OPTIMIZING FOR THE RADEON™ RDNA ARCHITECTURE

LOU KRAMER
DEVELOPER TECHNOLOGY ENGINEER, AMD

WHO AM I?

Lou Kramer

Developer Technology Engineer

at AMD since Nov. 2017

I work closely with game studios to make their games look amazing and run fast on AMD GPUs ☺

WHY THIS TALK?

On July 7th 2019, we released a new GPU architecture with our Radeon™ RX 5700 cards!

→ Radeon™ New Architecture (RDNA)

Today, we have several products based on RDNA

WHY THIS TALK?

RDNA is present in a bunch of different products

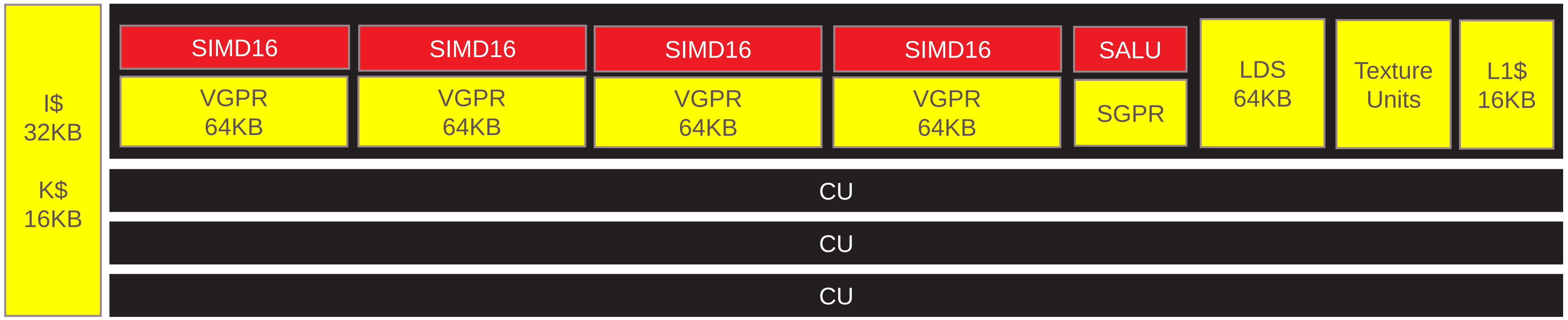
Design goals of RDNA

- Scalability
- Special focus on
    - Geometry handling
    - Cache flushes
    - Amount of work in flight needed
    - Latency

AGENDA

- Architecture
  - Compute Unit (**CU**) ⟷ Work Group Processor (**WGP**)
  - GCN ⟷ RDNA
  - Highlights of changes
- Optimizations
  - Texture access
  - Workload distribution
  - Shader optimizations

COMPUTE UNIT (CU)

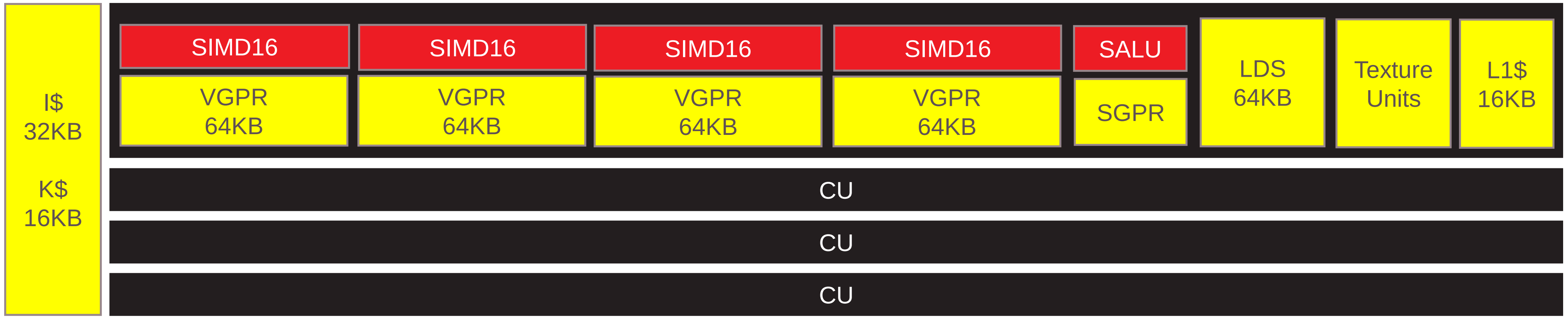


A GCN based GPU has several Compute Units - a CU has:

- 4 SIMD16 + VGPRs
- 1 Scalar ALU + SGPRs
- 1 L1 Cache
- …

This is where the shaders get executed!

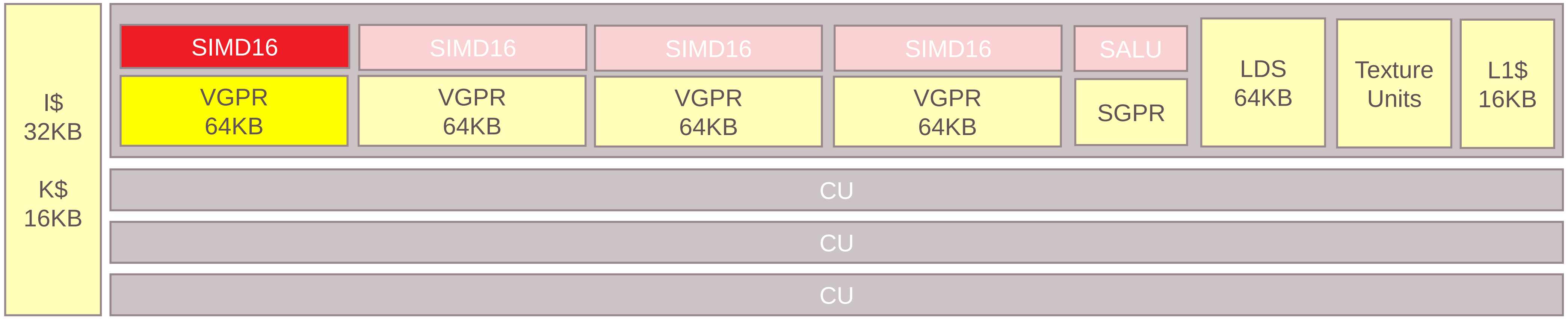
COMPUTE UNIT (CU)

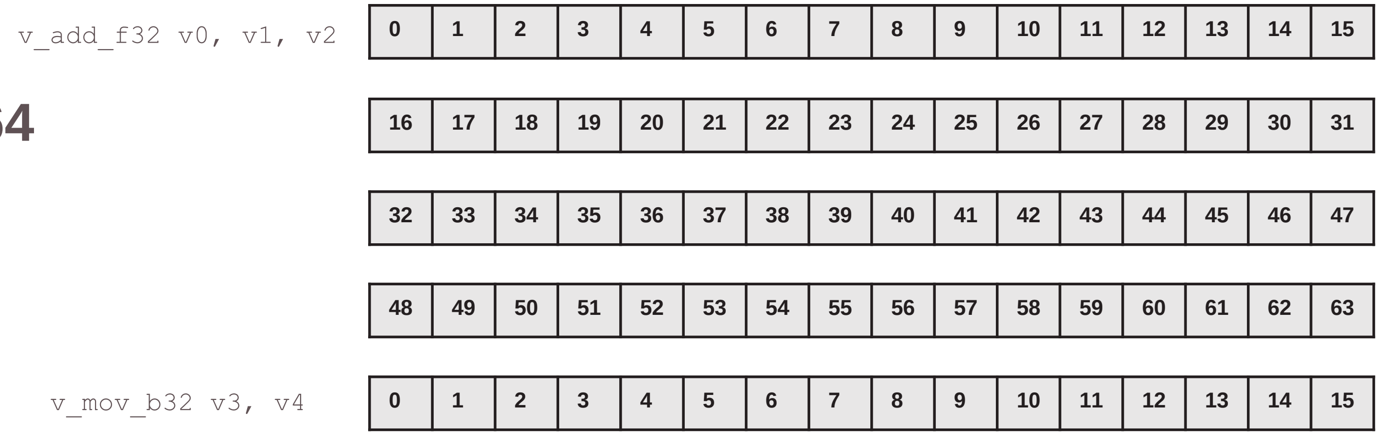


4 CUs share 1 Instruction and Constant Cache

This is where the shaders get executed!

COMPUTE UNIT (CU)



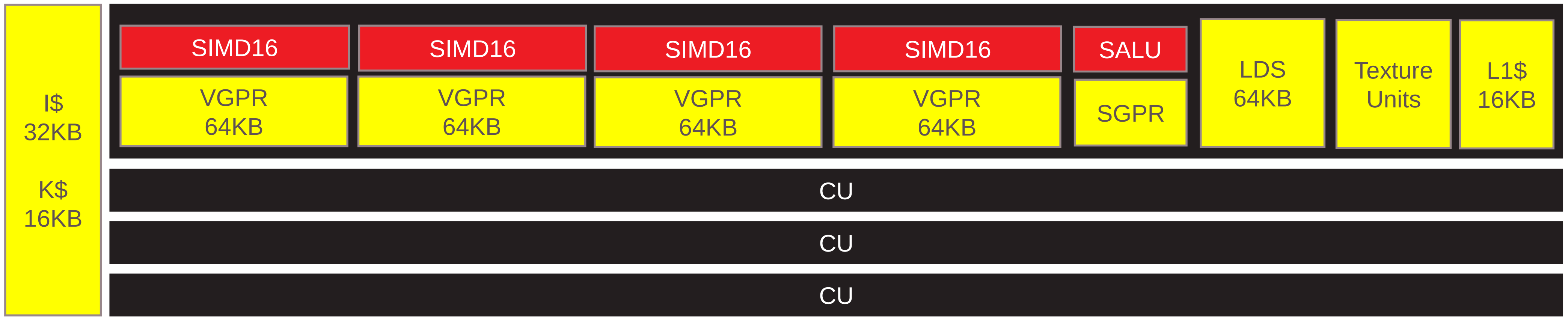


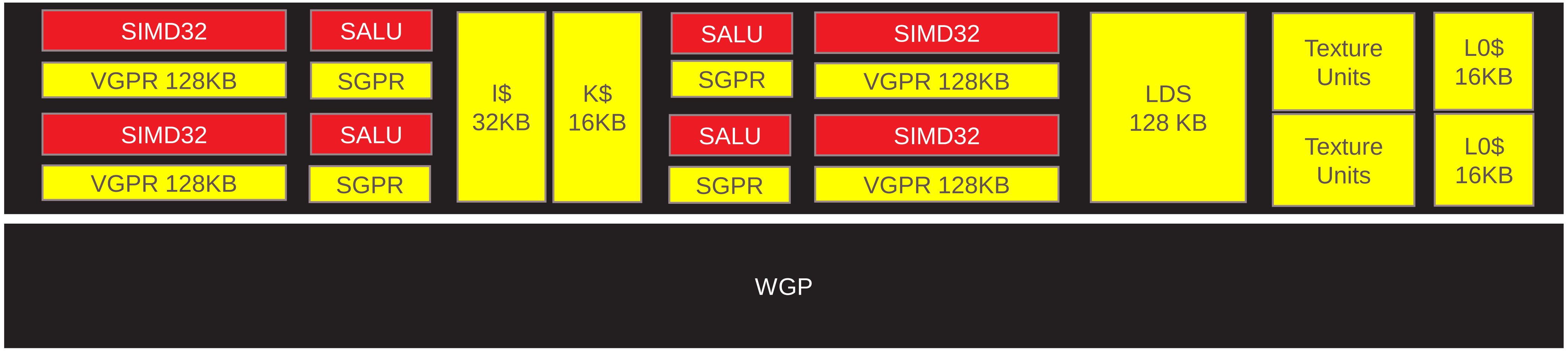
Each SIMD16 executes wavefronts of size **64**

In Lockstep -> 4 cycle instruction

4x SIMD16 = **64** threads ☺

CU ⟷ WORK GROUP PROCESSOR (WGP)



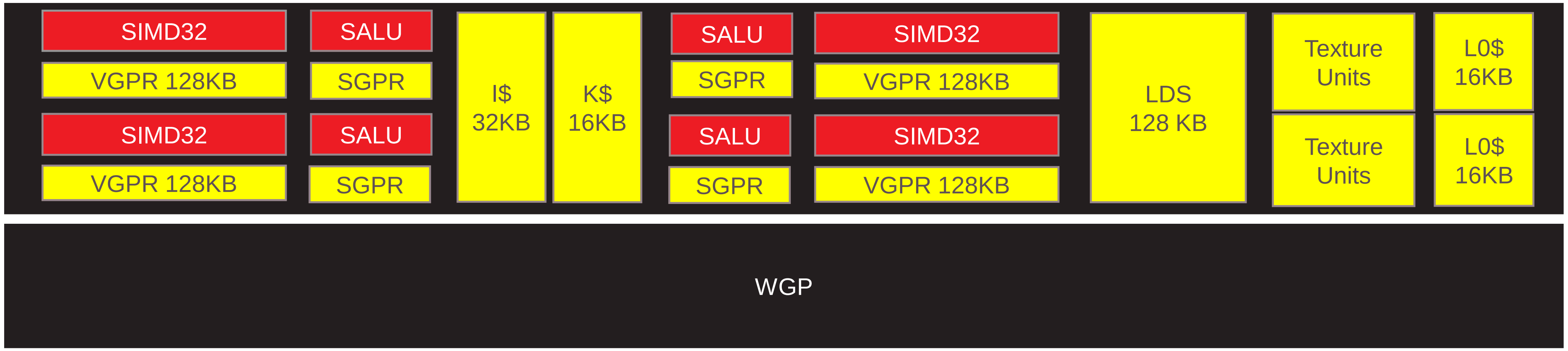


CU ⟷ WORK GROUP PROCESSOR (WGP)

A RDNA based GPU has several Work Group Processors - a WGP has:

- 4 SIMD**32** + VGPRs
- **4** Scalar ALUs + SGPRs
- **2 L0** Cache
- **1** Instruction and Constant cache
- …

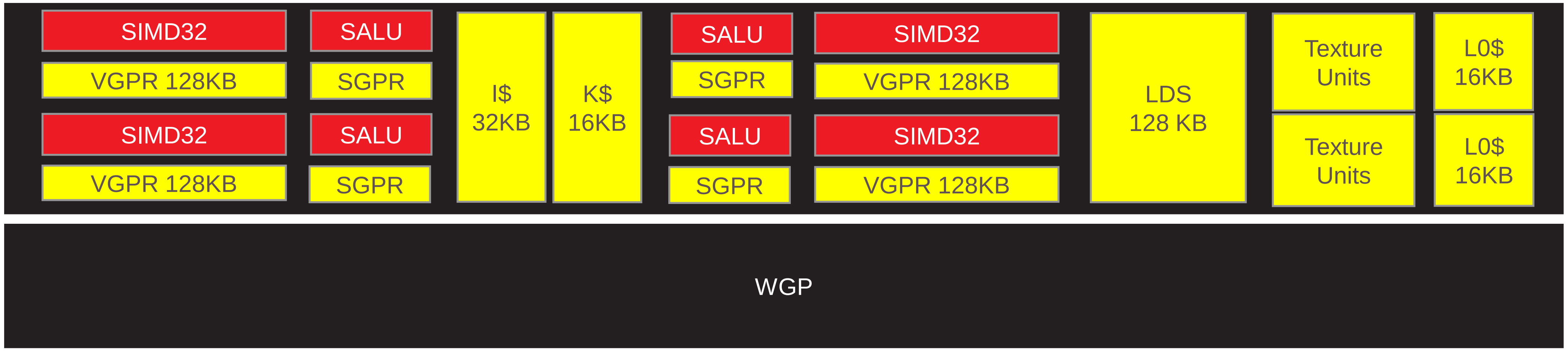
This is where the shaders get executed!



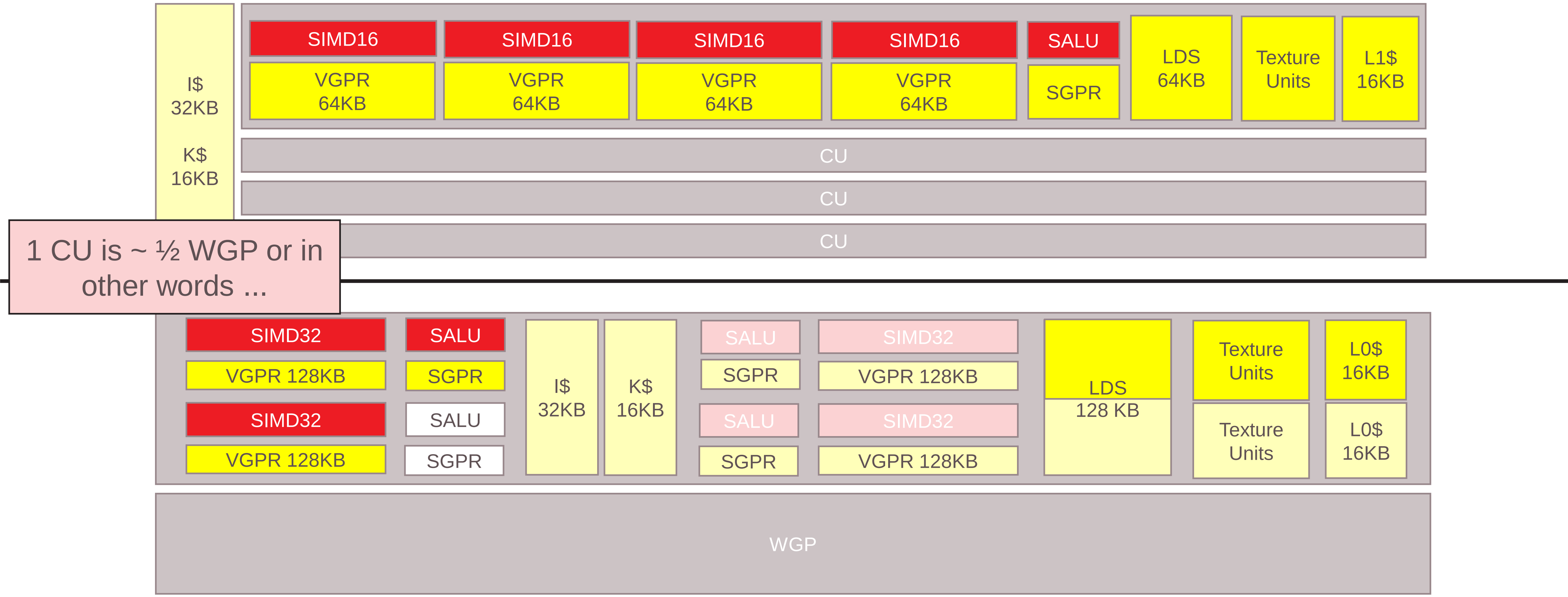
CU ⟷ WORK GROUP PROCESSOR (WGP)

5 WGPs share a L1 cache … more on this later ☺

This is where the shaders get executed!

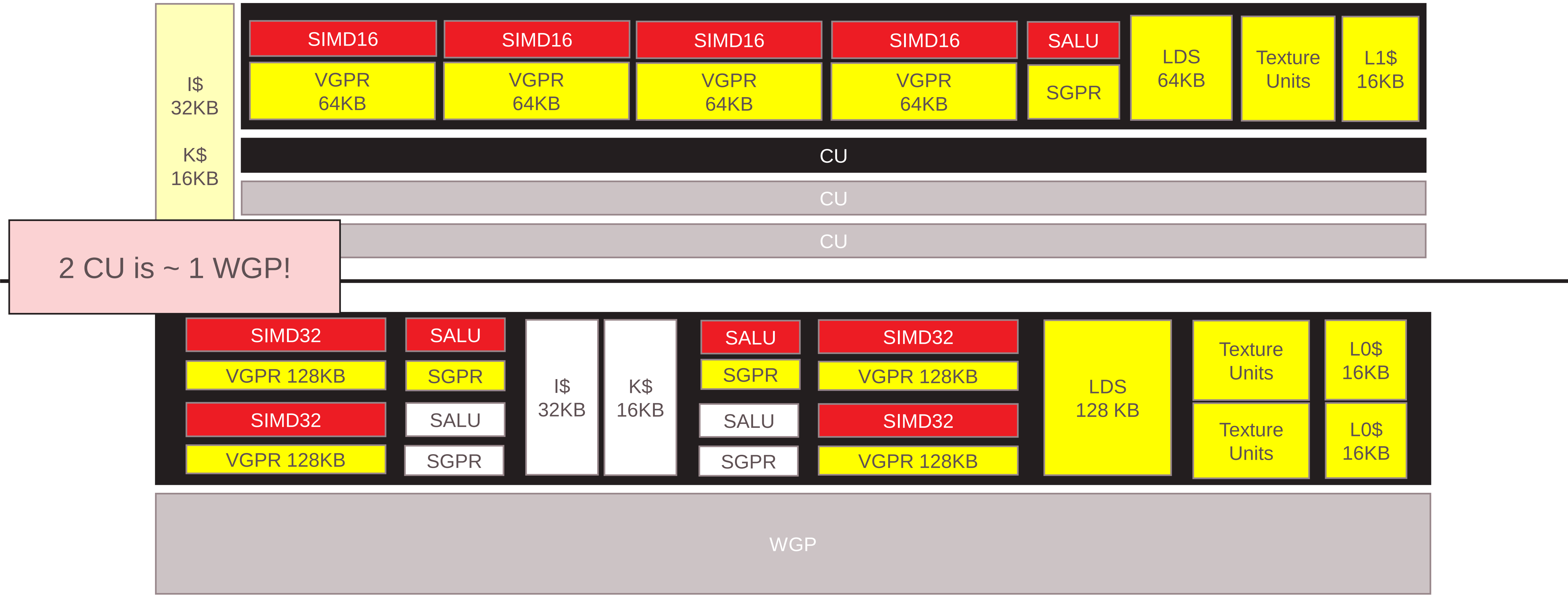


CU ⟷ WORK GROUP PROCESSOR (WGP)

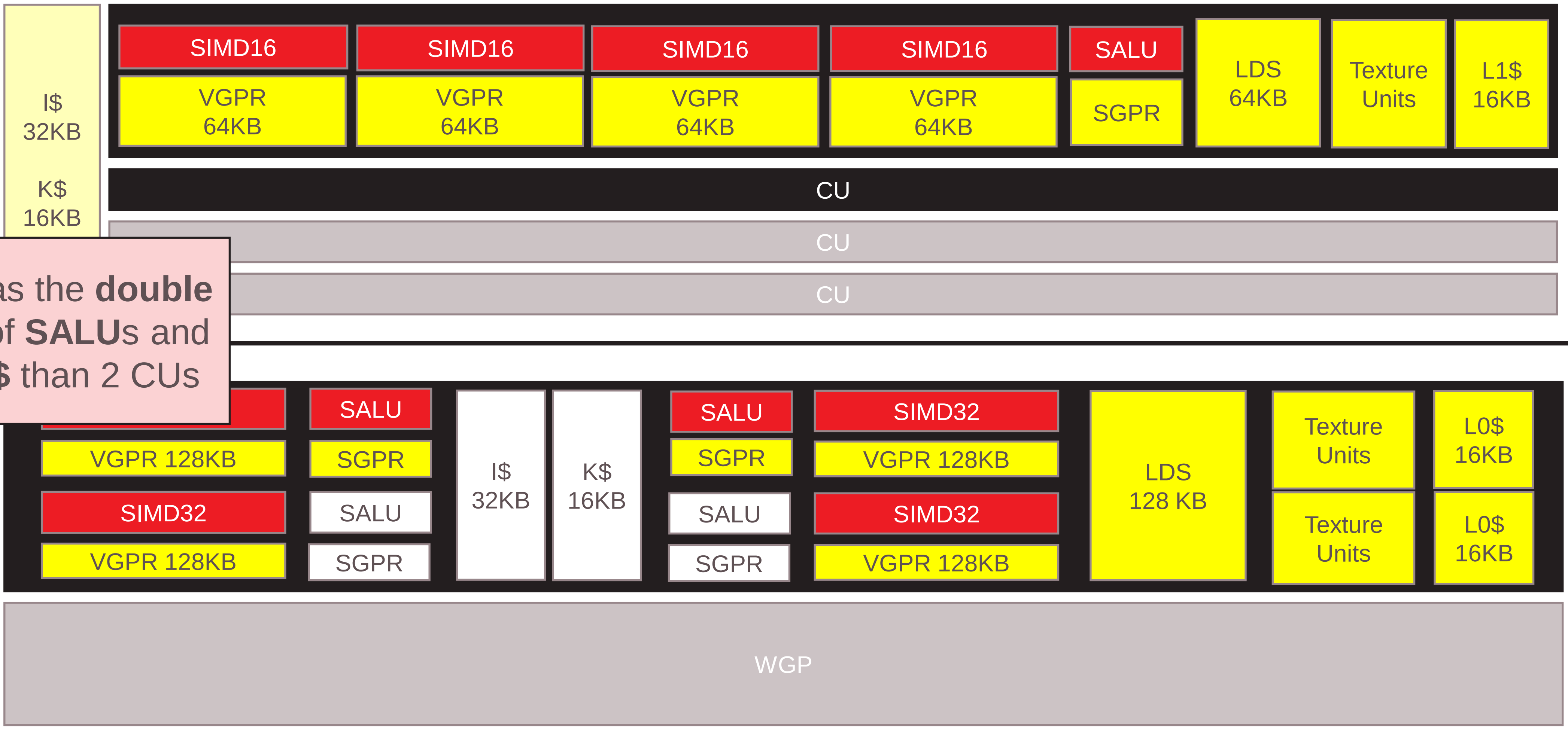


1 CU is ~ ½ WGP or in other words ...

CU ⟷ WORK GROUP PROCESSOR (WGP)



CU ⟷ WORK GROUP PROCESSOR (WGP)

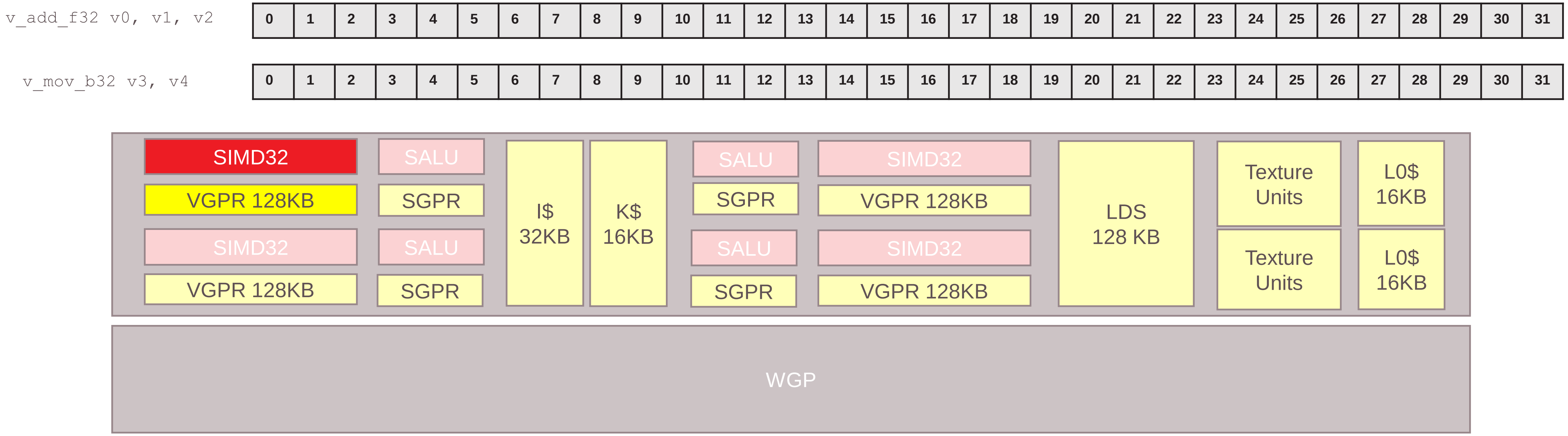


1 WGP has the **double amount** of **SALU**s and **I$** and **K$** than 2 CUs

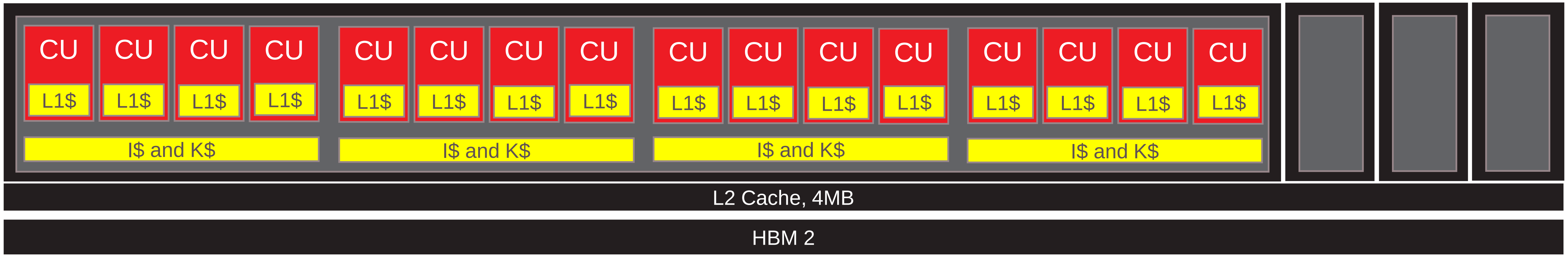
CU ⟷ WORK GROUP PROCESSOR (WGP)

Each SIMD32 executes wavefronts of size 32 **natively**

Single instruction cycle



GCN ⟷ RDNA



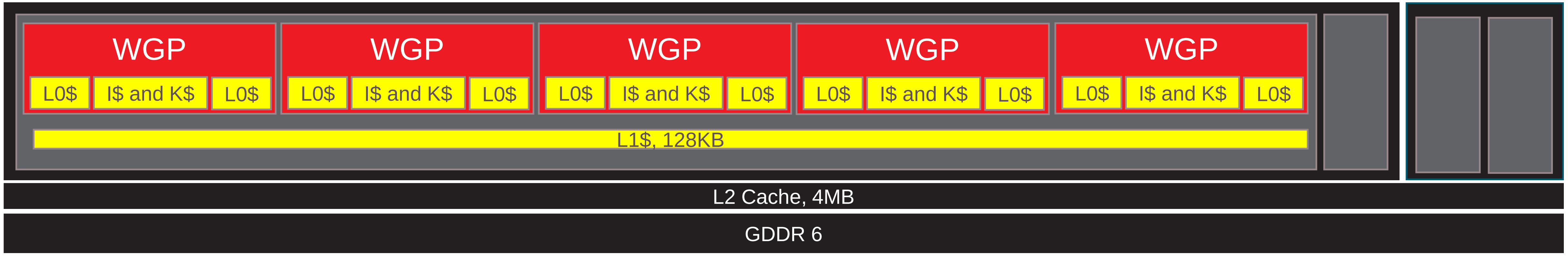
Example above: Radeon™ RX Vega 64

- 64 CUs
- Global L2 Cache
- Global Memory: HBM 2

GCN ⟷ RDNA

Example below: Radeon™ RX 5700 XT

- 20 WGPs (~40 CUs)
- 1 L1 Cache per 5 WGPs
- Global L2 Cache
- Global Memory: GDDR 6

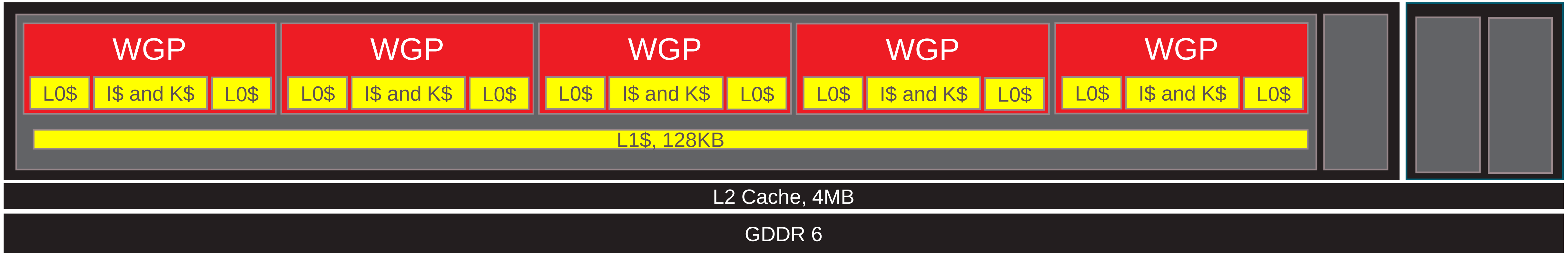


GCN ⟷ RDNA

Example below: Radeon™ RX 5700 XT

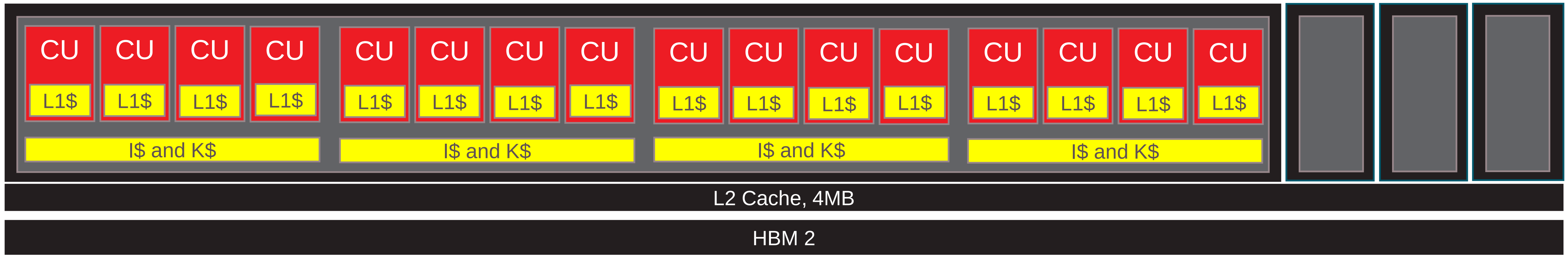
- 20 WGPs (~40 CUs)
- **1 L1 Cache per 5 WGPs**
- Global L2 Cache
- Global Memory: GDDR 6

There is no equivalent to the L1 cache on GCN. This is a **new** level of cache!

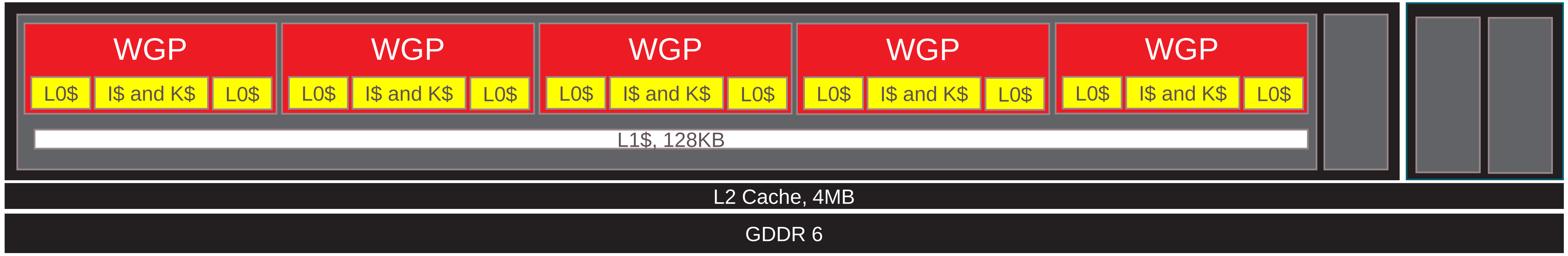


GCN ⟷ RDNA

RadeonTM RX Vega 64



RadeonTM RX 5700 XT



HIGHLIGHT OF CHANGES

RDNA	GCN
WGP	CU
L0, L1, L2, L3	L1, L2, L3
Wave32 native, Wave64 via dual issue of Wave32	Wave64 (4x SIMD16)
Single cycle instruction	Four cycle instruction
4 triangles/clock (after culling), >> 4 (before culling)	2-4 triangles/clock (culled/unculled)

HIGHLIGHT OF CHANGES

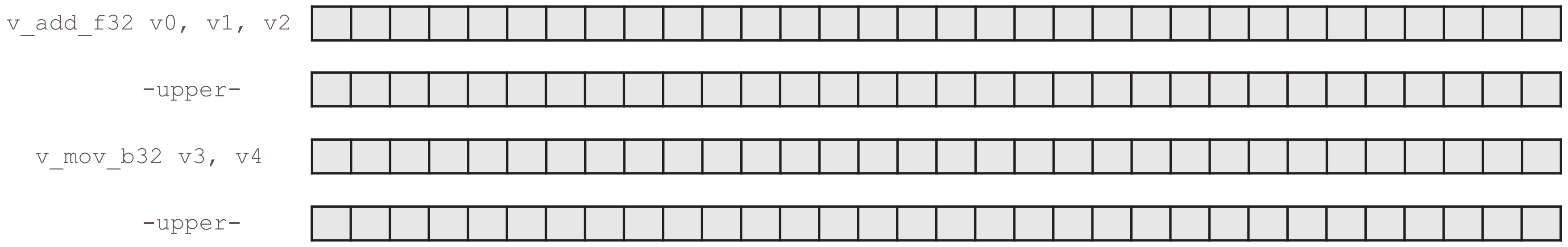
RDNA	GCN
WGP **≈ 2 CUs: double SALU & I$ & K$**	CU
L0, L1, L2, L3	L1, L2, L3
Wave32 native, Wave64 via dual issue of Wave32	Wave64 (4x SIMD16)
Single cycle instruction	Four cycle instruction
4 triangles/clock (after culling), >> 4 (before culling)	2-4 triangles/clock (culled/unculled)

HIGHLIGHT OF CHANGES

RDNA	GCN
WGP	CU
L0, **L1**, L2, L3	L1, L2, L3
Wave32 native, Wave64 via dual issue of Wave32	Wave64 (4x SIMD16)
Single cycle instruction	Four cycle instruction
4 triangles/clock (after culling), >> 4 (before culling)	2-4 triangles/clock (culled/unculled)

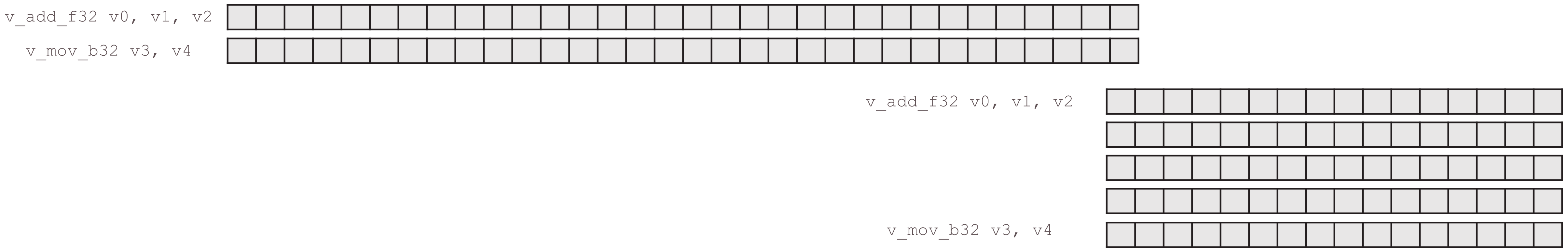
HIGHLIGHT OF CHANGES

RDNA	GCN
WGP	CU
L0, L1, L2, L3	L1, L2, L3
Wave32 native, Wave64 via dual issue of Wave32	Wave64 (4x SIMD16)
Single cycle instruction	Four cycle instruction
4 triangles/clock (after culling), >> 4 (before culling)	2-4 triangles/clock (culled/unculled)



HIGHLIGHT OF CHANGES

RDNA	GCN
WGP	CU
L0, L1, L2, L3	L1, L2, L3
Wave32 native, Wave64 via dual issue of Wave32	Wave64 (4x SIMD16)
Single cycle instruction	Four cycle instruction
4 triangles/clock (after culling), >> 4 (before culling)	2-4 triangles/clock (culled/unculled)



HIGHLIGHT OF CHANGES

RDNA	GCN
WGP	CU
L0, L1, L2, L3	L1, L2, L3
Wave32 native, Wave64 via dual issue of Wave32	Wave64 (4x SIMD16)
Single cycle instruction	Four cycle instruction
4 triangles/clock (after culling), **>> 4** (before culling)	2-4 triangles/clock (culled/unculled)

OPTIMIZATIONS

OPTIMIZATIONS

TEXTURE ACCESS

Caches

Access Pattern

WORKLOAD DISTRIBUTION

Wave32 / Wave64

SHADER OPTIMIZATIONS

Wave / subgroup operations

TEXTURE ACCESS

TEXTURE ACCESS - CACHES

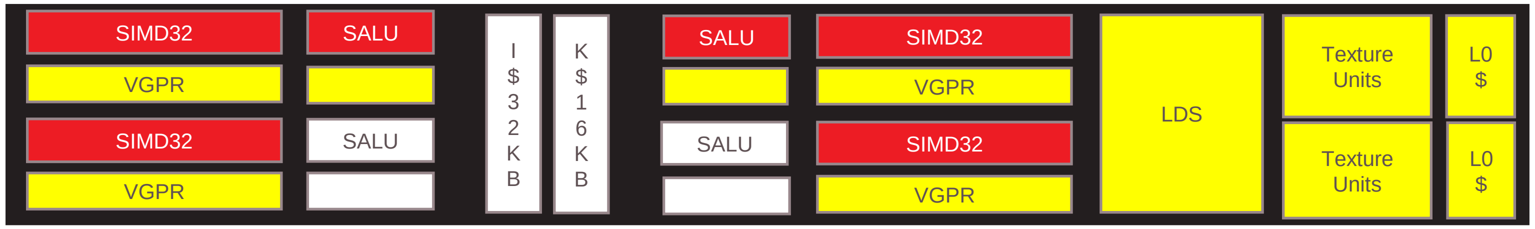
When loading from memory, we want as many cache hits and as few cache misses as possible ☺

+	-
We have one more level of Cache \o/ -> L1 Cache	The two L0 caches on one WGP are not coherent ☹

TEXTURE ACCESS - CACHES

When loading from memory, we want as many cache hits and as few cache misses as possible ☺

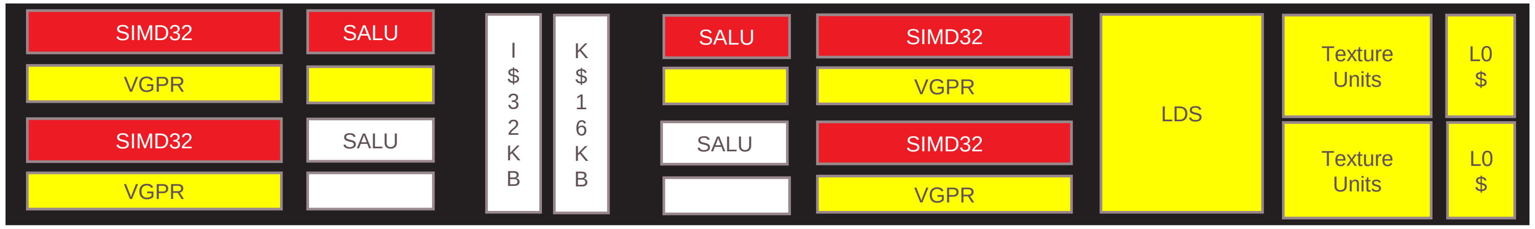
+	-
We have one more level of Cache \o/ -> L1 Cache	**The two L0 caches on one WGP are not coherent ☹**



TEXTURE ACCESS - CACHES

When loading from memory, we want as many cache hits and as few cache misses as possible ☺

+	-
We have one more level of Cache \o/ -> L1 Cache	**The two L0 caches on one WGP are not coherent ☹**



L0 caches not coherent on one WGP

- does **not** affect threads within a **single wavefront**
- **can** affect threads across one **single thread group** if thread group size **> 32** 🤔

TEXTURE ACCESS - CACHES

When loading from memory, we want as many cache hits and as few cache misses as possible ☺

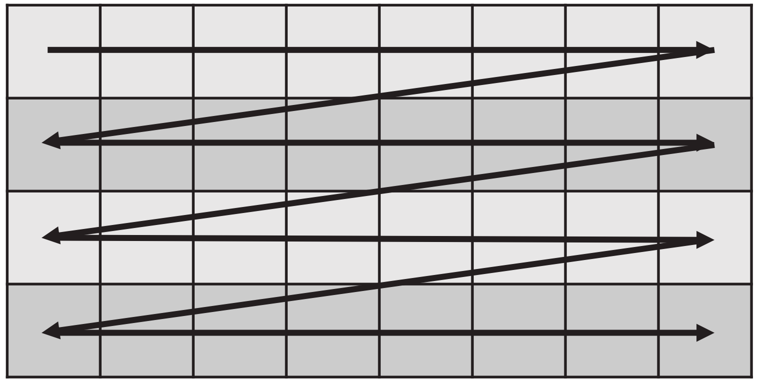
+	-
We have one more level of Cache \o/ -> L1 Cache	The two L0 caches on one WGP are not coherent ☹
L2 Cache has more clients -> less cache flushes	
Increased Cache Line Size -> **128B**	**Potentially need to adjust memory alignments**

L0 caches not coherent on one WGP

- does **not** affect threads within a **single wavefront**
- **can** affect threads across one **single thread group** if thread group size **> 32** 🤔

TEXTURE ACCESS

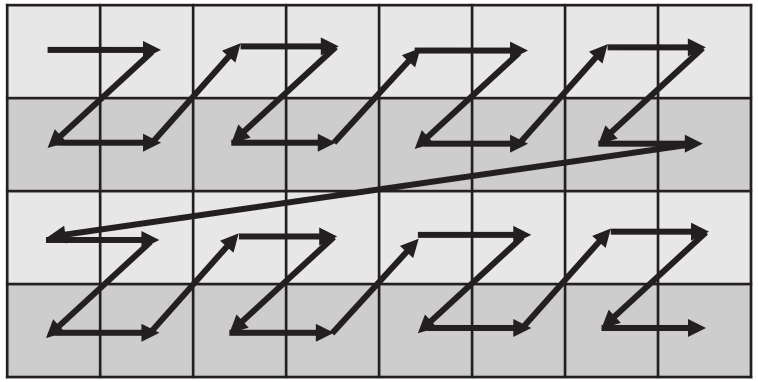
The thread indices in a compute shader are organized in a ROW_MAJOR pattern, matching a linear texture



TEXTURE ACCESS

Texture access is – however – optimized for the standard swizzle

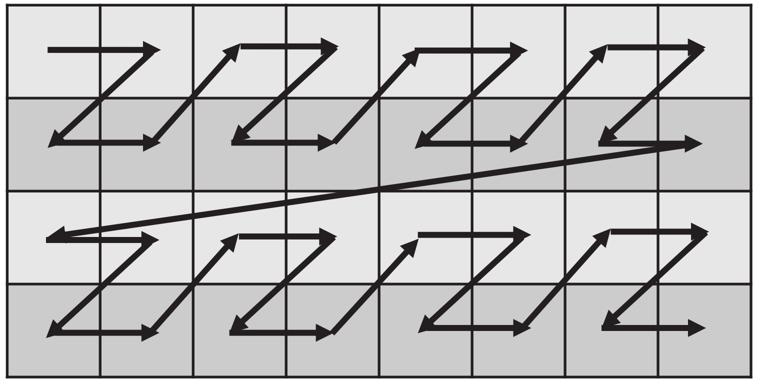
See also Microsoft‘s documentation about texture layouts: https://docs.microsoft.com/en-us/windows/win32/api/d3d12/ne-d3d12-d3d12_texture_layout



TEXTURE ACCESS

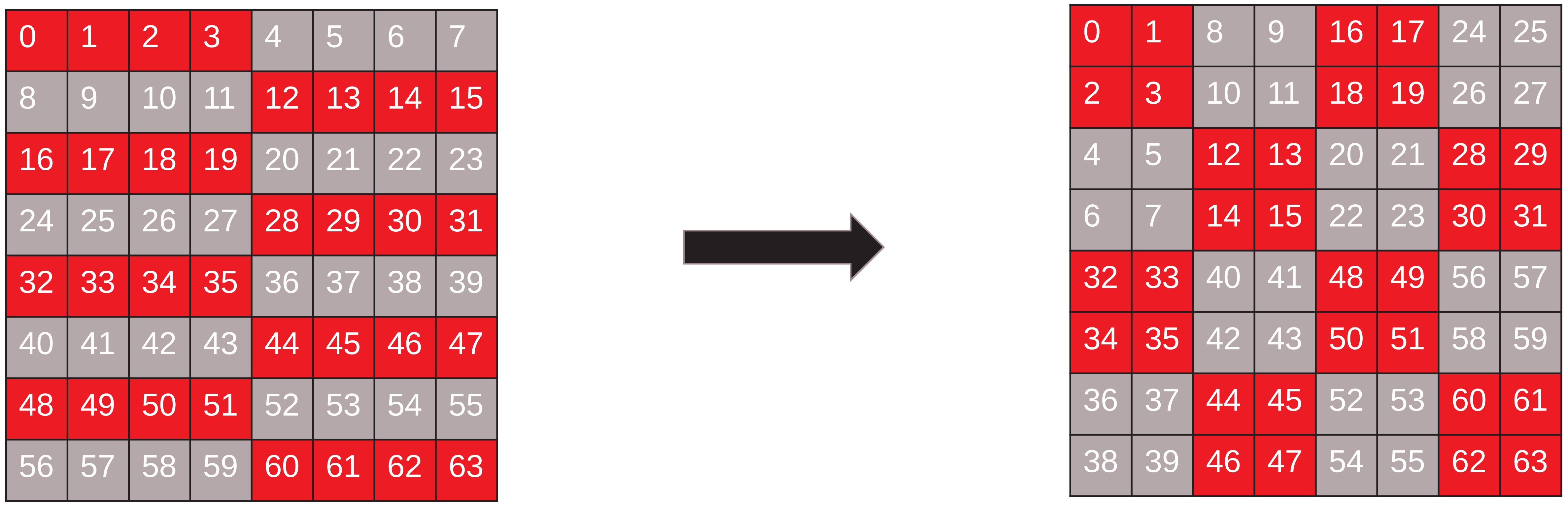
Texture access is – however – optimized for the standard swizzle

See also Microsoft‘s documentation about texture layouts:
https://docs.microsoft.com/en-us/windows/win32/api/d3d12/ne-d3d12-d3d12_texture_layout



This is the pattern in which textures are laid out - neighboring pixels are stored close together in memory

MORTON-LIKE ORDERING

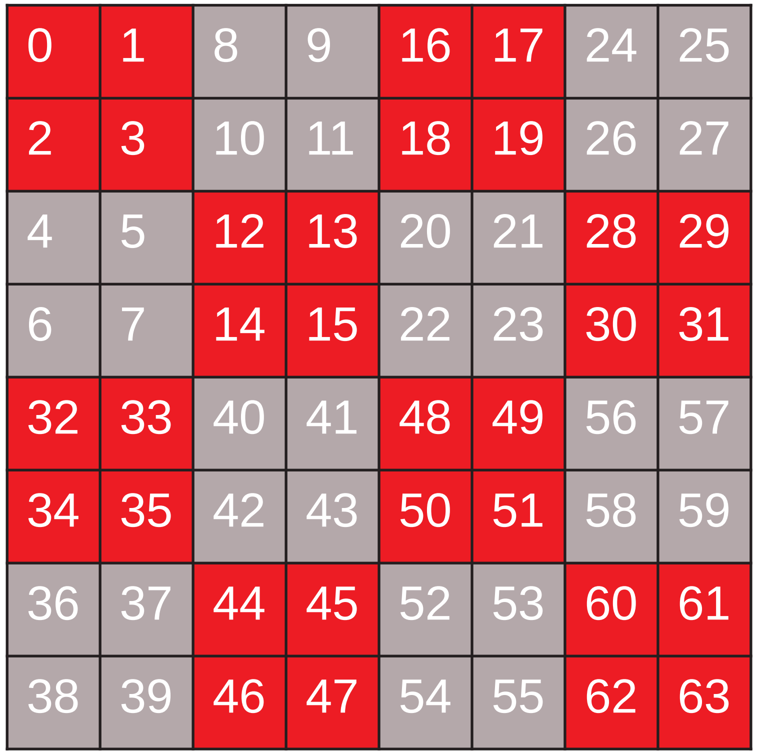


0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	8	9	16	17	24	25
2	3	10	11	18	19	26	27
4	5	12	13	20	21	28	29
6	7	14	15	22	23	30	31
32	33	40	41	48	49	56	57
34	35	42	43	50	51	58	59
36	37	44	45	52	53	60	61
38	39	46	47	54	55	62	63

MORTON-LIKE ORDERING

```
x = (((index >> 2) & 0x0007) & 0xFFFE) | index & 0x0001
y = ((index >> 1) & 0x0003) | (((index >> 3) & 0x0007) & 0xFFFC)
```



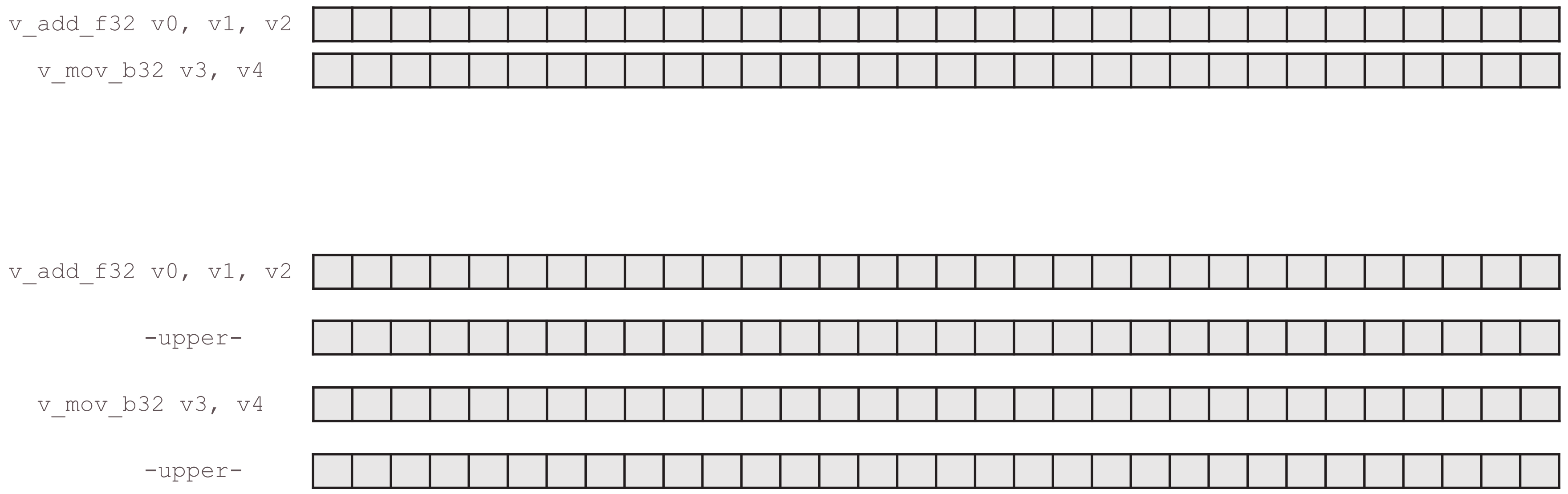
0	1	8	9	16	17	24	25
2	3	10	11	18	19	26	27
4	5	12	13	20	21	28	29
6	7	14	15	22	23	30	31
32	33	40	41	48	49	56	57
34	35	42	43	50	51	58	59
36	37	44	45	52	53	60	61
38	39	46	47	54	55	62	63

WORKLOAD DISTRIBUTION

WORKLOAD DISTRIBUTION

On RDNA, the shader can run either in **Wave32** or **Wave64** mode

→ **Not controllable** within the shader – the driver chooses the mode for the shader



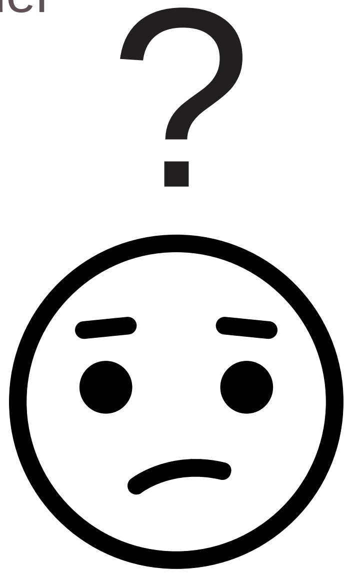
WORKLOAD DISTRIBUTION

On RDNA, the shader can run either in **Wave32** or **Wave64** mode

→ **Not controllable** within the shader – the driver chooses the mode for the shader

But how to design our shaders now?

For Wave32 or Wave64?



WORKLOAD DISTRIBUTION

On RDNA, the shader can run either in **Wave32** or **Wave64** mode

→ **Not controllable** within the shader – the driver chooses the mode for the shader

!

☺

But how to design our shaders now?

For Wave32 or **Wave64**?

→ For Wave64!

WORKLOAD DISTRIBUTION

A multiple of 64 as thread group size works well for **both**

Wave64

and

Wave32

This is good news for GCN cards ☺

WORKLOAD DISTRIBUTION

A multiple of 64 as thread group size works well for **both**

Wave64

and

Wave32

This is good news for GCN cards ☺

That's it?

WORKLOAD DISTRIBUTION

Arrange the thread groups within the dispatch in multiples of 64

→ Same as for GCN

Arrange the **threads within** a **thread group** in multiples of **32**

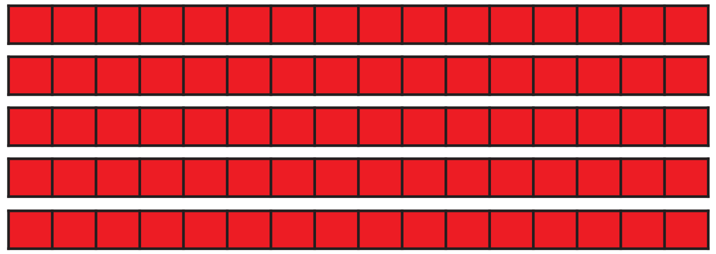
→ GCN does not care about this

→ Since all 64 threads always had to run in lock step unless **all** threads are inactive

→ Not true for RDNA ☺

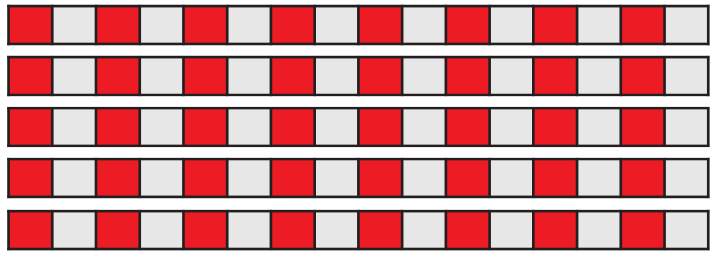
GCN WAVE 64

```
v_add_f32 v0, v1, v2

v_mov_b32 v3, v4
```



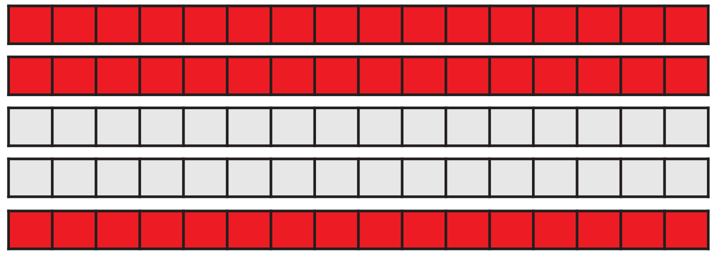
Full wavefront is executed

```
v_add_f32 v0, v1, v2

v_mov_b32 v3, v4
```



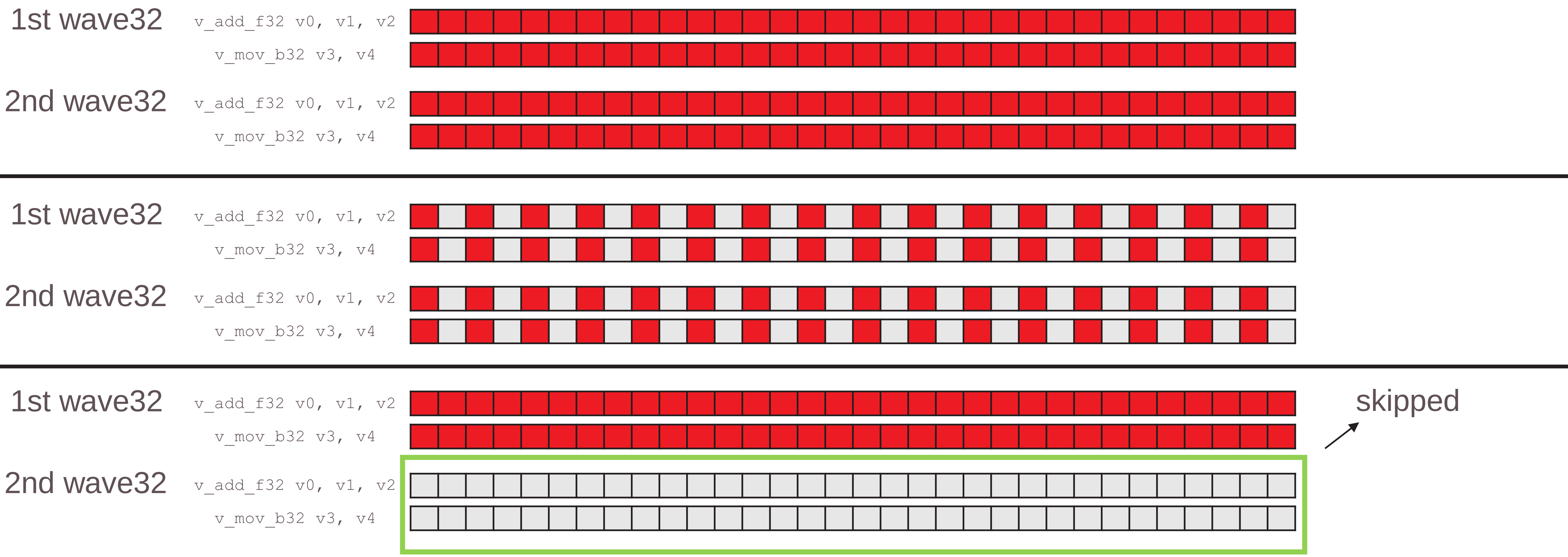
Full wavefront is executed

```
v_add_f32 v0, v1, v2

v_mov_b32 v3, v4
```

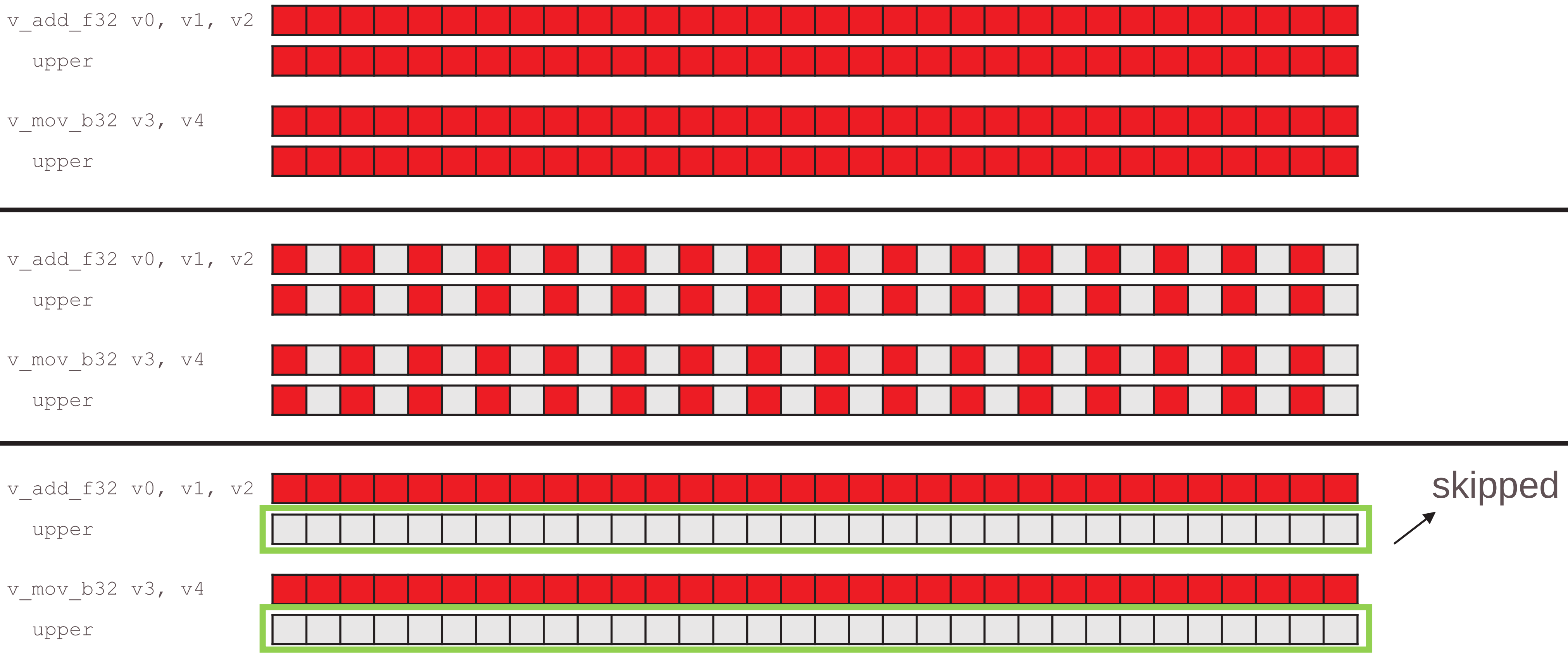


Full wavefront is executed

RDNA WAVE 32


1st wave32
v_add_f32 v0, v1, v2
v_mov_b32 v3, v4
2nd wave32
v_add_f32 v0, v1, v2
v_mov_b32 v3, v4
1st wave32
v_add_f32 v0, v1, v2
v_mov_b32 v3, v4
2nd wave32
v_add_f32 v0, v1, v2
v_mov_b32 v3, v4
1st wave32
v_add_f32 v0, v1, v2
v_mov_b32 v3, v4
skipped
2nd wave32
v_add_f32 v0, v1, v2
v_mov_b32 v3, v4

RDNA WAVE 64


v_add_f32 v0, v1, v2
upper
v_mov_b32 v3, v4
upper
v_add_f32 v0, v1, v2
upper
v_mov_b32 v3, v4
upper
v_add_f32 v0, v1, v2
upper
skipped
v_mov_b32 v3, v4
upper

CONCLUSION

Use a multiple of **64** as thread group size

- Works well on RDNA both for
  - Wave32
  - Wave64
- Works well on GCN

Group your active threads within a single thread group / wavefront

- Inactive groups of 32 threads can be skipped on RDNA

SHADER OPTIMIZATIONS

SHADER OPTIMIZATIONS

Loading data from global memory can be quite expensive

To share data between threads of a single thread group, we can use Local Data Share (LDS)

```
groupshared float data[32];
```

→ Faster than global memory!

What about **threads** of a **single wavefront**?

SHADER OPTIMIZATIONS

Loading data from global memory can be quite expensive

To share data between threads of a single thread group, we can use LDS

→ Faster than global memory!

What about threads of a single wavefront?

→ Make use of Data Parallel Processing (DPP) or LDS Permute

SHADER OPTIMIZATIONS

DPP and LDS Permute is **nothing new** on RDNA

→ Works also on GCN \o/

But some specifics have changed 🤔

But first … what are DPP and LDS Permute exactly 😋

DATA PARALLEL PROCESSING (DPP)

id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
v0	1	8	3	1	5	6	1	6	5	7	4	5	2	3	0	0	4	2	3	5	6	4	7	3	2	7	9	4	6	0	0	7

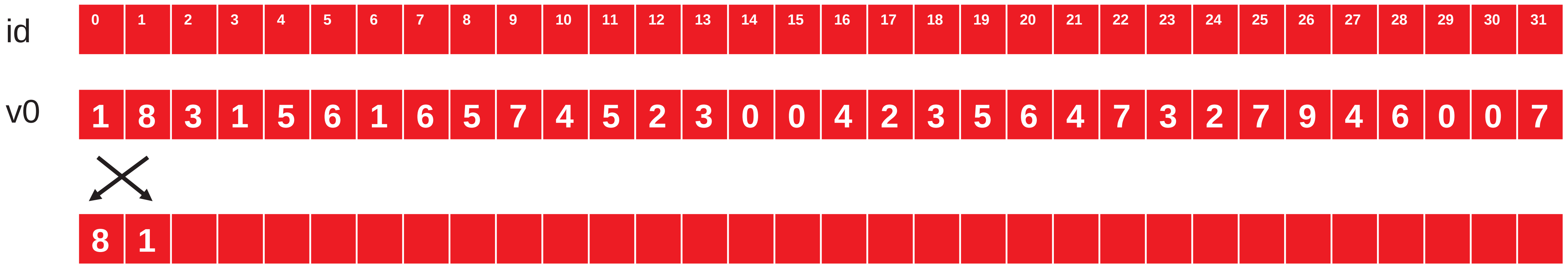
DPP can be used to exchange data between threads of a single wavefront

Thread 0 stores the value 1 in register v0

Thread 1 stores the value 8 in register v0

"Use DPP" so that thread 0 reads value 8 and thread 1 reads value 1

DATA PARALLEL PROCESSING (DPP)



id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
v0	1	8	3	1	5	6	1	6	5	7	4	5	2	3	0	0	4	2	3	5	6	4	7	3	2	7	9	4	6	0	0	7
	8	1																														

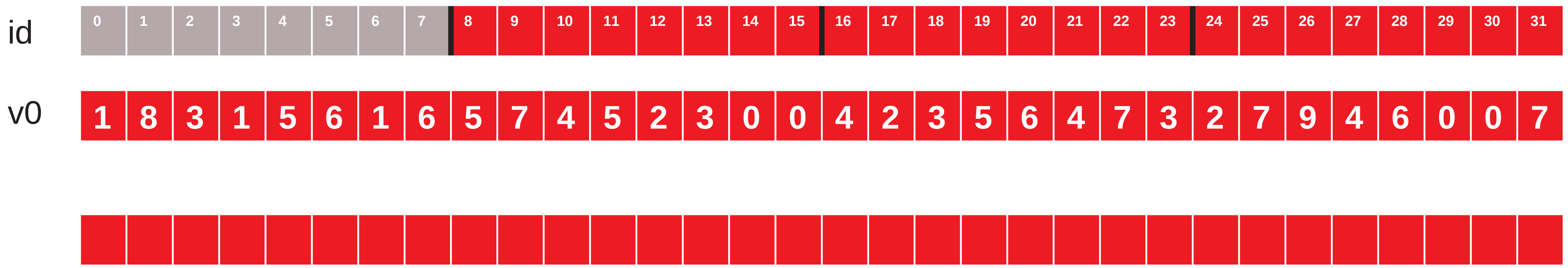
DPP can be used to exchange data between threads of a single wavefront

Thread 0 stores the value 1 in register v0

Thread 1 stores the value 8 in register v0

"Use DPP" so that thread 0 reads value 8 and thread 1 reads value 1

DATA PARALLEL PROCESSING (DPP)

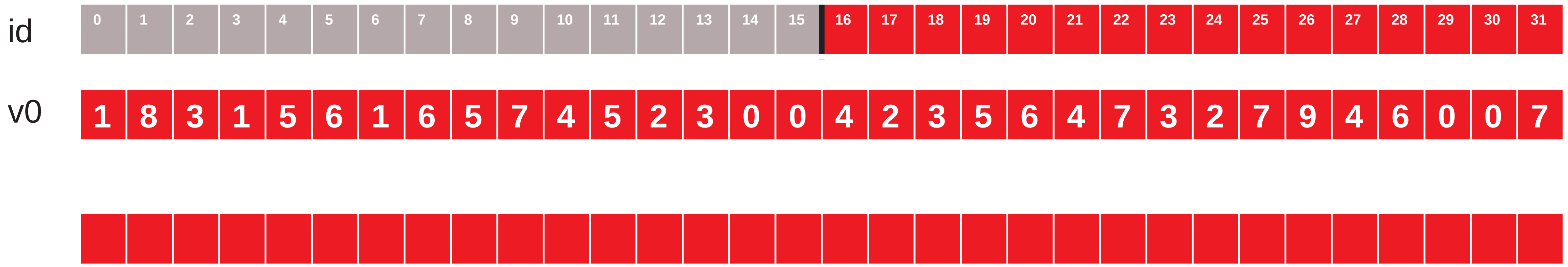


There are two DPP modes available on RDNA:

DPP instructions that operate on a group of **8 threads**: **DPP8**

Supports **arbitrary** swizzles

DATA PARALLEL PROCESSING (DPP)



id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
v0	1	8	3	1	5	6	1	6	5	7	4	5	2	3	0	0	4	2	3	5	6	4	7	3	2	7	9	4	6	0	0	7

There are two DPP modes available on RDNA:

DPP instructions that operate on a group of **16 threads: DPP16**

Supports a set of **predefined** swizzles

Permute of four threads

Row shift left by 1-15 threads

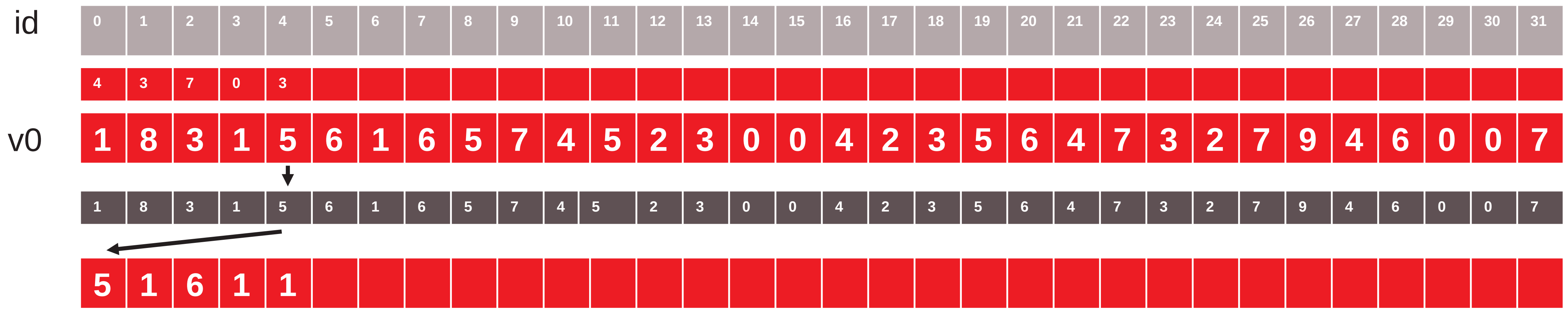
Row shift right by 1-15 threads

Row rotate right by 1-15

Mirror threads within half row (8 threads)

Mirror threads within row

LDS PERMUTE



We can do data exchange using LDS permute across 32 threads

- All active lanes write data to a temporary buffer
- All active lanes read data from the temporary buffer

Uses LDS hardware, but does not write to LDS memory

Uses additional VGPRs

DPP & LDS PERMUTE - IMPLEMENTATION

Some general guidelines:

- DPP limited to groups of 16
- Prefer to shuffle only across groups of 8
- Avoid shuffles across more than 32 threads

DPP & LDS PERMUTE - IMPLEMENTATION

Some general guidelines:

There is only DPP8 or DPP16

- DPP **limited** to groups of 16
- Prefer to shuffle only across groups of 8
- Avoid shuffles across more than 32 threads

DPP & LDS PERMUTE - IMPLEMENTATION

Some general guidelines:

- DPP limited to groups of 16
- Prefer to shuffle only **across groups of 8**
- Avoid shuffles across more than 32 threads

DPP8: Supports **arbitrary** swizzles

DPP & LDS PERMUTE - IMPLEMENTATION

Some general guidelines:

- DPP limited to groups of 16
- Prefer to shuffle only across groups of 8
- **Avoid shuffles across more than 32 threads**

LDS permute **limited** to 32 threads

Needs to use other instructions (e.g., readFirstLane)

DPP & LDS PERMUTE - IMPLEMENTATION

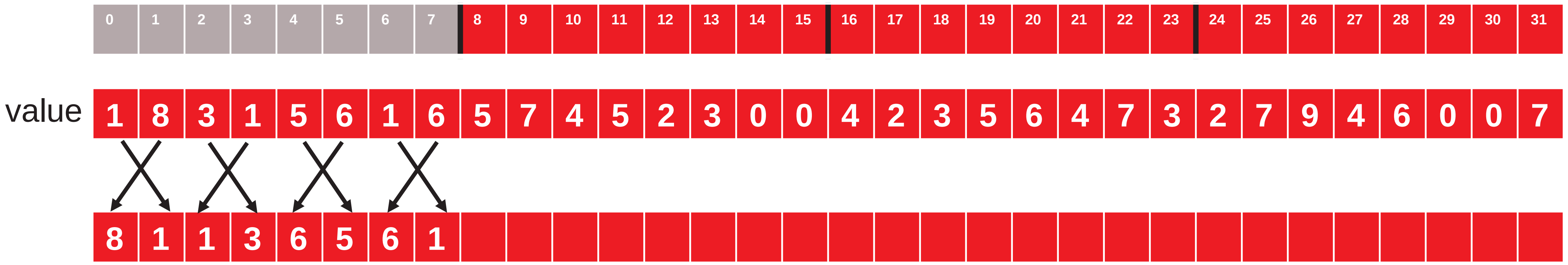
How to use DPP and LDS Permute in our shader?

Obviously there are no low level intrinsics in HLSL/GLSL

Use wave / subgroup operations – they can get translated into dpp or lds permute instructions at the ISA level

→ Follow previous guidelines!

DPP & LDS PERMUTE - IMPLEMENTATION



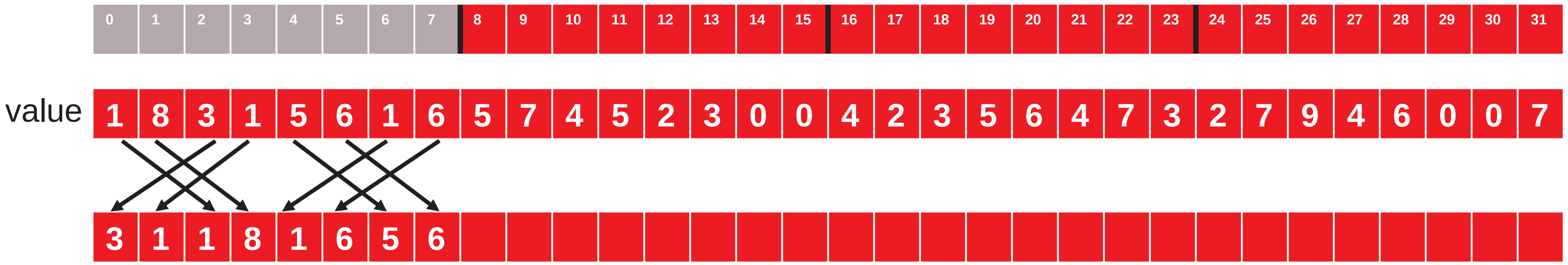
HLSL, SM6.0:

```
value = QuadReadAcrossX(value)
```

GLSL, subgroup operations:

```
value = subgroupQuadSwapHorizontal(value)
```

DPP & LDS PERMUTE - IMPLEMENTATION



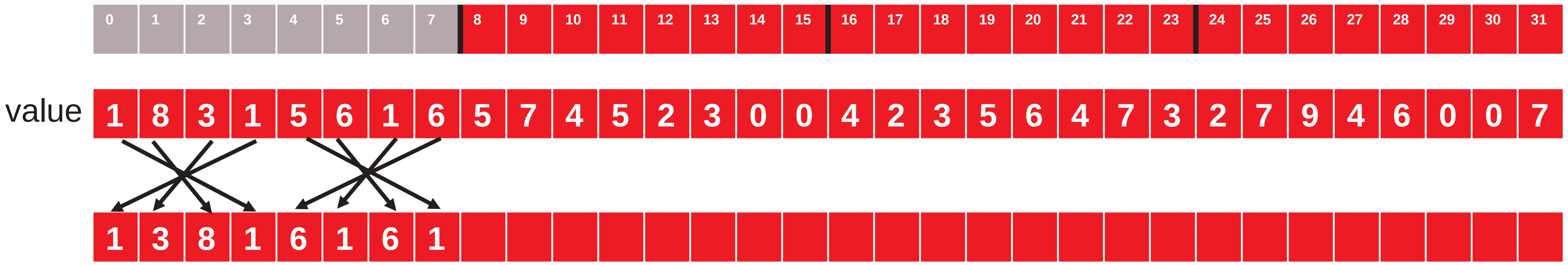
HLSL, SM6.0:

```
value = QuadReadAcrossY(value)
```

GLSL, subgroup operations:

```
value = subgroupQuadSwapVertical(value)
```

DPP & LDS PERMUTE - IMPLEMENTATION



HLSL, SM6.0:

```
value = QuadReadAcrossDiagonal(value)
```

GLSL, subgroup operations:

```
value = subgroupQuadSwapDiagonal(value)
```

DPP & LDS PERMUTE - IMPLEMENTATION

0	1	2	3
1	8	3	1

```
float result = v;
float result += subgroupQuadSwapHorizontal(v);
float result += subgroupQuadSwapVertical(v);
float result += subgroupQuadSwapDiagonal(v);
```

```
v_add_f32_dpp v26, v4, v4  quad_perm:[1, 0, 3, 2] row_mask:0xf bank_mask:0xf bound_ctrl:0
v_add_f32_dpp v26, v4, v26  quad_perm:[2, 3, 0, 1] row_mask:0xf bank_mask:0xf bound_ctrl:0
v_add_f32_dpp v4, v4, v26  quad_perm:[3, 2, 1, 0] row_mask:0xf bank_mask:0xf bound_ctrl:0
```

13	13	13	13

OPTIMIZATIONS – APPLIED

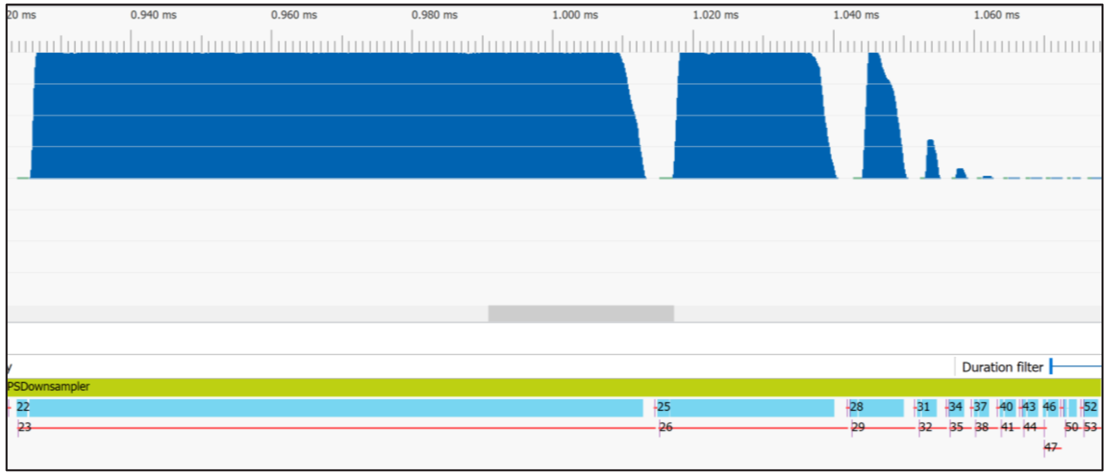
CASE STUDY: DOWNSAMPLING

All the following optimizations are showcased on a **texture downsampler** for **mipmap generation**

A common approach to generate the mipmap levels is using a **pixel shader**, **one pass per mip**

Limitations and bottlenecks of a pixel shader approach:

- **Barriers** between the mips
- Data exchange between the mips via **global memory**



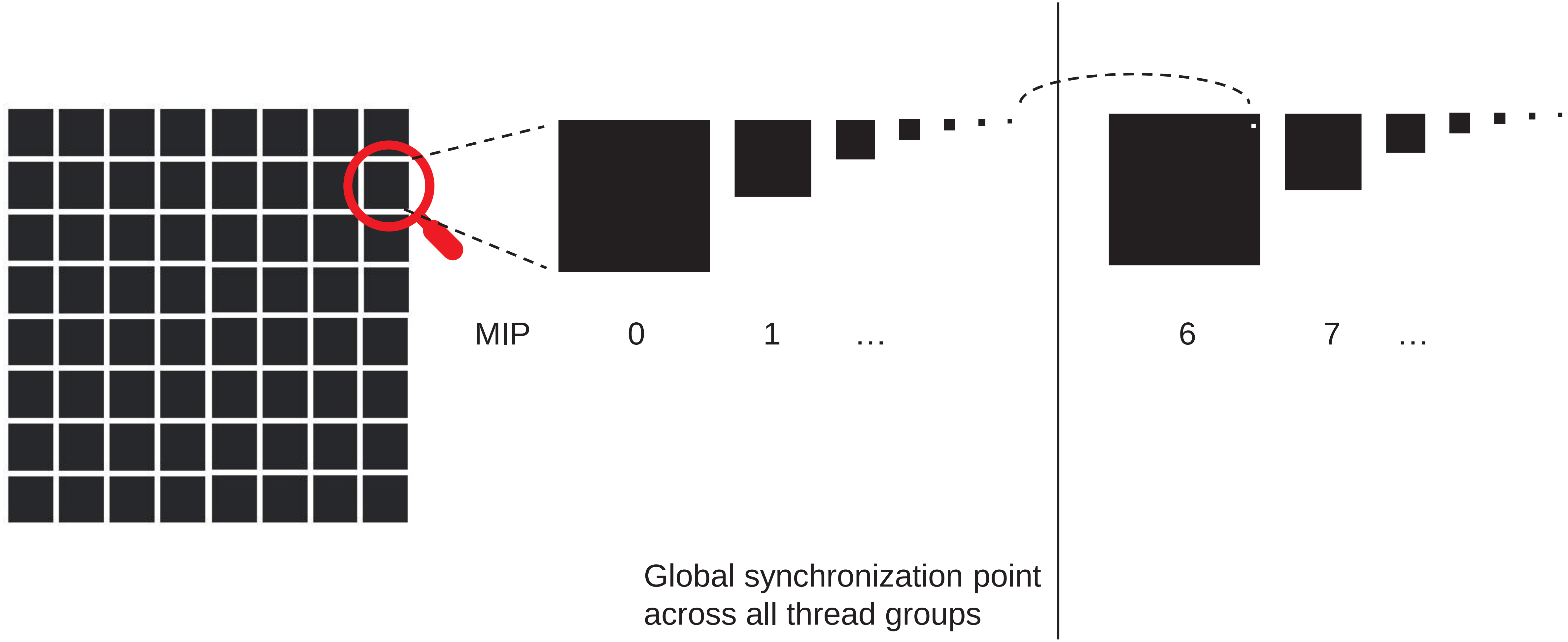
SINGLE PASS DOWNSAMPLER (SPD)

GPUOpen‘s FidelityFX Single Pass Downsampler (SPD) uses a **single pass compute shader** to generate **all mip levels**

Basic concept of SPD:

- Threadgroup of 256 threads downsamples a tile of 64x64 down to 1x1
- Last active threadgroup computes the remaining mips
- Can downsample a texture of size 4096x4096 to 1x1

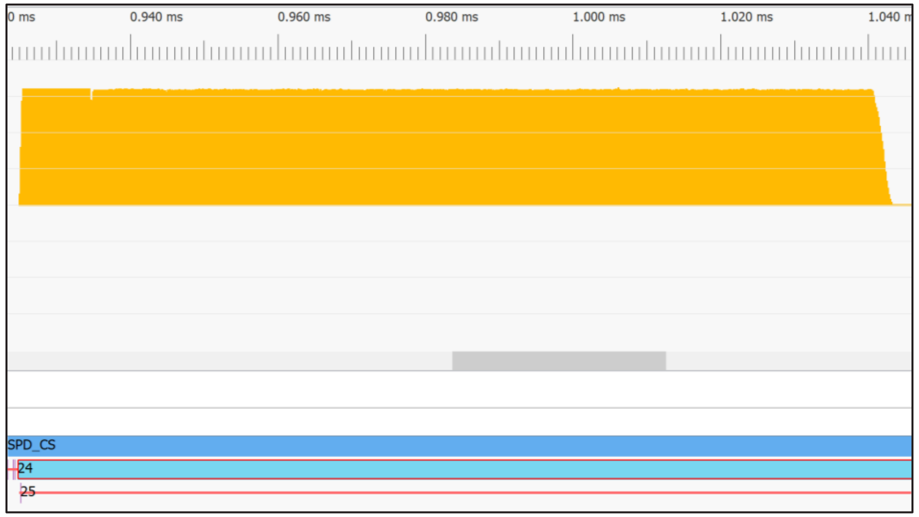
SINGLE PASS DOWNSAMPLER (SPD)



SINGLE PASS DOWNSAMPLER (SPD)

Advantages:

- **No barriers**
- Data exchange between the mips via LDS or DPP except for mip 6
- Can **overlap work** with other dispatches/draw calls due to no barriers between the mip generation



OPTIMIZATIONS – APPLIED TO SPD

TEXTURE ACCESS

How to load the source texture?

WORKLOAD DISTRIBUTION

How to distribute the work to the available threads?

How to share the data between the threads?

SHADER OPTIMIZATIONS

Usage of DPP

FP16 support

TEXTURE ACCESS

TEXTURE ACCESS

A time consuming part of SPD is loading the data of the source image texture

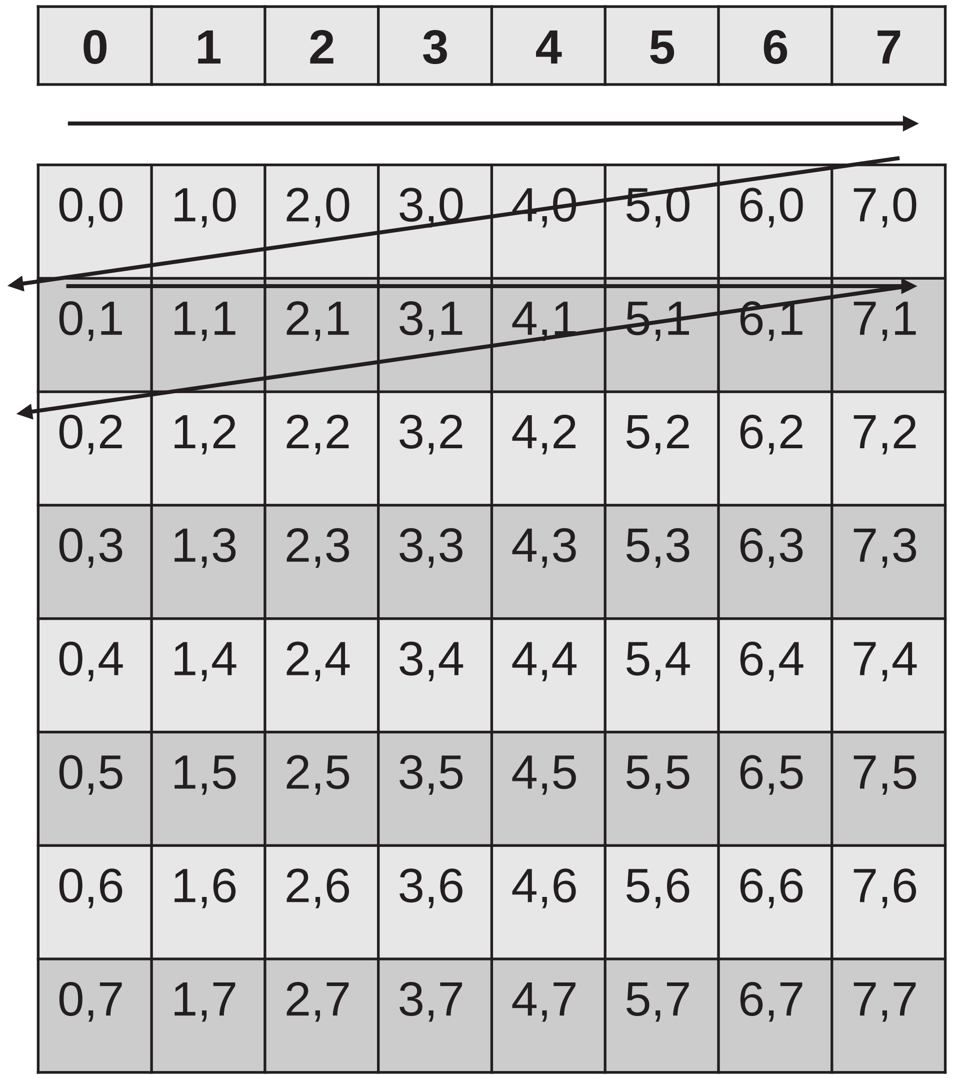
Especially for high resolution images this is critical

For low resolution images, when the data fits in the cache, it‘s less sensitive to the chosen access pattern

TEXTURE ACCESS

Common approach, e.g. compute shader

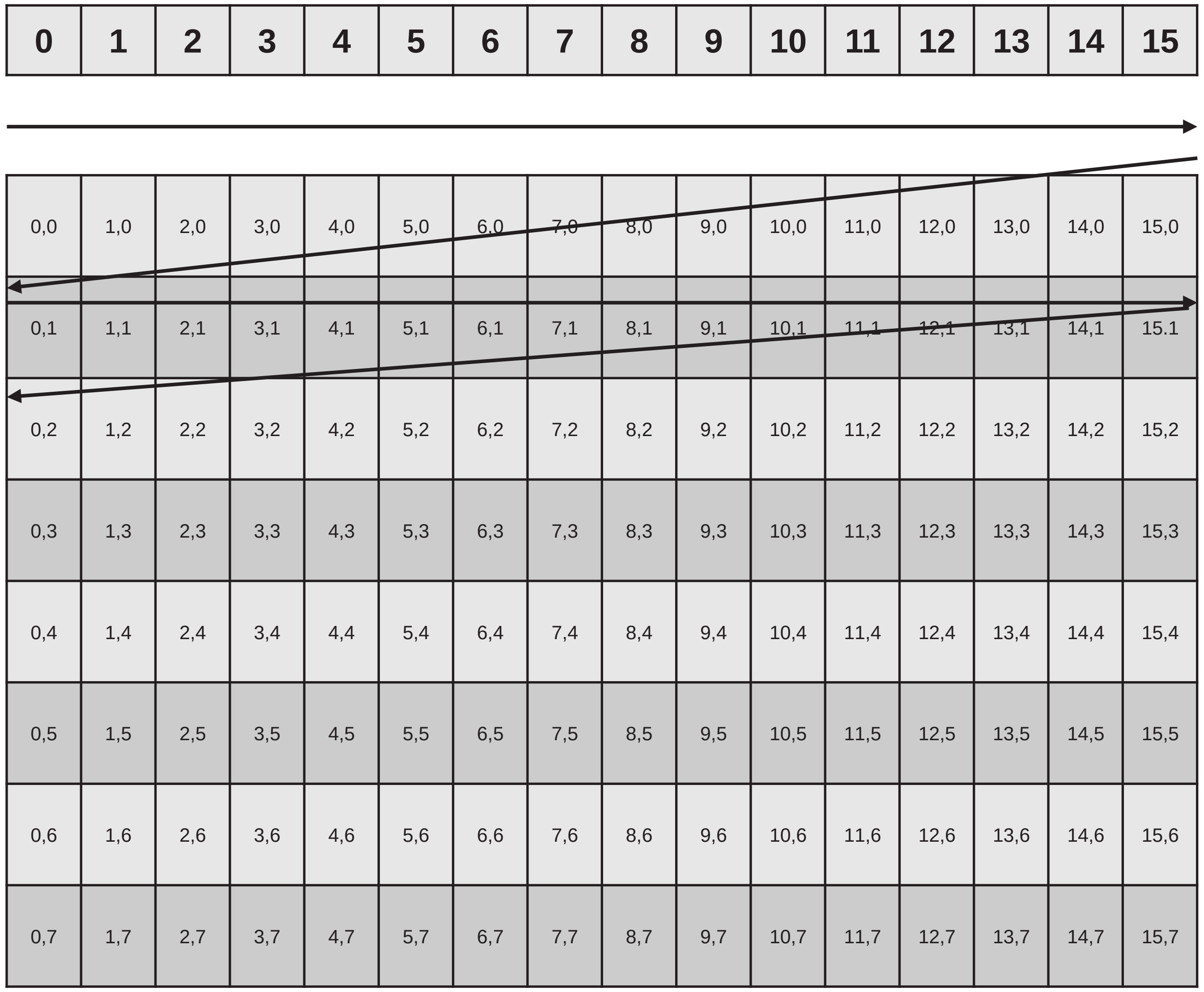
[**numthreads**(8,8,1)]



TEXTURE ACCESS

SPD has a thread group size of 256

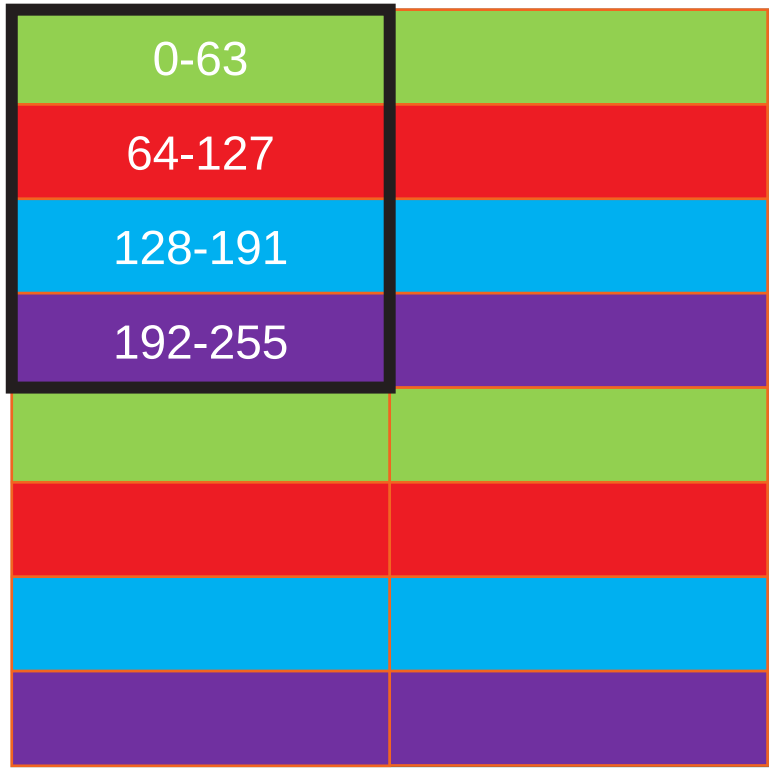
[**numthreads**(256,1,1)]

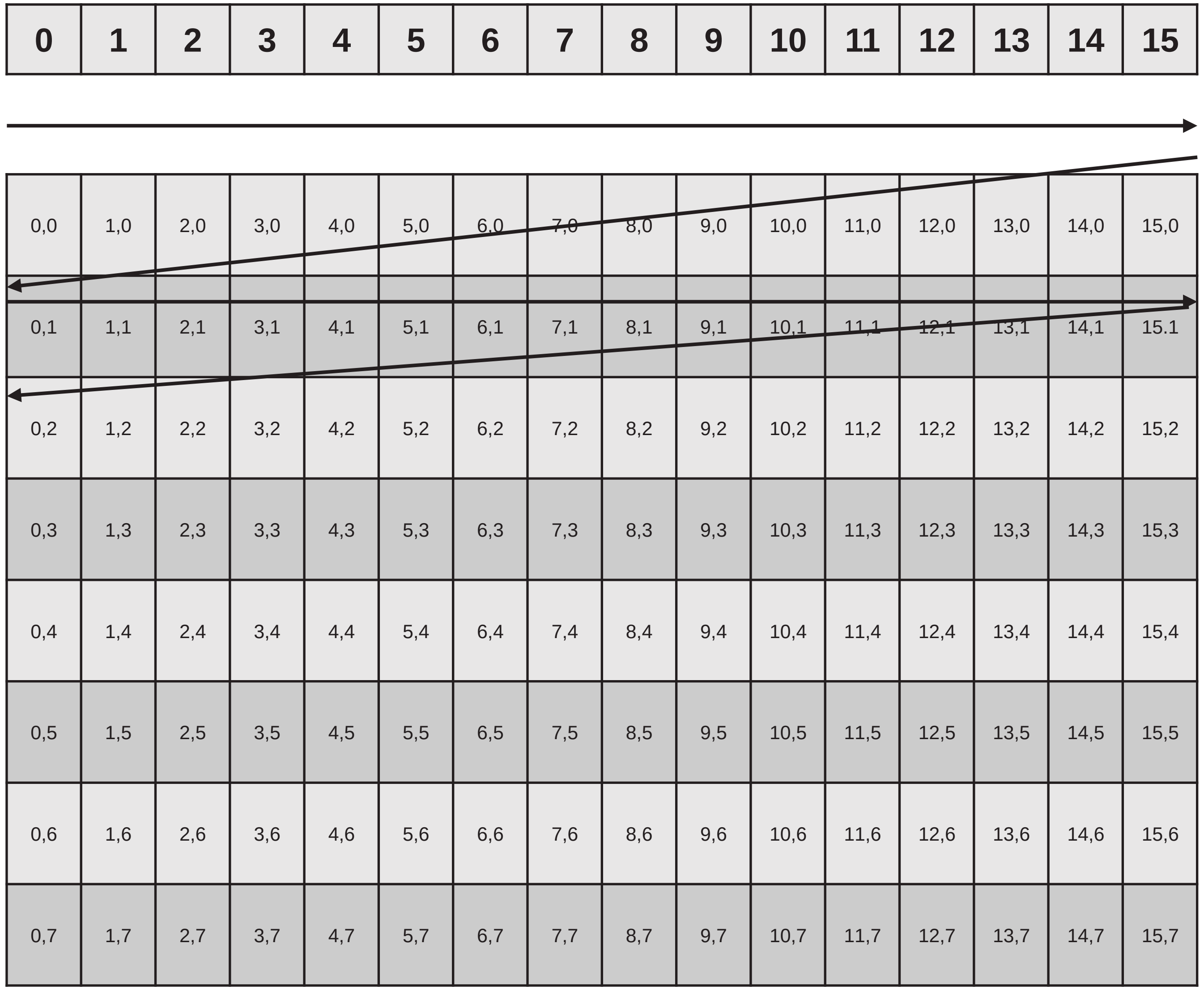


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

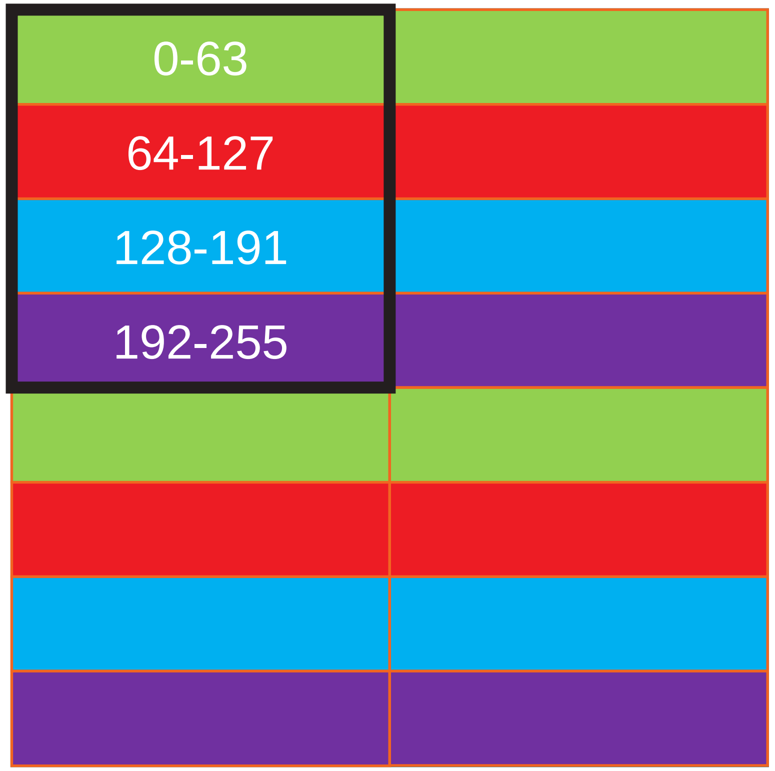
0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0	10,0	11,0	12,0	13,0	14,0	15,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1	10,1	11,1	12,1	13,1	14,1	15.1
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2	10,2	11,2	12,2	13,2	14,2	15,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	10,3	11,3	12,3	13,3	14,3	15,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4	9,4	10,4	11,4	12,4	13,4	14,4	15,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5
0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6	9,6	10,6	11,6	12,6	13,6	14,6	15,6
0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7	9,7	10,7	11,7	12,7	13,7	14,7	15,7

TEXTURE ACCESS





TEXTURE ACCESS



Texel index

0,0	1,0	2,0	3,0	4,0	5,0
0	0	1	1	2	2
0	0	1	1	2	2
16	16	17	17	18	18
16	16	17	17	18	18

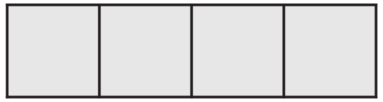
…

32,0	33,0	34,0	35,0	36,0	37,0
0	0	1	1	2	2
0	0	1	1	2	2
16	16	17	17	18	18
16	16	17	17	18	18

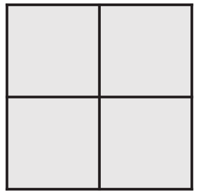
TEXTURE ACCESS

Disadvantages:

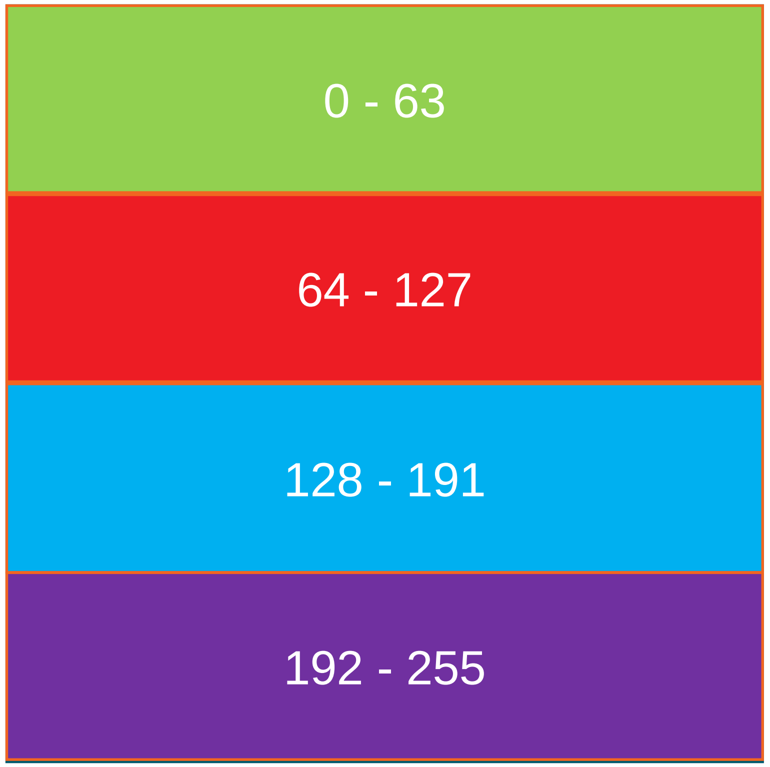
- We can't use quad swizzle to compute value for mip 2
- Thread 0-3 are computing consecutive texels in a lane

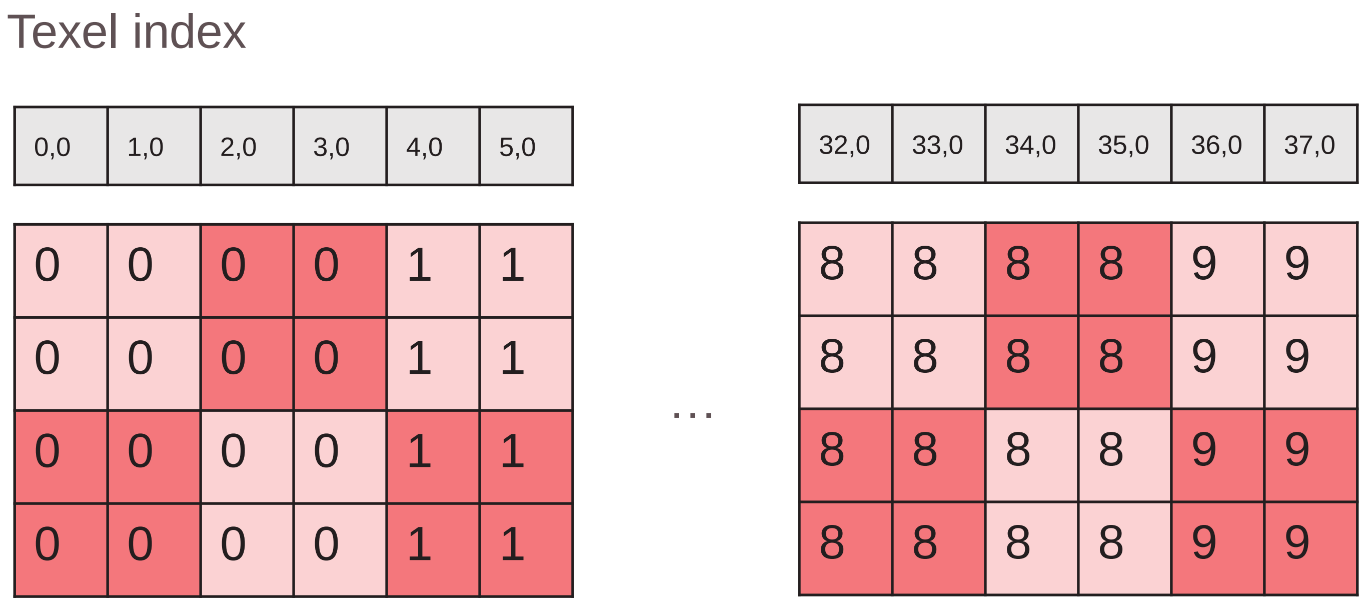


- For quad swizzle, we need thread 0-3 arranged in a quad pattern



TEXTURE ACCESS





0,0	1,0	2,0	3,0	4,0	5,0
0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	0	1	1

…

32,0	33,0	34,0	35,0	36,0	37,0
8	8	8	8	9	9
8	8	8	8	9	9
8	8	8	8	9	9
8	8	8	8	9	9

TEXTURE ACCESS

Advantage:

- We can compute value for mip 1 and mip 2 within one thread
- No inter-thread communication needed

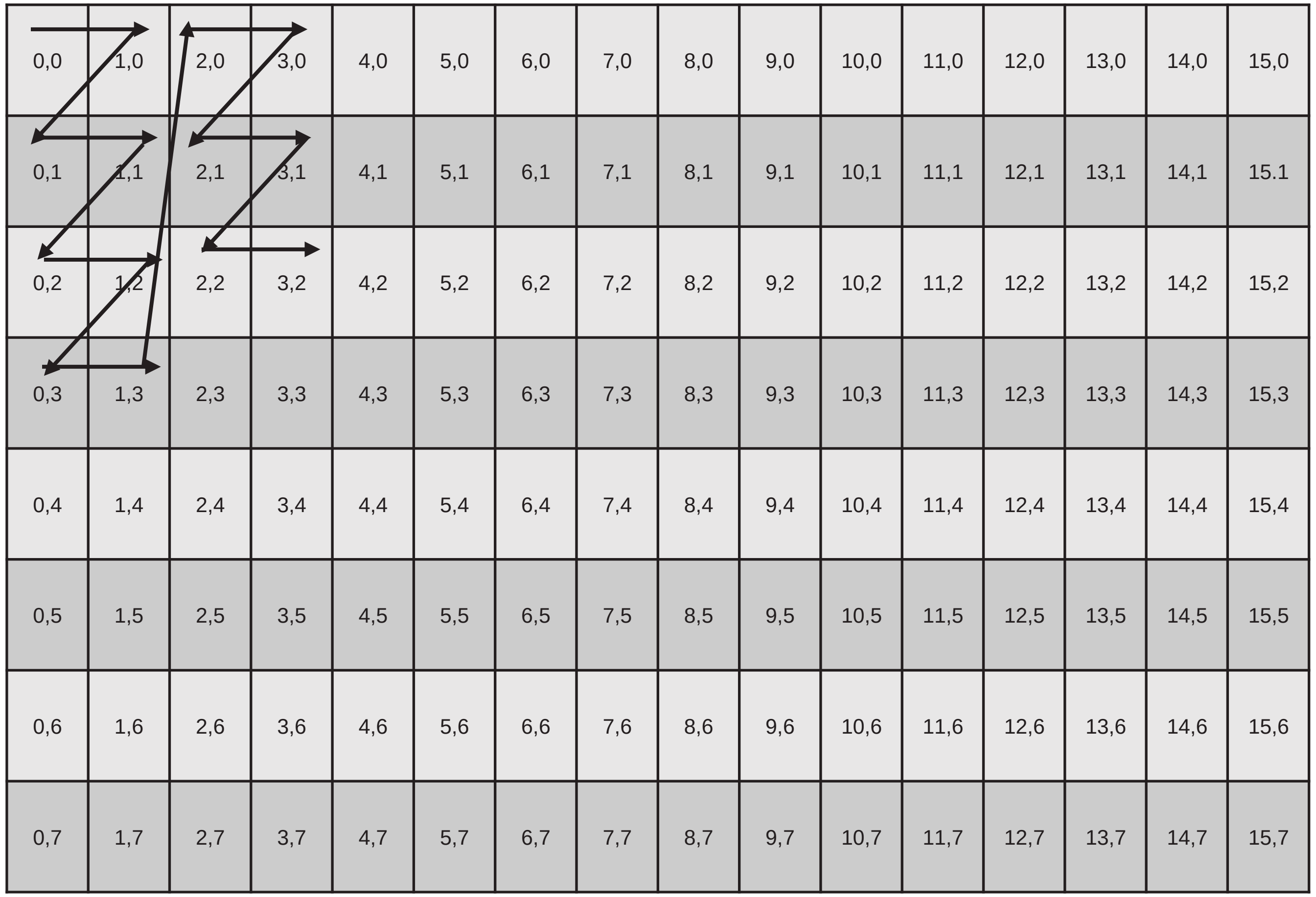
Disadvantage:

- For mip 3, we can‘t use quad swizzle because thread lanes are not in a quad pattern

TEXTURE ACCESS

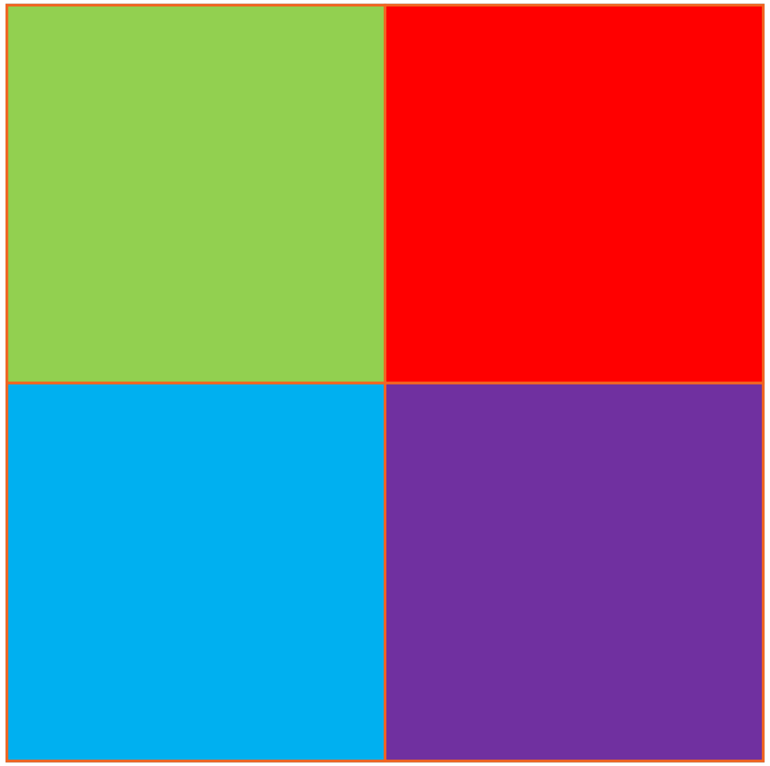
Use a morton-like ordering to rearrange the threads in a 2x2 swizzle

0	1	8	9	16	17	24	25	64	65	72	73	80	81	88	89
2	3	10	11	18	19	26	27	66	67	74	75	82	83	90	91



0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0	10,0	11,0	12,0	13,0	14,0	15,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1	10,1	11,1	12,1	13,1	14,1	15.1
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2	10,2	11,2	12,2	13,2	14,2	15,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	10,3	11,3	12,3	13,3	14,3	15,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4	9,4	10,4	11,4	12,4	13,4	14,4	15,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5
0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6	9,6	10,6	11,6	12,6	13,6	14,6	15,6
0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7	9,7	10,7	11,7	12,7	13,7	14,7	15,7

TEXTURE ACCESS



Texel index

0,0	1,0	2,0	3,0	4,0	5,0
0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	0	1	1

…

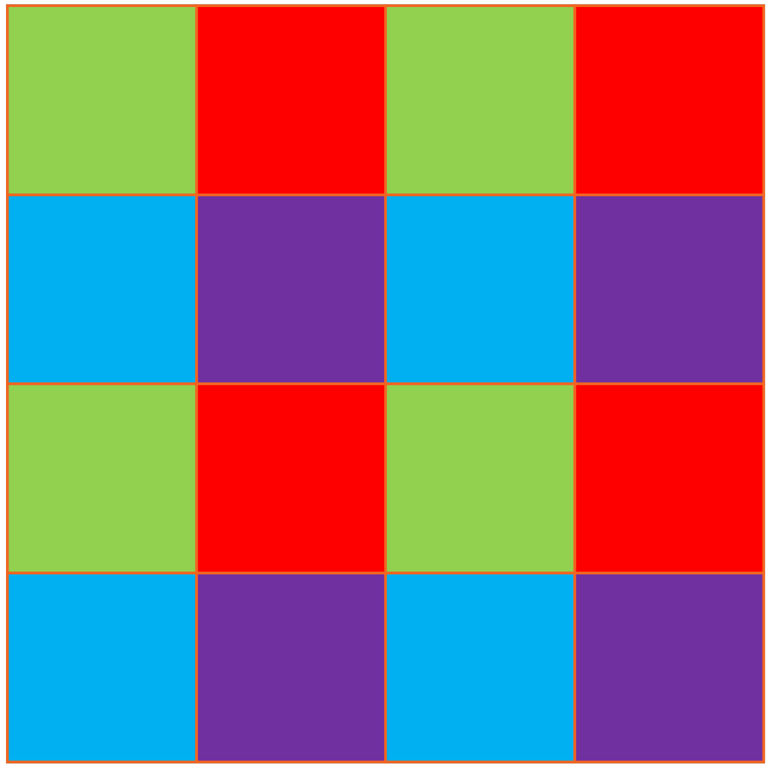
32,0	33,0	34,0	35,0	36,0	37,0
64	64	64	64	65	65
64	64	64	64	65	65
64	64	64	64	65	65
64	64	64	64	65	65

TEXTURE ACCESS

Advantage:

- We can compute value for mip 1 and mip 2 within one thread
- No inter-thread communication needed
- For mip 3, we **can** use quad swizzle because thread lanes are in a quad pattern

TEXTURE ACCESS



Texel index

0,0	1,0	2,0	3,0	4,0	5,0
0	0	1	1	8	8
0	0	1	1	8	8
2	2	3	3	10	10
2	2	3	3	10	10

…

32,0	33,0	34,0	35,0	36,0	37,0
0	0	1	1	8	8
0	0	1	1	8	8
2	2	3	3	10	10
2	2	3	3	10	10

TEXTURE ACCESS

Advantage:

- For mip 2, we can use quad swizzle because thread lanes are in a quad pattern

Disadvantage:

- For mip 3, we need either shuffleXor across 16 threads or LDS

PERFORMANCE COMPARISON

Performance gain from the initial approach to the **last approach** using a Morton-like ordering
→ **~8%**

when

- Downsampling a texture of size 4096x4096
- RGBA16 FLOAT
- Generating 12 mips

System specs:
Radeon™ RX 5700 XT
AMD Radeon™ driver 20.1.4
Ryzen 9 3900X

TEXTURE ACCESS - CONCLUSION

Use a **2x2 thread swizzle** – matches the standard texture layout

→ Morton-like ordering

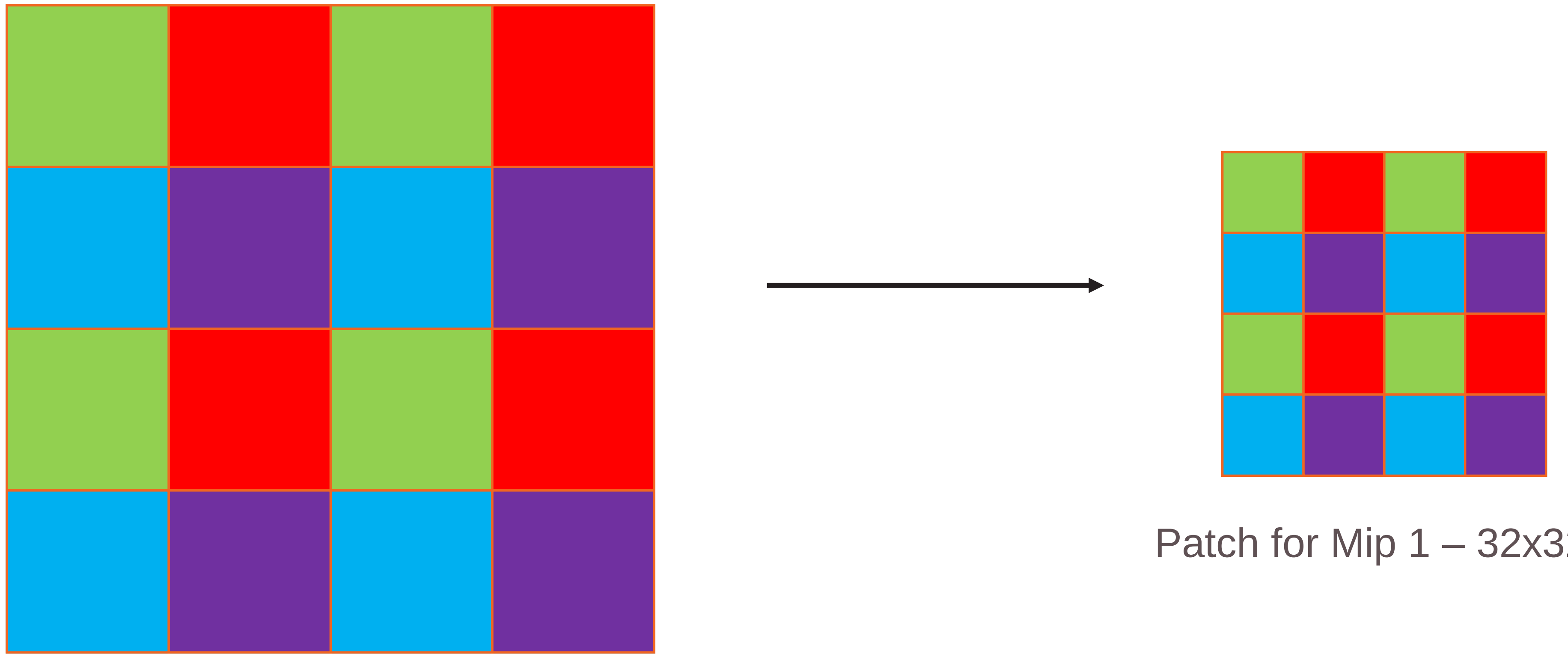
Loading more neighboring texels than 2x2 in one thread does not pay off, as the single load/sample instructions across all threads are not fetching neighboring texels anymore

Using a 2x2 swizzle has also another advantage: quad operations become an option ☺

WORKLOAD & DATA DISTRIBUTION

DATA EXCHANGE BETWEEN THE MIPS

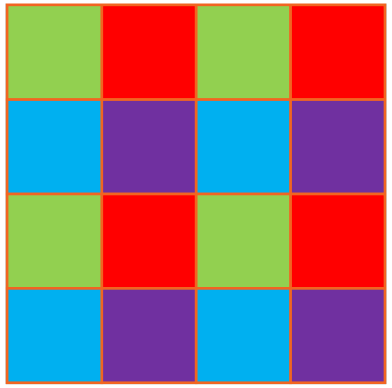
After loading the data from the source image we compute mip 1

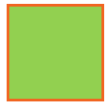


Patch for Mip 1 – 32x32

DATA EXCHANGE BETWEEN THE MIPS

How to compute mip 2?





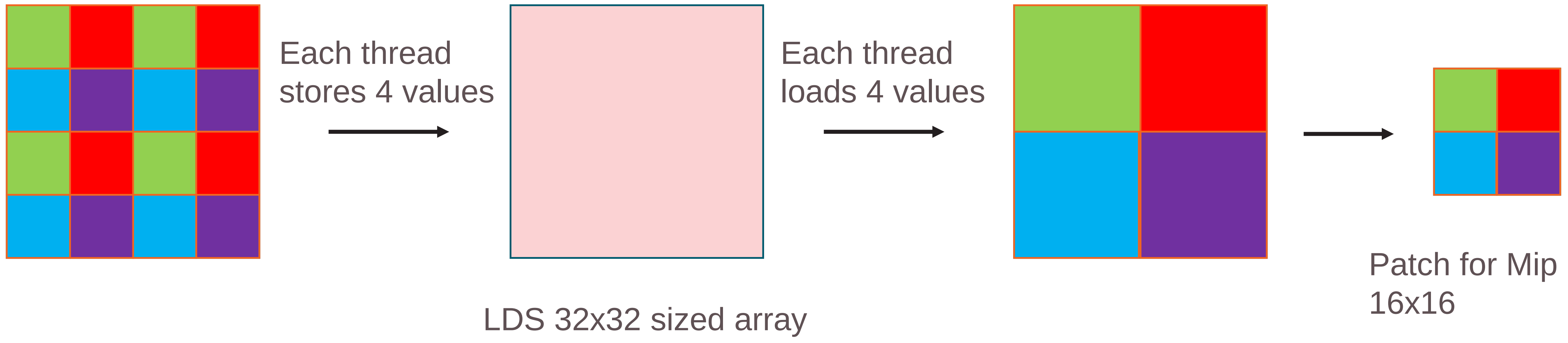
This is a 8x8 patch, values hold by 64 consecutive threads

As outlined before, we can use quad swizzles or LDS to move the data between the threads

Let's ignore quad swizzles for now and use only LDS

DATA EXCHANGE BETWEEN THE MIPS

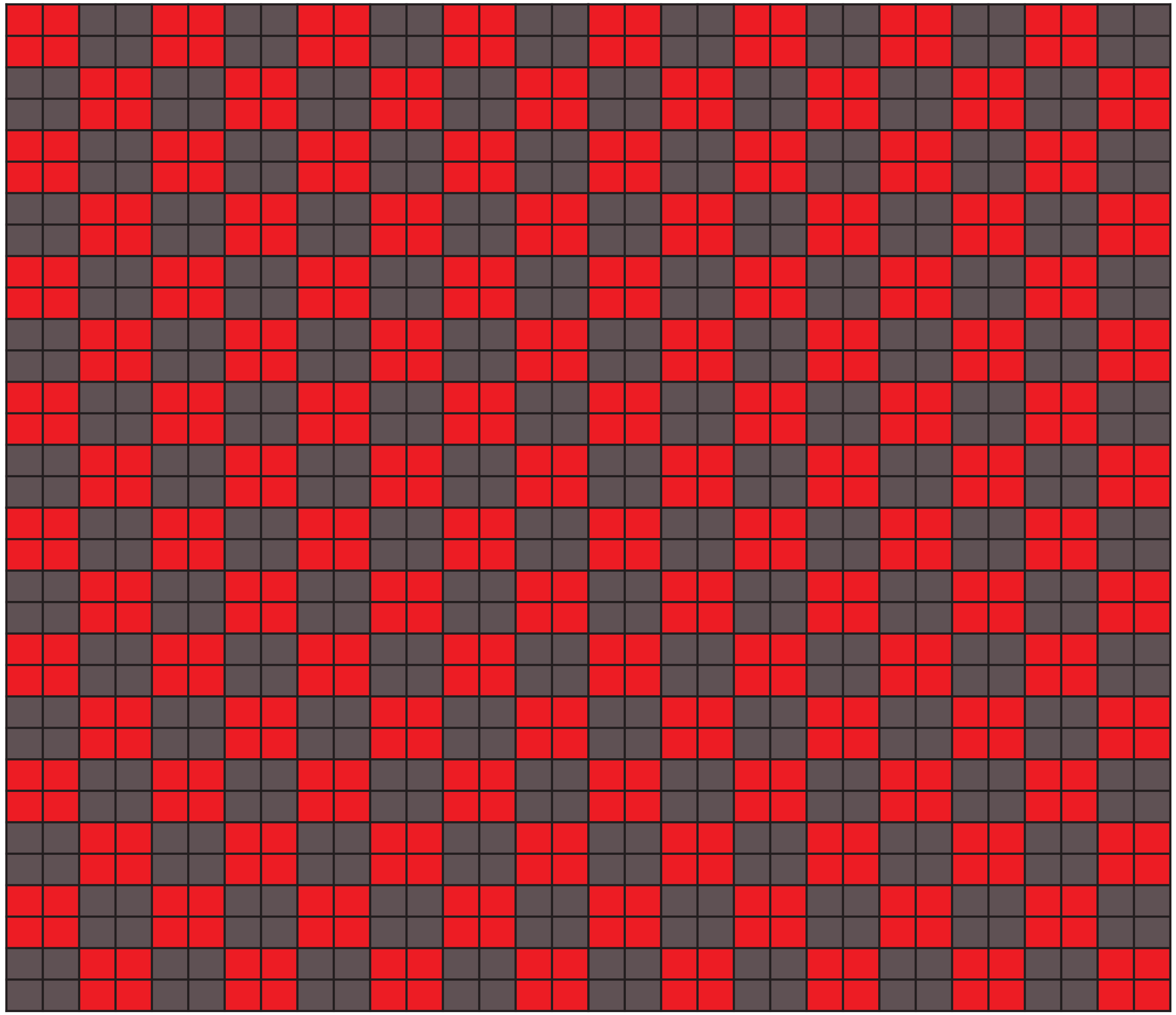
How to compute mip 2?



LDS

```
groupshared float4 spd_lds[32][32];
```

We need to store to and load from every entry in the LDS-array



WORK DISTRIBUTION

From mip 1 to mip 2, we need all threads:

```
Load(threadId.x * 2, threadId.y * 2)
Load(threadId.x * 2 + 1, threadId.y * 2)
Load(threadId.x * 2, threadId.y * 2 + 1)
Load(threadId.x * 2 + 1, threadId.y * 2 + 1)
```

WORK DISTRIBUTION

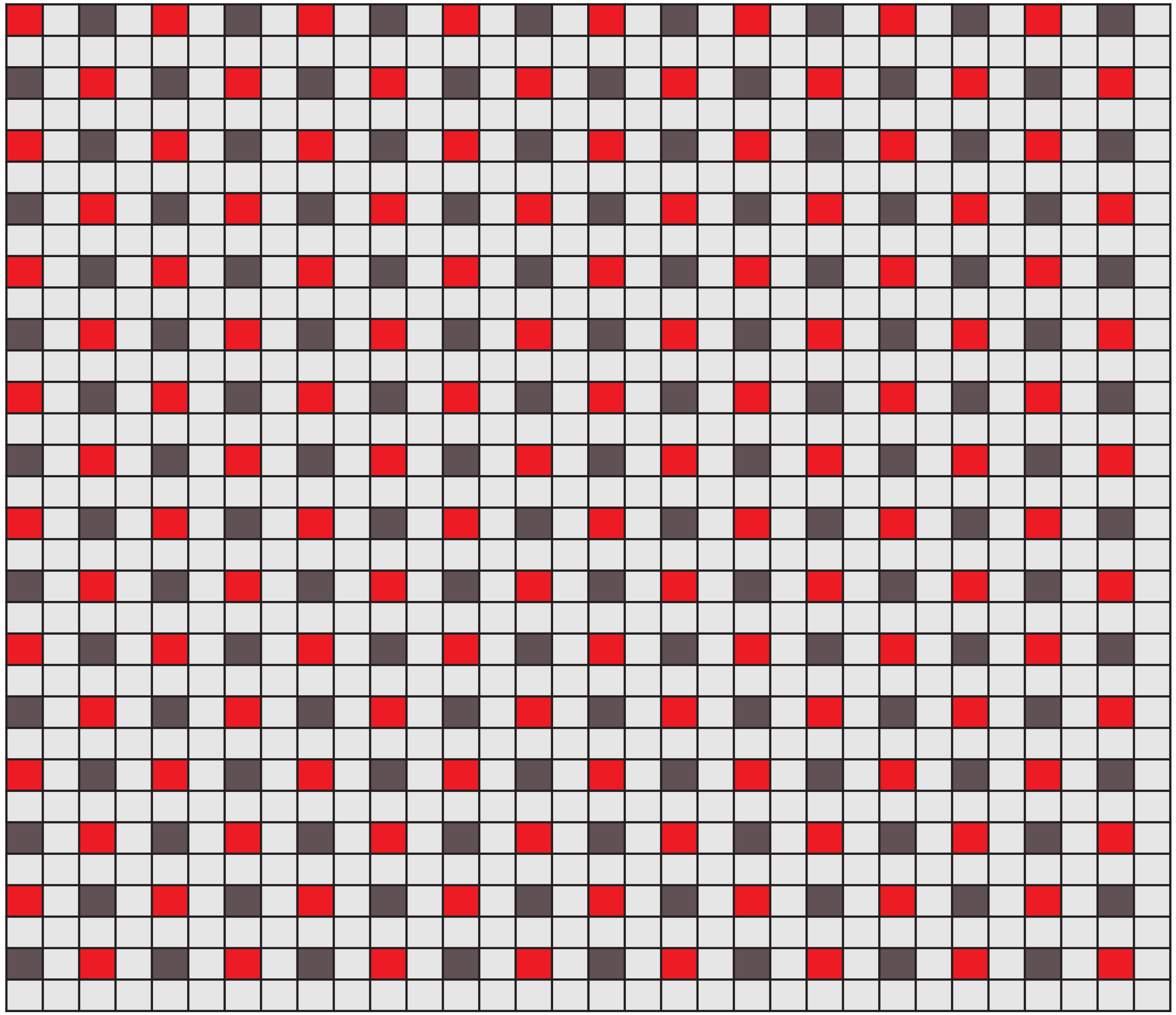
From mip 1 to mip 2, active threads

0	1	8	9	16	17	24	25	64	65	72	73	80	81	88	89
2	3	10	11	18	19	26	27	66	67	74	75	82	83	90	91
4	5	12	13	20	21	28	29	68	69	76	77	84	85	92	93
6	7	14	15	22	23	30	31	70	71	78	79	86	87	94	95
32	33	40	41	48	49	56	57	96	97	104	105	112	113	120	121
34	35	42	43	50	51	58	59	98	99	106	107	114	115	122	123
36	37	44	45	52	53	60	61	100	101	108	109	116	117	124	125
38	39	46	47	54	55	62	63	102	103	110	111	118	119	126	127
128	129	136	137	144	145	152	153	192	193	200	201	208	209	216	217
130	131	138	139	146	147	154	155	194	195	202	203	210	211	218	219
132	133	140	141	148	149	156	157	196	197	204	205	212	213	220	221
134	135	142	143	150	151	158	159	198	199	206	207	214	215	222	223
160	161	168	169	176	177	184	185	224	225	232	233	240	241	248	249
162	163	170	171	178	179	186	187	226	227	234	235	242	243	250	251
164	165	172	173	180	181	188	189	228	229	236	237	244	245	252	253
166	167	174	175	182	183	190	191	230	231	238	239	246	247	254	255

LDS

The result of one 2x2 tile can be stored in entry **(0,0),** (1,0), (0,1) or (1,1)

- We can overwrite these entries since we just used them to compute our result
  No additional synchronization is needed!



WORK DISTRIBUTION

Due to the nature of a downsampler, the further we go down the mip chain, the less threads we need

→ Which threads are we keeping active, which ones not?

For Mip 3, we only need every **4th thread**

```
If (threadIndex % 4 == 0){
   Load(threadId.x * 2, threadId.y * 2)
   Load(threadId.x * 2 + 2, threadId.y * 2)
   Load(threadId.x * 2, threadId.y * 2 + 2)
   Load(threadId.x * 2 + 2, threadId.y * 2 + 2)}
```

WORK DISTRIBUTION

From mip 1 to mip 2, active threads

0	**1**	**8**	**9**	**16**	**17**	**24**	**25**	**64**	**65**	**72**	**73**	**80**	**81**	**88**	**89**
2	**3**	**10**	**11**	**18**	**19**	**26**	**27**	**66**	**67**	**74**	**75**	**82**	**83**	**90**	**91**
4	**5**	**12**	**13**	**20**	**21**	**28**	**29**	**68**	**69**	**76**	**77**	**84**	**85**	**92**	**93**
6	**7**	**14**	**15**	**22**	**23**	**30**	**31**	**70**	**71**	**78**	**79**	**86**	**87**	**94**	**95**
32	**33**	**40**	**41**	**48**	**49**	**56**	**57**	**96**	**97**	**104**	**105**	**112**	**113**	**120**	**121**
34	**35**	**42**	**43**	**50**	**51**	**58**	**59**	**98**	**99**	**106**	**107**	**114**	**115**	**122**	**123**
36	**37**	**44**	**45**	**52**	**53**	**60**	**61**	**100**	**101**	**108**	**109**	**116**	**117**	**124**	**125**
38	**39**	**46**	**47**	**54**	**55**	**62**	**63**	**102**	**103**	**110**	**111**	**118**	**119**	**126**	**127**
128	**129**	**136**	**137**	**144**	**145**	**152**	**153**	**192**	**193**	**200**	**201**	**208**	**209**	**216**	**217**
130	**131**	**138**	**139**	**146**	**147**	**154**	**155**	**194**	**195**	**202**	**203**	**210**	**211**	**218**	**219**
132	**133**	**140**	**141**	**148**	**149**	**156**	**157**	**196**	**197**	**204**	**205**	**212**	**213**	**220**	**221**
134	**135**	**142**	**143**	**150**	**151**	**158**	**159**	**198**	**199**	**206**	**207**	**214**	**215**	**222**	**223**
160	**161**	**168**	**169**	**176**	**177**	**184**	**185**	**224**	**225**	**232**	**233**	**240**	**241**	**248**	**249**
162	**163**	**170**	**171**	**178**	**179**	**186**	**187**	**226**	**227**	**234**	**235**	**242**	**243**	**250**	**251**
164	**165**	**172**	**173**	**180**	**181**	**188**	**189**	**228**	**229**	**236**	**237**	**244**	**245**	**252**	**253**
166	**167**	**174**	**175**	**182**	**183**	**190**	**191**	**230**	**231**	**238**	**239**	**246**	**247**	**254**	**255**

From mip 2 to mip 3, active threads

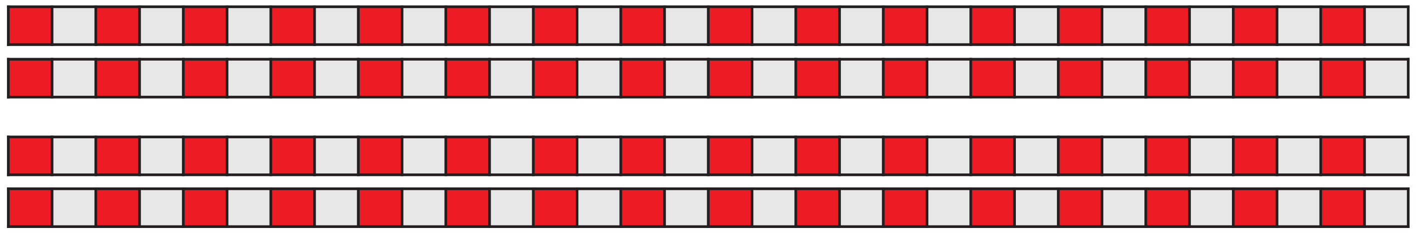
0	1	**8**	9	**16**	17	**24**	25	**64**	65	**72**	73	**80**	81	**88**	89
2	3	10	11	18	19	26	27	66	67	74	75	82	83	90	91
4	5	**12**	13	**20**	21	**28**	29	**68**	69	**76**	77	**84**	85	**92**	93
6	7	14	15	22	23	30	31	70	71	78	79	86	87	94	95
32	33	**40**	41	**48**	49	**56**	57	**96**	97	**104**	105	**112**	113	**120**	121
34	35	42	43	50	51	58	59	98	99	106	107	114	115	122	123
36	37	**44**	45	**52**	53	**60**	61	**100**	101	**108**	109	**116**	117	**124**	125
38	39	46	47	54	55	62	63	102	103	110	111	118	119	126	127
128	129	**136**	137	**144**	145	**152**	153	**192**	193	**200**	201	**208**	209	**216**	217
130	131	138	139	146	147	154	155	194	195	202	203	210	211	218	219
132	133	**140**	141	**148**	149	**156**	157	**196**	197	**204**	205	**212**	213	**220**	221
134	135	142	143	150	151	158	159	198	199	206	207	214	215	222	223
160	161	**168**	169	**176**	177	**184**	185	**224**	225	**232**	233	**240**	241	**248**	249
162	163	170	171	178	179	186	187	226	227	234	235	242	243	250	251
164	165	**172**	173	**180**	181	**188**	189	**228**	229	**236**	237	**244**	245	**252**	253
166	167	174	175	182	183	190	191	230	231	238	239	246	247	254	255

WORK DISTRIBUTION

Looks a lot like our previous example … ☹

🟥 0	1	🟥 8	9	🟥 16	17	🟥 24	25
2	3	10	11	18	19	26	27
🟥 4	5	🟥 12	13	🟥 20	21	🟥 28	29
6	7	14	15	22	23	30	31
🟥 32	33	🟥 40	41	🟥 48	49	🟥 56	57
34	35	42	43	50	51	58	59
🟥 36	37	🟥 44	45	🟥 52	53	🟥 60	61
38	39	46	47	54	55	62	63

🟥 64	65	🟥 72	73	🟥 80	81	🟥 88	89
66	67	74	75	82	83	90	91
🟥 68	69	🟥 76	77	🟥 84	85	🟥 92	93
70	71	78	79	86	87	94	95
🟥 96	97	🟥 104	105	🟥 112	113	🟥 120	121
98	99	106	107	114	115	122	123
🟥 100	101	🟥 108	109	🟥 116	117	🟥 124	125
102	103	110	111	118	119	126	127



```
v_add_f32 v0, v1, v2
  v_mov_b32 v3, v4

v_add_f32 v0, v1, v2
  v_mov_b32 v3, v4
```

WORK DISTRIBUTION

Due to the nature of a downsampler, the further we go down the mip chain, the less threads we need

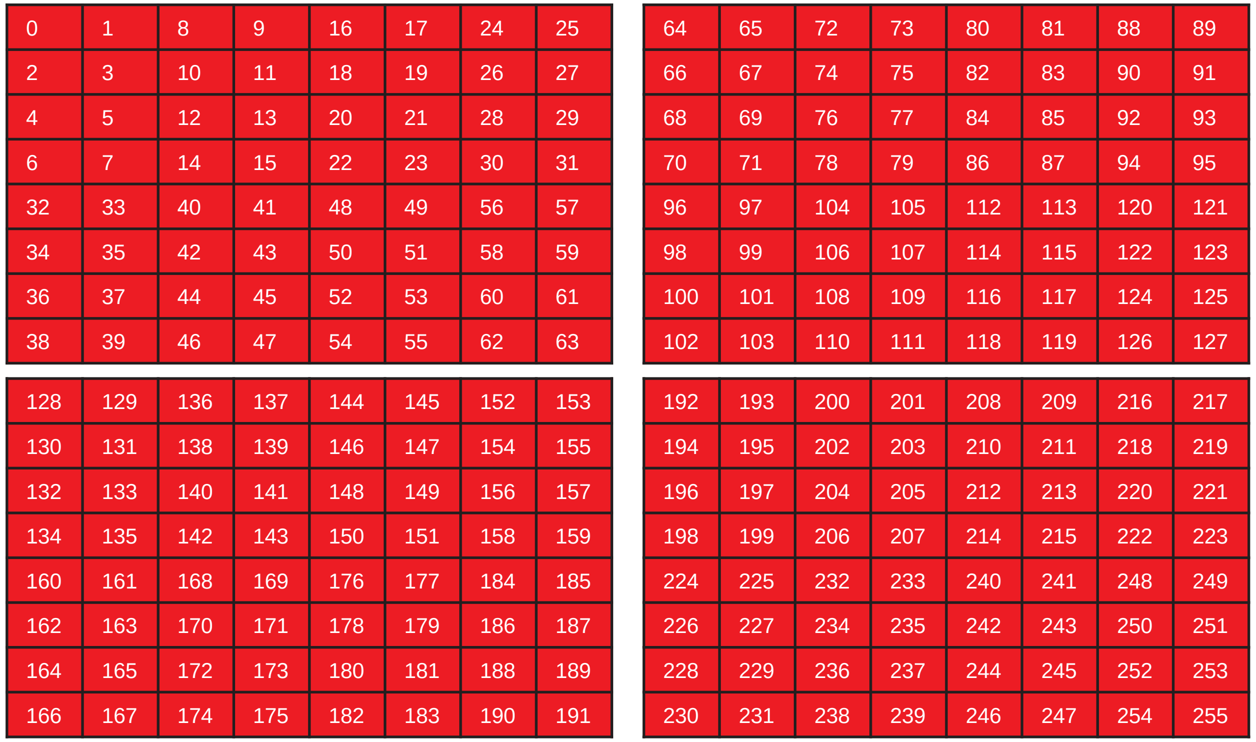
→ Which threads are we keeping active, which ones not?

For Mip 3, we only need every **4th thread**

```
If (threadIndex < 64 == 0){
   Load(threadId.x * 4, threadId.y * 4)
   Load(threadId.x * 4 + 2, threadId.y * 4)
   Load(threadId.x * 4, threadId.y * 4 + 2)
   Load(threadId.x * 4 + 2, threadId.y * 4 + 2)}
```

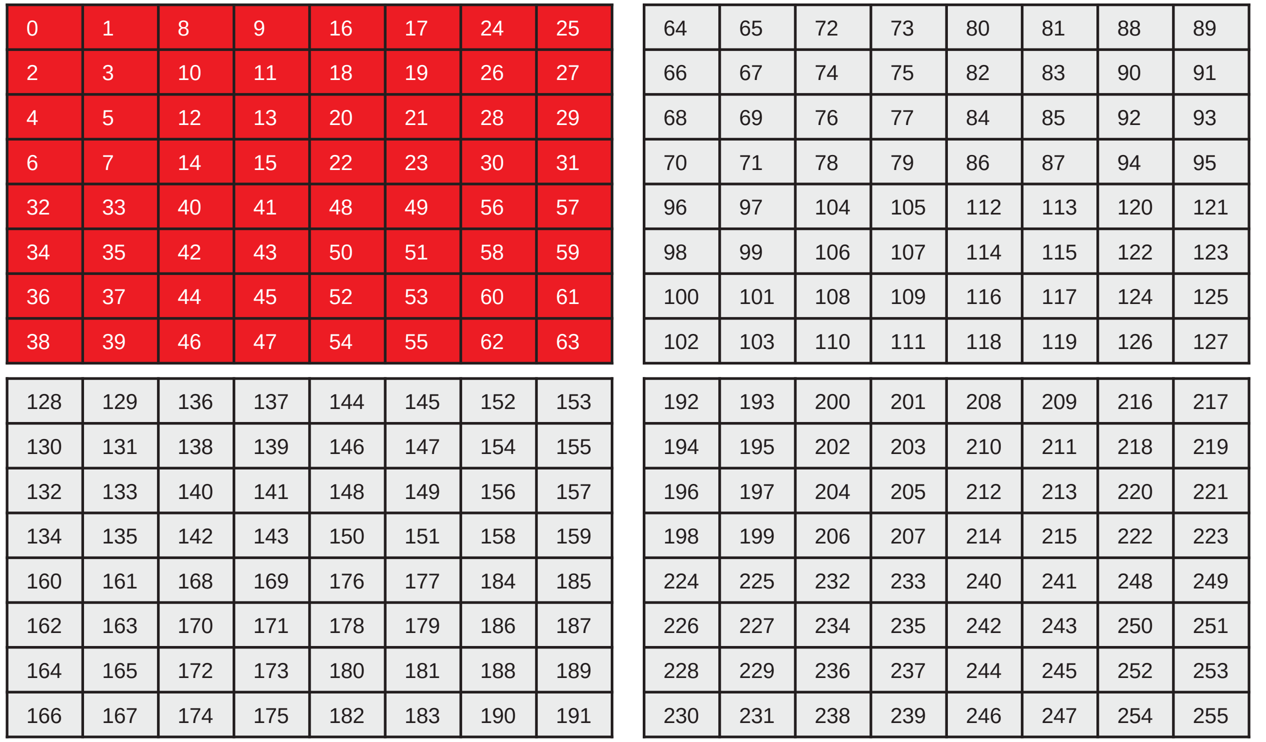
WORK DISTRIBUTION

From mip 1 to mip 2, active threads



0	1	8	9	16	17	24	25	64	65	72	73	80	81	88	89
2	3	10	11	18	19	26	27	66	67	74	75	82	83	90	91
4	5	12	13	20	21	28	29	68	69	76	77	84	85	92	93
6	7	14	15	22	23	30	31	70	71	78	79	86	87	94	95
32	33	40	41	48	49	56	57	96	97	104	105	112	113	120	121
34	35	42	43	50	51	58	59	98	99	106	107	114	115	122	123
36	37	44	45	52	53	60	61	100	101	108	109	116	117	124	125
38	39	46	47	54	55	62	63	102	103	110	111	118	119	126	127
128	129	136	137	144	145	152	153	192	193	200	201	208	209	216	217
130	131	138	139	146	147	154	155	194	195	202	203	210	211	218	219
132	133	140	141	148	149	156	157	196	197	204	205	212	213	220	221
134	135	142	143	150	151	158	159	198	199	206	207	214	215	222	223
160	161	168	169	176	177	184	185	224	225	232	233	240	241	248	249
162	163	170	171	178	179	186	187	226	227	234	235	242	243	250	251
164	165	172	173	180	181	188	189	228	229	236	237	244	245	252	253
166	167	174	175	182	183	190	191	230	231	238	239	246	247	254	255

From mip 2 to mip 3, active threads



0	1	8	9	16	17	24	25	64	65	72	73	80	81	88	89
2	3	10	11	18	19	26	27	66	67	74	75	82	83	90	91
4	5	12	13	20	21	28	29	68	69	76	77	84	85	92	93
6	7	14	15	22	23	30	31	70	71	78	79	86	87	94	95
32	33	40	41	48	49	56	57	96	97	104	105	112	113	120	121
34	35	42	43	50	51	58	59	98	99	106	107	114	115	122	123
36	37	44	45	52	53	60	61	100	101	108	109	116	117	124	125
38	39	46	47	54	55	62	63	102	103	110	111	118	119	126	127
128	129	136	137	144	145	152	153	192	193	200	201	208	209	216	217
130	131	138	139	146	147	154	155	194	195	202	203	210	211	218	219
132	133	140	141	148	149	156	157	196	197	204	205	212	213	220	221
134	135	142	143	150	151	158	159	198	199	206	207	214	215	222	223
160	161	168	169	176	177	184	185	224	225	232	233	240	241	248	249
162	163	170	171	178	179	186	187	226	227	234	235	242	243	250	251
164	165	172	173	180	181	188	189	228	229	236	237	244	245	252	253
166	167	174	175	182	183	190	191	230	231	238	239	246	247	254	255

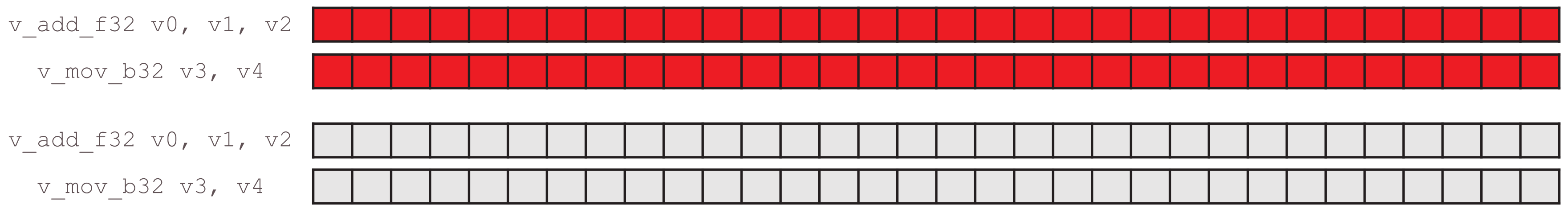
WORK DISTRIBUTION

Downsampling 2160x3840, RGBA16 FLOAT:

~7% performance gain compared to previous strategy

0	1	8	9	16	17	24	25
2	3	10	11	18	19	26	27
4	5	12	13	20	21	28	29
6	7	14	15	22	23	30	31
32	33	40	41	48	49	56	57
34	35	42	43	50	51	58	59
36	37	44	45	52	53	60	61
38	39	46	47	54	55	62	63

64	65	72	73	80	81	88	89
66	67	74	75	82	83	90	91
68	69	76	77	84	85	92	93
70	71	78	79	86	87	94	95
96	97	104	105	112	113	120	121
98	99	106	107	114	115	122	123
100	101	108	109	116	117	124	125
102	103	110	111	118	119	126	127



System specs:
Radeon^{TM} RX 5700 XT
AMD Radeon^{TM} driver 20.1.4
Ryzen 9 3900X

SHADER OPTIMIZATIONS

DATA EXCHANGE BETWEEN THE MIPS

What about using DPP / LDS to exchange data between the threads?

Idea:

Access the values of the other threads within a wavefront using using wave operations

Each wavefront downsamples a 32^2 patch down to 1^2 using ShuffleXor

→ LDS is then needed to shuffle the 4 output values across the 4 threadgroups

DATA EXCHANGE BETWEEN THE MIPS

What about using DPP / LDS to exchange data between the threads?

Idea:
Access the values of the other threads within a wavefront using using wave operations
Each wavefront downsamples a 32^2 patch down to 1^2 using ShuffleXor
→ LDS is then needed to shuffle the 4 output values across the 4 threadgroups

⚠ Assumes a thread group size of 64 🤔

DATA EXCHANGE BETWEEN THE MIPS

What about using DPP / LDS to exchange data between the threads?

Idea:
Access the values of the other threads within a wavefront using using wave operations
Each wavefront downsamples a 32^2 patch down to 1^2 using ShuffleXor
→ LDS is then needed to shuffle the 4 output values across the 4 threadgroups

⚠ Assumes a thread group size of 64 🤔

→ This is potentially not a problem!
→ In **Vulkan**®, when using subgroup operations, the wavefront size is fixed to 64
→ ShuffleXor IS a subgroup operation -> wavefront size is 64

DATA EXCHANGE BETWEEN THE MIPS

What about using DPP / LDS to exchange data between the threads?

Idea:
Access the values of the other threads within a wavefront using using wave operations
Each wavefront downsamples a 32^2 patch down to 1^2 using ShuffleXor
→ LDS is then needed to shuffle the 4 output values across the 4 threadgroups

⚠ Assumes a thread group size of 64 🤔

→ This is potentially not a problem!
→ In **Vulkan**®, when using subgroup operations, the wavefront size is fixed to 64
→ ShuffleXor IS a subgroup operation -> wavefront size is 64

Not true for **DX12**!
If we use wave operations, it still can run either Wave32 or Wave64

DATA EXCHANGE BETWEEN THE MIPS

What about using DPP / LDS to exchange data between the threads?

Idea:

Access the values of the other threads within a wavefront using using wave operations

Each wavefront downsamples a 32^2 patch down to 1^2 using ShuffleXor

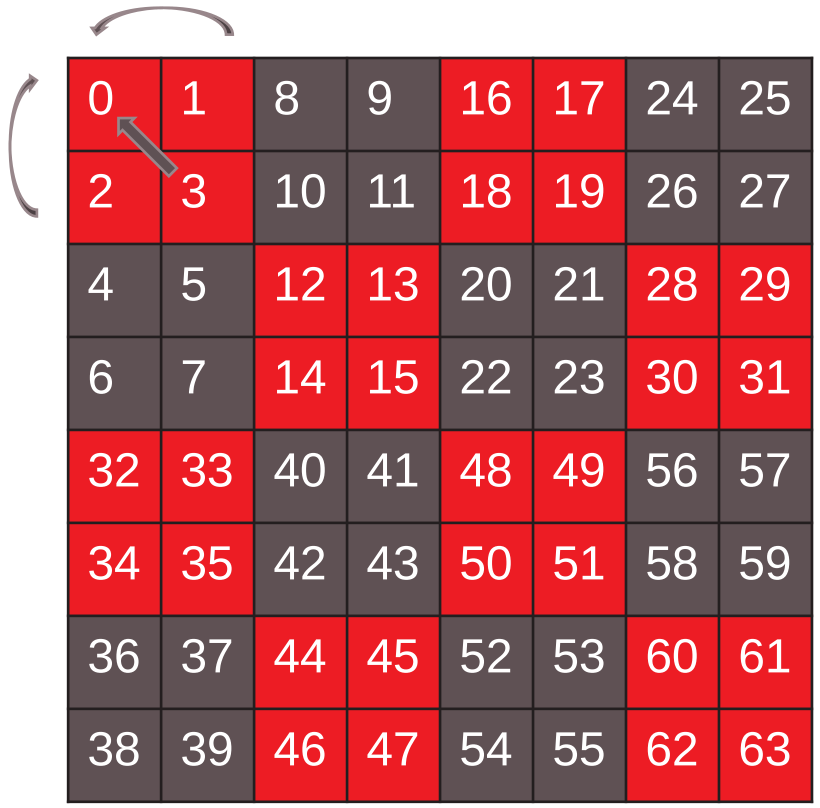
→ LDS is then needed to shuffle the 4 output values across the 4 threadgroups

⚠ Assumes a thread group size of 64 🤔

- → This is potentially not a problem!
- → In **Vulkan**®, when using subgroup operations, the wavefront size is fixed to 64
- → ShuffleXor IS a subgroup operation -> wavefront size is 64

In **Vulkan**®, the wavefront size is fixed to 64 unless you use an extension to enable variable subgroup size

QUAD SHUFFLE

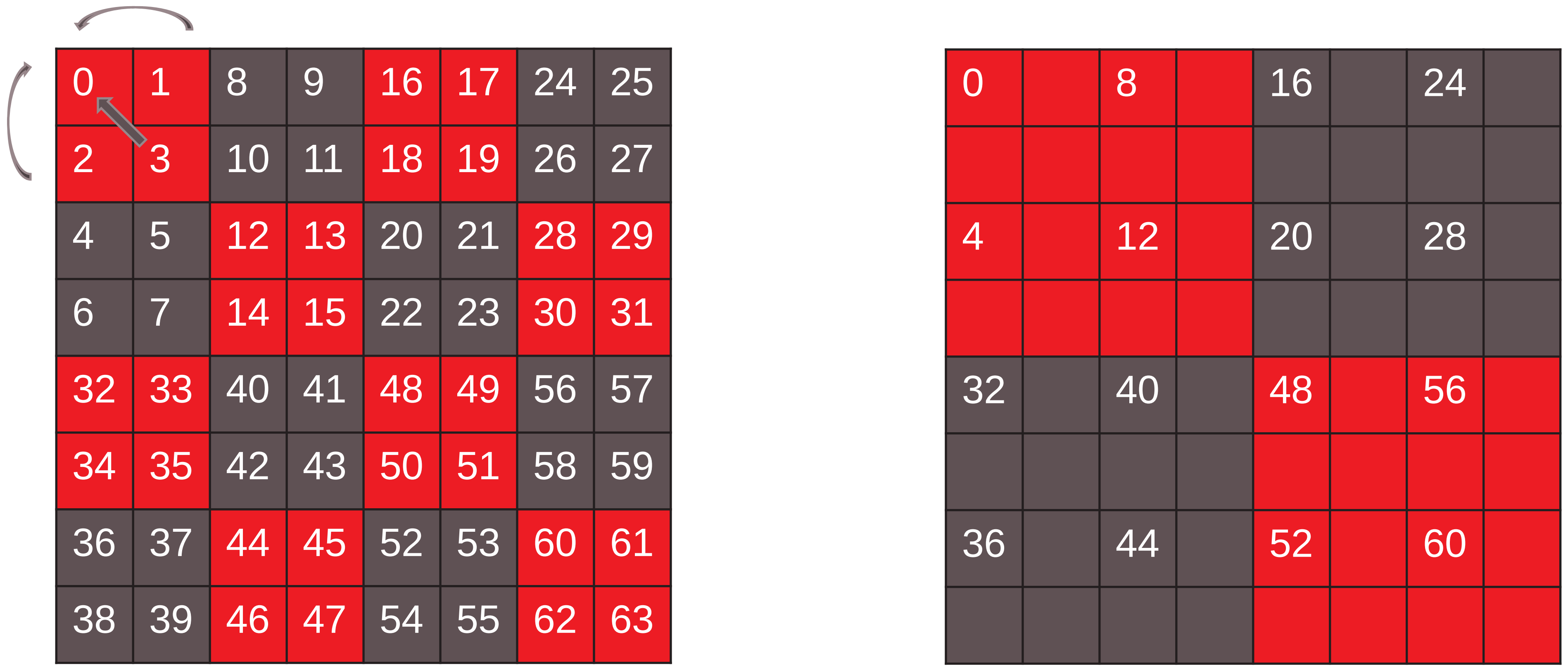


Thread 0 can access values of threads 1, 2, and 3 via ShuffleXor or Quad Operations

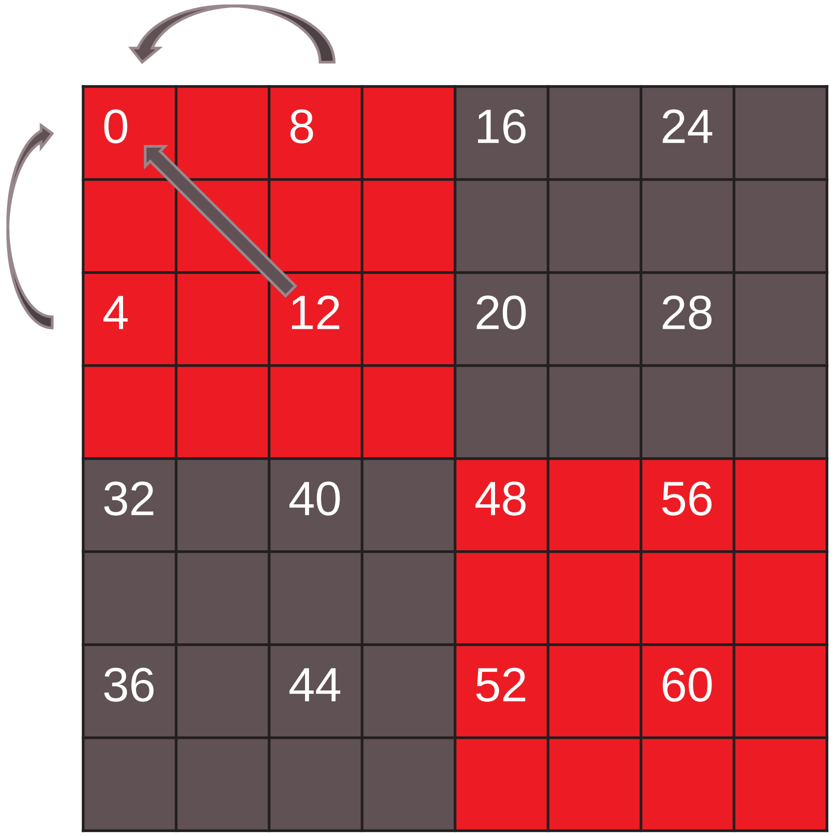
```
value += subgroupQuadSwapHorizontal(value);
value += subgroupQuadSwapVertical(value);
value *= 0.25;
```

ShuffleXor / Quad operations
- DPP limited to groups of 16 ✓
- Only shuffle across groups of 8 ✓
- Avoid shuffles across more than 32 threads ✓

QUAD SHUFFLE



SHUFFLEXOR

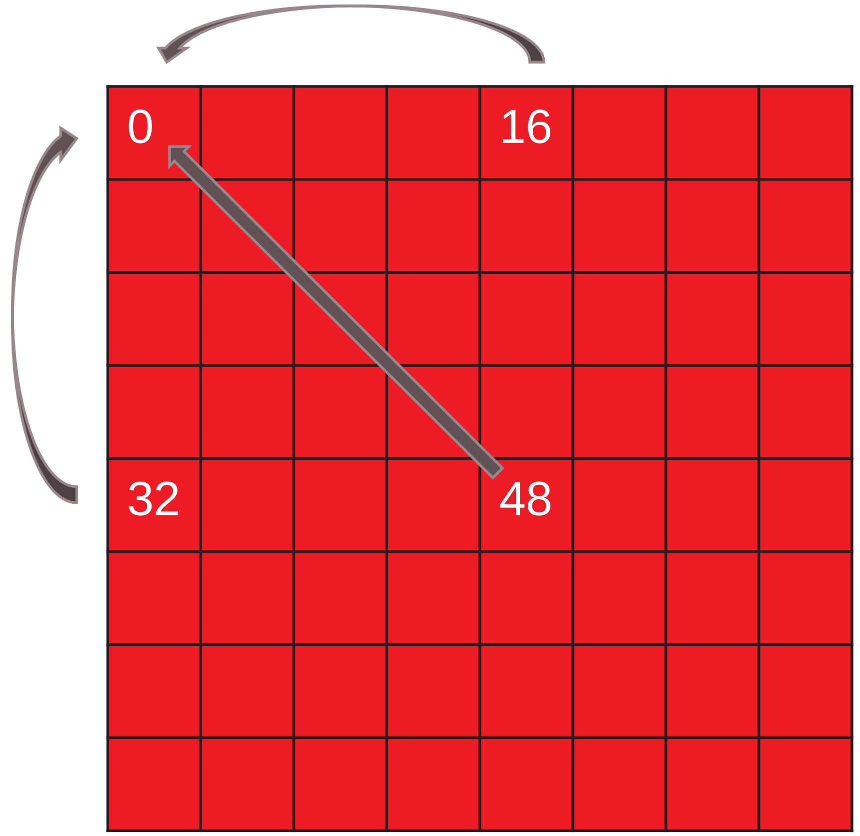


Thread 0 can access values of threads 4, 8, and 12 via ShuffleXor

```
value += subgroupShuffleXor(value,4);
value += subgroupShuffleXor(value,8);
value *= 0.25;
```

ShuffleXor / Quad operations
- DPP limited to groups of 16 ✓
- Only shuffle across groups of 8 ✗
- Avoid shuffles across more than 32 threads ✓

SHUFFLEXOR



Thread 0 can access values of threads 16, 32, and 48 via ShuffleXor or Quad Operations

```
value += subgroupShuffleXor(value,16);
value += subgroupShuffleXor(value,32);
value *= 0.25;
```

ShuffleXor / Quad operations

- DPP limited to groups of 16 ❌
- Only shuffle across groups of 8 ❌
- Avoid shuffles across more than 32 threads ❌

SHUFFLE ACROSS 64 THREADS

Impact varies between up to ~10% performance **drop** and a speed-up of about ~2%
→ Not a real improvement

Another problem: Requires wavefront size of 64. Not all GPUs are running wavefront size 64.

"Good" property:

- Requires less LDS

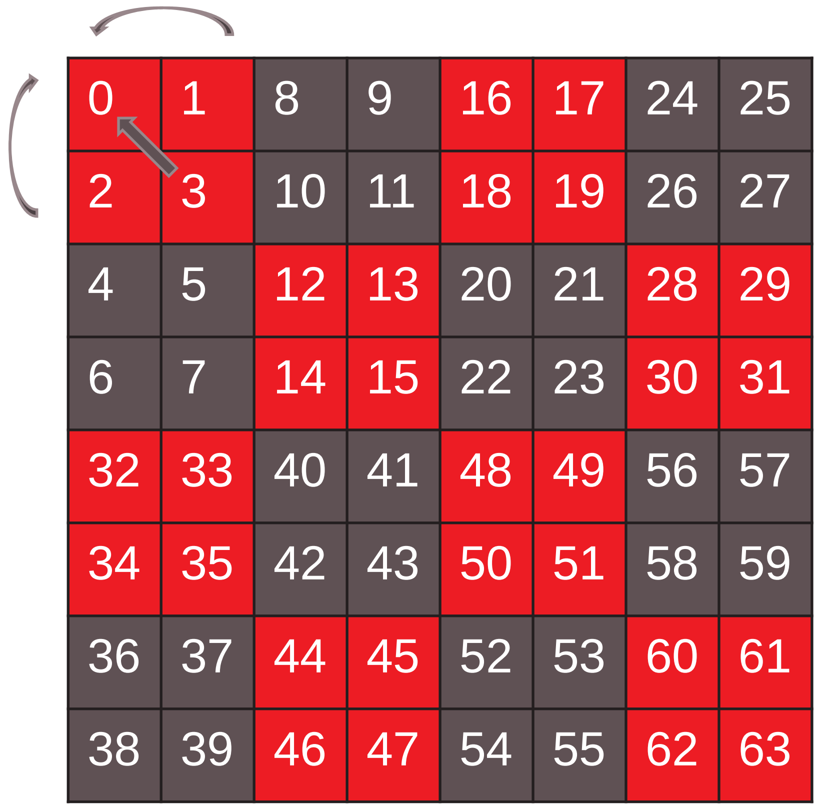
DATA EXCHANGE BETWEEN THE MIPS

What about using DPP / LDS permute to exchange data between the threads?

Use only quad reductions

-> LDS needed between each step (except mip 1 and mip 2) to group the data to a quad

QUAD SHUFFLE



Saves one round of LDS store and load completely

And this is for a time consuming mip ☺

Quad Shuffle:
- DPP limited to groups of 16 ✓
- Only shuffle across groups of 8 ✓
- Avoid shuffles across more than 32 threads ✓

QUAD OPERATIONS – PERF. NUMBERS

On a Radeon^{TM} RX 5700 XT card, for texture formats RGBA8 UNORM and RGBA16 FLOAT, texture resolutions

- 1080p → ~5%
- 1440p → ~5%
- 2160p → ~1%

There is an average speed-up of **~3.5%**

System specs:
Radeon^{TM} RX 5700 XT
AMD Radeon^{TM} driver 20.1.4
Ryzen 9 3900X

QUAD OPERATIONS – PERF. NUMBERS

On a Radeon™ RX 5700 XT card, for texture formats RGBA8 UNORM and RGBA16 FLOAT, texture resolutions

- 1080p → ~5%
- 1440p → ~5%
- 2160p → ~1%

Performance improvement gets lower the higher the resolution gets.

For high resolutions we are mainly bandwidth bound by loading the source texture!

There is an average speed-up of **~3.5%**

System specs:
Radeon™ RX 5700 XT
AMD Radeon™ driver 20.1.4
Ryzen 9 3900X

QUAD OPERATIONS

It's very compiler dependant. But the more wave / subgroup operations we observe in the wild, the better it gets ☺

Some other nice things besides pure performance

- Requires less LDS
  - We do not need LDS between mip 1 and mip 2 (32x32 patch to 16x16 patch)
- Requires less VGPRs

→ Can be important factors when overlapping SPD with other passes

QUAD OPERATIONS

It's very compiler dependant. But the more wave / subgroup operations we observe in the wild, the better it gets ☺

Some other nice things besides pure performance

- Requires less LDS
  - We do not need LDS between mip 1 and mip 2 (32x32 patch to 16x16 patch)
- Requires less VGPRs

Radeon™ RX 5700 XT, Driver: 20.1.4	**Vulkan®**	**DX12**
No subgroup operations	**48** VGPRs	**45** VGPRs
Subgroup operations	**40** VGPRs	**41** VGPRs

→ Can be important factors when overlapping SPD with other passes

FP16

Since many textures have a format with bits per pixel (bpp) smaller or equal to 16bit, we can consider using FP16

On RDNA,

filtering of 4-channel FP16 textures is now full-rate ☺

→ Writing and reading back FP16 is efficient!

FP16 – EXAMPLE PERF. NUMBERS

FP16 proved to be beneficial especially for small resolution textures

For high resolutions no difference could be measured

On a Radeon™ RX 5700 XT card, for texture formats RGBA8 UNORM and RGBA16 FLOAT, texture resolutions

- 256^2 → ~40%
- 1024^2 → 15%
- 1080p → 0-2%
- 1440p → 0-2%

System specs:
Radeon™ RX 5700 XT
AMD Radeon™ driver 20.1.4
Ryzen 9 3900X

FP16 – EXAMPLE PERF. NUMBERS

FP16 proved to be beneficial especially for small resolution textures

For high resolutions no difference could be measured

On a Radeon™ RX 5700 XT card, for texture formats RGBA8 UNORM and RGBA16 FLOAT, texture resolutions

- 256^2 → ~40%
- 1024^2 → 15%
- 1080p → 0-2%
- 1440p → 0-2%

Performance improvement gets lower the higher the resolution gets.

For high resolutions we are mainly bandwidth bound by loading the source texture!

System specs:
Radeon™ RX 5700 XT
AMD Radeon™ driver 20.1.4
Ryzen 9 3900X

FP16 – EXAMPLE PERF. NUMBERS

Same as with wave / subgroup operations, we have nice properties on top:

- Requires less LDS
- Requires less VGPRs – show numbers

Radeon™ RX 5700 XT, Driver: 20.1.4	**Vulkan®**	**DX12**
No subgroup operations	**48** VGPRs	**45** VGPRs
No subgroup operations – FP16	**38** VGPRs	**40** VGPRs
Subgroup operations	**40** VGPRs	**41** VGPRs
Subgroup operations – FP16	**28** VGPRs	**37** VGPRs

→ Can be important factors when overlapping SPD with other passes

SUMMARY

RDNA	GCN
WGP	CU
L0, **L1**, L2, L3	L1, L2, L3
Wave32 **native**, Wave64	Wave64 (4x SIMD16)
Single cycle instruction	**Four** cycle instruction
4 triangles/clock (after culling), **>>** 4 (before culling)	**2**-4 triangles/clock (culled/unculled)

SUMMARY

- Re-calculate your thread indices using a Morton-like ordering
  - Keep the 2x2 pattern per thread

- Distribute your work to the threads so that you can skip entire waves as much as possible
  - Think here in terms of wavefront size 32 ☺

- Use subgroup operations when possible – but pay attention on your shuffle scheme
  - Don‘t shuffle across more than 32 threads, if possible stick to 8 threads

- Consider FP16 where applicable

Q&A

lou.kramer@amd.com

AMD GPUOpen

EPYC RADEON RYZEN AMD

DISCLAIMER & ATTRIBUTION