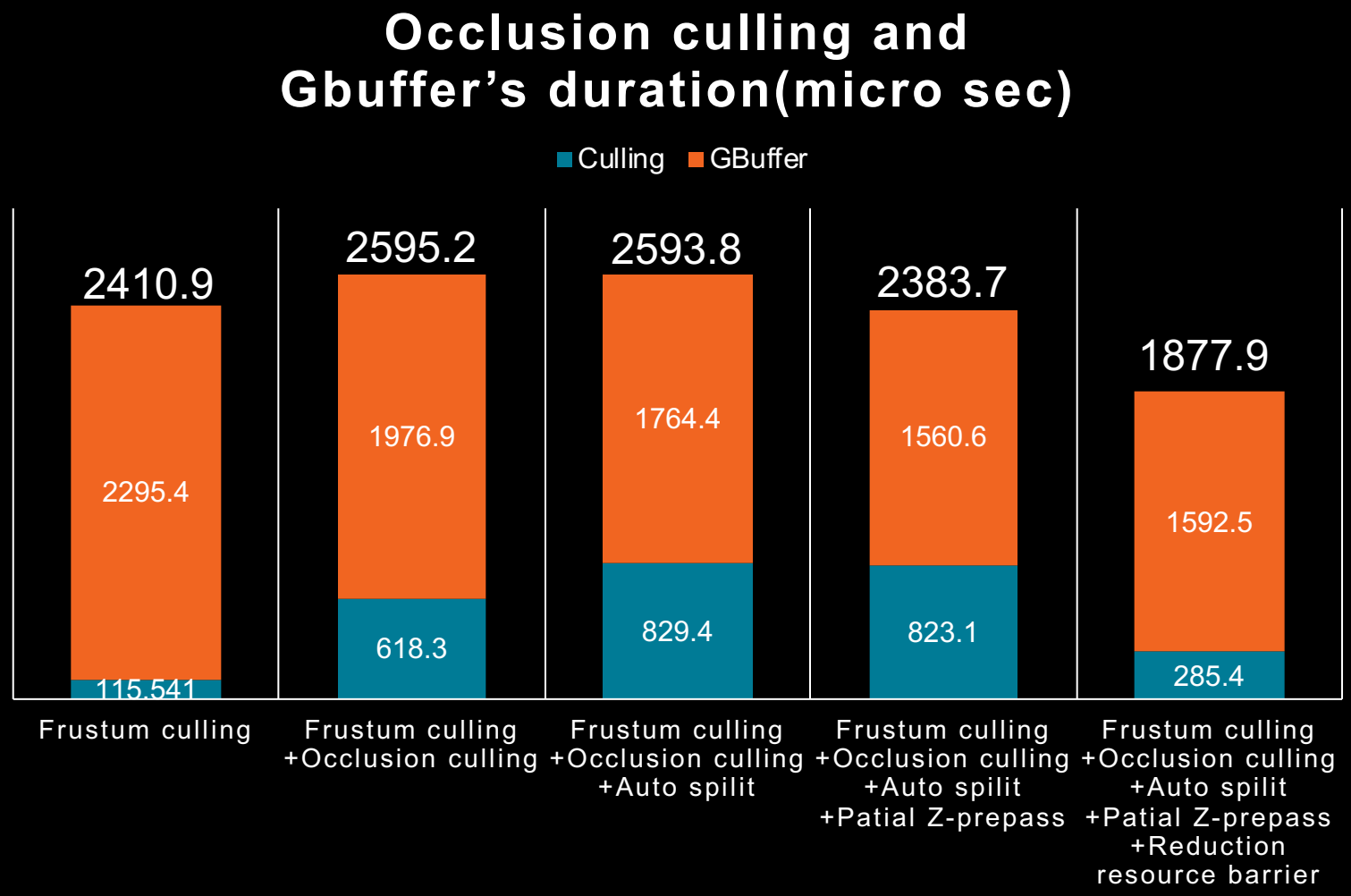
4% of your application's frame is consumed by barriers.

It is recommended that **no more than 5 % of your frame** be spent in barriers.

The barriers in your application resulted **447 cache invalidations** and **213 cache flushes.**

	Invalidate	Flush
Instruction (I)	0	–
Level 1 Constant (K)	117	–
Level 1 Data(L1)	117	–
Level 2 (L2)	106	106
Color (CB)	52	52
Depth (DB)	55	55

Comparison of each method



API DirectX 12, GPU AMD Radeon RX480 , Radeon Profiler

Root Signature

- DirectX12 uses similar RootSignature to DX11 & Consoles
- Determined at runtime, not at shader build
  - To provide customized optimization for each IHV
  - For AMD, use RootParamater as table
  - For NVIDIA, use RootParamater to optimize ConstantBuffer access

Root(AMD and Intel)
DescriptorTable(CBV 0-14)
DescriptorTable(SRV 0-32)
DescriptorTable(UAV 0-8)
DescriptorTable(Sampler)

Root(NVIDIA)
RootCBV(0)
RootCBV(1)
RootCBV(2)
RootCBV(3)
DescriptorTable(CBV 4-14)
DescriptorTable(SRV 0-32)
DescriptorTable(UAV 0-8)
DescriptorTable(Sampler)

Memory management

- In the first implementation, memory Evict started at around 50% memory usage
  - Pretty conservative

- Many spikes occurred during gameplay
  - In Resident Evil 2, controls loading and disposal for each room caused spikes every time the character moved

  - Even occurred when loading UI for pause menus

Memory management

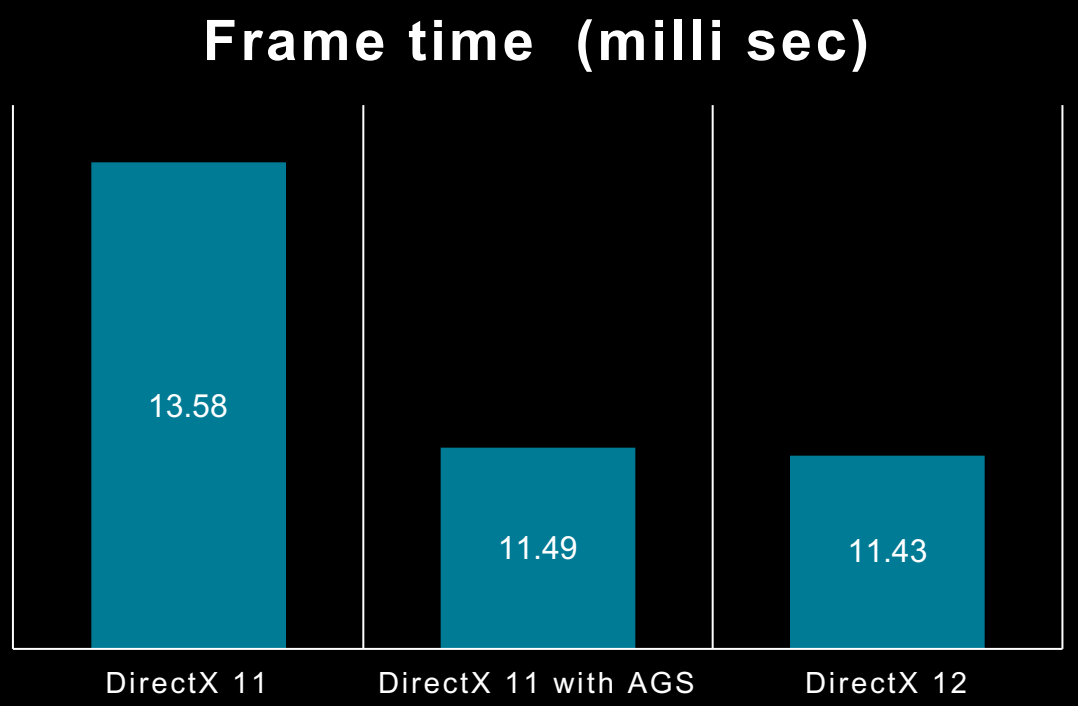
- Do not Evict until memory is exhausted
  - To prevent micro Evicts
- When the memory usage rate exceeds 90%, unreferenced memory is Evicted

Comparison after all optimizations

- 24% frame time saving!

Comparison of DirectX 11 / DirectX 12

- Profile Resident Evil 2 in game



GPU AMD Radeon RX480 , OCAT

Future works

- AsyncCompute
  - Used for Consoles
  - Implementaion was incompatible for PC

- Shader model 6.0
  - Some tests and trials were done
  - Not enough time to ensure stability

Optimization recap

- Although optimizations from console are useful, it may be inadequate by itself
- Reducing resource barrier is important
  - Big impact on performance!
  - Effectiveness of other optimization methods can be affected by the resource barrier
- Paging spikes decreased when memory management was done all at once rather than doing it in small increments
  - May be due to game design
  - Worked well even at around 90% utilization

Tips

Pre-bake PipelineStateObject

- Creating PipelineStateObject at runtime is slow
- It would be better if we can Pre-bake PipelineStateObject beforehand.

Pre-bake PipelineStateObject

- We pre-bake PSO before the final package
  - Included in assets created on the engine

```
{
    "Name":"LightCulling",
    "CS":["lightCulling.hlsl","CS_LightCulling",""]
},
{
    "Name":"IndirectIllumination",
    "CS":["deferred.hlsl","IndirectIlluminationCS",""]
},
```

- RTV, DSV and index bit stride are not included at first
- We use the collected information to pre-bake the PSO for the final package
  - Much smoother for the end-user.

Load PipelineStateObject at runtime

- Compile in the background during asset loading
  - Compute shader : Create immediately on another thread
  - Other shaders : Create if it is on the collected information.

- However, if the build of PipelineStateObject is not completed beforehand, the CPU is blocked

Quality Assurance (QA)

- Quality Assurance for PC version frequently suffer from GPU crashes
  - Various factors such as CPU, GPU, display, etc
- However, crash dumps were not useful for debugging GPU crashes
  - No way to trace
  - RE ENGINE does not offer functions to replay command lists… yet
- In DirectX 12, use WriteBufferImmediate
  - Read back executing shader name to the buffer for each drawing command
  - Able to know the shader name that was running at the time of crash
  - In DirectX 11, AGS supports BreadcrumbBuffer as same function.

Acknowledgments

- Big thanks to RE ENGINE dev team’s contribution and to the support of IHVs
  - Many bugs were fixed by the driver team!

Questions?

References

- GPU-Driven Rendering Pipelines, Ulrich Haar (Ubisoft Entertainment), Sebastian Aaltonen (Ubisoft Entertainment)
- Optimizing the Graphics Pipeline with Compute, Graham Wihlidal
- Improved Culling for Tiled and Clustered Rendering, Michal Drobot
- The Devils in the details, Tiago Sousa (idTech), Jean Geffroy (idTech) Siggraph 2016
- AMD GeometryFX
- Rendering with Conviction, Stephen Hill
- Moving to DirectX 12: Lessons Learned, Tiago Rodrigues
- Graphics optimization of the latest title in Capcom, Hitoshi Mishima CEDEC 2018