RDNA Architecture

Forward-looking statement

This presentation contains forward-looking statements concerning Advanced Micro Devices, Inc. (AMD) including, but not limited to, the features, functionality, performance, availability, timing, pricing, expectations and expected benefits of AMD's current and future products, which are made pursuant to the Safe Harbor provisions of the Private Securities Litigation Reform Act of 1995. Forward-looking statements are commonly identified by words such as "would," "may," "expects," "believes," "plans," "intends," "projects" and other terms with similar meaning. Investors are cautioned that the forward-looking statements in this presentation are based on current beliefs, assumptions and expectations, speak only as of the date of this presentation and involve risks and uncertainties that could cause actual results to differ materially from current expectations. Such statements are subject to certain known and unknown risks and uncertainties, many of which are difficult to predict and generally beyond AMD's control, that could cause actual results and other future events to differ materially from those expressed in, or implied or projected by, the forward-looking information and statements. Investors are urged to review in detail the risks and uncertainties in AMD's Securities and Exchange Commission filings, including but not limited to AMD's Quarterly Report on Form 10-Q for the quarter ended March 30, 2019

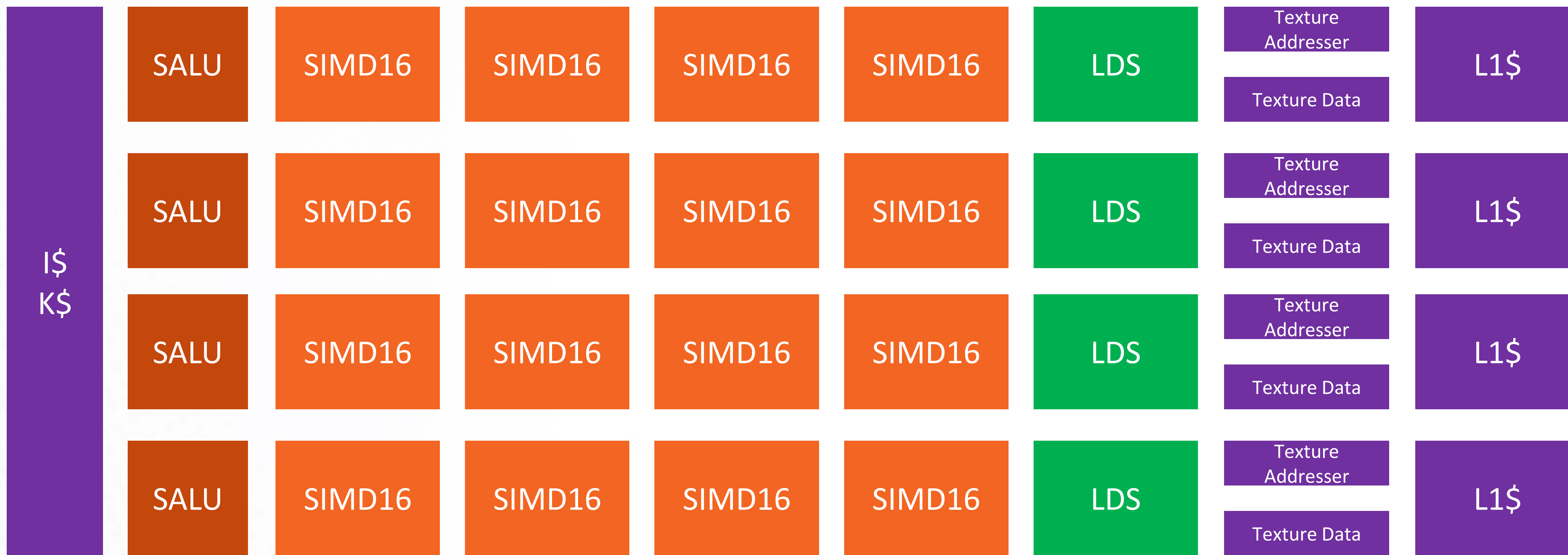
Highlights of the RDNA Workgroup Processor (WGP)

- Designed for lower latency and higher effective IPC
- Native Wave32 with support for Wave64 via dual-issue
- Single-cycle instruction issue
- Co-execution of transcendental arithmetic operations
- Resources of two Compute Units available to a single workgroup
- 2x scalar execution resources
- Vector memory improvements

AMD

GCN Compute Units

- 4 Compute Units:

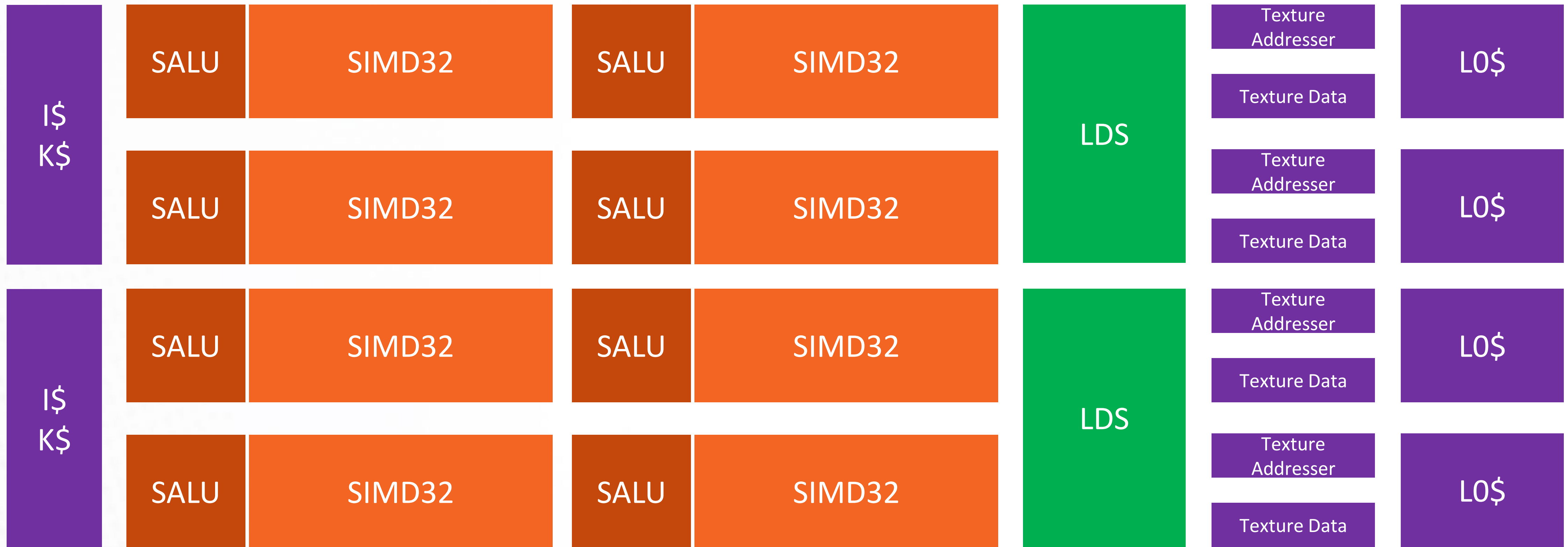


- RX590 has 36 CU, RX Vega64 has 64 CU

AMD

RDNA Workgroup Processors (WGP)

- 2 Workgroup Processors:



- “Navi” has 20 WGP, corresponding to 40 CU

AMD

4-cycle instruction issue on GCN

- Each wave is assigned to one SIMD16, up to 10 waves per SIMD16
- Each SIMD16 issues 1 instruction every 4 cycles
- Vector instructions throughput is 1 every 4 cycles

Cycle	0	1	2	3	4	5	6	7			
SALU	SIMD0	SIMD1	SIMD2	SIMD3	SIMD0	SIMD1	SIMD2	SIMD3			
SIMD0	0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63			
SIMD1		0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63		
SIMD2			0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63	
SIMD3				0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63

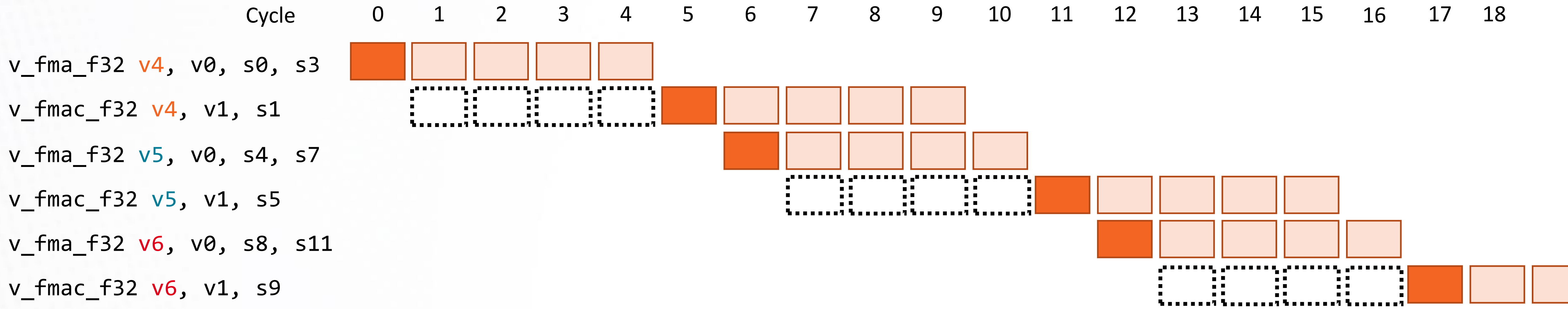
AMD

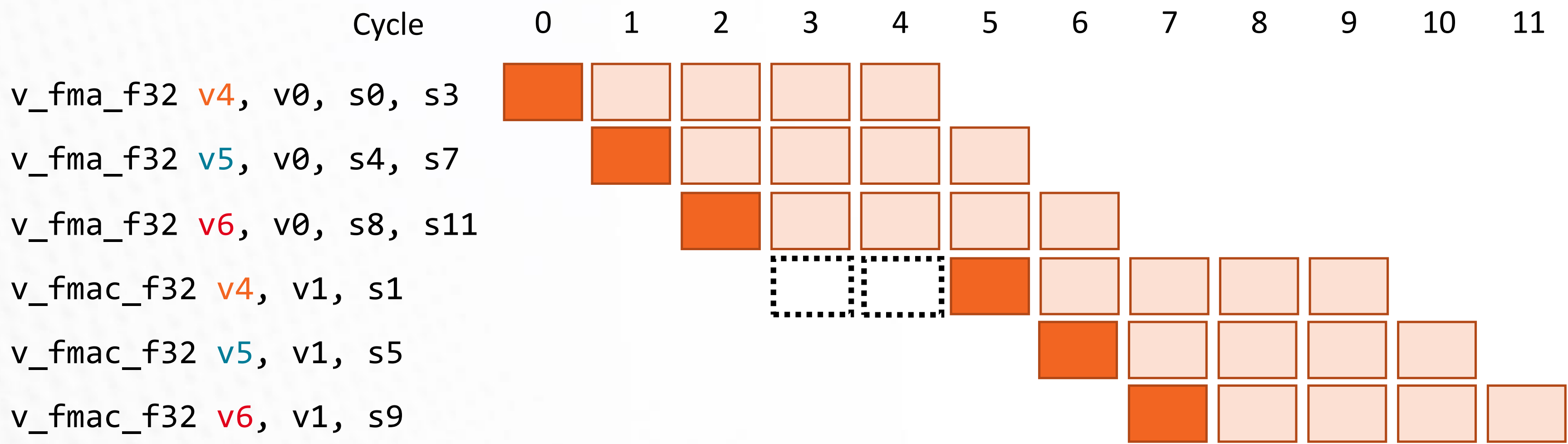
Single-cycle instruction issue on RDNA

- Each wave is assigned to one SIMD32, up to 20 waves per SIMD32
- Each SIMD32 issues 1 instruction every cycle
- Vector instruction throughput is 1 every cycle (for Wave32)
- 5 cycles of latency are exposed (automatic dependency check in hardware)
  - Dependency stalls can be filled by other waves

AMD

Single-cycle instruction issue: ILP and scheduling matters

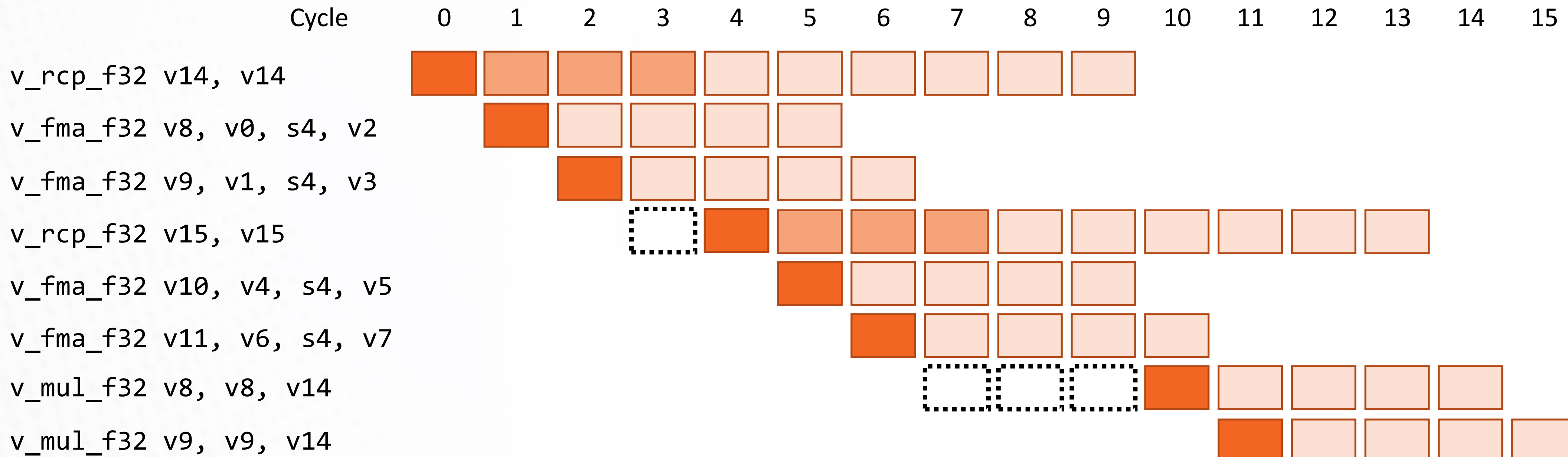




AMD

Transcendental math co-execution

- rcp/rsq/sqrt/log/exp/sin/cos
- Transcendental instructions are ¼ rate (like GCN)
- Non-transcendental instructions can execute in parallel

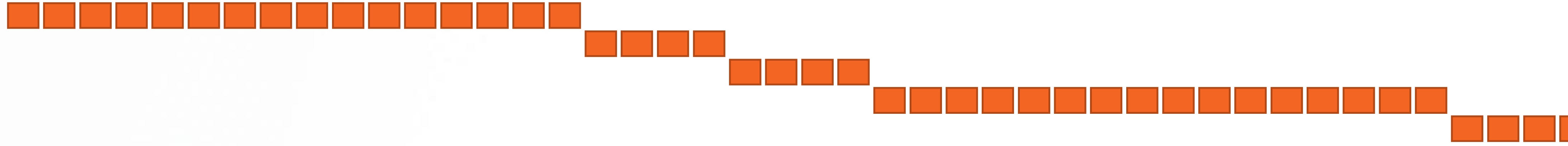


AMD

Instruction latency: GCN vs. RDNA

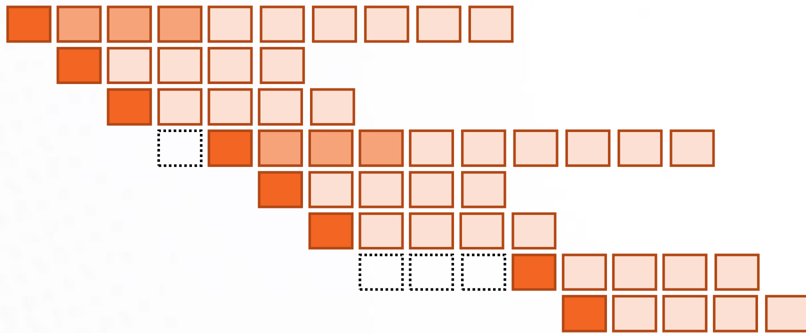
- Wave64 on GCN

```
v_rcp_f32 v14, v14
v_fma_f32 v8, v0, s4, v2
v_fma_f32 v9, v1, s4, v3
v_rcp_f32 v15, v15
v_fma_f32 v10, v4, s4, v5
v_fma_f32 v11, v6, s4, v7
v_mul_f32 v8, v8, v14
v_mul_f32 v9, v9, v14
```



- 2x Wave32 on RDNA – one per SIMD32

```
v_rcp_f32 v14, v14
v_fma_f32 v8, v0, s4, v2
v_fma_f32 v9, v1, s4, v3
v_rcp_f32 v15, v15
v_fma_f32 v10, v4, s4, v5
v_fma_f32 v11, v6, s4, v7
v_mul_f32 v8, v8, v14
v_mul_f32 v9, v9, v14
```



- With small dispatches, RDNA behaves significantly better than GCN

AMD

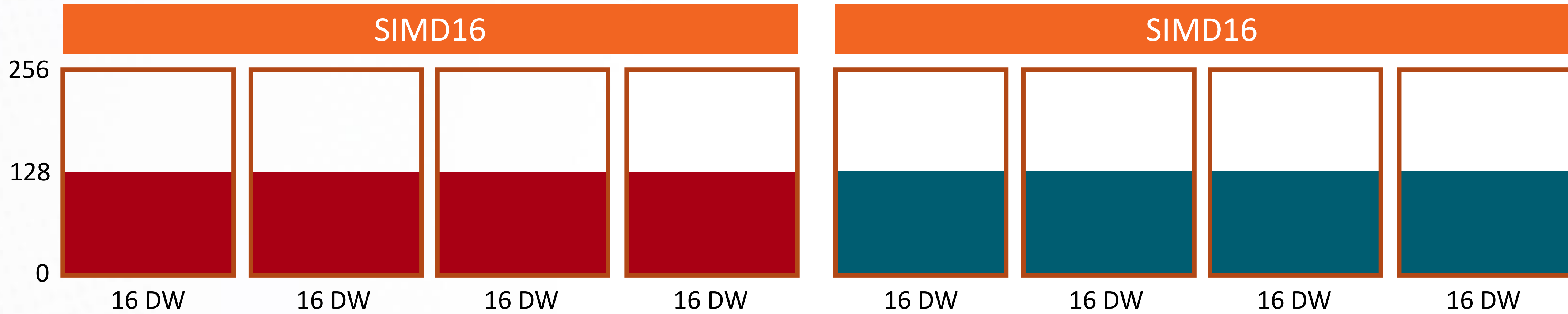
Vector register file (VGPRs)

- Each SIMD32 has 1024 physical registers
- Divided among waves, up to 256 each
- Wave64 "counts double"
- Examples:
  - 4x Wave32 with 256 VGPRs
  - 2x Wave64 with 256 VGPRs
  - 16x Wave32 with 64 VGPRs
  - 8x Wave64 with 64 VGPRs
- Occupancy in "# of threads per SIMD lane" is unchanged from GCN
  - Occupancy 4 on GCN = 16 threads per lane
  - RDNA equivalent: 16x Wave32 or 8x Wave64
- Call to action:
  - Think about occupancy in terms of "# of threads per SIMD lane"

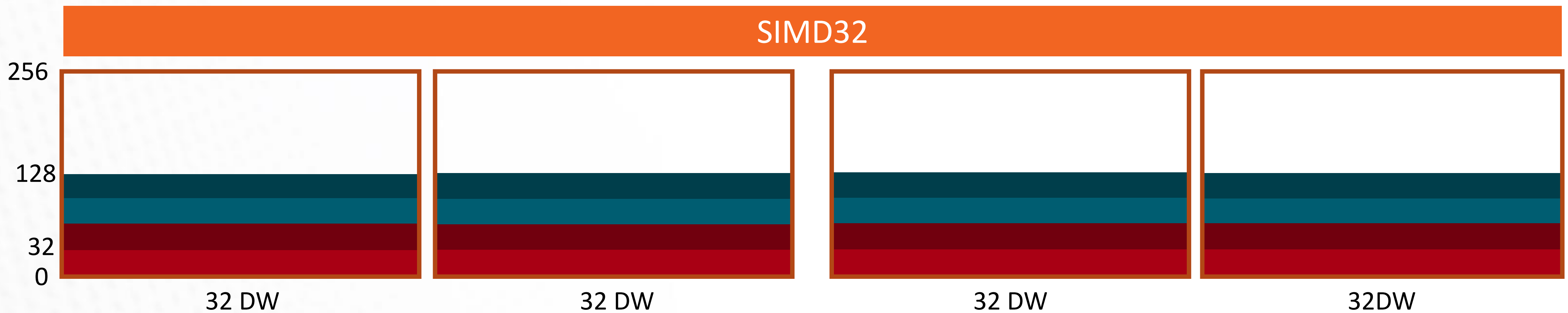
AMD

Vector register-based occupancy illustrated

- GCN: Wave64, 128 VGPR allocation



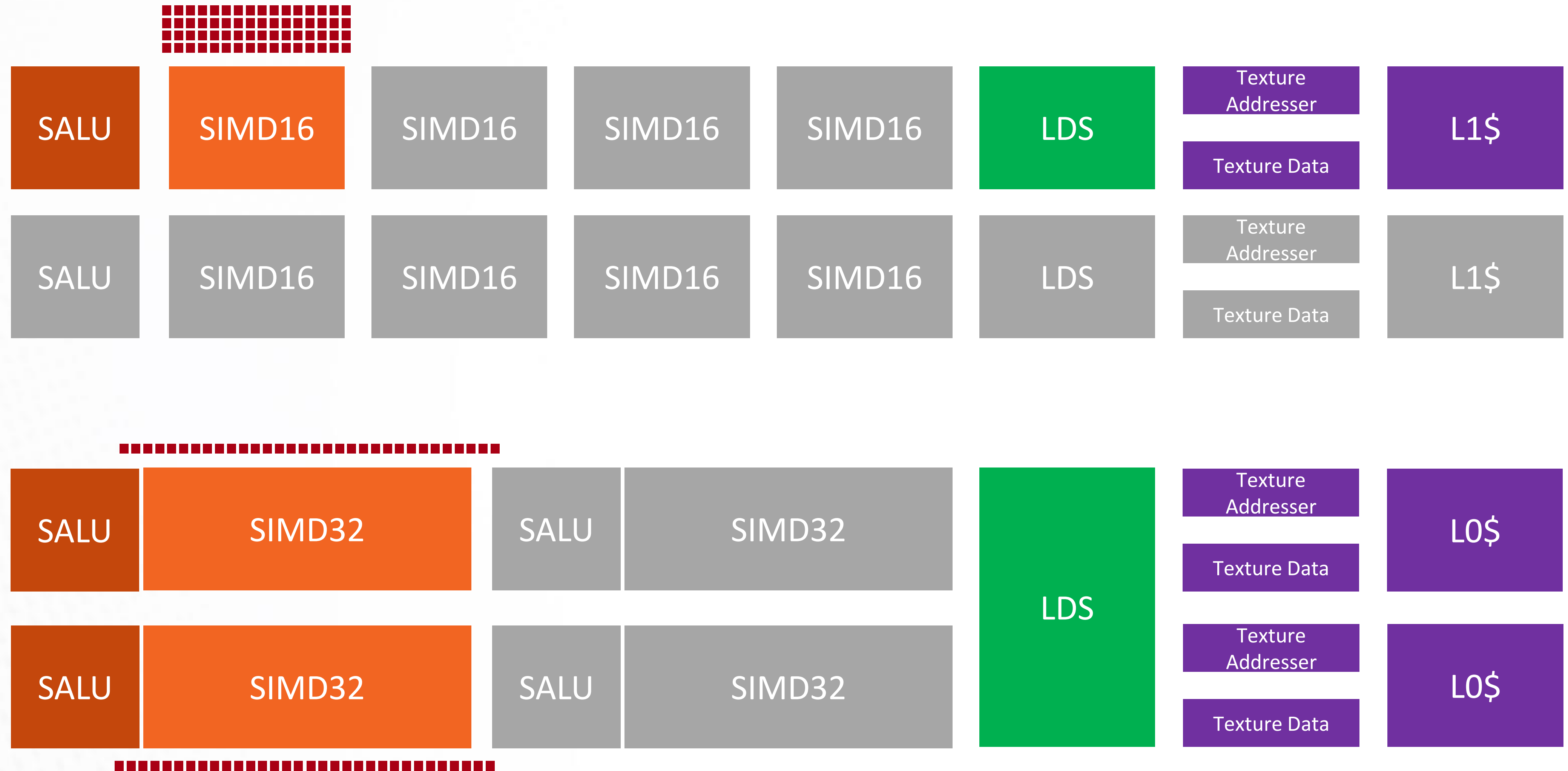
- RDNA: Wave32, 128 VGPR allocation



AMD

Keeping the SIMD busy: GCN vs. RDNA

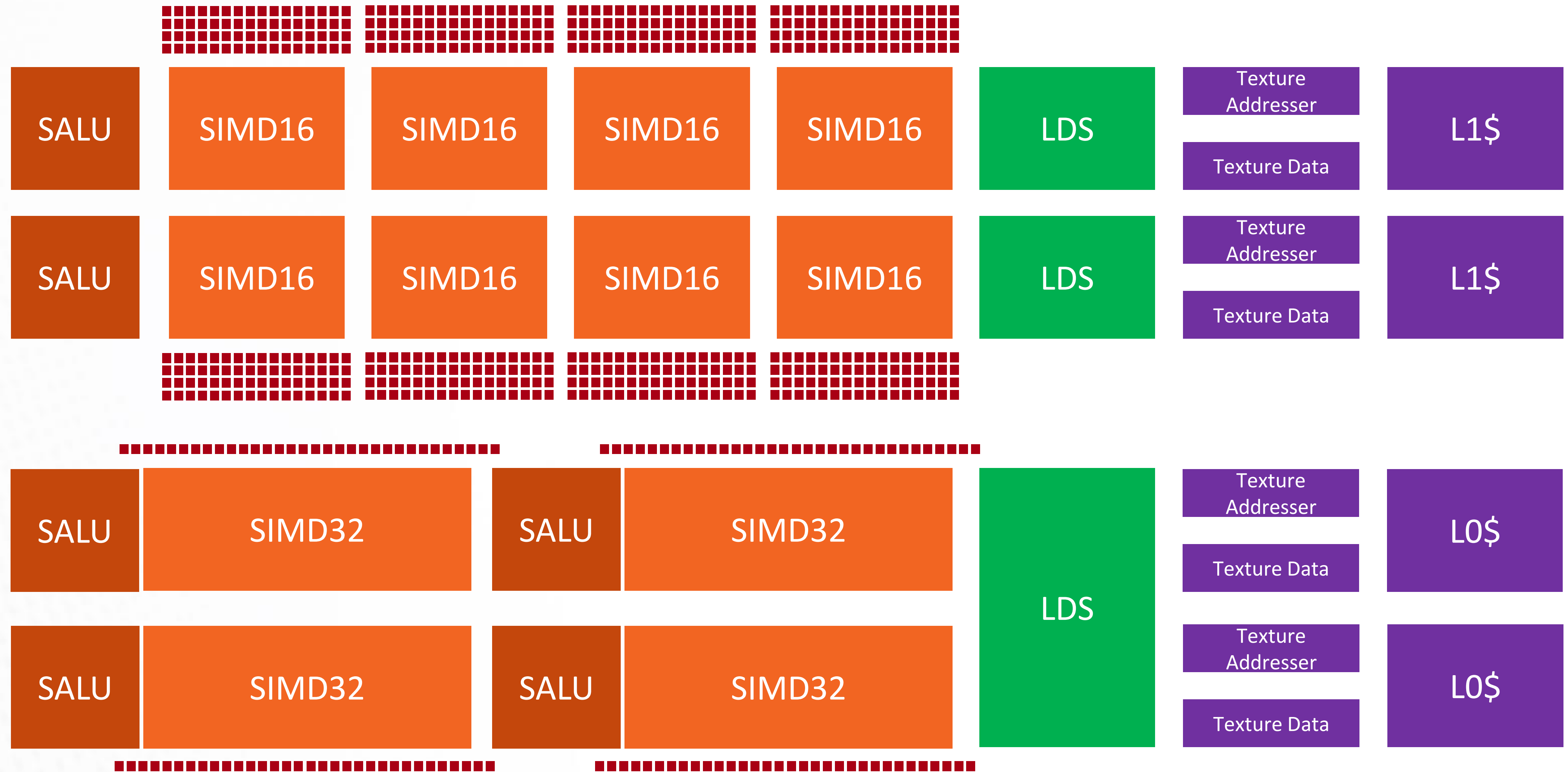
- Example: Small dispatch, 64 threads only



AMD

Keeping the SIMD busy: GCN vs. RDNA

- RDNA requires much fewer threads for all blocks to light up:



AMD

Keeping the SIMD busy: GCN vs. RDNA

- 2 CU require 2*4*64 = **512 threads** to be able to reach 100% ALU utilization
- WGP requires 4*32 = **128 threads** to be able to reach 100% ALU utilization
  - Only achieved with high instruction level parallelism (ILP)
  - Graphics workloads often have 3 independent streams (RGB / XYZ)
  - 256 threads / WGP often reach >90% ALU utilization in practice[1]
- Additional threads are needed on both GCN and RDNA to hide memory latency
  - Unless you can fill the wait with ALU (this is extremely rare)
  - # of threads required for memory latency hiding has reduced as well, but not as much
- Fewer threads required overall to keep the machine busy
  - Utilization ramps up more quickly after barriers
  - High VGPR counts hurt slightly less

- Call to action:
  - Keep ILP in mind when writing shader code

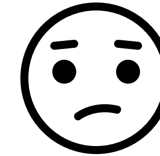
1 Observed in pixel shaders in a trace of a typical game rendering workload

What (**not**) to do for ILP

- Naïve idea: run multiple work items in a single thread

```
image_load v[0:1], …
s_waitcnt vmcnt(0)
v_mul_f32 v0, v0, s0
v_mul_f32 v1, v1, s0
image_store v[0:1], ...
```

```
image_load v[0:1], …
image_load v[2:3], …
s_waitcnt vmcnt(1)
v_mul_f32 v0, v0, s0
v_mul_f32 v1, v1, s0
s_waitcnt vmcnt(0)
v_mul_f32 v2, v2, s0
v_mul_f32 v3, v3, s0
image_store v[0:1], ...
image_store v[2:3], ...
```

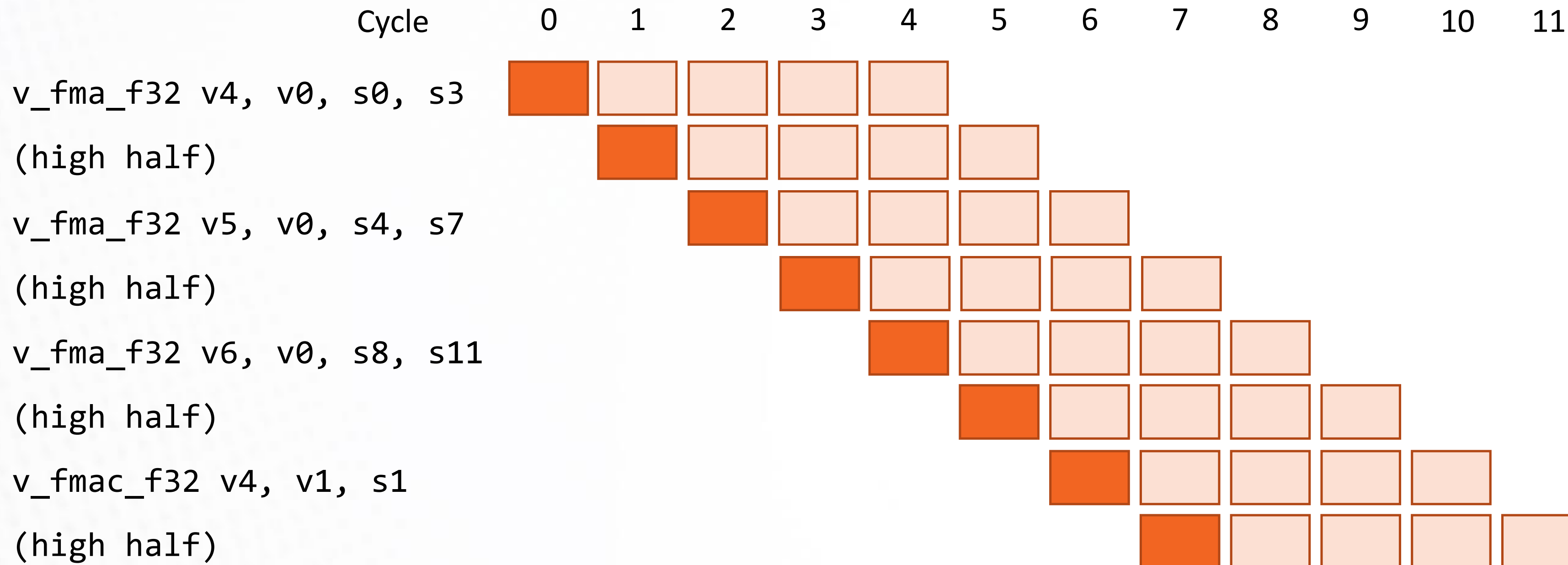


- Problems:
  - Code bloat (mind the I$ size!)
  - Higher VGPR count
  - Increases the effective dispatch granularity; more waste along the edges
- Don't panic: extra waves are a really good source of parallelism

AMD

Wave64 via dual-issue

- Vector instructions of Wave64 execute as 2x Wave32
- Same instructions, no code bloat



- When low or high half of EXEC is 0, that half skips execution

AMD

Wave32 vs. Wave64

Wave32	Wave64
Lower latency / wave lifetime Quicker ramp-up of WGPs after barriers More efficient for partially filled waves Tighter memory access patterns	Allows higher occupancy (# threads per lane) More efficient for attribute interpolation

- Compiler makes the decision
  - Compute and vertex shaders usually as Wave32, pixel shaders usually as Wave64
  - Heuristics will continue to be tuned for the foreseeable future

- Call to action:
  - Make sure barriers in shaders are semantically correct!
    - The compiler removes unnecessary barriers
  - Enable variable subgroup size, especially when using wave/subgroup intrinsics (Vulkan extension pending…)
  - Workgroup size: keep it a multiple of 64

AMD

LDS per workgroup processor

- 128 kB per workgroup processor
- Shared memory for compute, attributes for pixel shaders
- Up to 64 kB per workgroup
- Read / write / atomic throughput of up to 32 dwords per cycle (doubled relative to GCN)
- 32 banks
  - Same as “Vega”
  - Mind the bank conflicts!

AMD

Additional IPC improvements

- Overall goal: fewer move instructions
- Dual scalar source

```
s_buffer_load_dwordx2 s[0:1], …
s_waitcnt lgkmcnt(0)
v_mov_b32 v1, s0
v_fma_f32 v0, v0, v1, s1
```

```
s_buffer_load_dwordx2 s[0:1], …
s_waitcnt lgkmcnt(0)
v_fma_f32 v0, v0, s0, s1
```

- All VALU instructions support immediates (96-bit instructions)

```
s_mov_b32 s0, 0x3f490fdb    ; pi/4
v_mul_f32 v0, |v0|, s0
```

```
v_mul_f32 v0, |v0|, 0x3f490fdb
```

- Optional NSA (non-sequential address) encoding for image instructions
  - Avoids moves
  - Simplified register allocation helps reduce VGPR pressure

```
v_mov_b32 v7, v14
image_sample v[0:1], v[7:8], ...
```

```
image_sample v[0:1], [v14, v8], ...
```


AMD

Workgroup Processor summary

- Wave32 and single-cycle issue for better latency
- Co-execution of transcendental instructions
- Higher memory and LDS bandwidth, lower latency

- Calls to action:
  - Check your barriers!
  - Enable variable subgroup sizes once API support is there
  - Workgroup size: keep a multiple of 64
  - Keep an eye on instruction level parallelism (ILP), but don’t panic!
  - Calculate occupancy as "# threads per SIMD lane“
  - Worry a little less about VGPR pressure
  - Worry a little more about LDS bank conflicts
  - Use `ShuffleXor` instead of `Shuffle` when possible

AMD

Recap

- Explicit APIs expose more of the nitty-gritty details
  - Barriers and what caches get flushed
  - Blocks which can’t read/write compressed data (DCC and present ☺)

- GCN was compute/throughput focused, RDNA is graphics/latency focused
- Fix many bottlenecks found over the years
  - Geometry handling
  - Reduce cache flushes
  - Less “sensitive” compared to GCN (less work in flight needed, lower latency, etc.)

- Enable a more scalable architecture
  - Pave the way for a whole family of new GPUs
  - New features, different configurations, etc. coming down the line

AMD

Memory Hierarchy

GCN (RX Vega 64)

Graphics Core

Shader Engine

Shader Array

CU CU CU CU | CU CU CU CU | CU CU CU CU | CU CU CU CU

L1$ L1$ L1$ L1$ | L1$ L1$ L1$ L1$ | L1$ L1$ L1$ L1$ | L1$ L1$ L1$ L1$

I$ and K$ | I$ and K$ | I$ and K$ | I$ and K$

Shader Engine | Shader Array

Shader Engine | Shader Array

Shader Engine | Shader Array

L2$

Scalable Data Fabric

HBM2

RDNA (RX 5700 XT)

Graphics Core

Shader Engine

Shader Array

WGP | WGP | WGP | WGP | WGP

L0$ I$ and K$ L0$ | L0$ I$ and K$ L0$ | L0$ I$ and K$ L0$ | L0$ I$ and K$ L0$ | L0$ I$ and K$ L0$

L1$

Shader Array

Shader Engine | Shader Array | Shader Array

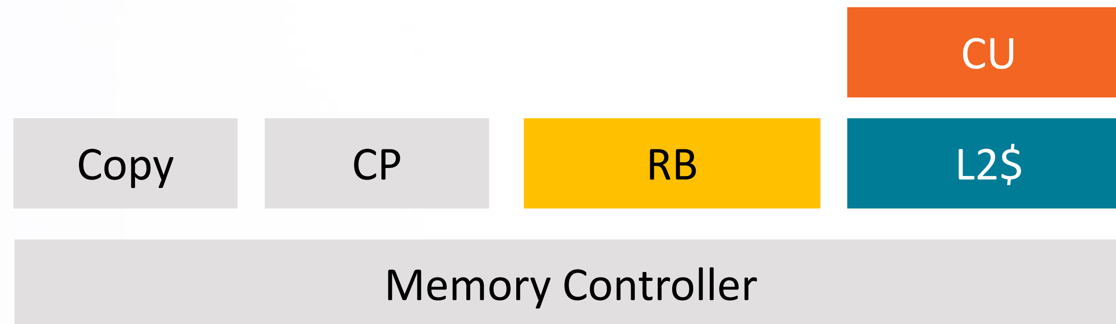
L2$

Scalable Data Fabric

GDDR6

Memory Hierarchy - L2 Clients

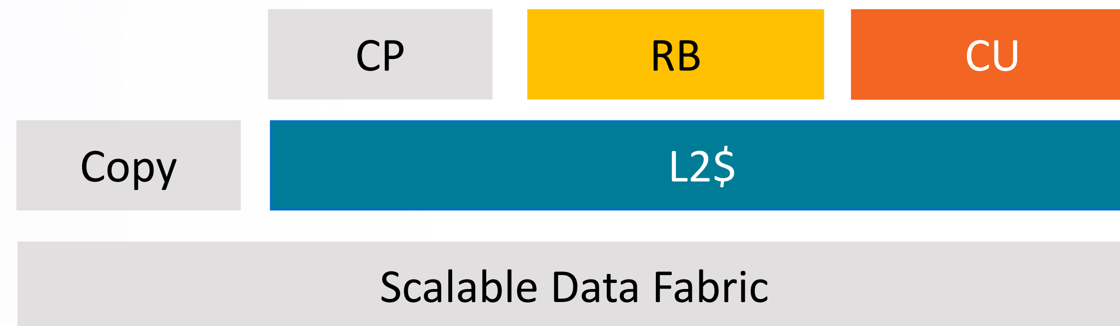
- On the Polaris architecture only the CUs are clients of L2.
  Copy Engine, CP and Render Backend directly write to memory.
  → Lots of L2 flushes.



AMD

Memory Hierarchy - L2 Clients

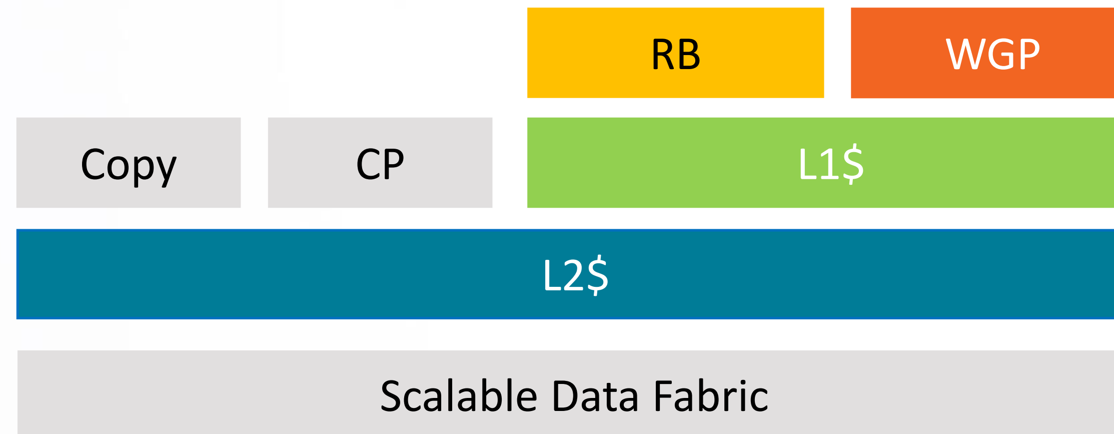
- On the Polaris architecture only the CUs are clients of L2.
  Copy Engine, CP and Render Backend directly write to memory.
  → Lots of L2 flushes.
- With the “Vega” architecture CP and the Render Backend became clients of L2.
  → Reduced number of L2 flushes. Uploads via copy queue still require an L2 flush.



AMD

Memory Hierarchy - L2 Clients

- On the Polaris architecture only the CUs are clients of L2.
  Copy Engine, CP and Render Backend directly write to memory.
  → Lots of L2 flushes.
- With the “Vega” architecture CP and the Render Backend became clients of L2.
  → Reduced number of L2 flushes. Uploads via copy queue still require a flush.
- On RDNA the Copy Engine is now a client of L2, too.
  → You should rarely observe a L2 flush on “Navi”.



AMD

Caches – Some numbers

RX Vega 64	Size	Cache Line Size	Read/Write
Instruction Cache (I$)	32KB per 4 CUs	32B	Read-only
Scalar Cache (K$)	16KB per 4 CUs	32B	Read-only

RX 5700 XT	Size	Cache Line Size	Read/Write
Instruction Cache (I$)	32KB per WGP (~2CUs)	64B	Read-only
Scalar Cache (K$)	16KB per WGP (~2CUs)	64B	Read-only

→ Twice as much I$ and K$, balancing out requirement for twice as many scalar resources

AMD

Caches – Some numbers

RX Vega 64	Size	Cache Line Size	Read/Write
Instruction Cache (I$)	32KB per 4 CUs	32B	Read-only
Scalar Cache (K$)	16KB per 4 CUs	32B	Read-only
L1 Cache	16KB per CU	64B	Read-only

RX 5700 XT	Size	Cache Line Size	Read/Write
Instruction Cache (I$)	32KB per WGP (~2CUs)	64B	Read-only
Scalar Cache (K$)	16KB per WGP (~2CUs)	64B	Read-only
L0 Cache	2x 16KB per WGP (~2CUs)	128B	Read-only

→ Twice as much I$ and K$, balancing out requirement for twice as many scalar resources
→ Higher bandwidth due to 128B cache lines. Fills up the chip with fewer memory requests.

AMD

Caches – Some numbers

RX Vega 64	Size	Cache Line Size	Read/Write
Instruction Cache (I$)	32KB per 4 CUs	32B	Read-only
Scalar Cache (K$)	16KB per 4 CUs	32B	Read-only
L1 Cache	16KB per CU	64B	Read-only
L2 Cache	4MB	64B	Read/Write

RX 5700 XT	Size	Cache Line Size	Read/Write
Instruction Cache (I$)	32KB per WGP (~2CUs)	64B	Read-only
Scalar Cache (K$)	16KB per WGP (~2CUs)	64B	Read-only
L0 Cache	2x 16KB per WGP (~2CUs)	128B	Read-only
L1 Cache	128KB per shader array	128B	Read-only
L2 Cache	4MB	128B	Read/Write

→ Twice as much I$ and K$, balancing out requirement for twice as many scalar resources
→ Higher bandwidth due to 128B cache lines. Fills up the chip with fewer memory requests.
→ Lower latency over “Vega” due to 512KB additional caches (L1).

AMD

DCC Everywhere

"Vega"	Compressed Reads	Compressed Writes
CU	✓	✗
Render Backend	✓	✓
Present Queue	✗	N/A

→ Decompression barrier before writing to UAV.

→ Decompression barrier before Present.

"Navi"	Compressed Reads	Compressed Writes
WGP	✓	✓
Render Backend	✓	✓
Present Queue	✓	N/A

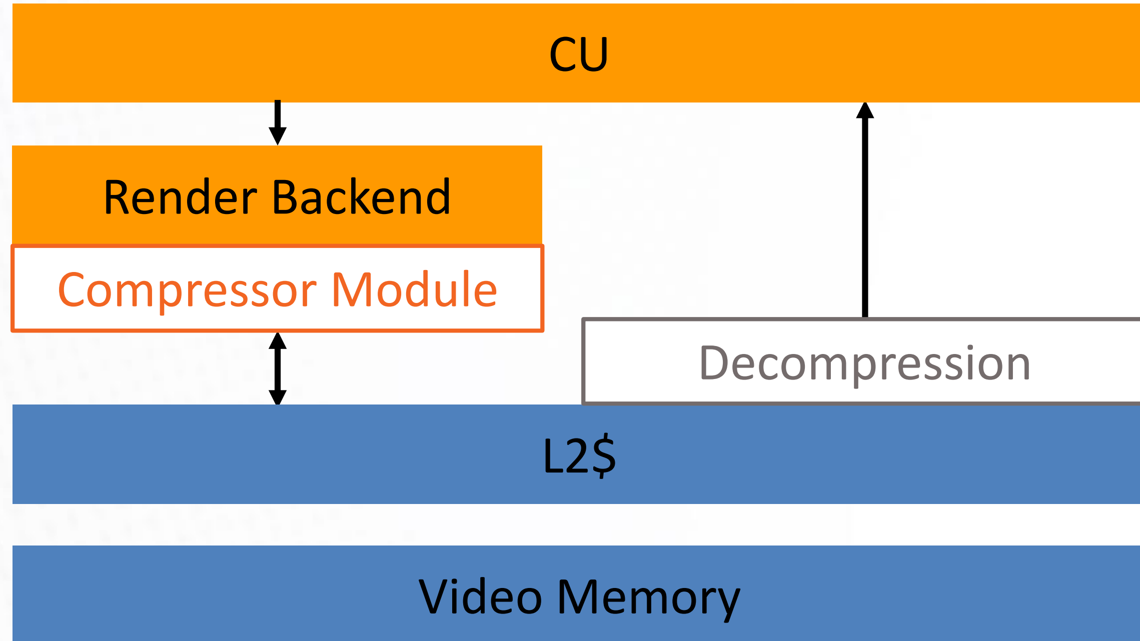
→ Expect more textures to stay compressed.
→ Compute all the things!

AMD

DCC Everywhere

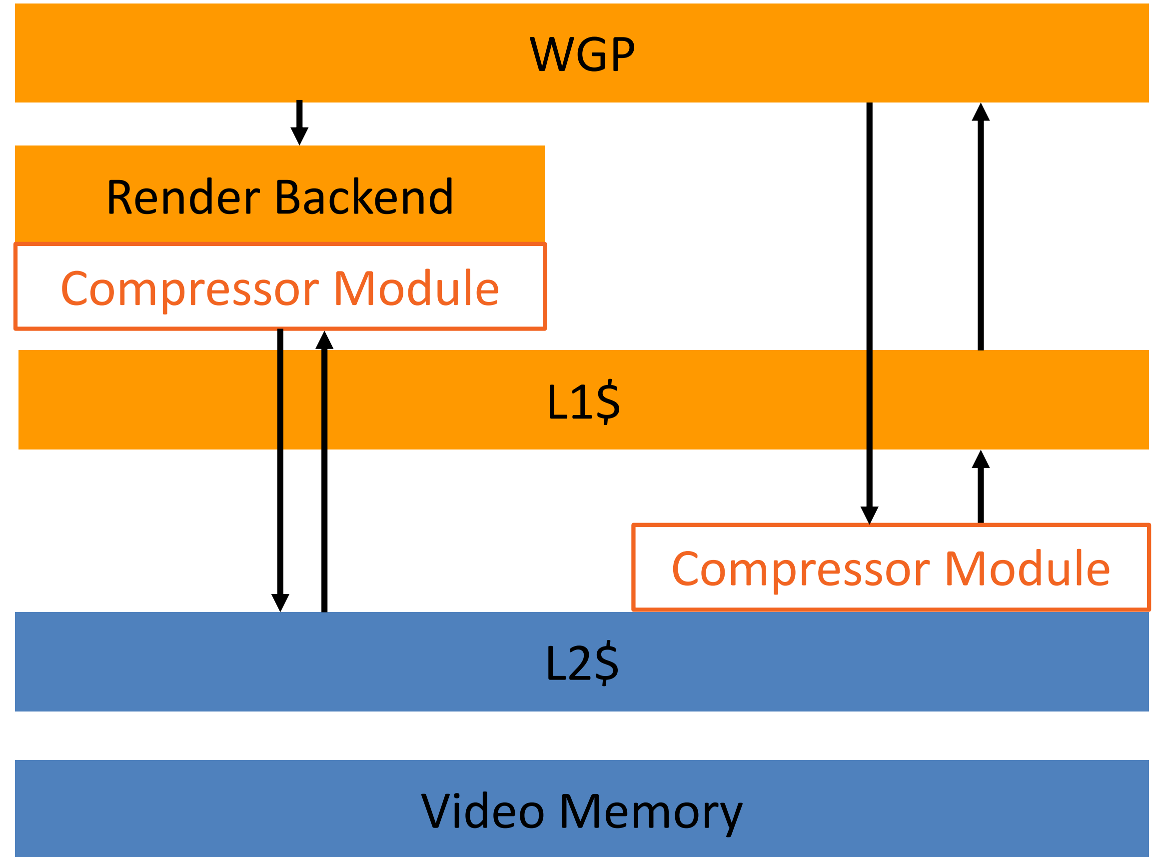
Compressed
Decompressed

- On “Vega”:



→ Bandwidth to VMEM stays unaffected.

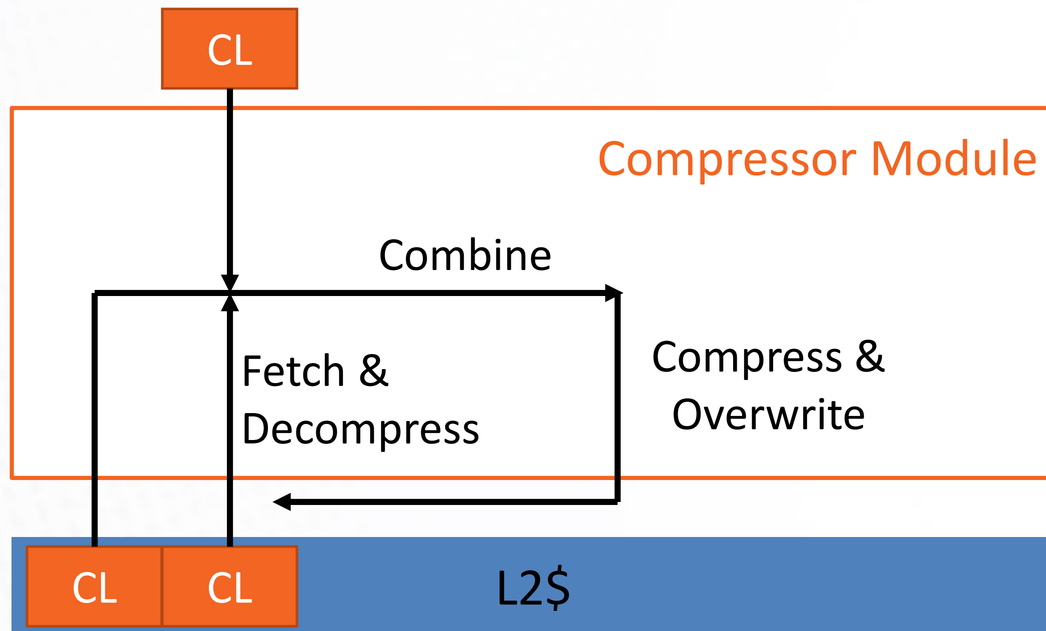
- On “Navi”:



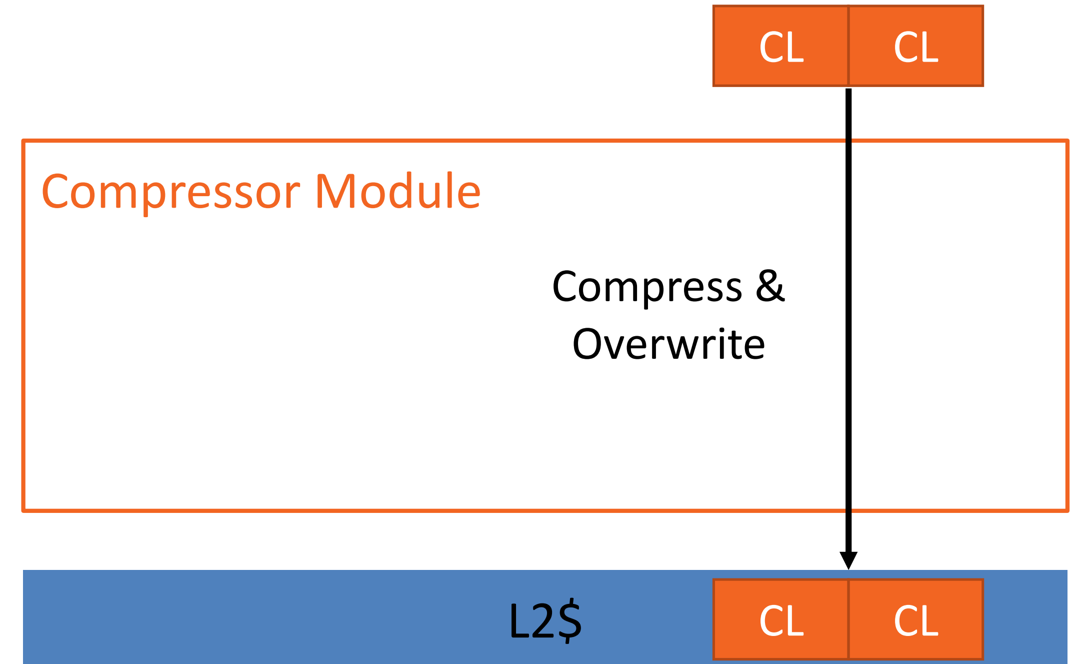
AMD

DCC Everywhere – Compressed Writes

- Scattered Write



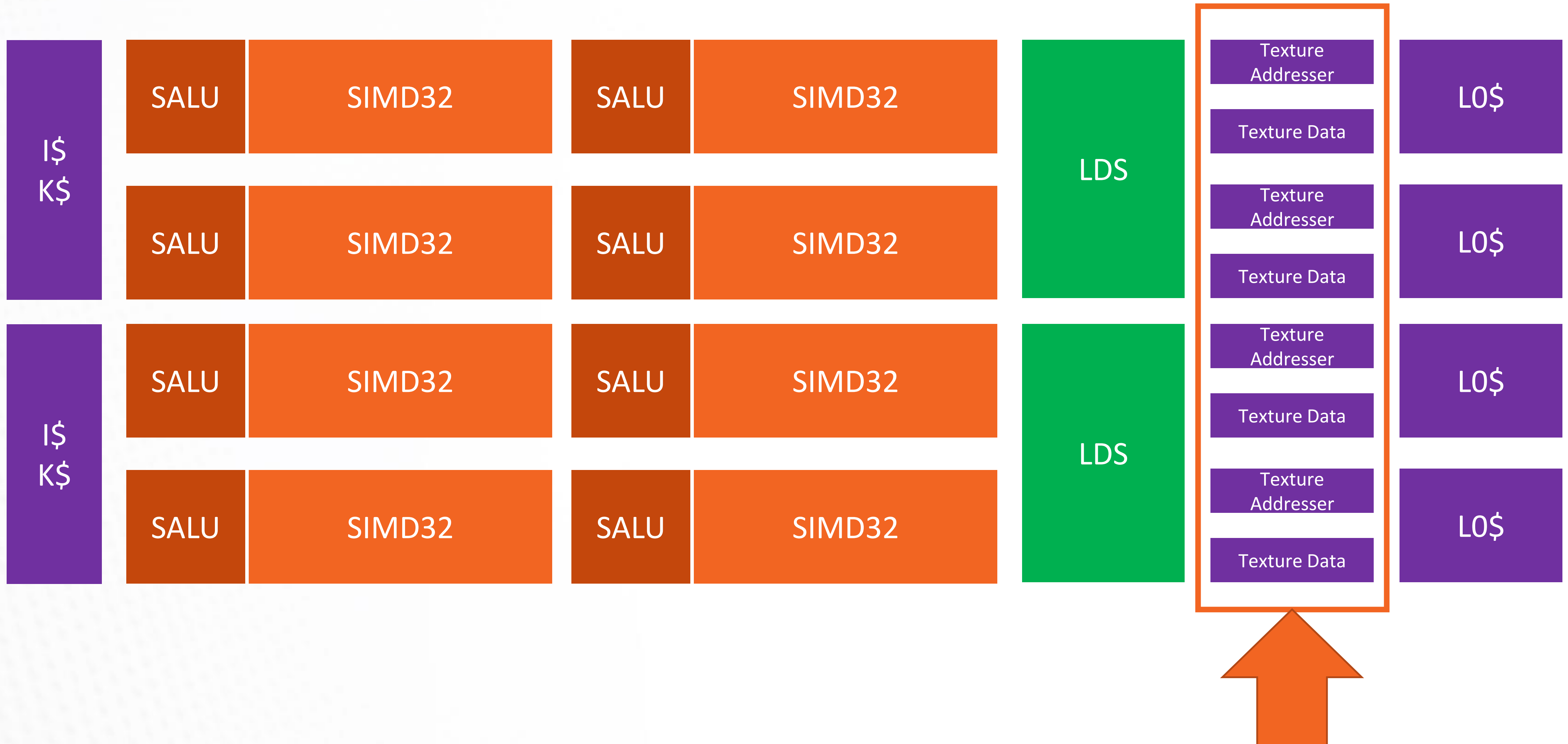
- 256B Coalesced Write
  (4bpp in Wave64, 8bpp on Wave32)



→ Prefer coalesced stores of at least 256B per wave for full efficiency.
→ Good rule of thumb: Write 8x8 blocks to images with 8x8 workgroup size.

AMD

Back to the WGP – Texture Units



AMD

Texture Unit - Changes to TA/TD

Feature	GCN	RDNA
Load addressing	4 to 16 (coalesced) addresses/clk	32 addresses/clk
Load data processing	4 to 16 (coalesced) dwords/clk	32 dwords/clk

→ Easier to reach maximum bandwidth via loads.

AMD

Texture Unit - Changes to TA/TD

Feature	GCN	RDNA
Load addressing	4 to 16 (coalesced) addresses/clk	32 addresses/clk
Load data processing	4 to 16 (coalesced) dwords/clk	32 dwords/clk
Store addressing	4 to 16 (coalesced) addresses/clk	32 addresses/2 clk
Store data processing	4 to 16 (coalesced) dwords/clk	32 dwords/2 clk

→ Easier to reach maximum bandwidth via loads.
→ Easier to reach maximum bandwidth via stores.

AMD

Texture Unit - Changes to TA/TD

Feature	GCN	RDNA
Load addressing	4 to 16 (coalesced) addresses/clk	32 addresses/clk
Load data processing	4 to 16 (coalesced) dwords/clk	32 dwords/clk
Store addressing	4 to 16 (coalesced) addresses/clk	32 addresses/2 clk
Store data processing	4 to 16 (coalesced) dwords/clk	32 dwords/2 clk
Filtering 64bit texels	2 components/clk	4 components/clk

→ Easier to reach maximum bandwidth via loads.
→ Easier to reach maximum bandwidth via stores.
→ Improve FP16 with full rate 4 channel sampling.

AMD

Load / Store Queue – “Vega”

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
```

VMCNT

0

AMD

Load / Store Queue – “Vega”

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
```

VMCNT

1

Vector stores and loads increment VMCNT.

AMD

Load / Store Queue – "Vega"

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
```

VMCNT

2

AMD

Load / Store Queue – "Vega"

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
```

VMCNT

2

Wait here until VMCNT is 0.

AMD

Load / Store Queue – “Vega”

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
```

VMCNT

0

That means we also wait for the store!

AMD

Load / Store Queue – "Vega"

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
```

VMCNT

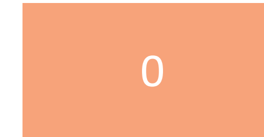
0

AMD

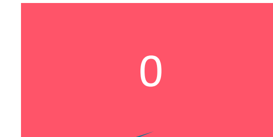
Load / Store Queues - RDNA

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
s_waitcnt vscnt(0)
s_barrier
```

VMCNT



VSCNT



“Navi” adds a separate queue for stores.

AMD

Load / Store Queues - RDNA

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
s_waitcnt vscnt(0)
s_barrier
```

VMCNT

0

VSCNT

1

Stores increment VSCNT.

AMD

Load / Store Queues - RDNA

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
s_waitcnt vscnt(0)
s_barrier
```

VMCNT	VSCNT
1	1

Loads still increment VMCNT.

AMD

Load / Store Queues - RDNA

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
s_waitcnt vscnt(0)
s_barrier
```

VMCNT

0

VSCNT

1

Wait until all loads return
and VMCNT drops to 0.

AMD

Load / Store Queues - RDNA

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
s_waitcnt vscnt(0)
s_barrier
```

VMCNT: 0

VSCNT: 1

Now we **don't** wait for the store to finish!

AMD

Load / Store Queues - RDNA

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
s_waitcnt vscnt(0)
s_barrier
```

Waiting for the store happens here.

VMCNT	VSCNT
0	1

Load / Store Queues - RDNA

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
s_waitcnt vscnt(0)
s_barrier
```

VMCNT

0

VSCNT

0

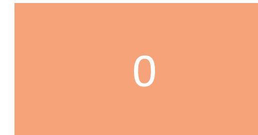
We are good to go once the store operation returns.

AMD

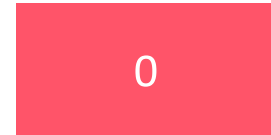
Load / Store Queues - RDNA

```
...
buffer_store_dword v2, v0, s[12:15], 0 idxen
buffer_load_dword v0, v1, s[8:11], 0 idxen
s_waitcnt vmcnt(0)
v_add_f32 v0, v2, v0
s_waitcnt vscnt(0)
s_barrier
```

VMCNT



VSCNT



- Optimization for the general case.
  - It's very likely that you will see `s_waitcnt vscnt(0)` only in front of a `s_barrier` or in front of atomic operations.
  - `s_endpgm` implicitly waits on all counters.
- Non atomic stores are now true "fire and forget".

Load / Store Queues - RDNA

Effectively sequences like this will now run faster by default.

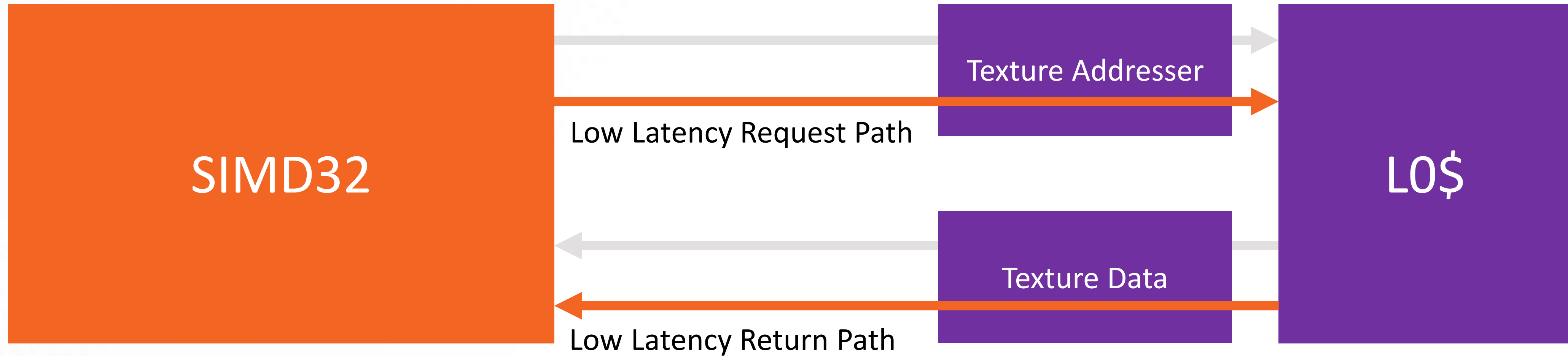
→You might want to consider interleaving loads and stores again.

→This has the additional benefit of saving VGPRs.

```
buffer_load_dword v1, v0, s[4:7], 0 idxen
v_add_u32 v2, 1, v0
s_waitcnt vmcnt(0)
buffer_store_dword v1, v0, s[8:11], 0 idxen
buffer_load_dword v1, v2, s[4:7], 0 idxen
v_add_u32 v3, 2, v0
s_waitcnt vmcnt(0)
buffer_store_dword v1, v2, s[8:11], 0 idxen
buffer_load_dword v1, v3, s[4:7], 0 idxen
v_add_u32 v2, 3, v0
s_waitcnt vmcnt(0)
buffer_store_dword v1, v3, s[8:11], 0 idxen
buffer_load_dword v1, v2, s[4:7], 0 idxen
v_add_u32 v3, 4, v0
s_waitcnt vmcnt(0)
buffer_store_dword v1, v2, s[8:11], 0 idxen
buffer_load_dword v1, v3, s[4:7], 0 idxen
v_add_u32 v2, 5, v0
s_waitcnt vmcnt(0)
buffer_store_dword v1, v3, s[8:11], 0 idxen
buffer_load_dword v1, v2, s[4:7], 0 idxen
v_add_u32 v3, 6, v0
s_waitcnt vmcnt(0)
buffer_store_dword v1, v2, s[8:11], 0 idxen
buffer_load_dword v1, v3, s[4:7], 0 idxen
v_add_u32 v2, 7, v0
s_waitcnt vmcnt(0)
buffer_store_dword v1, v3, s[8:11], 0 idxen
buffer_load_dword v1, v2, s[4:7], 0 idxen
```

RDNA Fast Loads

- Texture Unit has low latency path for Loads.



- In general we need to keep the order with respect to Samples.
- Worst-case: Loads are as slow as Samples.

→Replace all `texture.Sample(PointSampler, texcoord)` with `texture.Load(location)`!

→Separate Loads and Samples if feasible.

Memory Hierarchy – Take Away

- 128B cache lines
  → You may want to adjust your memory alignments
- Addition of L1 gives access to more cache
  → Easier to run at peak ALU
- Independent loads and stores
  → Faster by default and opportunity for VGPR savings
- Higher bandwidth via PCIe® 4, load in parallel via PCIe® and copy queue (SDMA) on L2
  → Stream all the things on the copy queue
- DCC everywhere
  → Fully overwrite 256B blocks on store.
  → Another reason to move to compute ☺

RDNA

- RDNA - An all new architecture
- Focus on graphics & latency
  - Reduced latency throughout the whole pipeline
  - Higher efficiency on many graphics workloads
- A new infrastructure
  - New memory, interconnect, etc.
  - New display controllers
- Available July 7th!

AMD

DISCLAIMER AND ATTRIBUTIONS

DISCLAIMER

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

“Vega” and “Navi” are codenames for AMD architectures, and are not product names. GD-122.

©2019 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, Radeon, Ryzen and combinations thereof are trademarks of Advanced Micro Devices, Inc. Vulkan and the Vulkan logo are registered trademarks of the Khronos Group Inc. PCIe and PCI Express are registered trademarks of the PCI-SIG Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

AMD