COMPUTE SHADERS

LOU KRAMER

DEVELOPER TECHNOLOGY ENGINEER, AMD

AGENDA

- Introduction to Compute Shaders.
  - Software.
  - Hardware.
- Memory on RDNA™2.
  - Caches.
  - Groupshared memory (aka LDS).
  - Texture Access.
- Execution model on RDNA™2.
  - Divergence.
  - Scalarization.
- Export.
- Conclusion.

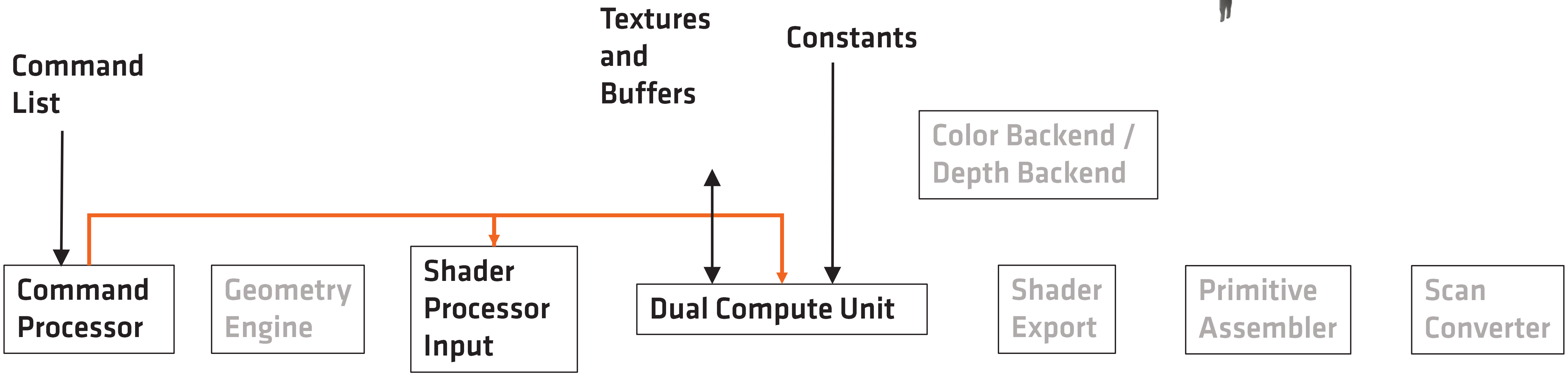
INTRODUCTION

THE COMPUTE PIPELINE

- In case you watched my talk from last year at the GIC‘20 ...
- Remember the compute pipeline?

Compute Shader Stage

ON THE RDNA™2 ARCHITECTURE



Command List

Textures and Buffers

Constants

Color Backend / Depth Backend

Command Processor

Geometry Engine

Shader Processor Input

Dual Compute Unit

Shader Export

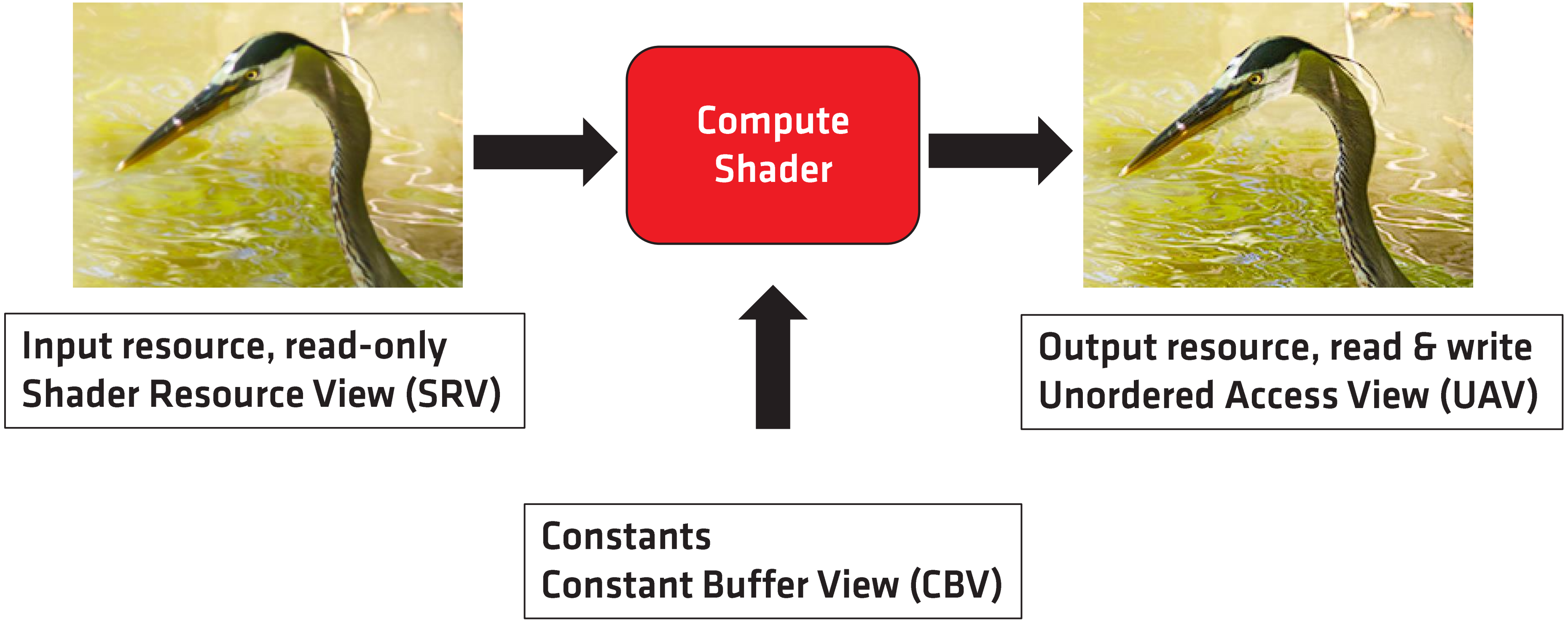
Primitive Assembler

Scan Converter

🟧 Compute Pipeline

DISPATCH A COMPUTE SHADER - EXAMPLE

From a developer‘s point of view:
What commands do we need to submit (to the command processor) to schedule a compute shader dispatch?



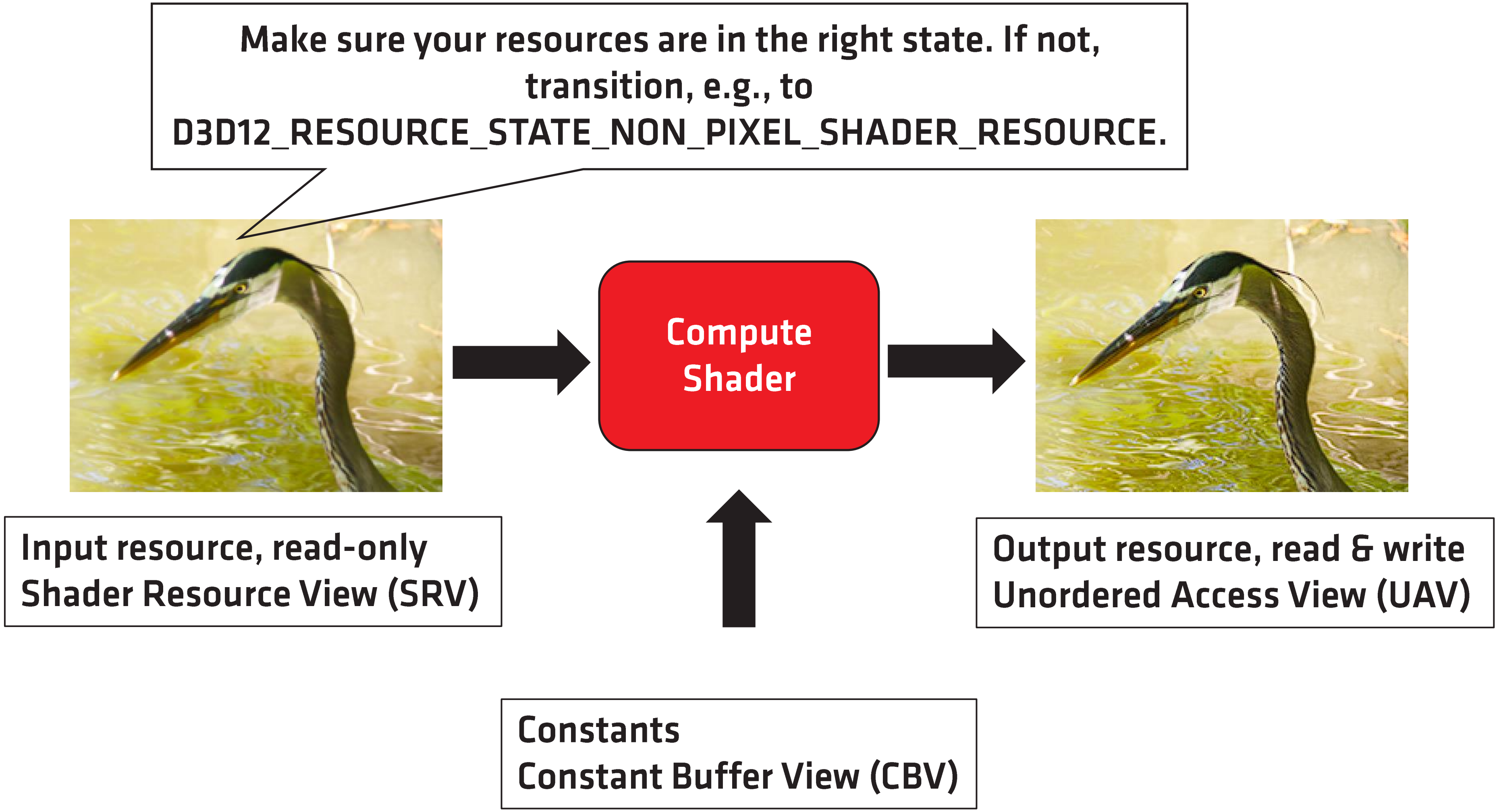
Compute Shader

Input resource, read-only
Shader Resource View (SRV)

Output resource, read & write
Unordered Access View (UAV)

Constants
Constant Buffer View (CBV)

DISPATCH A COMPUTE SHADER - EXAMPLE



Make sure your resources are in the right state. If not, transition, e.g., to D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE.

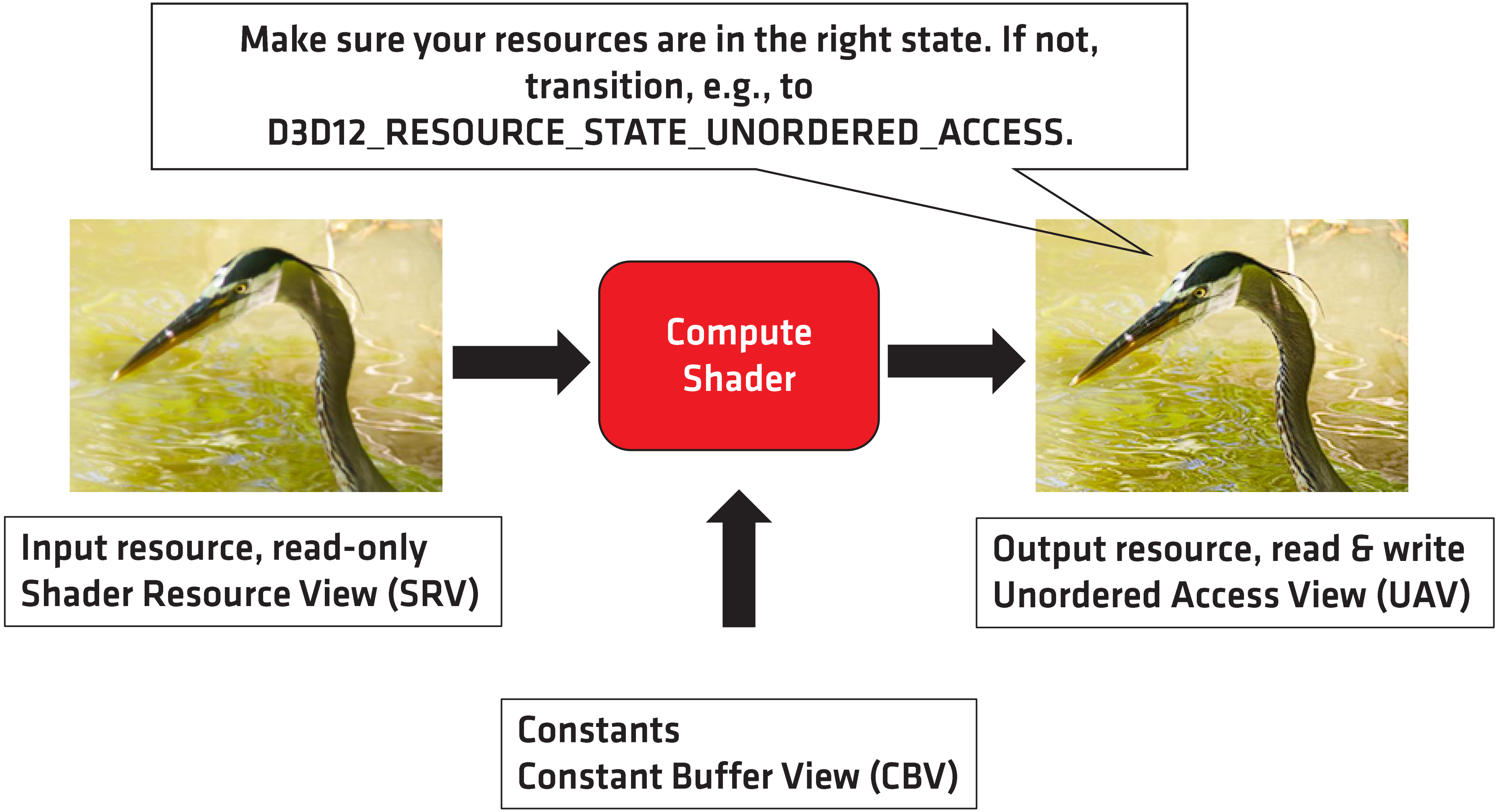
Compute Shader

Input resource, read-only
Shader Resource View (SRV)

Output resource, read & write
Unordered Access View (UAV)

Constants
Constant Buffer View (CBV)

DISPATCH A COMPUTE SHADER - EXAMPLE



Make sure your resources are in the right state. If not, transition, e.g., to D3D12_RESOURCE_STATE_UNORDERED_ACCESS.

Compute Shader

Input resource, read-only
Shader Resource View (SRV)

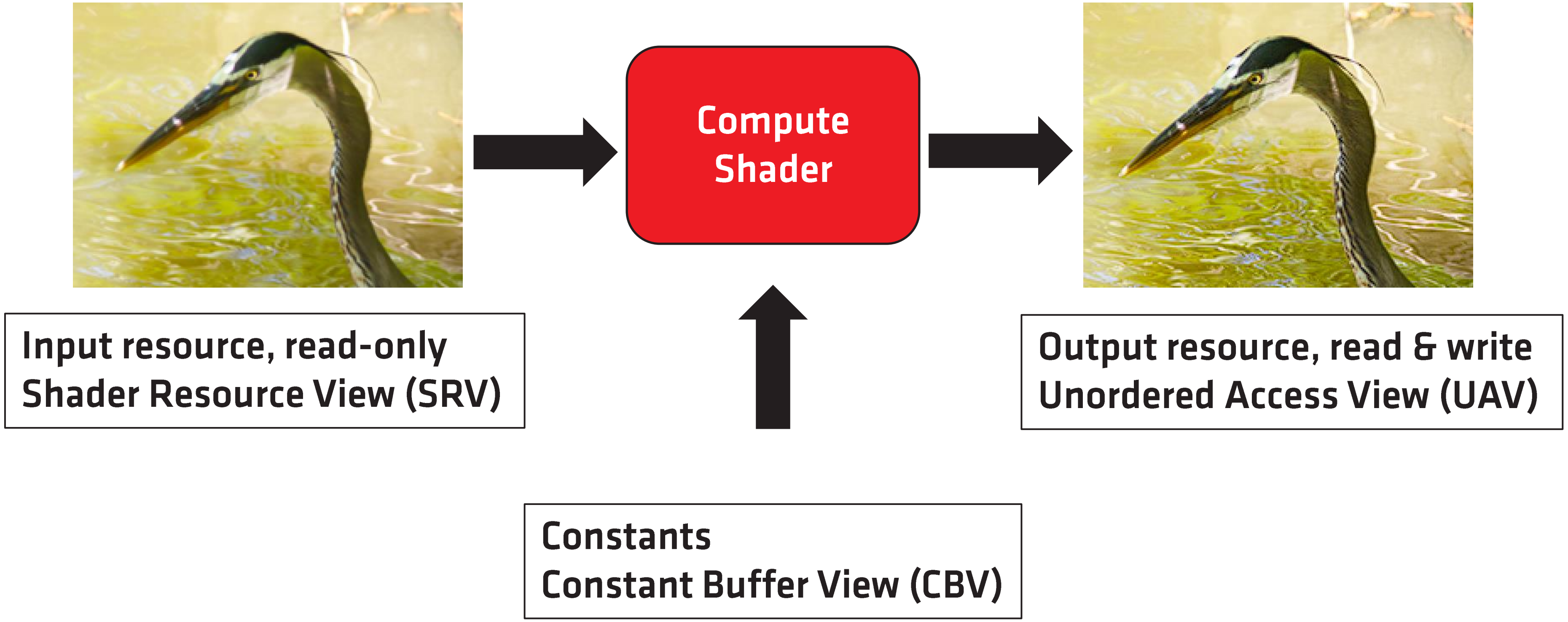
Output resource, read & write
Unordered Access View (UAV)

Constants
Constant Buffer View (CBV)

DISPATCH A COMPUTE SHADER - EXAMPLE

Avoid COMMON state whenever possible
(and this is not specific to Compute Shaders ☺).

Can lead to a significant performance drop!



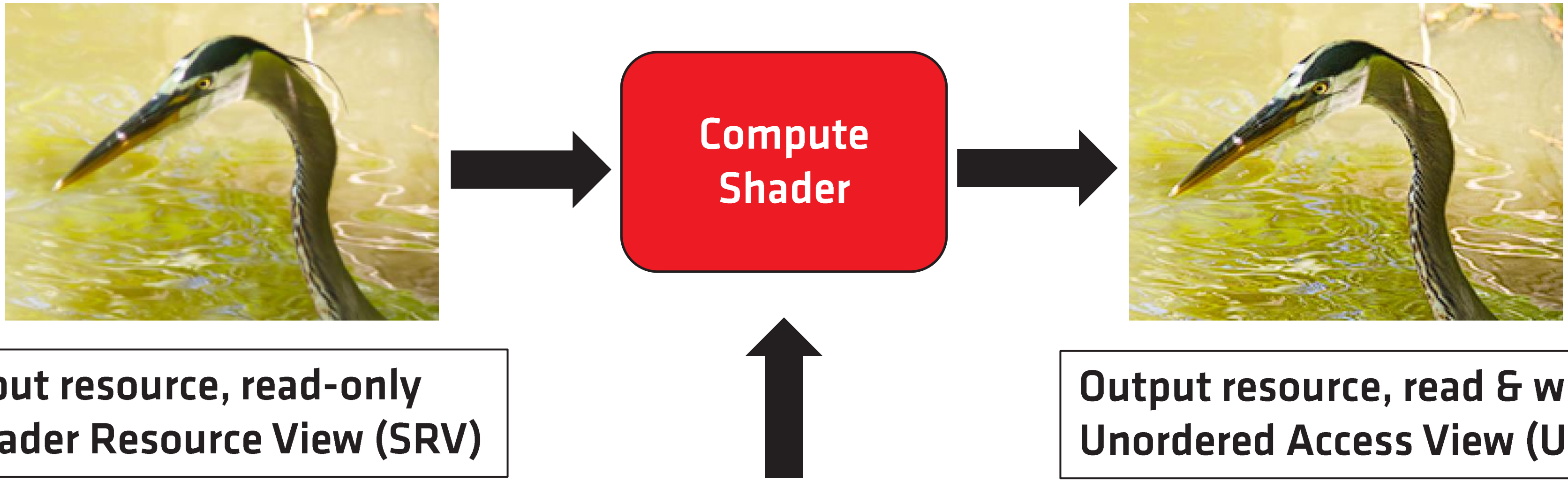
Compute Shader

Input resource, read-only
Shader Resource View (SRV)

Output resource, read & write
Unordered Access View (UAV)

Constants
Constant Buffer View (CBV)

DISPATCH A COMPUTE SHADER - EXAMPLE



Compute Shader

Input resource, read-only
Shader Resource View (SRV)

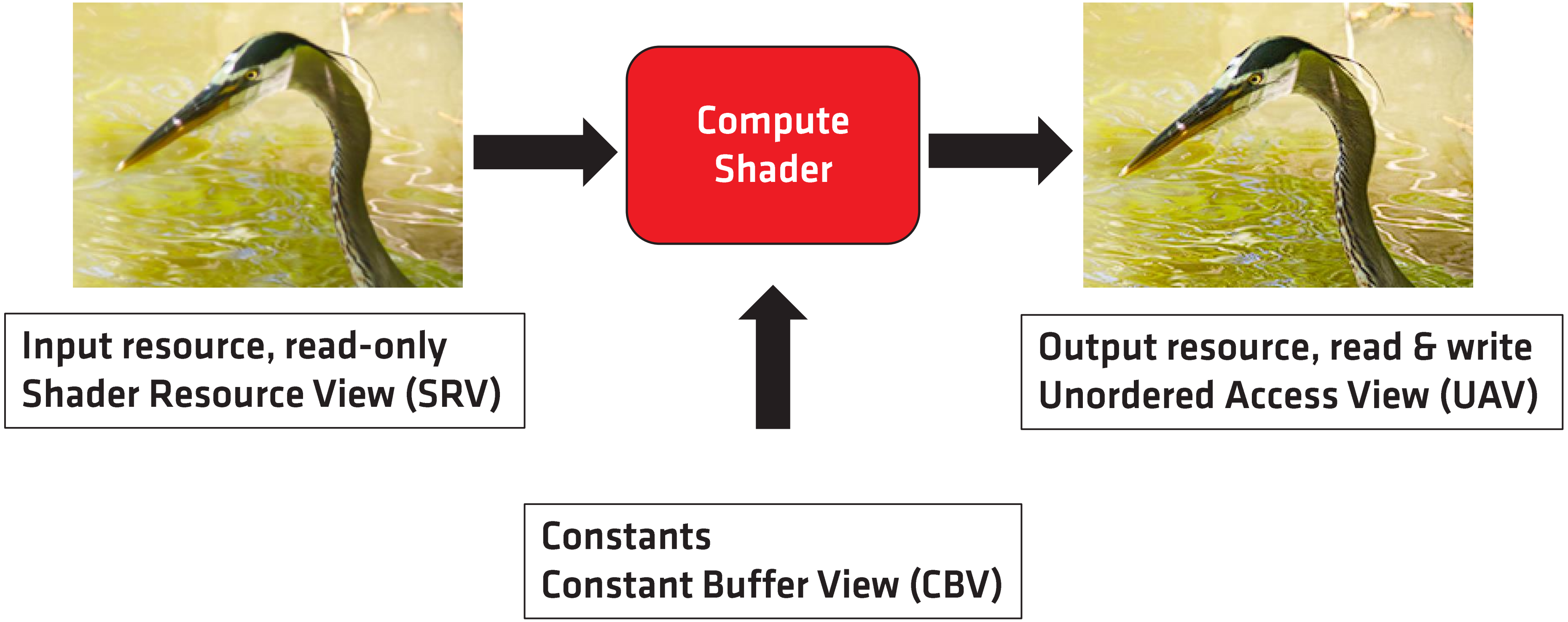
Output resource, read & write
Unordered Access View (UAV)

Constants
Constant Buffer View (CBV)

Update constants if necessary, e.g., with current per-frame data.

DISPATCH A COMPUTE SHADER - EXAMPLE

- "Bind" the resources – so the GPU knows which resources to access and how you refer to them in the shader.
- "Bind" your compute pipeline – it contains your compute shader you want to run.
- Dispatch!



Compute Shader

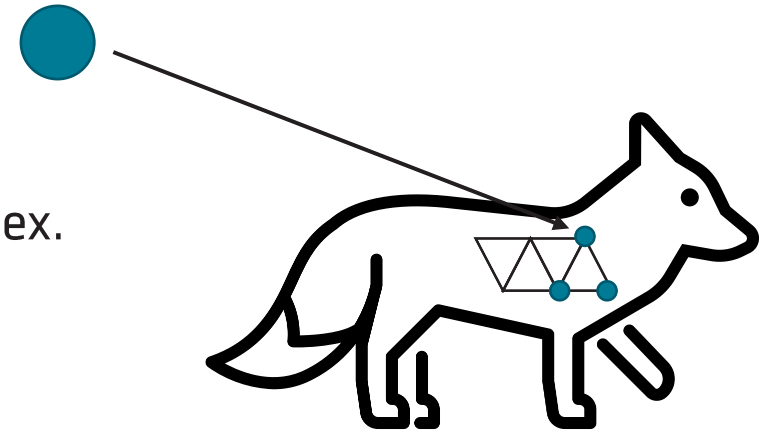
Input resource, read-only
Shader Resource View (SRV)

Output resource, read & write
Unordered Access View (UAV)

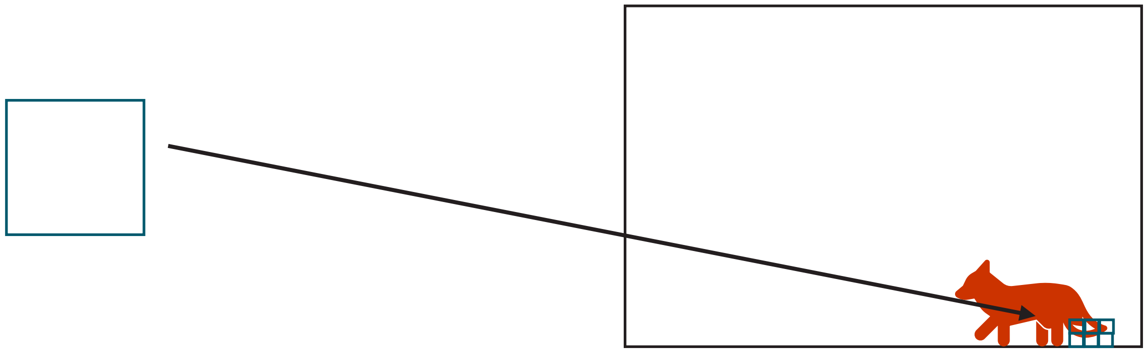
Constants
Constant Buffer View (CBV)

DISPATCH

A single workitem for vertex shaders is a vertex.



A single workitem for pixel shaders is a pixel.

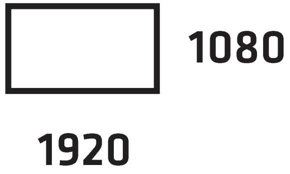


A single workitem for compute shaders is called a thread.

Abstract thingy ...

DISPATCH – FULLSCREEN PASS

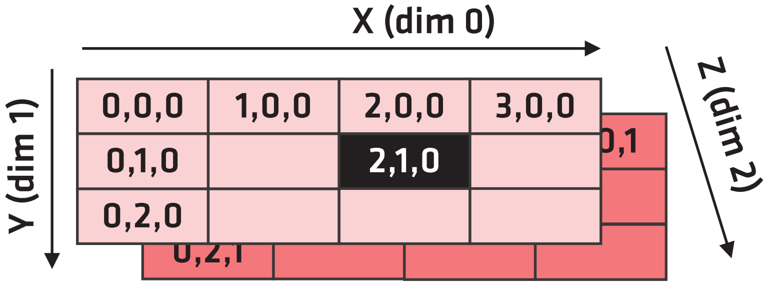
A fullscreen pass runs on every pixel in the output screen:



**For compute shaders,
we need to explicitly specify the number of threads.**

This means, we need 1920x1080 pixel shader invocations.

HIERARCHY OF WORK ITEMS



Dispatch(4,3,2);
The dispatch call invokes 4 * 3 * 2 = 24 thread groups in undefined order.

Thread (5,1,0)

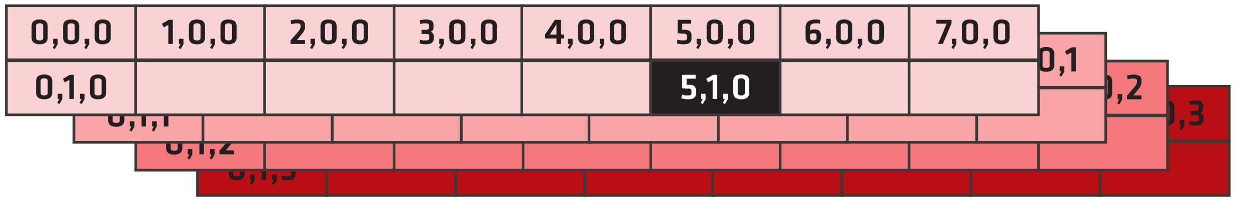
SV_GroupThreadID: (5,1,0)
SV_GroupID: (2,1,0)
SV_DispatchThreadID: (2,1,0) * (8,2,4) + (5,1,0) = (21, 3, 0)
SV_GroupIndex: 0 * 8 * 2 + 1 * 8 + 5 = 13

In the compute shader, the thread group size is declared using

[numthreads(8, 2, 4)]

-> each thread group has 64 threads.

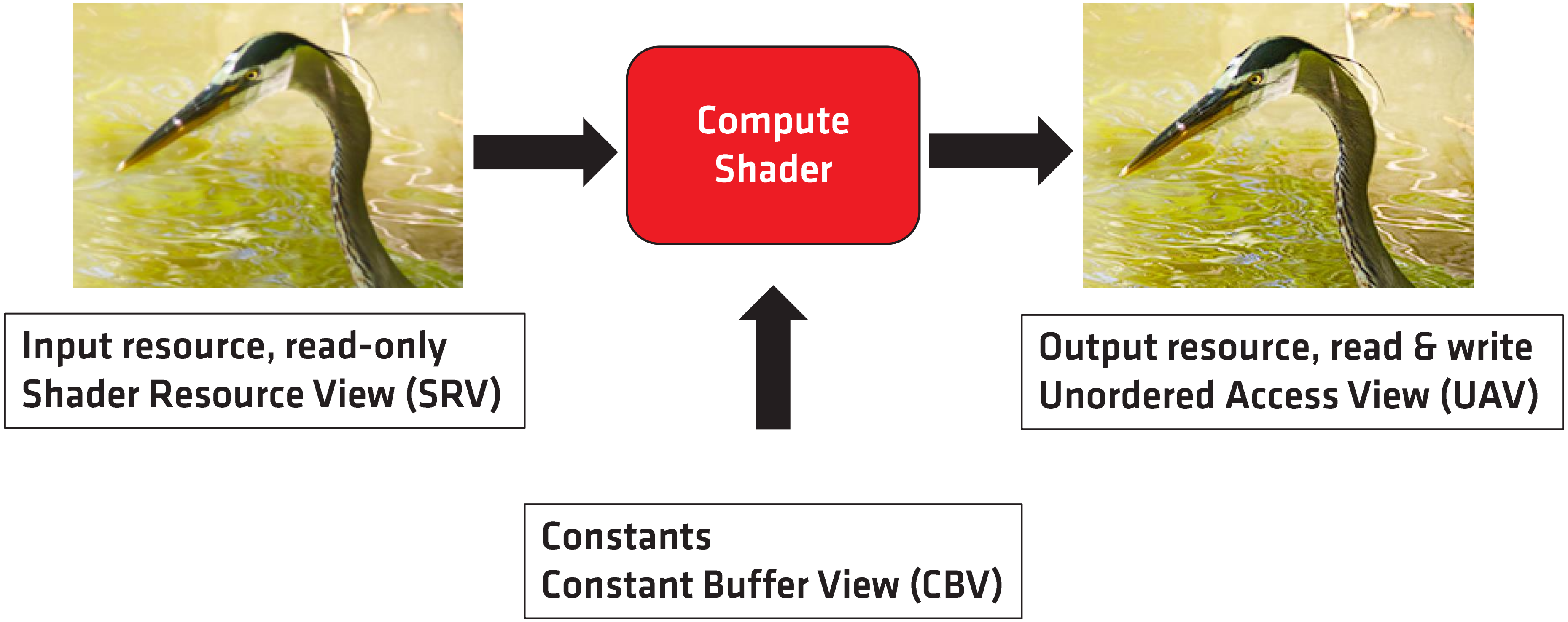
Thread group (2,1,0)



based on: https://docs.microsoft.com/en-us/windows/desktop/direct3dhlsl/sv-dispatchthreadid

DISPATCH A COMPUTE SHADER – FULLSCREEN PASS

- Output resource: **1920x1080** texels.
- Approach: each thread produces **1** output texel -> need **1920x1080** threads.
- If we choose a thread group size of **8x8**, we need **240x135** thread groups.



Input resource, read-only
Shader Resource View (SRV)

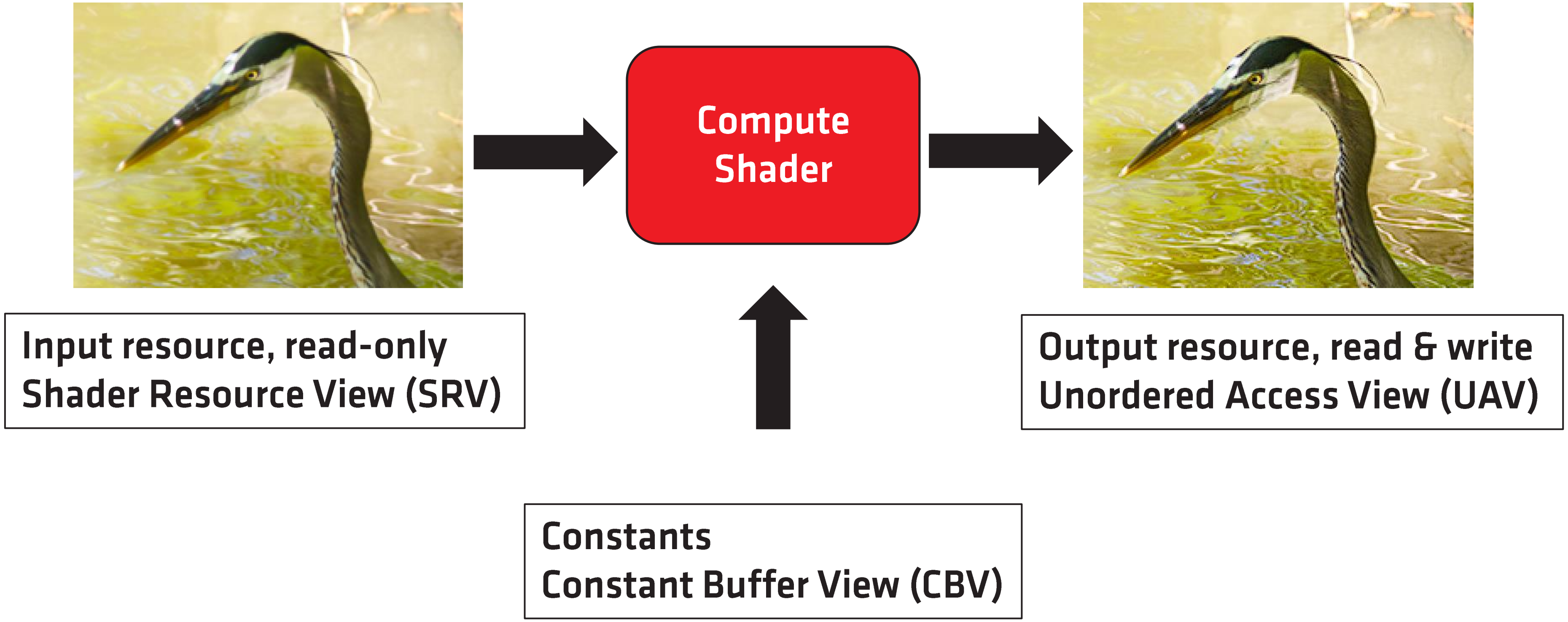
Compute Shader

Output resource, read & write
Unordered Access View (UAV)

Constants
Constant Buffer View (CBV)

DISPATCH A COMPUTE SHADER – FULLSCREEN PASS

- Output resource: **1920x1080** texels.
- Approach: each thread produces **4** output texel -> need **960x540** threads.
- If we choose a thread group size of **8x8**, we need **120x68** thread groups.



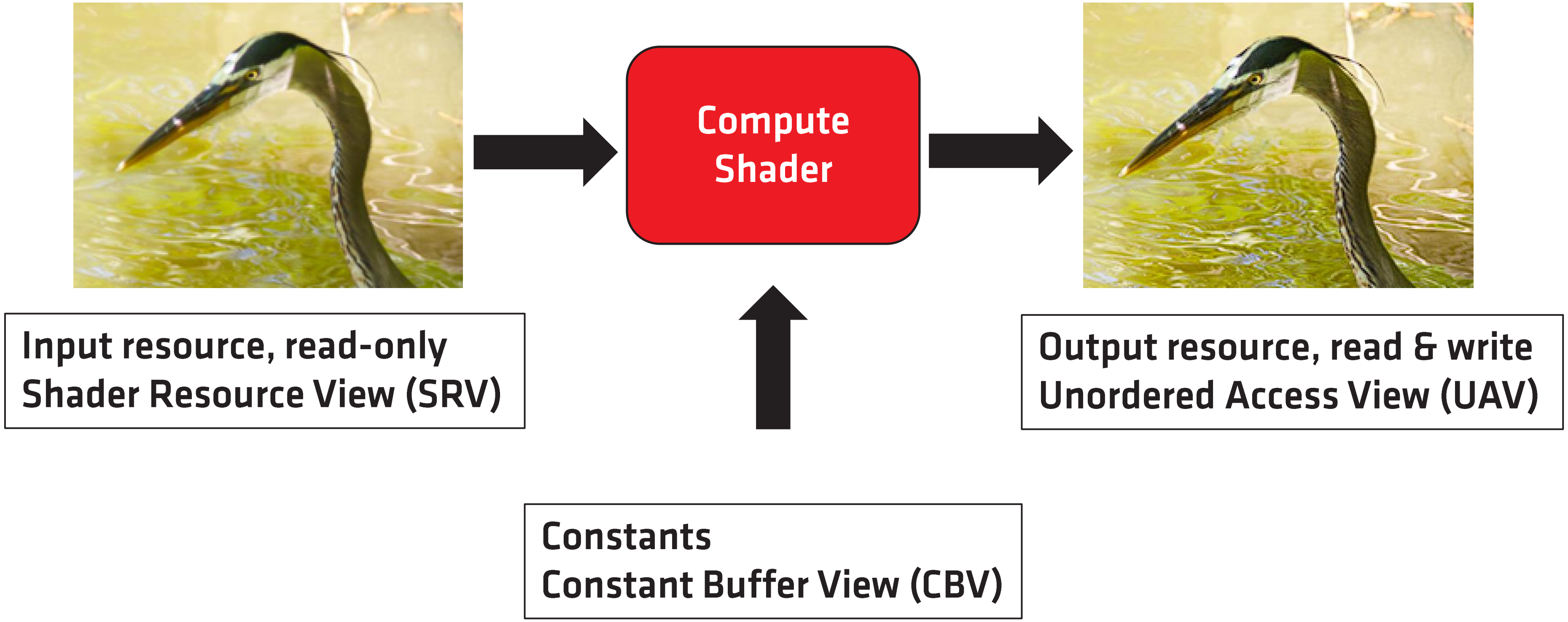
Input resource, read-only
Shader Resource View (SRV)

Output resource, read & write
Unordered Access View (UAV)

Constants
Constant Buffer View (CBV)

DISPATCH A COMPUTE SHADER – FULLSCREEN PASS

- Output resource: **1920x1080** texels.
- Approach: each thread produces **4** output texel -> need **960x540** threads.
- If we choose a thread group size of **8x4**, we need **120x135** thread groups.



Compute Shader

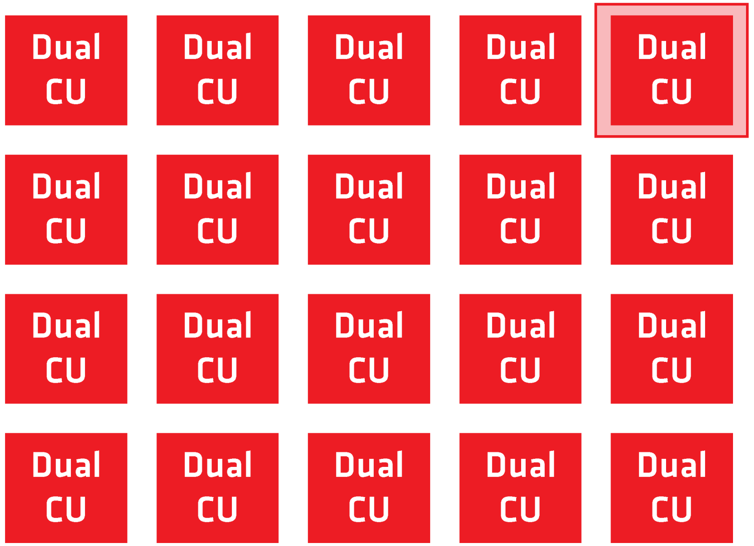
Input resource, read-only
Shader Resource View (SRV)

Output resource, read & write
Unordered Access View (UAV)

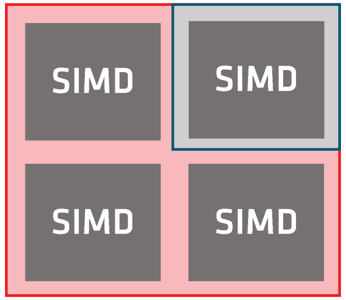
Constants
Constant Buffer View (CBV)

SCHEDULING OF THE WORKLOAD

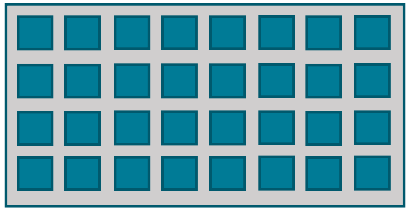
- Dual Compute Units are designed to execute parallel workloads!
- The number of Dual CUs depends on the card,
e.g., Radeon™ RX 6900 XT has 40 Dual CUs.



…



4 x 32-wide SIMDs per Dual CU.



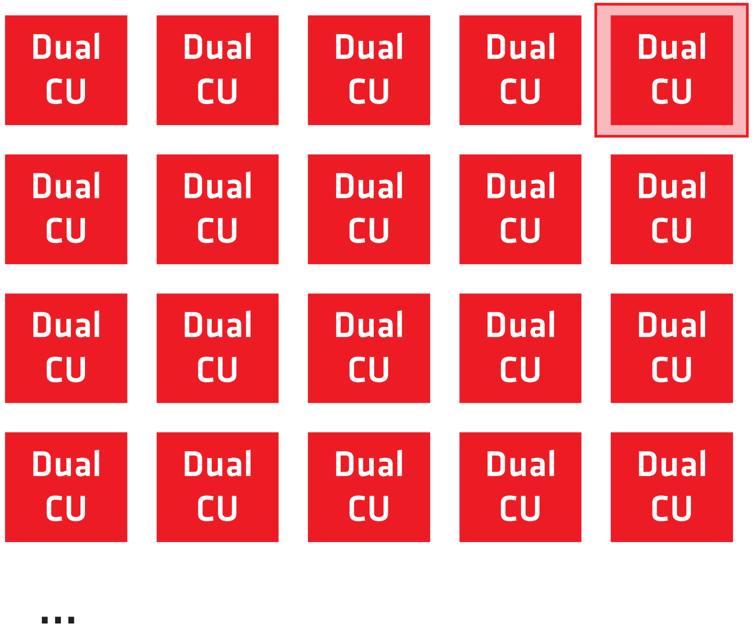
32 threads per SIMD.

One vector register (VGPR) holds one value per thread.

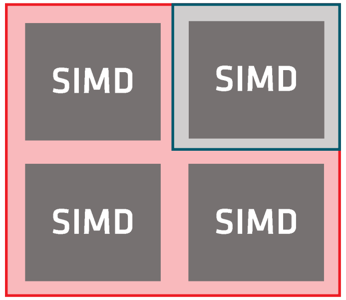
One scalar register (SGPR) holds one value per wave.

SCHEDULING OF THE WORKLOAD

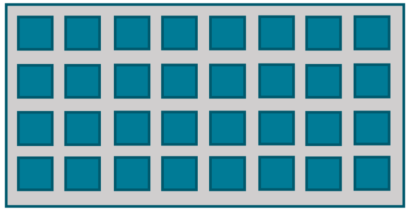
- Dual Compute Units are designed to execute parallel workloads!
- The number of Dual CUs depends on the card,
  e.g., Radeon™ RX 6900 XT has 40 Dual CUs.



Each thread group gets scheduled to an available Dual CU.



4 x 32-wide SIMDs per Dual CU.



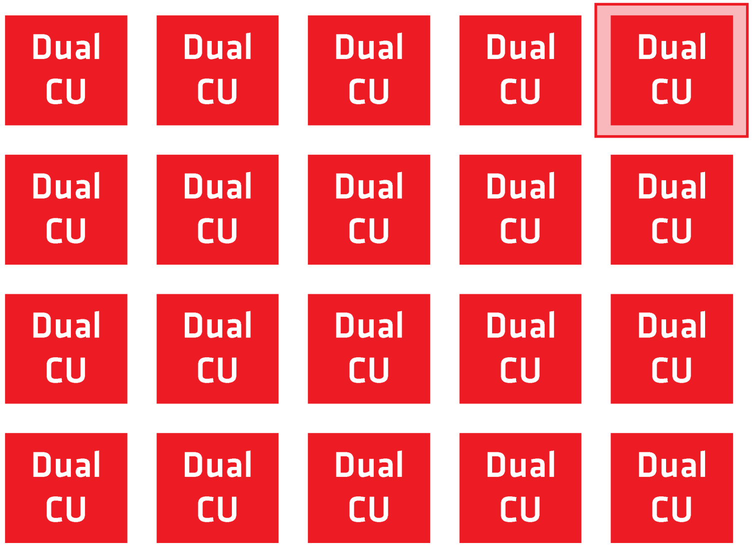
32 threads per SIMD.

One vector register (VGPR) holds one value per thread.

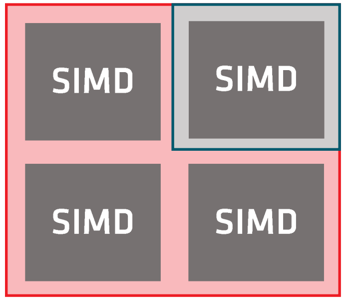
One scalar register (SGPR) holds one value per wave.

SCHEDULING OF THE WORKLOAD

- Dual Compute Units are designed to execute parallel workloads!
- The number of Dual CUs depends on the card, e.g., Radeon™ RX 6900 XT has 40 Dual CUs.

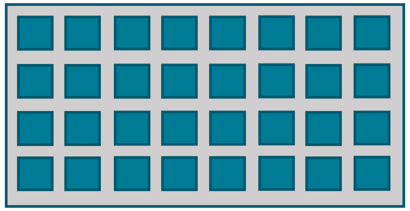


...



4 x 32-wide SIMDs per Dual CU.

The threads get scheduled to a SIMD in wavefronts:
Either 32 threads or 64 threads.



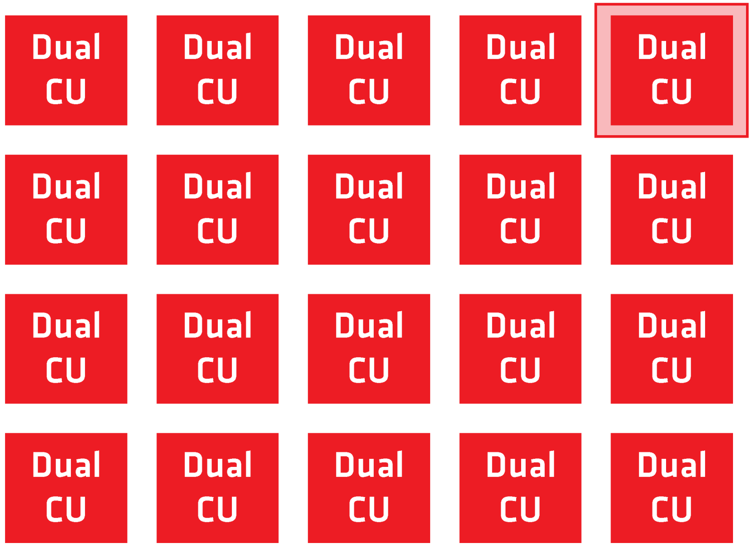
32 threads per SIMD.

One vector register (VGPR) holds one value per thread.

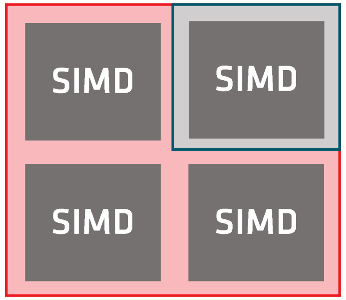
One scalar register (SGPR) holds one value per wave.

SCHEDULING OF THE WORKLOAD

- Dual Compute Units are designed to execute parallel workloads!
- The number of Dual CUs depends on the card, e.g., Radeon™ RX 6900 XT has 40 Dual CUs.

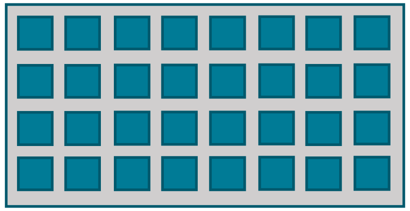


...



4 x 32-wide SIMDs per Dual CU.

The 32 threads on the SIMD. In case of wavefront 64, another 32 threads get scheduled right after the first batch.



32 threads per SIMD.

One vector register (VGPR) holds one value per thread.

One scalar register (SGPR) holds one value per wave.

MEMORY & COMMUNICATION

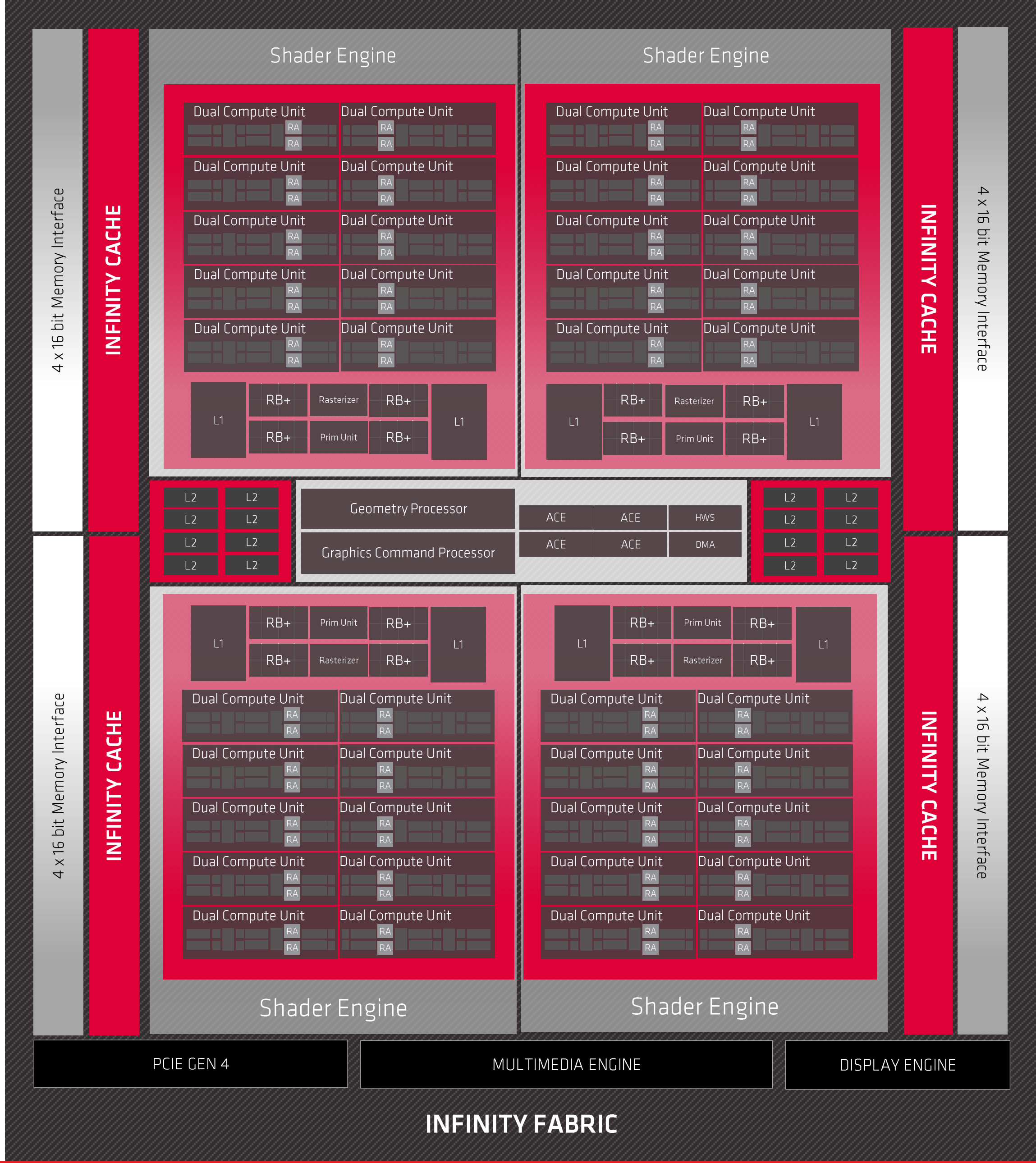
MEMORY & CACHES

Global Memory -> Local Device Memory.

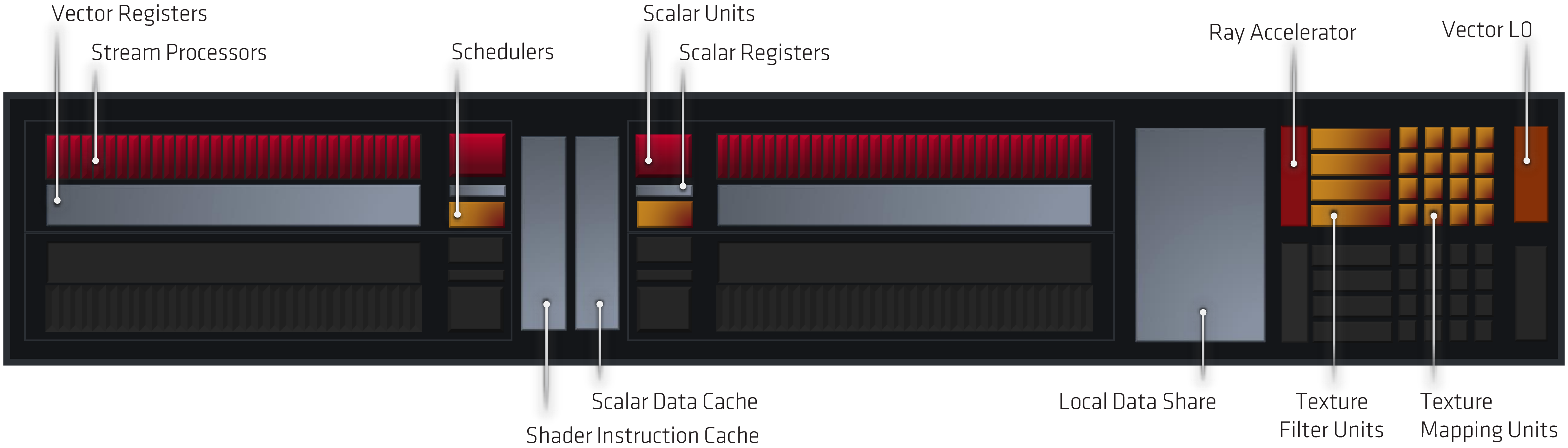
Caches.

- Infinity Cache (Global).
- L2 Cache (Global).
- L1 Cache (Shader Array → 5 Dual CUs).
- L0 Cache (CU).

Groupshared Memory, aka Local Data Share (LDS) (Dual CU).

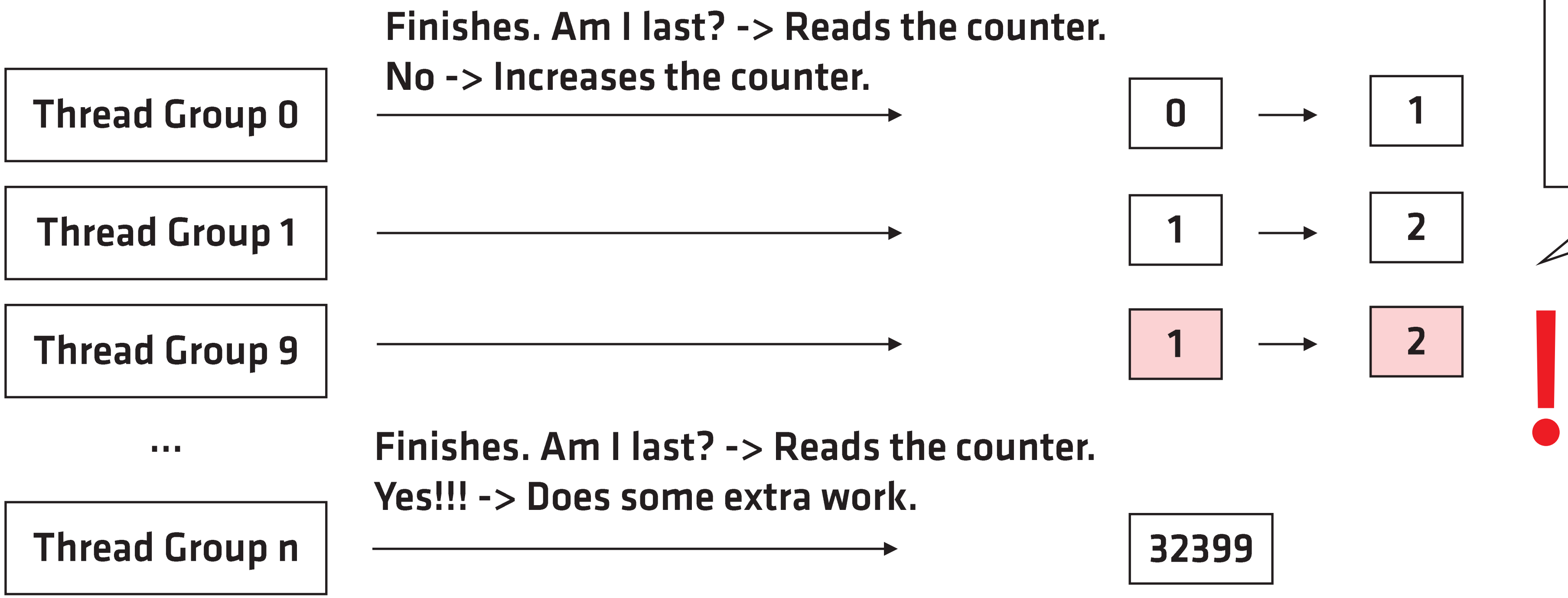


THE RDNA™2 COMPUTE UNIT



COMMUNICATION WITHIN A DISPATCH

- Communication between all thread groups, e.g., via a global atomic counter.
  - E.g., in case you want to figure out the last active thread group.
  - Each thread group increases the counter by completion.
  - The last active thread group will know it's last by reading the counter.



Finishes. Am I last? -> Reads the counter.
No -> Increases the counter.

Thread Group 0 → 0 → 1

Thread Group 1 → 1 → 2

Thread Group 9 → 1 → 2 !

…

Finishes. Am I last? -> Reads the counter.
Yes!!! -> Does some extra work.

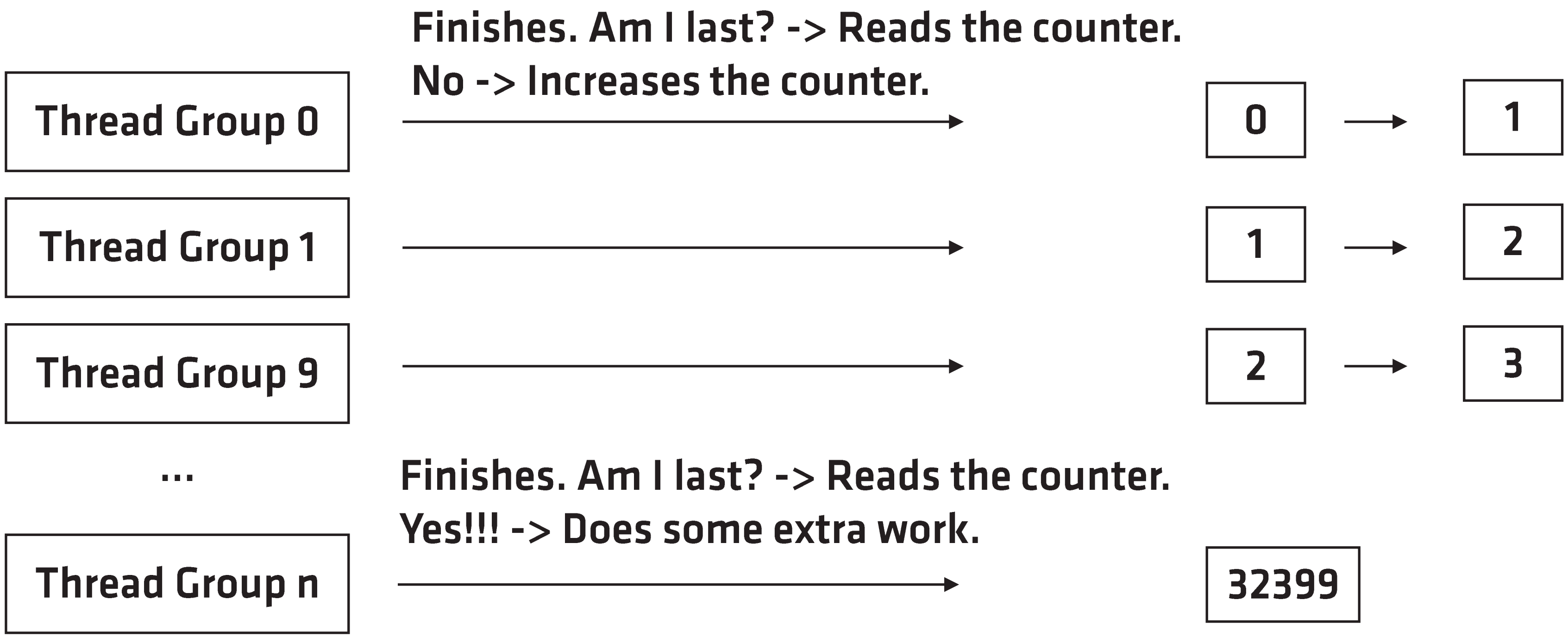
Thread Group n → 32399

The counter needs to be visible to all thread groups.

Otherwise …

COMMUNICATION WITHIN A DISPATCH

- Communication between all thread groups, e.g., via a global atomic counter.
  - E.g., in case you want to figure out the last active thread group.
  - Each thread group increases the counter by completion.
  - The last active thread group will know it’s last by reading the counter.



Finishes. Am I last? -> Reads the counter.
No -> Increases the counter.

Finishes. Am I last? -> Reads the counter.
Yes!!! -> Does some extra work.

The counter needs to be visible to all thread groups.
→
Needs to be at least in the **L2 Cache** – first cache that is global.

Needs to be marked as **globallycoherent** in the shader: bypasses L0 and L1.

✓

COMMUNICATION WITHIN A THREAD GROUP

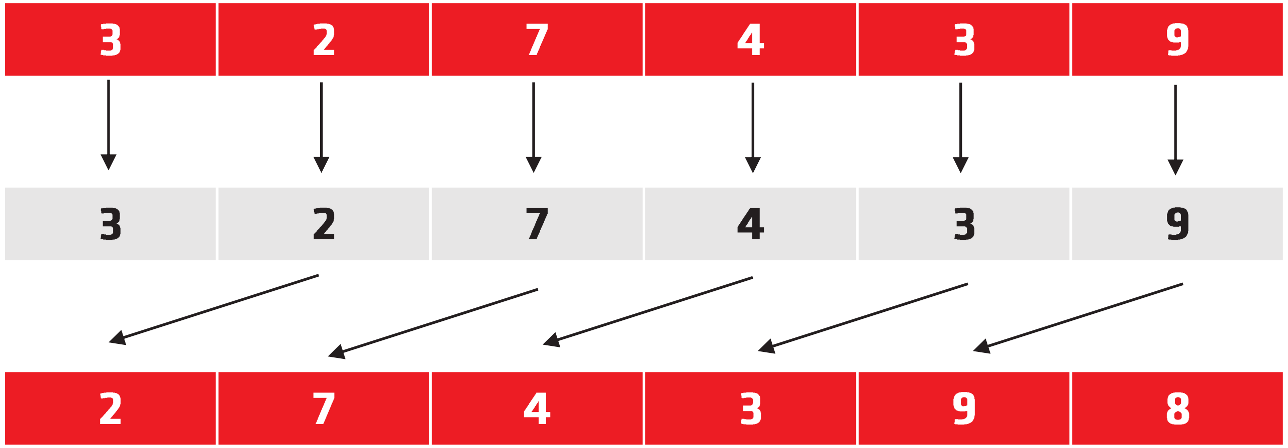
- Communication between threads of a **single thread group**.
- All threads are on the same Dual Compute Unit.
- We can share data between threads within a thread group using **groupshared memory**.
- E.g., useful if multiple threads have to access the same data.

```
float a = Compute(threadIndex);


some_lds[threadIndex] = a;

GroupMemoryBarrierWithGroupSync();

float b = some_lds[(threadIndex + 1) % 64];
```

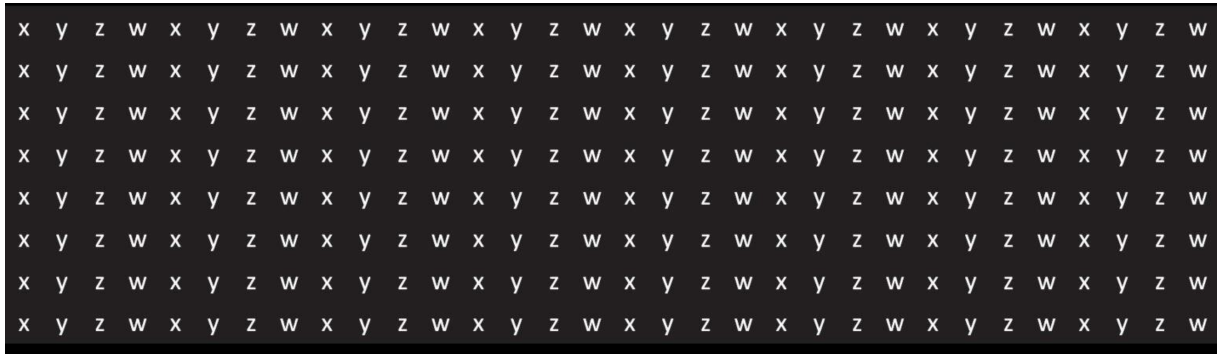


3	2	7	4	3	9
3	2	7	4	3	9
2	7	4	3	9	8

GROUP SHARED MEMORY – LOCAL DATA SHARE (LDS)

- LDS is banked on RDNA™2 and GCN.
- Bank conflicts increase latency of instructions – try to avoid them!

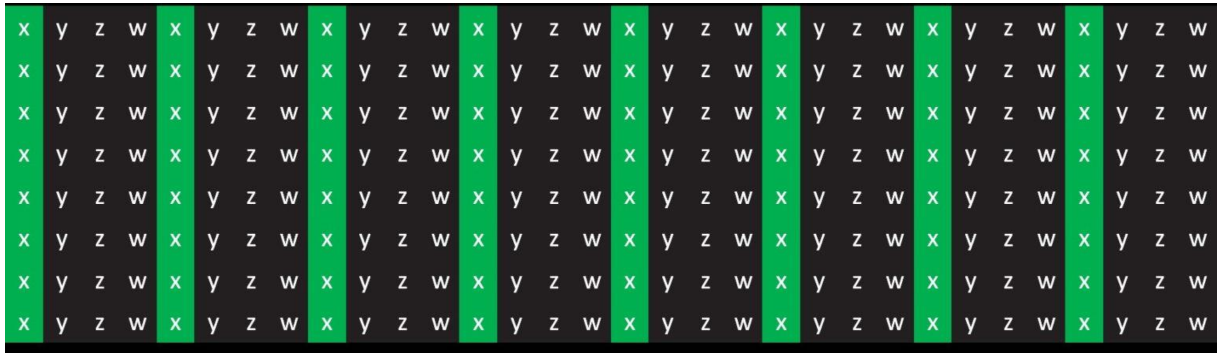
Float4 array:



x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w

Reading X

→ 8 bank conflicts.

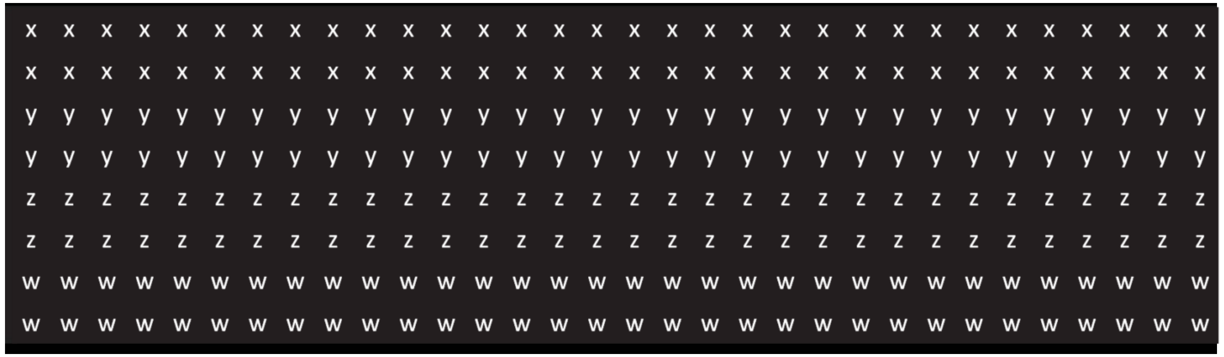


x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y z w

GROUP SHARED MEMORY – LOCAL DATA SHARE (LDS)

- LDS is banked on RDNA™2 and GCN.
- Bank conflicts increase latency of instructions – try to avoid them!

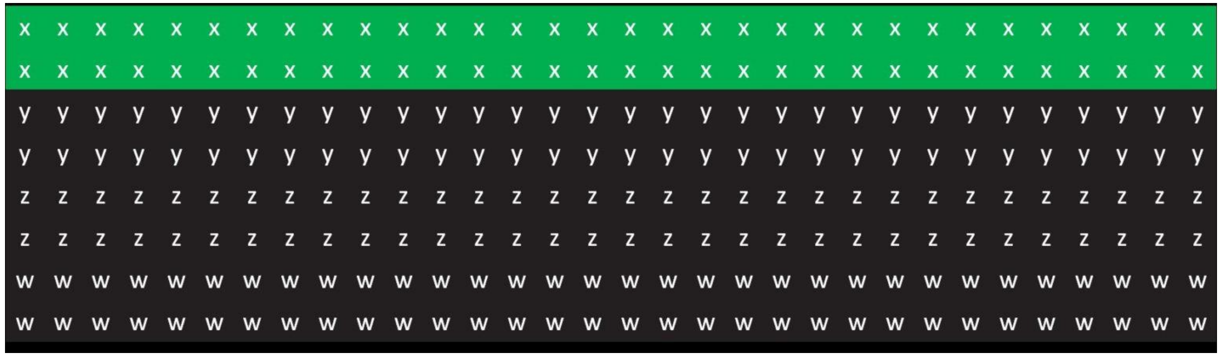
Array of floats:



x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Reading X

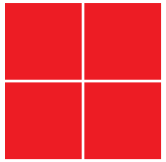
→ 2 bank conflicts.



x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

COMMUNICATION WITHIN A WAVEFRONT

- Communication between threads of a **single wavefront.**
- All threads are on the same SIMD.
- A thread can access the data of another thread within the same wavefront using wave operations, e.g.,
  - `QuadReadAcrossX`
  - `QuadReadAcrossY`



MEMORY & CACHES

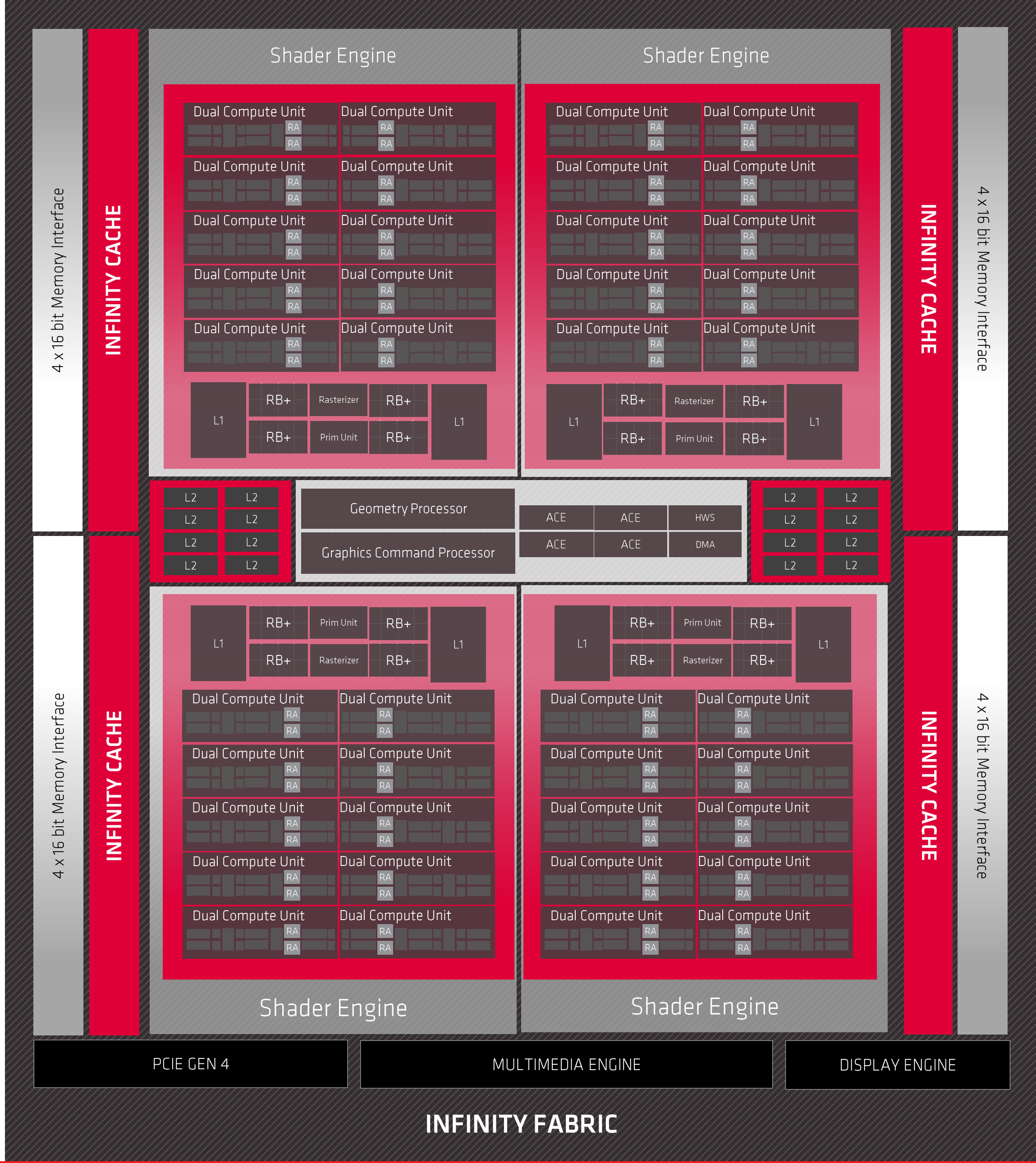
Global Memory -> Local Device Memory.

But how to access data in the first place?

Caches.

- Infinity Cache (Global).
- L2 Cache (Global).
- L1 Cache (Shader Array → 5 Dual CUs).
- L0 Cache (CU).

Groupshared Memory, aka Local Data Share (LDS) (Dual CU).



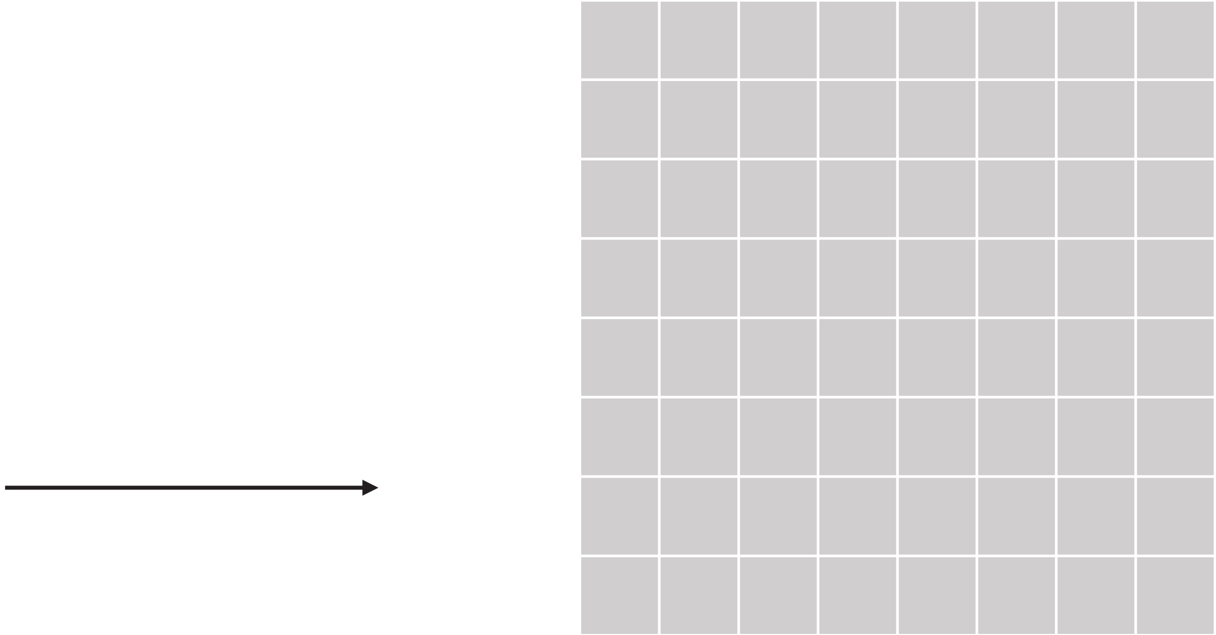
TEXTURE ACCESS – READING & WRITING

- Contiguous reads and writes are faster than scattered.
- More likely that the requested data lies within a cache line.

```
[numthreads(64,1,1)]
...
float4 color = inputTexture[threadID];
```

Prefer a quad pattern over a linear pattern:

```
[numthreads(8,8,1)]
...
float4 color = inputTexture[threadID];
```



TEXTURE ACCESS – READING & WRITING

- Contiguous reads and writes are faster than scattered.
- More likely that the requested data lies within a cache line.

- Writes to UAVs: Not just contiguous writes, but preferably contiguous writes of whole 256Byte blocks per wave.

→ Can help maximizing bandwidth in compute shaders.

- Rule of thumb:
  - 8x8 thread group writes 8x8 block of pixels.
  - Write to all channels if possible!

TEXTURE ACCESS – READING & WRITING

- For a output texture that has 4 channels, but you only compute the output for 3 of them:

```
output[threadID].xyz = color.xyz;
```

- However, just writing 3 channels and not all 4 channels means, that it's **not** a whole block of 256Bytes.

- In fact, it can be more efficient to load the 4th channel:
  - `color.w = output[threadID].w`
  - `output[threadID].xyzw = color.xyzw`

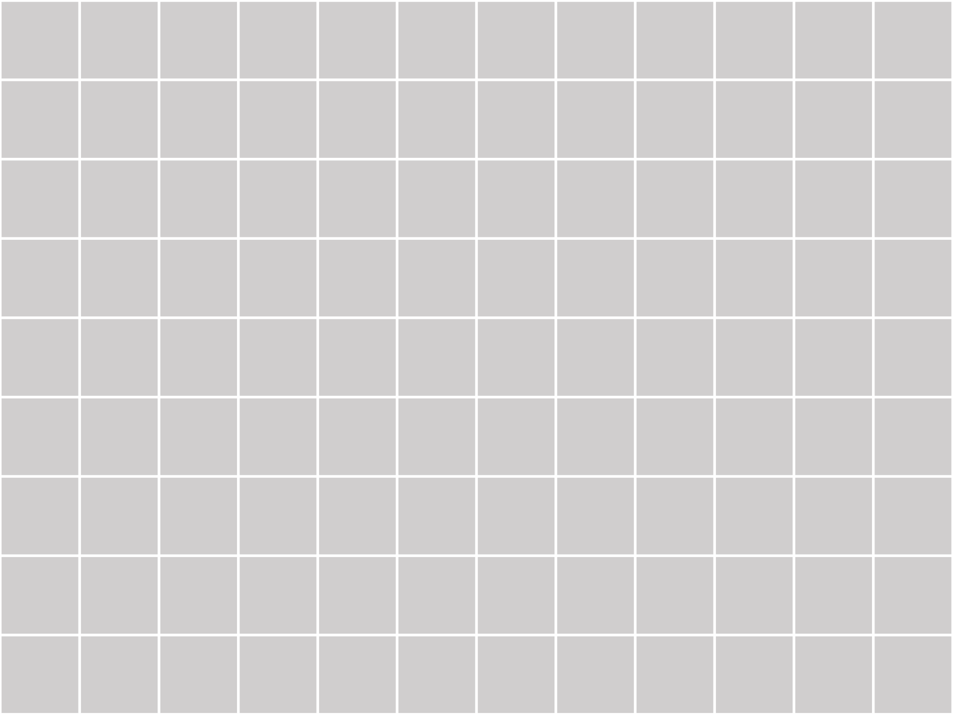
- If the 4th channel is unused, you could write a 'dummy' data:
  - `output[threadID].xyzw = float4(color.xyz, 1.0f);`

TEXTURE ACCESS – READING & WRITING

Or, if we go back to one of our previous examples: What to do when 1 thread writes out 4 output values?

A thread group size of 8x8 writing out 16x16 texels.

```
int2 index = threadID * 2;
output[index + int2(0,0)].xyzw = color0.xyzw;
output[index + int2(1,0)].xyzw = color1.xyzw;
output[index + int2(0,1)].xyzw = color2.xyzw;
output[index + int2(1,1)].xyzw = color3.xyzw;
```

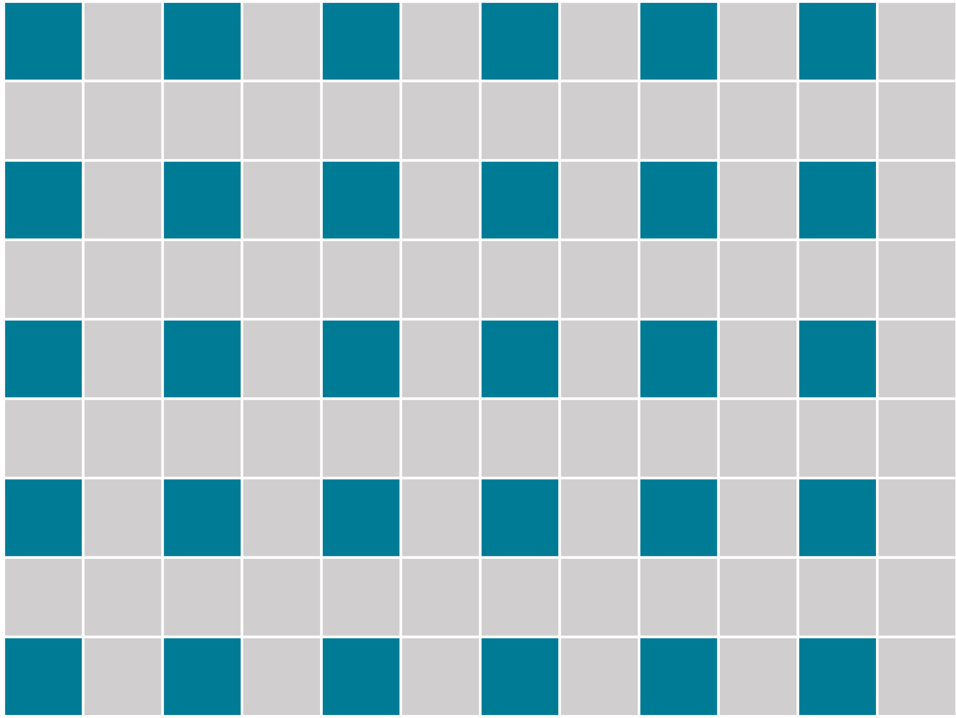


TEXTURE ACCESS – READING & WRITING

Or, if we go back to one of our previous examples: What to do when 1 thread writes out 4 output values?

A thread group size of 8x8 writing out 16x16 texels.

```
int2 index = threadID * 2;
output[index + int2(0,0)].xyzw = color0.xyzw;
output[index + int2(1,0)].xyzw = color1.xyzw;
output[index + int2(0,1)].xyzw = color2.xyzw;
output[index + int2(1,1)].xyzw = color3.xyzw;
```

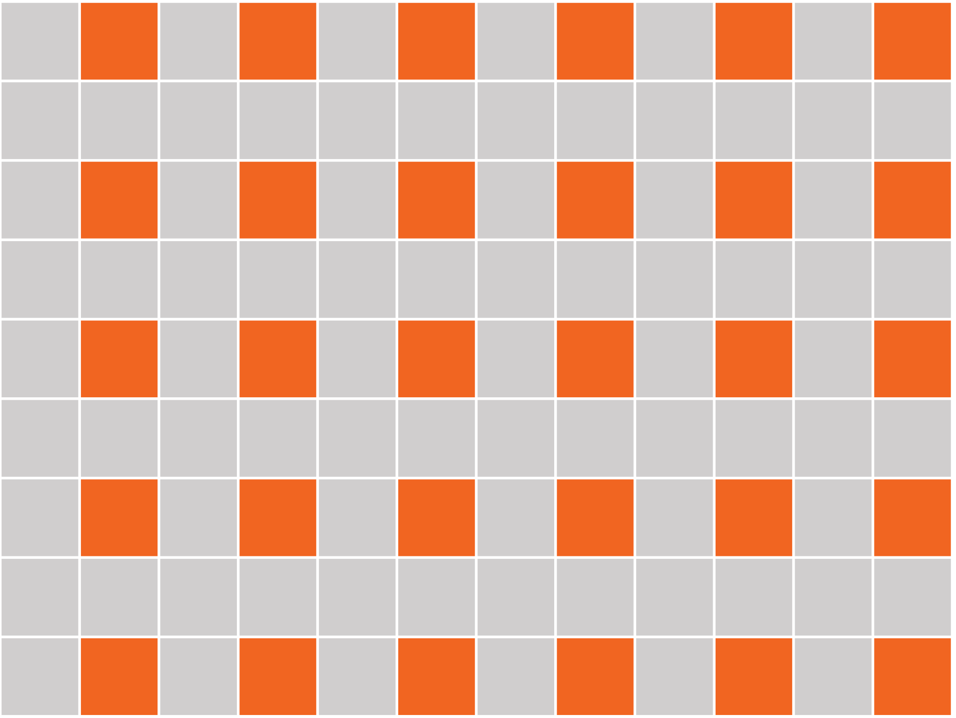


TEXTURE ACCESS – READING & WRITING

Or, if we go back to one of our previous examples: What to do when 1 thread writes out 4 output values?

A thread group size of 8x8 writing out 16x16 texels.

```
int2 index = threadID * 2;
output[index + int2(0,0)].xyzw = color0.xyzw;
output[index + int2(1,0)].xyzw = color1.xyzw;
output[index + int2(0,1)].xyzw = color2.xyzw;
output[index + int2(1,1)].xyzw = color3.xyzw;
```

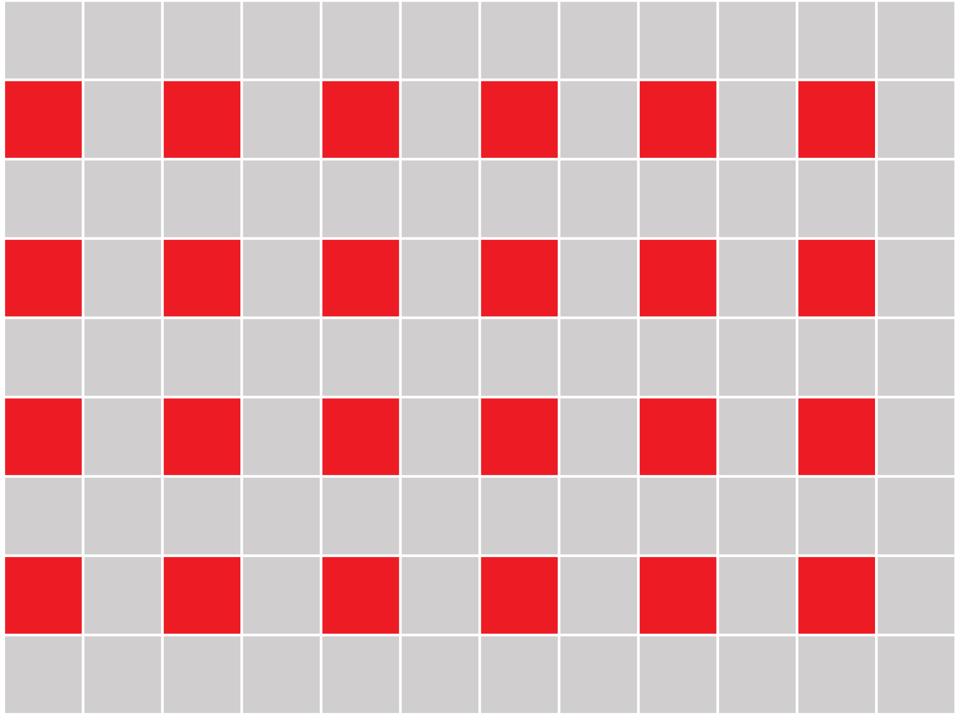


TEXTURE ACCESS – READING & WRITING

Or, if we go back to one of our previous examples: What to do when 1 thread writes out 4 output values?

A thread group size of 8x8 writing out 16x16 texels.

```
int2 index = threadID * 2;
output[index + int2(0,0)].xyzw = color0.xyzw;
output[index + int2(1,0)].xyzw = color1.xyzw;
output[index + int2(0,1)].xyzw = color2.xyzw;
output[index + int2(1,1)].xyzw = color3.xyzw;
```

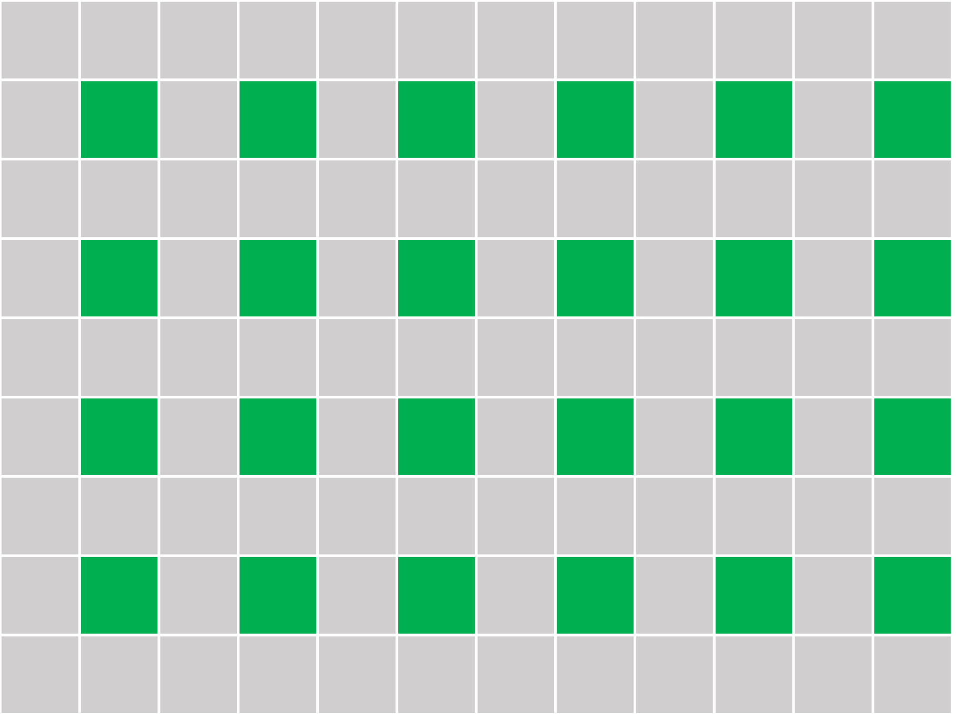


TEXTURE ACCESS – READING & WRITING

Or, if we go back to one of our previous examples: What to do when 1 thread writes out 4 output values?

A thread group size of 8x8 writing out 16x16 texels.

```
int2 index = threadID * 2;
output[index + int2(0,0)].xyzw = color0.xyzw;
output[index + int2(1,0)].xyzw = color1.xyzw;
output[index + int2(0,1)].xyzw = color2.xyzw;
output[index + int2(1,1)].xyzw = color3.xyzw;
```

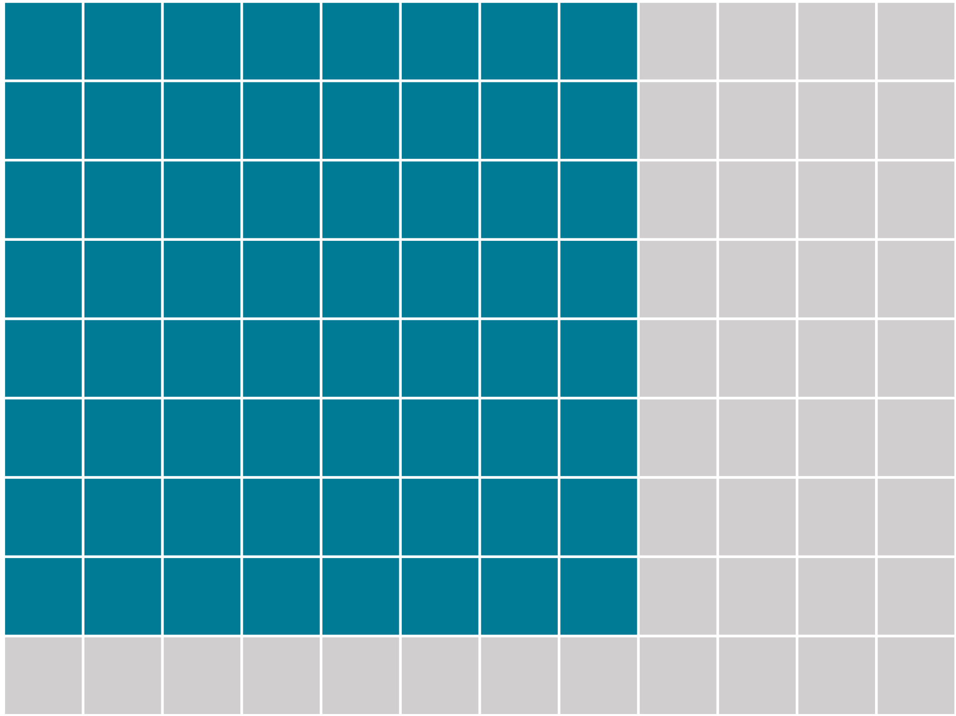


TEXTURE ACCESS – READING & WRITING

Or, if we go back to one of our previous examples: What to do when 1 thread writes out 4 output values?

A thread group size of 8x8 writing out 16x16 texels.

```
output[threadID + int2(0,0)].xyzw = color0.xyzw;
output[threadID + int2(8,0)].xyzw = color1.xyzw;
output[threadID + int2(0,8)].xyzw = color2.xyzw;
output[threadID + int2(8,8)].xyzw = color3.xyzw;
```

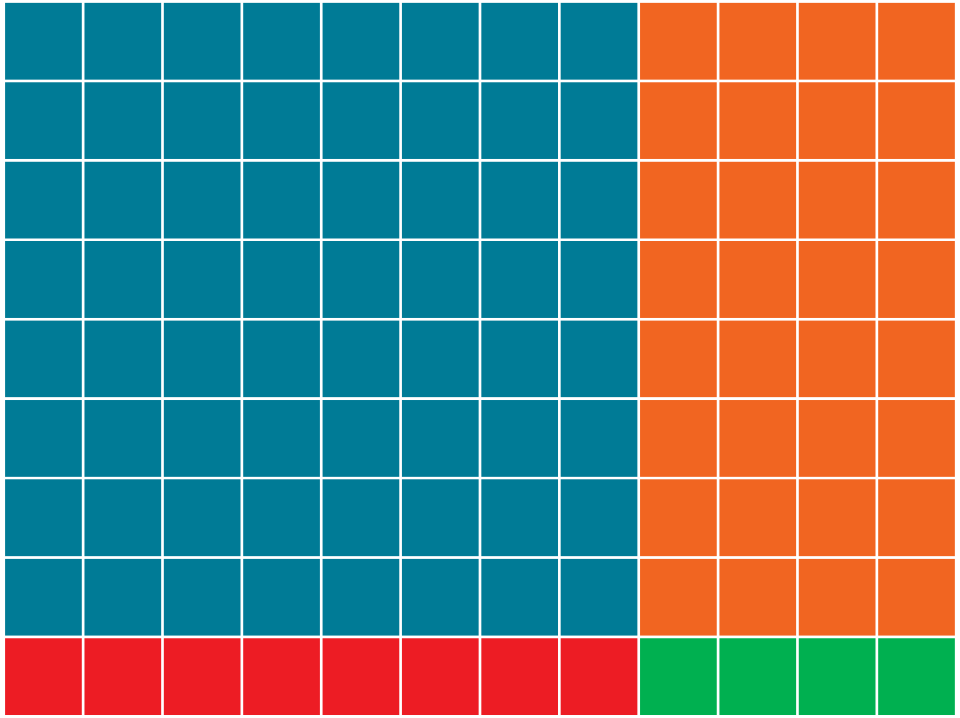


TEXTURE ACCESS – READING & WRITING

Or, if we go back to one of our previous examples: What to do when 1 thread writes out 4 output values?

A thread group size of 8x8 writing out 16x16 texels.

```
output[threadID + int2(0,0)].xyzw = color0.xyzw;
output[threadID + int2(8,0)].xyzw = color1.xyzw;
output[threadID + int2(0,8)].xyzw = color2.xyzw;
output[threadID + int2(8,8)].xyzw = color3.xyzw;
```



EXECUTION MODEL

EXECUTION OF A WAVEFRONT

```
…
block_a:
s_mov_b64       s[0:1], exec
; condition test, writes results to s[2:3]
s_mov_b64       s[2:3], condition
s_mov_b64       exec, s[2:3]
s_branch_execz block_c
block_b:
; ‘if‘ part: computeDetail();
s_not_b64       exec, exec
s_branch_execz block_d
block_c:
; ‘else‘ part: computeBasic();
block_d:
s_mov_b64       exec, s[0:1]
;code afterwards
…
```

```
groupshared float data[64];

[numthreads(8,8,1)]
void main(uint index : SV_GroupIndex)
{
  …
  if (condition)
    computeDetail();
  else
    computeBasic();
  …
}
```

EXECUTION OF A WAVEFRONT

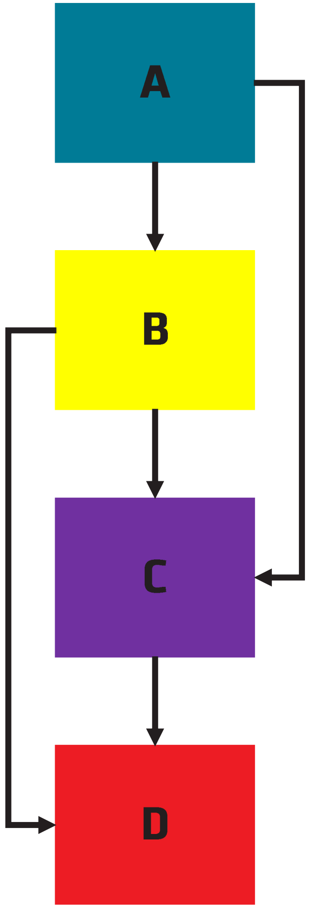
```
…
block_a:
s_mov_b64       s[0:1], exec
; condition test, writes results to s[2:3]
s_mov_b64       s[2:3], condition
s_mov_b64       exec, s[2:3]
s_branch_execz block_c
block_b:
; ‘if‘ part: computeDetail();
s_not_b64       exec, exec
s_branch_execz block_d
block_c:
; ‘else‘ part: computeBasic();
block_d:
s_mov_b64       exec, s[0:1]
;code afterwards
…
```

Save exec

exec = result of test per thread

Invert exec

Restore exec



EXECUTION OF A WAVEFRONT

…

`block_a:`

⬛⬛⬛⬛⬛⬛⬛⬛⬛⬛⬛⬛⬛ …

⬛⬛🟥⬛⬛🟥🟥🟥⬛⬛⬛🟥⬛ …

`block_b:`

🟥🟥⬛🟥🟥⬛⬛⬛🟥🟥🟥⬛🟥 …

`block_c:`

`block_d:`

⬛⬛⬛⬛⬛⬛⬛⬛⬛⬛⬛⬛⬛ …

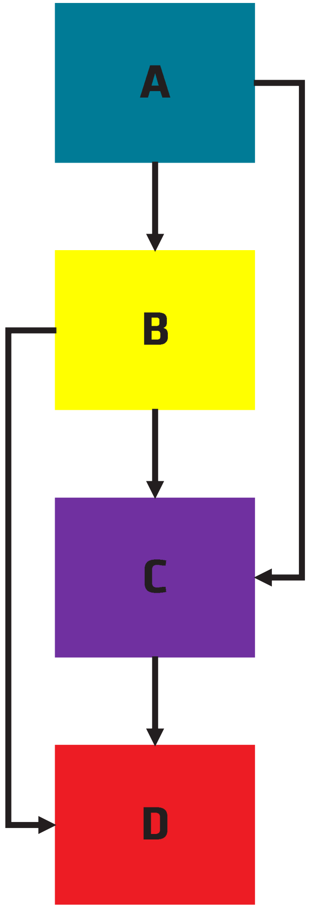
…

Save exec

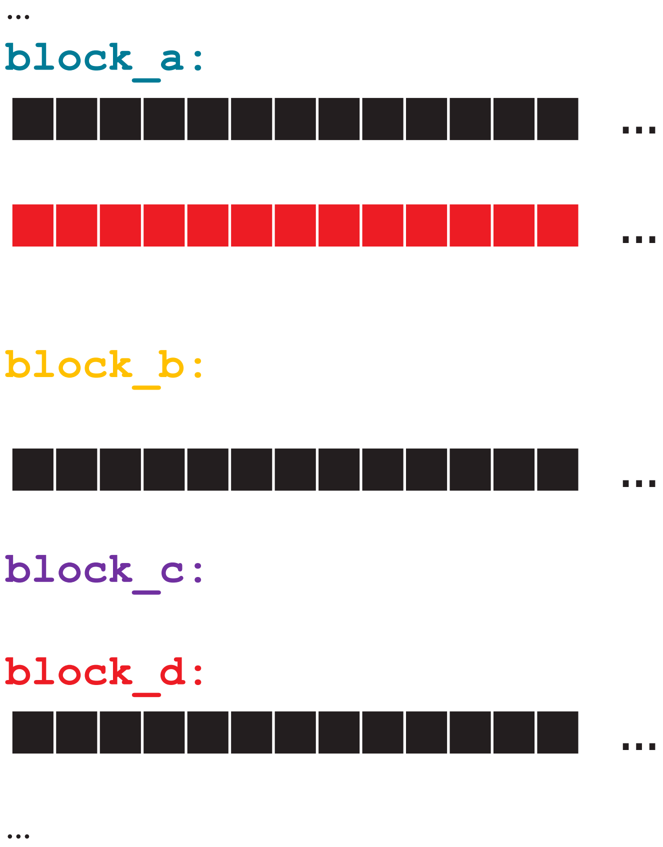
exec = result of test per thread

Invert exec

Restore exec



EXECUTION OF A WAVEFRONT



...

block_a:

...

...

block_b:

...

block_c:

block_d:

...

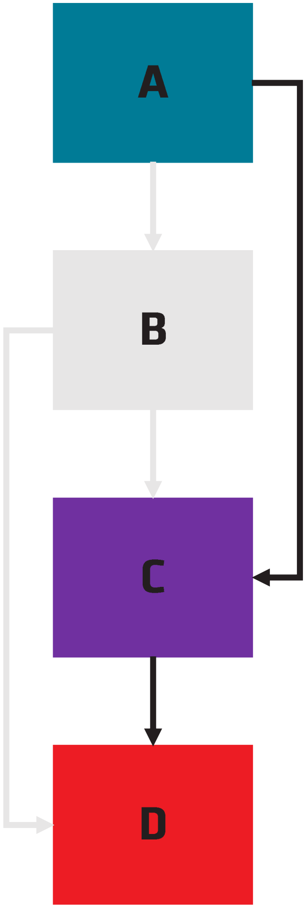
...

Save exec

exec = result of test per thread

Invert exec

Restore exec



EXECUTION OF A WAVEFRONT – TEXTURE ACCESS

The address to a texture is stored in scalar registers:

```
image_load v[5:8], [v22, v23, v5], s[12:19] dmask:0xf dim:SQ_RSRC_IMG_2D_ARRAY
```

Address to your texture.
Stored in scalar register s[12:19].

Your UV coordinates.
Stored in vector register v22, v23, v5.
Unique per thread.

The loaded values will be stored in v[5:8] (4 channels).
Unique per thread.

```
RWTexture2D imgSrc :register(u0);

[numthreads(8,8,1)]
void main(uint3 index : SV_GroupThreadID)
{
  …
   float4 value = imgSrc.Load(index.xy);
  …
}
```

EXECUTION OF A WAVEFRONT – TEXTURE ACCESS

- This becomes interesting, when there is an array of textures.
- Well … Not really if the index is constant.

image_load v[2:5], v[0:1], s[4:11] dmask:0xf dim:SQ_RSRC_IMG_2D

s[4:11] points to imgSrc[0].

```
RWTexture2D imgSrc []:register(u0);

[numthreads(8,8,1)]
void main(uint3 index : SV_GroupThreadID)
{
  …
   float4 value = imgSrc[0].Load(index.xy);
  …
}
```

EXECUTION OF A WAVEFRONT – TEXTURE ACCESS

- This becomes interesting, when there is an array of textures.
- What if the index is non-uniform though?
- Potentially, each thread could access a different texture within the array.

```
RWTexture2D imgSrc []:register(u0);

[numthreads(8,8,1)]
void main(uint3 index : SV_GroupThreadID)
{
  …
   float4 value = imgSrc[NonUniformResourceIndex(imgIndex)].Load(index.xy);

  …
}
```

By default, the compiler assumes the index is uniform.
We need to explicitly tell it is not.

EXECUTION OF A WAVEFRONT – TEXTURE ACCESS

```
RWTexture2D imgSrc []:register(u0);

[numthreads(8,8,1)]
void main(uint3 index : SV_GroupThreadID)
{
  …
   float4 value = imgSrc[NonUniformResourceIndex(imgIndex)].Load(index.xy);

  …
}
```

The address of the texture is still stored in scalar registers!

```
image_load v[3:6], v[0:1], s[8:15] dmask:0xf dim:SQ_RSRC_IMG_2D
```

This is the same for each thread ...

EXECUTION OF A WAVEFRONT – TEXTURE ACCESS

```
RWTexture2D imgSrc []:register(u0);

[numthreads(8,8,1)]
void main(uint3 index : SV_GroupThreadID)
{
  …
   float4 value = imgSrc[NonUniformResourceIndex(imgIndex)].Load(index.xy);

  …
}
```

The address of the texture is still stored in scalar registers!

```
image_load v[3:6], v[0:1], s[8:15] dmask:0xf dim:SQ_RSRC_IMG_2D
```

So for each unique texture, we have to have a separate call.

WATERFALL

`float4 value = imgSrc[`**`NonUniformResourceIndex`**`(imgIndex)].Load(index.xy);`

The compiler will iterate through all threads until every thread has the correct index. This is also called ‘waterfall’.

```
...
_L2
          v_readfirstlane_b32 s4, v2

          v_cmp_eq_u32_e32  vcc_lo, s4, v2
          s_and_saveexec_b32  s5, vcc_lo

          s_cbranch_execz  _L0

BVF0_0:
...
          image_load
          s_andn2_b32   s3, s3, exec_lo

          s_cbranch_scc0 _L1
_L0:
...
          s_mov_b32     exec_lo, s5
          s_and_b32     exec_lo, exec_lo, s3
          s_branch      _L2
...
```

Pick each descriptor one by one, load them into a scalar register.

Check if we picked the right descriptor for the thread.

If not, skip image load. Jump to _L0.

Load the images for all active threads.

Exit “waterfall”.

All threads that have not executed imageLoad yet are here.
Update exec mask.

Jump back to start of waterfall.

WATERFALL

`float4 value = imgSrc[NonUniformResourceIndex(imgIndex)].Load(index.xy);`

The compiler will iterate through all threads until every thread has the correct index. This is also called ‘waterfall’.

```
...
_L2
          v_readfirstlane_b32 s4, v2

          v_cmp_eq_u32_e32  vcc_lo, s4, v2
          s_and_saveexec_b32  s5, vcc_lo

          s_cbranch_execz  _L0

BVF0_0:
...
          image_load
          s_andn2_b32   s3, s3, exec_lo

          s_cbranch_scc0 _L1
_L0:
...
          s_mov_b32     exec_lo, s5
          s_and_b32     exec_lo, exec_lo, s3
          s_branch      _L2
...
```

Pick each descriptor one by one, load them into a scalar register.

Check if we picked the right descriptor for the thread.

If not, skip image load. Jump to _L0.

Load the images for all active threads.

Exit “waterfall”.

All threads that have not executed imageLoad yet are here.
Update exec mask.

Jump back to start of waterfall.

WATERFALL

`float4 value = imgSrc[NonUniformResourceIndex(imgIndex)].Load(index.xy);`

The compiler will iterate through all threads until every thread has the correct index. This is also called ‘waterfall’.

```
...
_L2
          v_readfirstlane_b32 s4, v2

          v_cmp_eq_u32_e32  vcc_lo, s4, v2
          s_and_saveexec_b32  s5, vcc_lo

          s_cbranch_execz  _L0

BVF0_0:
...
          image_load
          s_andn2_b32   s3, s3, exec_lo

          s_cbranch_scc0 _L1
_L0:
...
          s_mov_b32     exec_lo, s5
          s_and_b32     exec_lo, exec_lo, s3
          s_branch      _L2
...
```

Pick each descriptor one by one, load them into a scalar register.

Check if we picked the right descriptor for the thread.

If not, skip image load. Jump to _L0.

Load the images for all active threads.

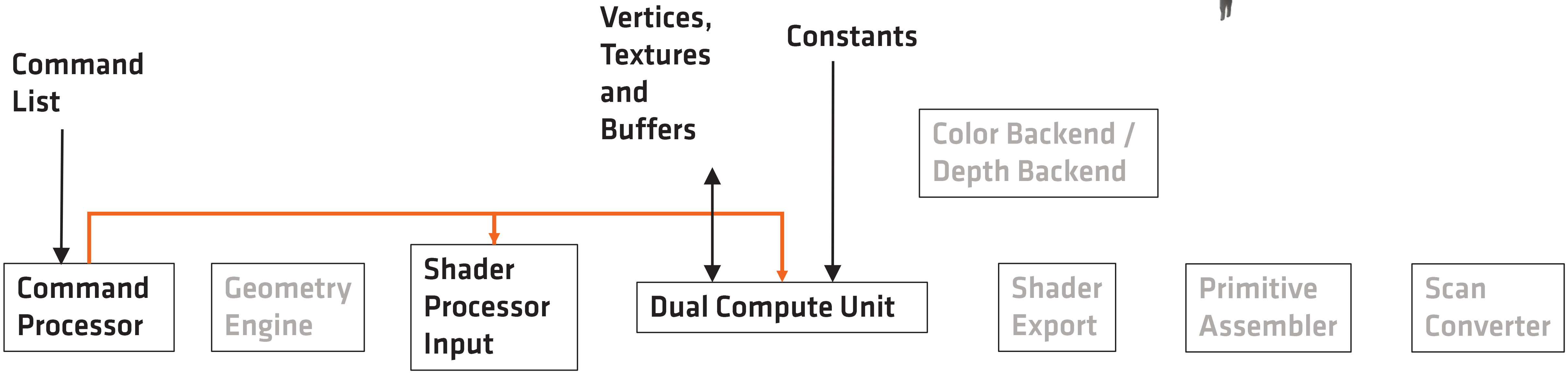
Exit “waterfall”.

All threads that have not executed imageLoad yet are here.
Update exec mask.

Jump back to start of waterfall.

EXPORT

ON THE RDNA™2 ARCHITECTURE



Command List

Vertices, Textures and Buffers

Constants

Color Backend / Depth Backend

Command Processor

Geometry Engine

Shader Processor Input

Dual Compute Unit

Shader Export

Primitive Assembler

Scan Converter

🟧 Compute Pipeline

SHADER EXPORT

Shader Export is not used for compute shaders.

- Pixel shaders can be blocked from exporting by other waves.
- Exporting takes time.
- While a shader is waiting to be able to export, it still occupies the resources.

→ Could prevent to launch new waves to already start their work.

Compute Shaders do not suffer from this.

Can be beneficial in highly varying workloads.

CONCLUSION

- Compute shaders are quite flexible.
- If the fixed function pipeline stages of the graphics pipeline are not needed, e.g., the rasterizer, considering to use compute shaders is worth a thought.
- Efficient use of caches is essential:
  - How are textures accessed?
  - How and which threads are sharing data?
- Keeping divergence low during shader execution is important:
  - Can we scalarize certain parts of the shader?
- Efficient write pattern.
  - Scattered write patterns can hurt performance a lot 😥.
  - Make sure to write out in **256Byte blocks per wave** to maximize bandwidth.

DISCLAIMER & ATTRIBUTES

Disclaimer:

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

©2021 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, Radeon, Ryzen and combinations thereof are trademarks of Advanced Micro Devices, Inc.
Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

AMD