RADEON

ADVANCED SHADER PROGRAMMING ON GCN

PRESENTED BY TIMOTHY LOTTES

ADVANCED SHADER PROGRAMMING

- Skipping the larger introduction on GCN
  - "The AMD GCN Architecture – A Crash Course" is a great refresher
  - http://www.slideshare.net/DevCentralAMD/gs4106-the-amd-gcn-architecture-a-crash-course-by-layla-mah
- Presenting a holistic view of shader programming
  - Focusing on how to reason about high-level design decisions
  - And presenting as many optimization tools as is possible in one hour
- Freely mixing Vulkan® and DirectX® 12 terms (talk applies to both APIs)
- Flood of technical background and strategies for optimization
  - Feel free to follow-up after the talk with questions: Timothy.Lottes@amd.com

TERMS

- I$ = L1 instruction cache

- K$ = scalar L1 data cache (aka “Konstant” cache)
- SALU = scalar ALU operation
- SGPR = scalar general purpose register
- SMEM = scalar memory operation

- V$ = vector L1 data cache
- VALU = vector ALU operation
- VGPR = vector general purpose register
- VMEM = vector memory operation

THE GPU

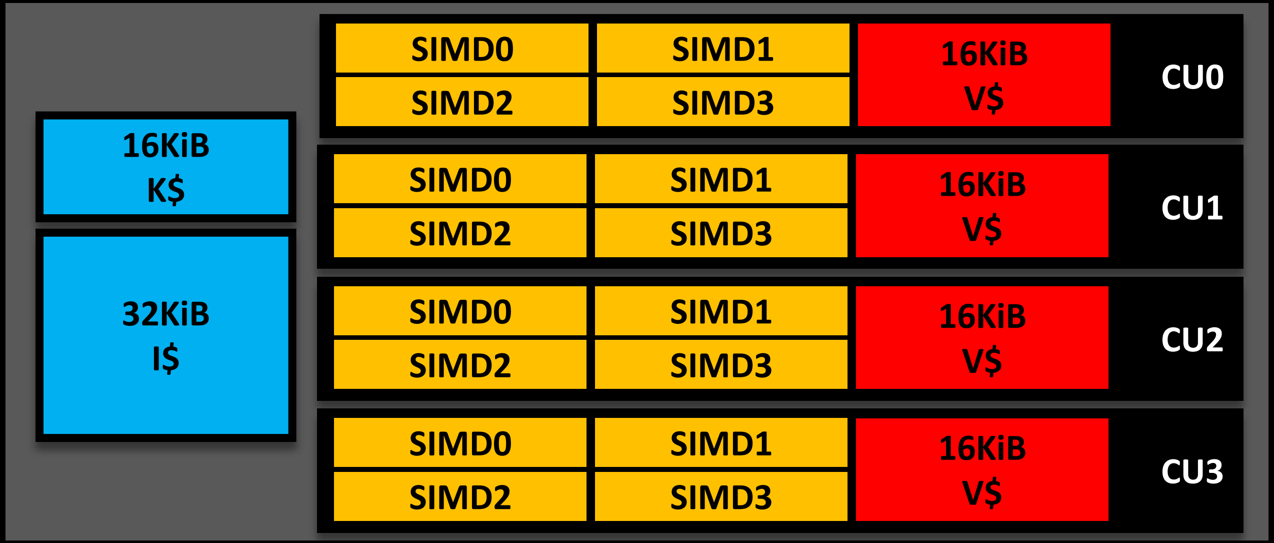
- A GPU is roughly a bunch of functional blocks connected by Queues
  - Can see the regular structure of blocks on GPU die shots

- Functional blocks have a fixed capacity/time
- Size of Queues control latency and volatility tolerance

- Shader Optimization
  - Keep the Queues fed with regular steady work
  - Adjust workload to avoid draining limiting Queue
  - Adjust workload to avoid being limited by fixed capacity/time

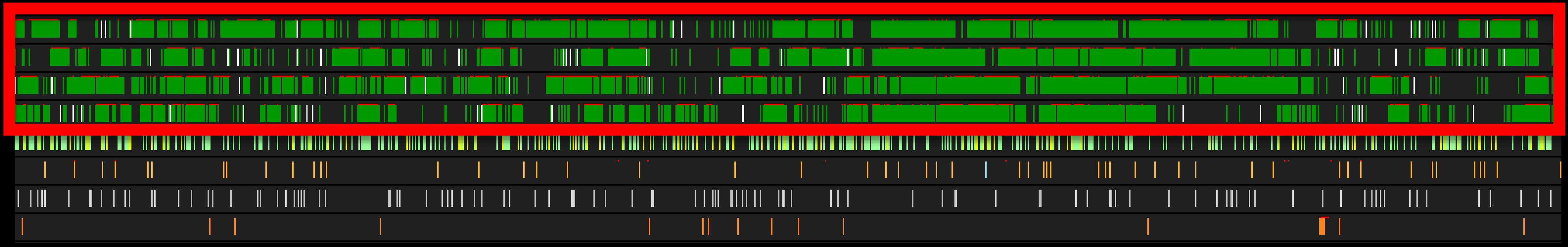
GCN BACKGROUND

- Example: R9 Nano
  - 1 GHz clock, 8192 Gflop/s, 512 Gbyte/s, 256 Gtex/s, 64 Grop/s
  - 2 MiB L2$, 64 CUs each with 16KiB V$ (Vector L1) and 4 SIMDs
  - Each SIMD has capacity for 10 waves (64 lanes/wave)
  - Each CU has a peak throughput of 64 VALU instructions/clock
  - Each 4 CUs share 16KiB K$ (Constant L1), and 32KiB I$ (Instruction L1)



MOTIVATIONS

- Optimization is about maximizing VALU utilization (getting work done)
  - A core component of that is optimizing for data flow in the GPU
  - Minimizing getting stalled on memory
- Showing a real GCN CU utilization plot across time below
  - The outlined box shows filled green for areas of utilized VALU cycles in each SIMD of the CU
  - See all the dead space = large amount of lost capacity to do work
  - This talk is ultimately about minimizing this VALU dead space



TUNING KNOBS

- Adjust data format and organization (packing)
- Adjust data access patterns (spatial and temporal)

- Adjust the amount of instructions required
- Adjust the binding complexity

- Adjust the way work is distributed across the chip
- Adjust the way work is separated into kernels and waves

- Adjust the amount of work which is running in parallel

LOOKING FIRST AT WORK DISTRIBUTION

- How to optimally leverage caches
- Methods for optimal grouping of work into workgroups
- Recommendations on sizing of work per PSO change

SHADER SIZE TARGETS TO STAY IN I$

- Example: R9 Nano

  Each 4 CUs share 16KiB K$, and 32KiB I$ (16/chip)

- Shader size targets to stay cached

  32KiB I$ / 4 to 8-bytes per Instruction = 4K to 8K instructions fit in the cache

- If multiple shaders need to share the cache, optimal to have smaller shaders

  4 shaders / cache = 1K to 2K instructions average per shader

  8 shaders / cache = 512 to 1K instructions average per shader

  16 shaders / cache = 256 to 512 instructions average per shader

OPTIMAL TO NOT CHANGE PSO TOO OFTEN

- Using the R9 Nano as an easy example to reason about (1ns/clk)

  1 GHz clock, 8192 Gflop/s, 512 Gbyte/s, 256 Gtex/s, 64 Grop/s

  Each 4 CUs share 16KiB K$, and 32KiB I$ (16/chip)

- Changing shaders

  Say L2 hit rate is double DRAM bandwidth: 1024 Gbyte/s (estimating, real number is different)

  32KiB cache * 16/chip = 512KiB to fully fill instruction caches

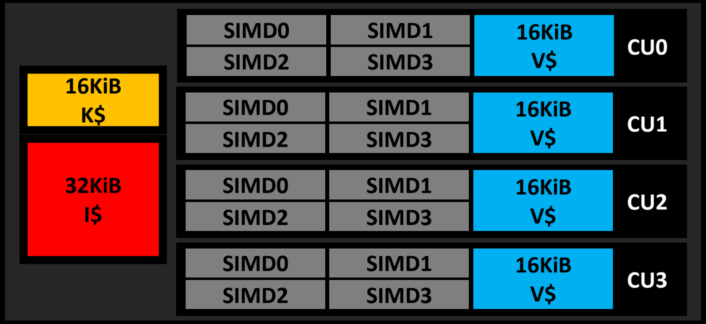
  Could rough estimate ability to fill instruction caches 2 million times per second

  Using estimate: 100 Hz * 1000 fills of instruction cache = estimate 5% of GPU L2 hit capacity

  Instruction fetch can ultimately eat against bandwidth available to do work

CACHES – MAGNITUDE OF DATA USED BY SHADER

- 4-8 bytes per instruction – I$
  - 1024 instruction shader = 4096-8192 bytes per wave
- 1 byte per texel for BC3 (aka DXT5) – V$
  - 16 images * 64 invocations * 1 byte = 1024 bytes per wave (more in practice due to over-fetch)
- 8 bytes per RGBA16F image store – V$
  - 2 image stores * 64 invocations * 8 bytes = 1024 bytes per wave (no DCC, assuming aligned output)
- 16-32 bytes per descriptor – K$
  - 16 descriptors = 256-512 bytes per wave (assuming packed and aligned)
- 4 bytes per 32-bit constant – K$
  - 64 constants = 256 bytes per wave (assuming packed and aligned)
- In order to amortize cost of {instructions, descriptors, constants}
  - Need a lot of reuse (hits in I$ and K$)
  - Want to avoid too many unique small draw/dispatch shaders



16KiB K$

32KiB I$

SIMD0 SIMD1 SIMD2 SIMD3 16KiB V$ CU0

SIMD0 SIMD1 SIMD2 SIMD3 16KiB V$ CU1

SIMD0 SIMD1 SIMD2 SIMD3 16KiB V$ CU2

SIMD0 SIMD1 SIMD2 SIMD3 16KiB V$ CU3

DATA CAN HAVE A RELATIVELY SHORT LIFETIME IN V$ HIERARCHY

- Using the R9 Nano as an easy example to reason about (1ns/clk)

  1 GHz clock, 8192 Gflop/s, 512 Gbyte/s, 256 Gtex/s, 64 Grop/s

  2 MiB L2$, 64 CUs each with 16KiB V$ and 4 SIMD

- Minimum bound on window of opportunity to hit in L2$

  512 Gbyte/s can fill 2 MiB L2$ in somewhere over 4096 clock cycles

  Data can have a relatively short lifetime in L2

- Can be a small window of opportunity to hit in V$

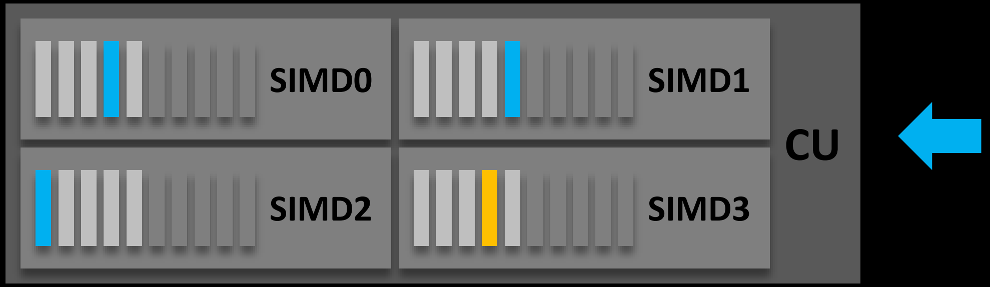
  64 CUs * 16KiB V$ = 1 MiB total V$

  Half the size of L2 and substantially higher bandwidth with L2 hits

  Window of reuse in V$ can be very short, organize loads to maintain L1 reuse

GRAPHICS WORK DISTRIBUTION

- GCN can distribute graphics waves across the chip dynamically
  - For fragment shaders there is no fixed mapping of ROP tile to one CU on the chip
- GCN prefers to group fragment waves temporally on a CU for better V$ usage
  - For example one wave per SIMD, 4 waves/CU, before moving to another CU
  - Work distributer can skip over a SIMD if SIMD is too full to issue work
    - This can increase V$ pressure but can better maintain GPU utilization
- Vertex Shader waves are launched one per CU before moving to another CU
  - Prefer to not push work into VS if it requires high amounts of V$ locality



Example VGPR limited at 50% wave occupancy
(5 waves/SIMD)
Only one VS wave active on the CU
Grey waves are existing waves
New blue fragment waves could only launch on 3 SIMDs

COMPUTE WORK DISTRIBUTION

- Compute shaders can enable fixed wave grouping via Workgroups

  Waves of a Workgroup share one V$

  Compute can be a way to increase V$ locality (use more than 4 waves/workgroup)

  Can adjust wave occupancy by LDS usage, tune waves to V$ ratio for high performance

- Compute workgroups dispatched 1 per CU before moving to another CU
- Compute mixed with graphics

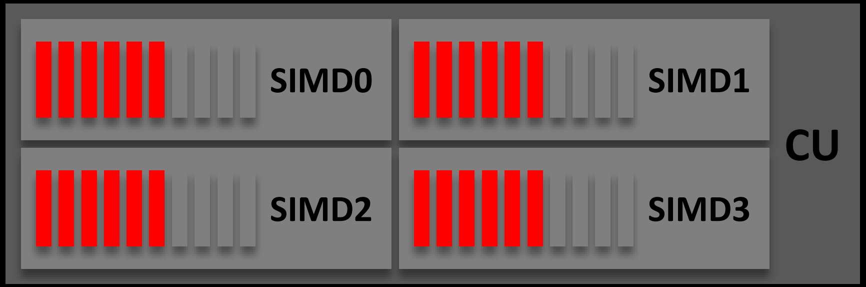
  Keep workgroup sized to fit well concurrently with graphics shaders

- Compute alone without graphics

  Running small workgroups can be bad for V$ locality

  CS without using barriers: can size workgroup to fill CU

  CS with barriers: size workgroup so a few fill CU



WAVE LIMITS EXTENSION

- VK_AMD_wave_limits
- Possible Vulkan extension, possible usage cases,
  - Throttling wave per pipeline stage to better mix compute and graphics
  - Enabling tuning of wave to V$ ratio
  - Limiting execution to specific groups of CUs for better I$,K$,V$ usage

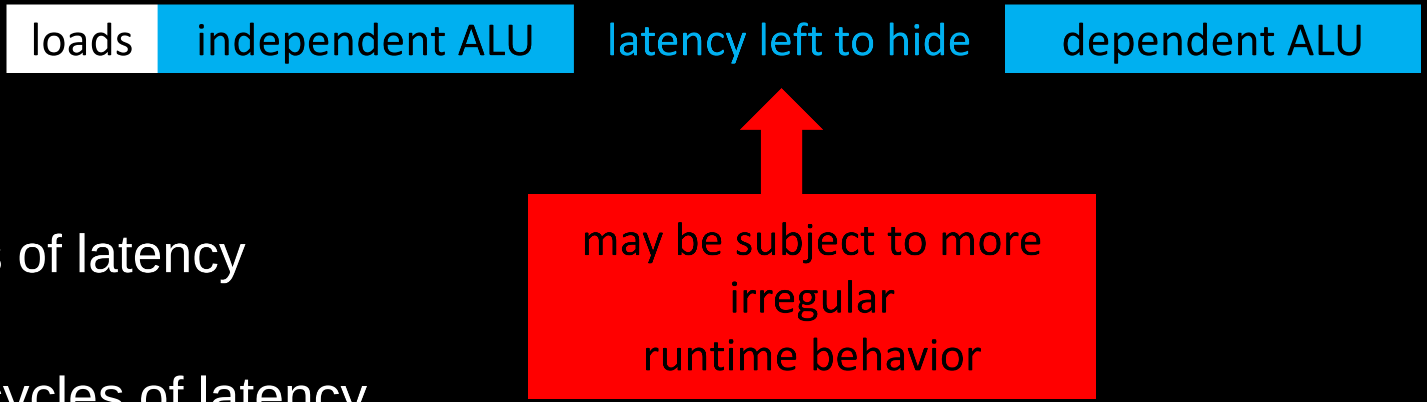
```
struct VkPipelineShaderStageCreateInfoWaveLimitAMD {
        VkStructureType sType;
        const void* pNext;
        float maxPercentageOfWavesPerCu;
        uint32_t* cuEnableMask; }
```

WAVE LAUNCH COSTS

- Costs
  - Mixed pipeline workloads waiting for right granularity of free CU resources
  - Setting USER-DATA SGPRs (preloaded scalar data before wave launches)
  - Wave launch building run-time generated descriptors
    - Root-table CBVs are 64-bit pointers converted to 128-bit descriptors at run-time
  - Initial SMEM loads for {constants and descriptors}
  - Filling the L1 caches for 1st use {shaders, constants and descriptors}
    - GCN4 adds shader prefetch
  - Deriving per-lane values used in shader execution (like lane index)
- Can amortize costs in compute by pulling lots of work before exit
  - Some of the most highly optimized kernels written at AMD use the “pinned workgroup” model

WAVE MULTI-ISSUE

- CU can issue to multiple functional units at the same time
  - But for a given clock, can only issue to functional units from different waves on the same SIMD
  - So prefer high enough wave occupancy as this can increase IPC (instructions per clock)
  - Try for good ILP (instruction level parallelism) to maximize latency hiding ability in the wave
  - Batch to minimize amount of latency to hide per individual load



loads | independent ALU | latency left to hide | dependent ALU

may be subject to more irregular runtime behavior

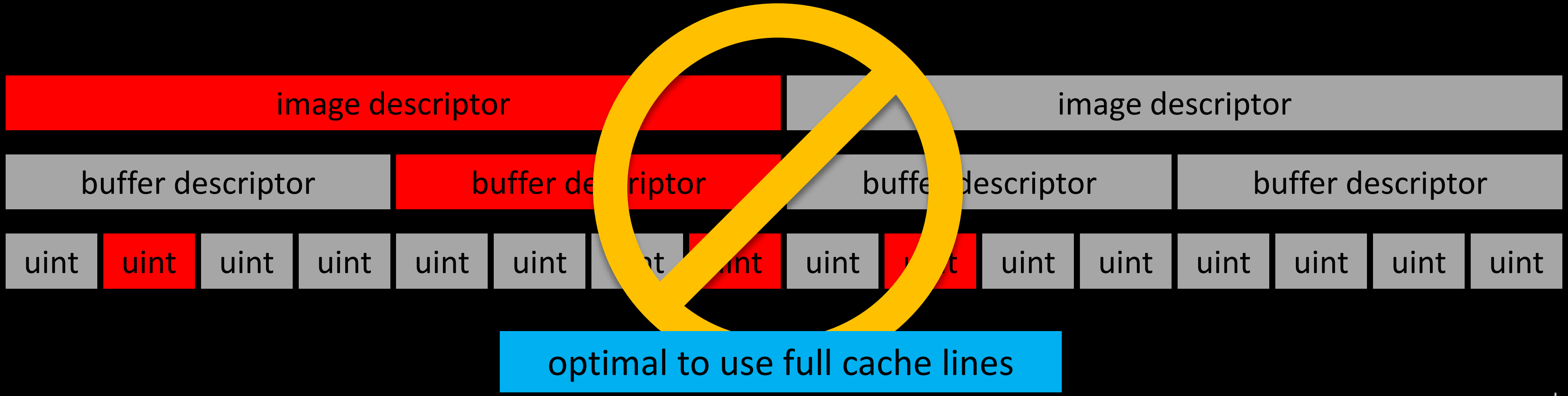
- Example of functional units
  - SALU – scalar ALU
  - SMEM – K$ access – tens of cycles of latency
  - VALU – vector ALU
  - VMEM – V$ access – hundreds of cycles of latency
  - LDS – workgroup shared memory
  - Export – graphics shader fixed function output

MOVING ON TO BINDINGS AND VMEM ACCESS

- Optimizing binding
- VMEM throughput
- Shader storage buffers (aka RWStructureBuffer in the other API)

CONSTANT AND DESCRIPTOR SET DATA

- Use full cache lines
  - 16KiB K$ / 64-byte lines = only 256 lines
  - Vulkan can pack 4 buffer or sampler descriptors in one cache line
  - Vulkan can pack 2 image descriptors in one cache line
  - Scattered usage can effectively be 2 to 4 times more expensive for descriptors
  - Scattered usage amplifies the amount of latency which needs to be hidden
  - Optimal to layout descriptors and constants aligned and grouped by usage



USER-DATA SGPR PRELOAD

- 16 SGPRs can be pre-loaded before shader start
  - App does not get 16, some used for driver and for fixed function depending on pipeline state
  - Rest used for Descriptor Set pointers, DYNAMIC Descriptors, Root Table entries, etc
  - Descriptor Sets (aka Descriptor Tables) are 32-bit values (one USER-DATA SGPR)

- Warning about spilling out of USER-DATA SGPRs
  - Too many root entries in DirectX 12, too many Sets or DYNAMIC Descriptors in Vulkan
  - USER-DATA SGPRs are hardware versioned = fast
  - Spill can be software versioned and requires indirection = slower

HIGH FREQUENCY CHANGES

- Using the R9 Nano as an easy example to reason about (1ns/clk)

  Each 4 CUs share 16KiB K$, and 32KiB I$ (16/chip)

- Levels for I$ and K$ reuse

  Looking at case where shader waves get distributed 1 per SIMD (4 per CU) before next CU

  16 waves sharing I$ * 64 lanes * 1 I$ = 1024 invocations

  First step of reuse is maximizing single I$ reuse

  16 waves sharing I$ * 64 lanes * 16 I$/chip = 16384 invocations

  Next step of reuse is filling up the machine more than once

- Short running draws/dispatches want to share shader

  With specialization ideally through things which fit in USER-DATA SGPRs

  For example change Descriptor Set or change Push Constant in Vulkan

  Keep usage of USER-DATA small to avoid launch overhead

DRAW-TO-DRAW HAZARDS

- DirectX12/Vulkan have multiple Table/Set binding points to remove hazards
  - Split your resources and bind Table/Set by frequency of update

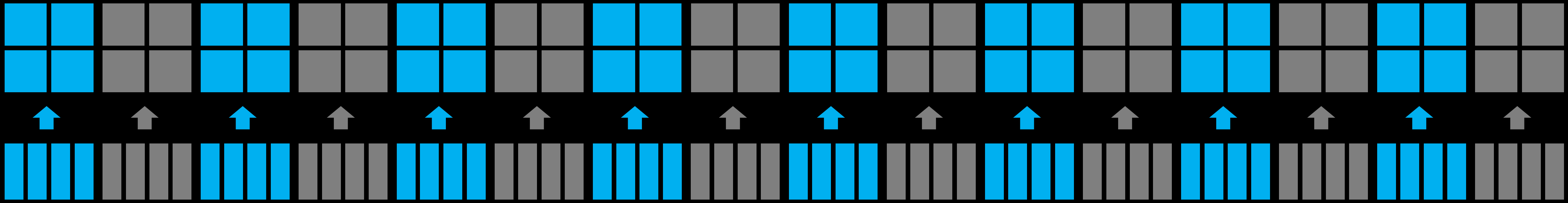
- Stay far away from any binding model construct which has draw-to-draw hazards
  - For example OpenGL ®, DirectX 11, and KHR_push_descriptors produce hazards
  - Effectively one “Descriptor Set” per stage, shared across draws
  - API designed around updating a fraction of the “Set” between draws = hazard
  - Need to version the “Set” = slow

VMEM

- Optimizing vector memory access

VMEM ACCESS

- One active lane has same addressing speed limit as 64 active lanes
  - But could save power and cache usage
- Image accesses are done in aligned groups of 4 lanes (2x2 fragment quad)
  - Keep this in mind for lane layout for compute waves
    - Want mip level to be the same for the group of 4 lanes
    - The point of 4 lane grouping is to avoid V$ bank conflicts
  - Want the whole wave to have good 2D locality (for 2D accesses)
    - Ordering of the {4 lane groups} inside the wave is not as important



VMEM THROUGHPUT UPPER BOUND PER CU

- Spec peak throughput for V$ hits (perf lower in practice)
  - 32-bit (or smaller) single-channel buffer loads / wave = 4 clocks (under specific cases)
  - multi-channel buffer loads / wave = 16 clocks
  - 128-bit (or smaller) non-filtered texels / wave = 16 clocks
  - 32-bit (or smaller) filtered texels / wave = 16 clocks
  - 64-bit filtered texels / wave = 32 clocks
  - 128-bit filtered texels / wave = 64 clocks

- Prefer larger granularity accesses (yellow)
  - Highest data/overhead ratio
  - Less VGPRs used for VMEM parameters for a given amount of data transferred
  - GPU has fixed limit on number of pending VMEM requests

IMAGE DIVERGENCE

- For 3D images sampled from all directions (SDF tracing)
  - Make sure to not flag for render target usage (forces possibly slower THIN tiling mode)
- Array layer divergence = no problem, no replay (fast)
- Mip divergence in quad = hardware replay loop (slow)
  - Addressing can only resolve one mip level per clock per 4 lanes
  - Coarse LOD in fragment shader forces same mip level
  - Watch out in other stages, can use wave operations to force same LOD
- Resource indexing divergence = software replay loop (slower)

RESOURCE INDEXING DIVERGENCE OVERHEAD AND LATENCY CHALLENGE

- Resource divergence blocks Instruction Level Parallelism (ILP)

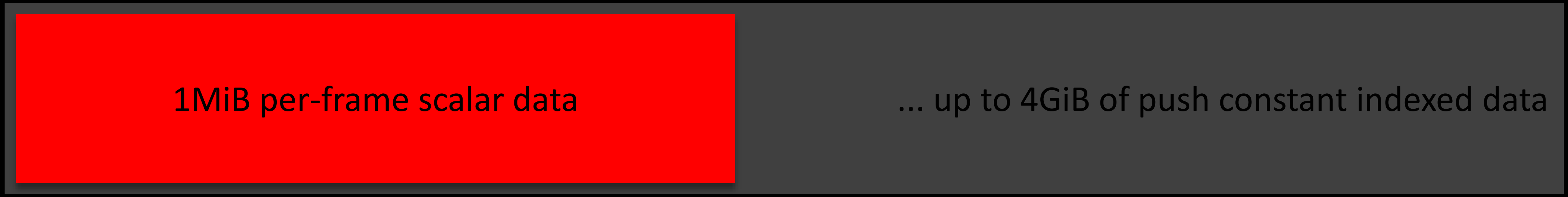
```
s_mov_b64 uniformSave,EXEC
s_mov_b64 uniformRemaining,EXEC
v_lshlrev_b64 offset,index,5 // index to descriptor offset
loop:
   v_readfirstlane_b32 uniformOffset,offset
   v_cmpx_eq uniformOffset,offset
   s_nop 5 // VALU sets EXEC required manual 5-cycle wait state (other wave can execute)
   s_buffer_load_dwordx8 ...,uniformOffset // fetch the descriptor
   s_waitcnt lgkm_cnt=0 // wait for descriptor to be loaded (other wave can execute)
   image_sample_... // do the fetch(es)
   ...
   s_cbranch_scc loop
s_mov_b64 EXEC,uniformSave
...
s_waitcnt 0 // wait for everything before 1st result usage (due to unknown # of fetches)
```

RESOURCES – SHADER STORAGE BUFFER OBJECT (SSBO)

- The ultimate memory access optimization tool on GCN
- Not limited to 64 KiB
  - One single descriptor can access up to 4GiB of DEVICE_LOCAL buffer memory for the app
  - If using SHADER_STORAGE_BUFFER_OBJECT_DYNAMIC
    - Descriptor is preloaded into USER-DATA SGPRs before shader executes (no indirection)
- Not limited to one type
  - Can do {1,2,4} component granularity accesses
  - Can have automatic type conversion (but limited API support right now)
- Provides {Read, Write, and Atomic} access
- In hardware same descriptor can be used in SMEM or VMEM path
  - Use with dynamically uniform addressing to get SMEM loads into SGPRs

SMEM BUFFER ACCESSES

- Hardware can load 1 to 16 DWORDs in one operation
  - GCN 3/4 added support for storing 1-4 DWORDS
- S_BUFFER_LOAD_DWORD addressing modes
  - [buffer descriptor base address + 20-bit unsigned byte immediate offset]
    - Optimal to place immediate indexed data in lower 1MiB of buffer
    - Good place for per-frame or per-view data
  - [buffer descriptor base address + 32-bit unsigned byte offset in an SGPR]
    - Largest fast access buffer is 4GiB in size



1MiB per-frame scalar data

... up to 4GiB of push constant indexed data

SMEM BUFFER ACCESS FORMAT SUPPORT

- Formats supported when accessing via a dynamically uniform address
  - SALU does not have any special format conversion hardware
- {1,2,4,8,16} component 32-bit {{signed,unsigned} integer, float}
- Leave other types packed in 32-bit values
  - 4 component 8-bit {signed,unsigned} integers
  - {2,4} component 16-bit {{signed,unsigned} integers, float}
- Unpack on usage later using SDWA
- Use packed on GCN5 for double rate math

VMEM TBUFFER HARDWARE FORMAT SUPPORT

- TBUFFER_<LOAD|STORE>_FORMAT_<F>
  - Same instructions used for vertex fetch, type provided in opcode instead of descriptor

- 4-bit DFMT – Data format
  - {1,2,4} component {8,16,32}-bit values
  - {3} component 32-bit values
  - 10:11:11
  - 10:10:10:2
  - 2:10:10:10

- 3-bit NFMT – Numeric Format
  - {unorm,snorm,uscaled,sscaled,uint,sint,float}

VMEM TBUFFER FORMAT SUPPORT OVERLAPPED WITH API

- The {uint,sint,float} cases using native types or SDWA unpack for {8,16}-bit

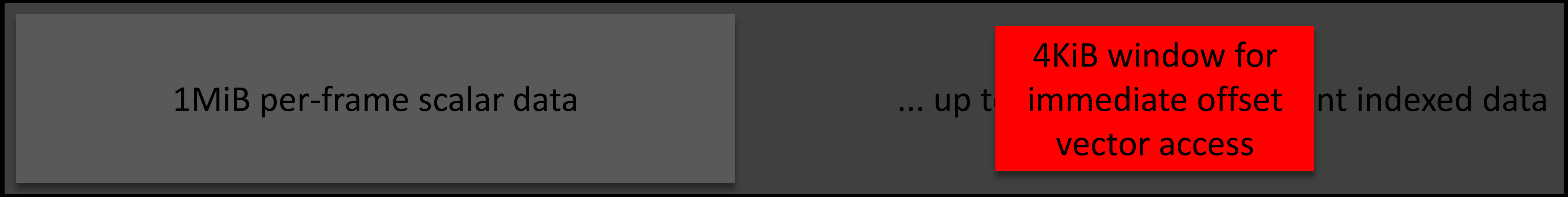
- The unpack<Unorm|Snorm><2x16|4x8>() instructions
  - Covers {unorm,snorm} for multi-component {8,16}-bit values
  - Efficient compiler support pending (will note on GPUOpen when it arrives)

- Currently missing API or extension support for
  - 10:11:11
  - 10:10:10:2
  - 2:10:10:10

VMEM BUFFER ADDRESSING

- GCN buffer addressing has more features than covered here
  - Including 64-bit addressing, but showing just the simplified addressing for SSBO usage here

- [descriptor base address + SGPR offset + VGPR offset + immediate offset]
  - 48-bit base address (in descriptor)
  - 32-bit SGPR byte offset (optional via 0 inline constant)
  - 32-bit VGPR byte offset (optional)
  - 12-bit unsigned immediate byte offset (included in opcode for up to 4KiB offset)



1MiB per-frame scalar data

... up t

4KiB window for immediate offset vector access

nt indexed data

DIVING INTO INSTRUCTION LEVEL DETAILS

- Ship One Shader in Vulkan
- Instruction built-in features
- DPP and wave-level programming
- SDWA and packed math

GCN INSTRUCTION SET ARCHITECTURE – MAJOR REVISIONS

- GCN 1st/2nd Generation : R9 390, etc
  - Base for comparison
- GCN 3rd/4th Generation : R9 380, FuryX, RX480, etc (1st GPU released 2014)
  - SMEM now designed to support scalar memory writes
  - Single-rate 16-bit operations
  - SDWA : Sub DWord Addressing provides 8-bit and 16-bit pack/unpack on register access
  - DPP : Data Parallel Processing allows VALU instructions to source from another lane
  - V_PERM_B32
  - DS_PERMUTE_B32, DS_BPERMUTE_B32
- GCN 5th Generation : “Vega” Chipset
  - Packed 16-bit operations (double-rate)
  - And more ...

ADAPTING TO GPU VARIATION IN GCN AND WITH OTHER VENDORS

- Compile one shader which uses multiple extensions from multiple IHVs

  Then ship this one shader!

- Leverage Vulkan SPIR-V Specialization Constants

  Specialization Constants to turn on/off usage of extensions, provide subgroup size, etc

- Run SPIR-V through fast linear SPIR-V to SPIR-V filter at load-time

  Set Specialization Constants based on capabilities and extensions on local machine

  Trim unsupported extensions

  Transform unsupported opcodes to OpUndef

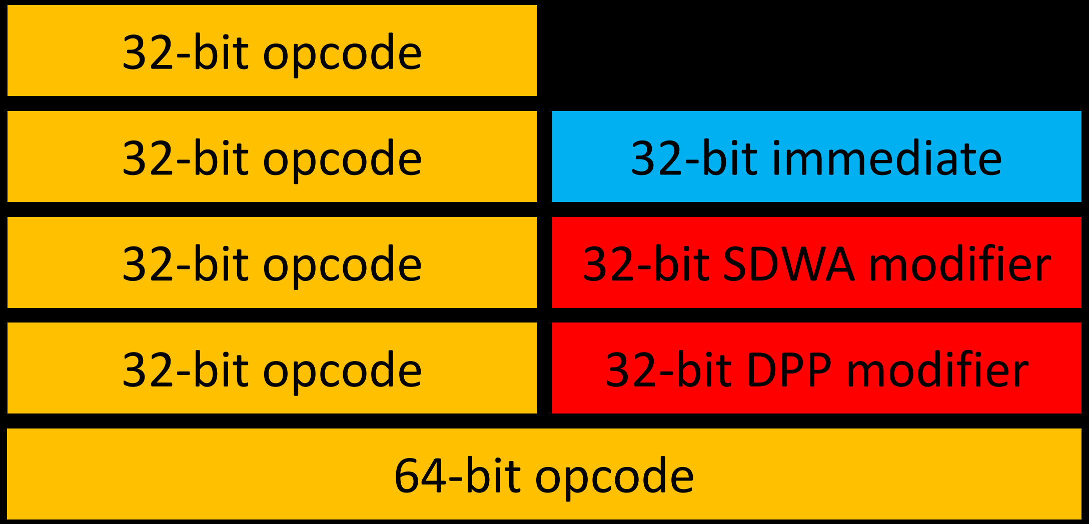
  SOS – Ship One Shader – Open source dependency-free C header, development in progress . . .

VULKAN AND SPIR-V SPECIALIZATION CONSTANT DETAILS

- Provide constants at PSO compile time so compiler can fold into optimizations
  - VkSpecializationInfo in Vk<Compute|Graphics>PipelineCreateInfo
  - Filled out for vkCreate<Compute|Graphics>Pipelines()
- GLSL syntax
  - `layout(constant_id=0) const bool name = false; // sets default value`
- Specialization constants can be used to size arrays, etc

DIVING INTO INSTRUCTION LEVEL DETAILS

- GCN has a variable-width ISA with 32-bit or 64-bit instructions
  - 64-bit instructions are for VMEM and 3 operand VALU operations
  - GCN3 32-bit opcode modifiers trade increased flexibility for increased I$ pressure



32-bit opcode

32-bit opcode | 32-bit immediate

32-bit opcode | 32-bit SDWA modifier

32-bit opcode | 32-bit DPP modifier

64-bit opcode

TERMS

- 32-bit opcodes

  VOP1 – Vector ALU Operation with 1 Source Operand

  VOP2 – Vector ALU Operation with 2 Source Operands

  VOPC – Vector ALU Operation Compare (2 Source Operands)

- 64-bit opcodes

  VOP3 – Vector ALU Operation with up to 3 Source Operands

  Adds input and output modifiers

FREE DESTINATION MODIFIERS – OMOD

- Only available for 32-bit and 64-bit floating point (not supported for 16-bit)

  Applies to SDWA and VOP3 (3 source) but not DPP and not packed math

- Provides free multiply on destination

  dst = 0.5 * operationResult

  dst = 1.0 * operationResult

  dst = 2.0 * operationResult

  dst = 4.0 * operationResult

- OMOD works on instructions which cannot produce denormals

  COS_F32, CUBE*_F32, LOG_F32, MAD_F32, RCP_F32, RSQ_F32, SIN_F32, SQRT_F32, etc

- Otherwise OMOD is only available when denormals are in flush-to-zero mode

  Default on PC Graphics APIs

FREE DESTINATION MODIFIERS – CLAMP

- GCN1/2 supports CLAMP only on floating point
- GCN3/4 supports CLAMP on integers as well
- GCN5 supports CLAMP on packed math

- CLAMP applies to SDWA, VOP3 (3 source), and packed math, but not DPP

- CLAMP on floating point = saturate(operation) = clamp(operation, 0.0, 1.0)
- CLAMP on integer stops overflow or underflow

  Not exposed in any PC API as of March 2017

FREE DESTINATION MODIFIERS – CLAMPED OMOD

- CLAMP applies after OMOD

- x = clamp(2.0 * (x + y), 0.0, 1.0)

- V_ADD_F32 v1 v1 v2 mul:2 clamp

FREE INPUT MODIFIERS

- Supported for floating point inputs, separate options for each input

- Supported with SDWA and VOP3 (3 source) but not DPP

  {src, -src, abs(src), -abs(src)}

- Supported with packed math on GCN5

  {src, -src} with different options for {hi, lo} 16-bit value

- Better to defer these until just when needed to help the compiler out

  Compiler needs to pattern match to find them

VECTOR INLINE CONSTANTS AND LITERALS

- Opcode forms, only 9-bit sources can use inline constant and literals
  - VOP1 (1 source): 9-bit SRC0
  - VOP2 (2 sources): 9-bit SRC0, 8-bit SRC1
  - VOPC (compare): 9-bit SRC0, 8-bit SRC1
  - VOP3 (3 sources): 9-bit SRC0, 9-bit SRC1, 9-bit SRC2 (no 32-bit literal support)
  - VOP* SDWA/DPP (2 sources): 8-bit SRC0, 8-bit SRC1
- Can only source either a 32-bit literal or a SGPR source per instruction
  - 32-bit literals only supported on 32-bit instructions
  - For VOP3, can use that SGPR source in any of the 3 sources via duplication
- Any 9-bit source can use an inline constant without being synthesized
  - Integers: -16, -15, -14, ... 0 ... 62, 63, 64
  - Floating point: -4.0, -2.0, -1.0, 0.0, 1.0, 2.0, 4.0 and 1.0/(2.0*pi)
  - For VOP3, all 3 sources can use a different inline constant

SCALAR INLINE CONSTANTS AND LITERALS

- Can use one 32-bit literal per instruction
  - With exception: instructions which already include 16-bit literals

- Any source can use a inline constant without being synthesized
  - Integers: -16, -15, -14, ... 0 ... 62, 63, 64
  - Floating point: -4.0, -2.0, -1.0, 0.0, 1.0, 2.0, 4.0
  - Sources can use a different inline constant

WAVE-LEVEL PROGRAMMING

- Leveraging SMEM and SALU for better performance

WAVE-LEVEL PROGRAMMING

- GCN1/2
  - V_READ<FIRST>LANE_B32 to transfer VGPR to SGPR
  - DS_SWIZZLE_B32 for sourcing another lane
- GCN3 design adds faster DPP support for sourcing another lane during operation

- Parallel reductions can return dynamically uniform values
  - Promote VMEM to SMEM operation
  - Leverage SGPRs to store loads instead of VGPRs for major VGPR savings

- Can use parallel operations to minimize global atomics

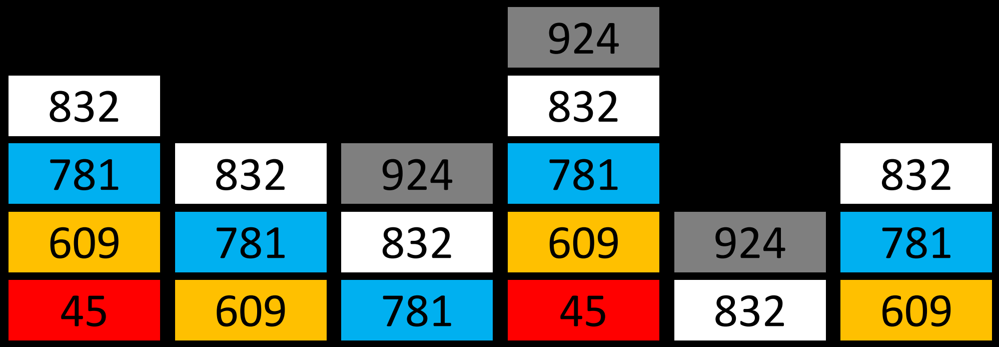
WAVE-LEVEL PROGRAMMING SUPPORT

- Available now
- Vulkan
  AMD_shader_ballot, ARB_shader_group_vote, ARB_shader_ballot, etc
- AGS in DirectX 11 and 12
  http://gpuopen.com/gaming-product/amd-gpu-services-ags-library/
- Available in the future via SM6 in DirectX 12
  https://msdn.microsoft.com/en-us/library/windows/desktop/mt733232%28v=vs.85%29.aspx

K$ ALL THE THINGS

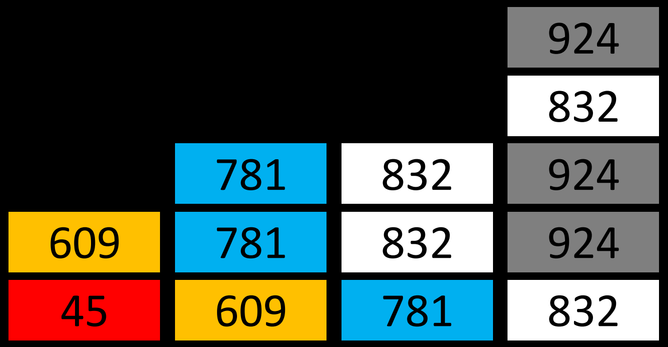
- Using the R9 Nano as an easy example to reason about (1ns/clk)
  - 2 MiB L2$, 64 CUs each with 16KiB V$ and 4 SIMD
  - Each 4 CUs share 16KiB K$, and 32KiB I$ (16/chip)
- Data cache per wave and per invocation
  - 80 waves / K$ and 1280 invocations / V$ @ 50% wave occupancy
  - 16KiB K$ / 80 waves = 204.8 bytes/wave average
  - 16KiB V$ / 1280 invocations = 12.8 bytes/invocation average
- K$ amplifies the amount of L1 cache available to work with
  - For example, DOOM leverages K$ for a 1.43x performance improvement
    - http://advances.realtimerendering.com/s2016/Siggraph2016_idTech6.pdf
  - Good {alignment, packing, wave operations} can enable taking advantage of K$
  - K$ typically requires 10x less hit latency to hide than V$ (varies, but good ballpark estimate)

THE POWER OF GOING DYNAMICALLY UNIFORM



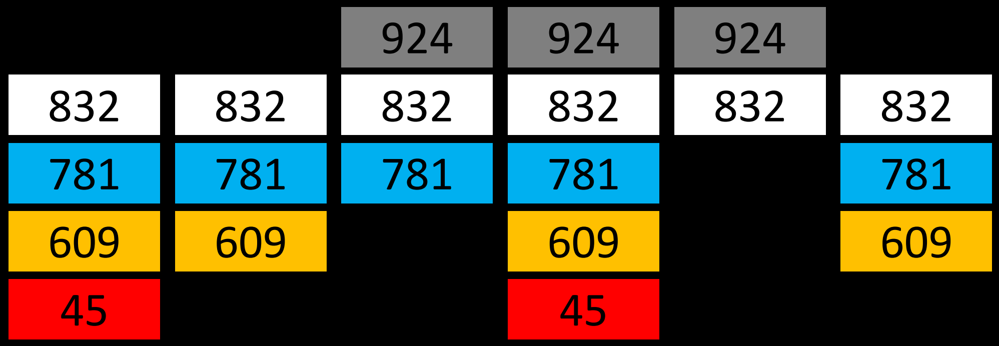
			924		
832			832		
781	832	924	781		832
609	781	832	609	924	781
45	609	781	45	832	609

pre-sorted per lane lists executed in list order



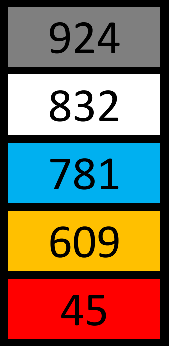
			924	uniform branch
			832	uniform branch
	781	832	924	3 way divergent branching
609	781	832	924	4 way divergent branching
45	609	781	832	4 way divergent branching

things processed in iteration **13 paths executed**



		924	924	924	
832	832	832	832	832	832
781	781	781	781		781
609	609		609		609
45			45		

process in “if(item==waveMin(item))” order



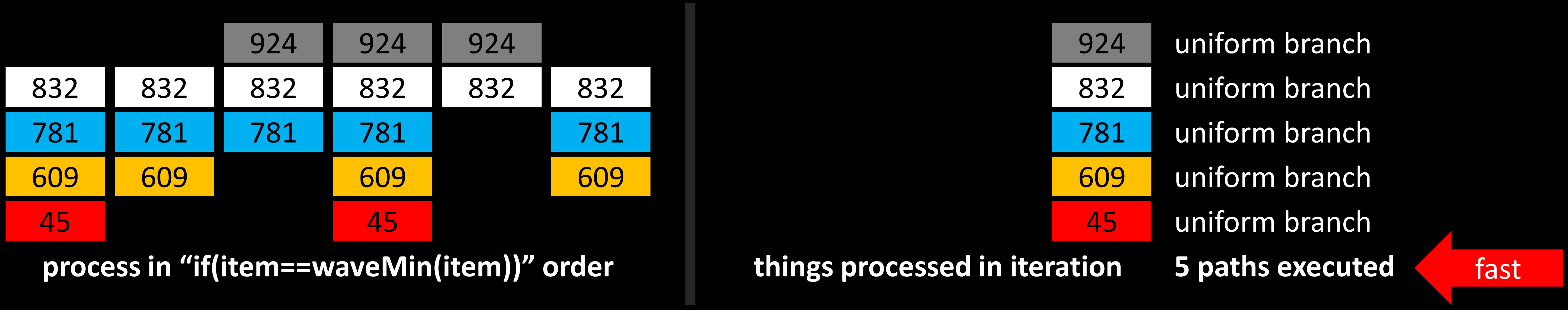
924	uniform branch
832	uniform branch
781	uniform branch
609	uniform branch
45	uniform branch

things processed in iteration **5 paths executed**

fast

THE POWER OF GOING DYNAMICALLY UNIFORM – CODE EXAMPLE

```
uint vgprThingIndex = FetchNextThing();
while(vgprThingIndex != 0) { // process all the things
    uint sgprThingIndex = minInvocationsNonUniformAMD(vgprThingIndex); // in uniform order
    if(vgprThingIndex == sgprThingIndex) {
        uvec4 sgprData = FetchThingData(sgprThingIndex); // VGPR savings!
        vgprThingIndex = FetchNextThing(); // reduce latency by fetching next index early
        switch(sgprData.x) { // coherent branching!
```



		924	924	924	
832	832	832	832	832	832
781	781	781	781		781
609	609		609		609
45			45		

process in “if(item==waveMin(item))” order

924	uniform branch
832	uniform branch
781	uniform branch
609	uniform branch
45	uniform branch
things processed in iteration	**5 paths executed**

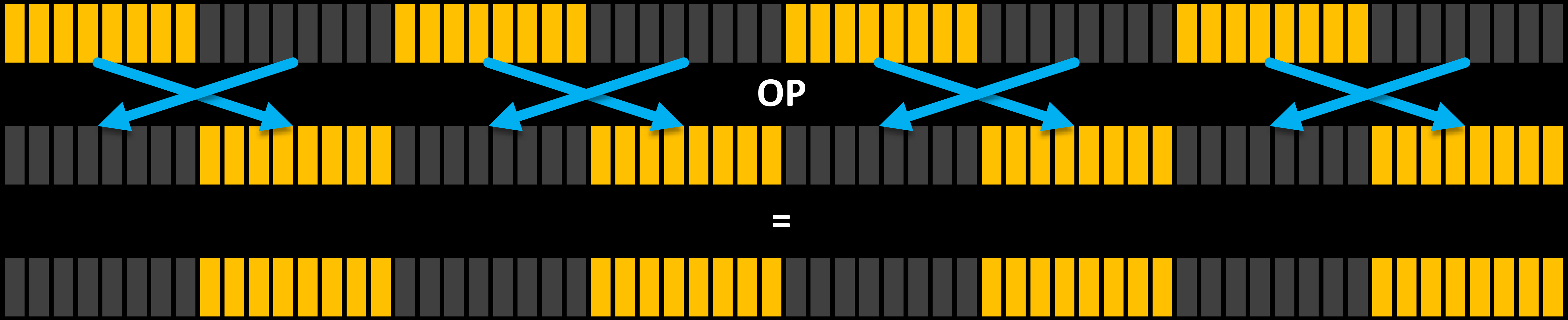
fast

DS_SWIZZLE_B32

- The first interface provided for cross-lane operations in GCN1
- Drives through LDS crossbar in hardware so requires S_WAITCNT
  - Higher latency than DPP, does permutation separate from any math operation
- Cluster of aligned 4 lanes
  - Full permutation
- Cluster of aligned 32 lanes
  - Controlled by 5-bit {AND,OR,XOR} masks
  - Can do clustered broadcast
  - Can do clustered mirror (reverse lanes in clusters)
  - Can swap pairs of clusters
- Wave of 64 lanes
  - Requires two V_READLANE_B32 ops to merge the two 32-lane cluster results

DPP – DATA PARALLEL PRIMITIVES

- Added on GCN3, ability to source one operand from another lane during operation
  - Fixed set of lane permutations on dedicated hardware fast path
  - Useful for building dynamically uniform values to remove divergence on GPU
  - Helps leveraging SMEM as a secondary L1 cache for data



DPP DESIGN DETAILS

- VOP_DPP provides a 32-bit opcode extension DWORD
- Can be applied to 32-bit opcodes (VOP1/VOP2/VOPC)
  - DPP is a modifier and not an extra instruction
- Enables a collection of features
  - Enables sourcing SRC0 VGPR from another lane of the wave (fixed lane permutations)
  - Supports an immediate write mask for ROWs and BANKs which disable operation store
  - Supports a flag which forces invalid or disabled lanes to read zero for SRC0
  - Supports additional NEG/ABS control for SRC0 and SRC1
- Hardware motivation
  - Fixed permutations on dedicated hardware fast path
  - Avoids latency of going to LDS crossbar
  - Avoid "shuffle()" aka DS_BPERMUTE_B32 for parallel operations due to high cost

DPP DETAILS – PERMUTATIONS (WRITE MASK CAN FURTHER SHAPE THESE)

- Cluster of aligned 4 lanes

  QUAD_PERM – Any permutation supported
- Cluster of aligned 8 lanes

  ROW_HALF_MIRROR – Reverse all lanes
- Cluster of aligned 16 lanes

  ROW_SL / ROW_SR – Shift left / right by {1 to 15} lanes
  ROW_RR – Rotate right by {1 to 15} lanes
  ROW_MIRROR – Reverse all lanes
  ROW_BCAST15 – Broadcast 15th lane of prior cluster to next cluster
- Cluster of aligned 32 lanes

  ROW_BCAST31 – Broadcast 31st lane of prior cluster to next cluster
- Full 64 lane wave

  WF_SL1 / WF_SR1 – Shift left / right by 1 lane
  WF_RL1 / WF_RR1 – Rotate left / right by 1 lane

DPP DISASSEMBLY EXAMPLE – WAVE MINIMUM

- Safe in divergent control flow using 8 VALU operations with DPP

```
s_orn2_saveexec_b64 s[4:5], 0 // save EXEC and switch to execute only on inactive lanes
v_mov_b32 v2, -1 // setup operation neutral value on inactive lanes
s_nand_b64 exec, 0, 0 // switch to full-wave execution
s_nop
v_min_u32 v2, v2, v2 row_shr:1 // first stage of 6-stage minimum reduction
s_nop // 1 clock wait state before result is valid, other wave VALU can issue here
v_min_u32 v2, v2, v2 row_shr:2
s_nop
v_min_u32 v2, v2, v2 row_shr:4
s_nop
v_min_u32 v2, v2, v2 row_shr:8
s_nop
v_min_u32 v2, v2, v2 row_bcast:15 row_mask:0xa
s_nop
v_min_u32 v2, v2, v2 row_bcast:31 row_mask:0xc
s_mov_b64 exec, s[4:5] // restore EXEC and return to set of active lanes at beginning of code block
v_readlane_b32  s4, v2, 63 // places result in SGPR
```

PACKING

SDWA / PACKED PREVIEW

- As of March 2017 the rest of this talk is a preview of what we are planning to bring
  - Both in hardware and software
  - Some of this may be subject to change

- Talk covers strategy for optimizing for GCN5 packed math
  - With backwards compatibility to previous GCN3 generation launched in 2014
  - Investing in packing can bring large gains on non-GCN5 hardware as well

- Watch for follow-up content on GPUOpen in 2017!
  - We will let you know when compiler support is ready for SDWA + packed math

PACK ALL THE VECTOR THINGS FOR HIGHER THROUGHPUT

- VGPR savings on GCN3 and up for better occupancy
  - Seen double digit performance improvements on some shaders as we bring up this feature

- Packed math on GCN5 for more perf per instruction
  - Tool to bring a VALU limited shader back to being bandwidth bound
  - Advantage dependent on shader

  - Once bandwidth bound, possibility to try to trade more VALU for less bandwidth
    - Compress all the things

PACKING STRATEGIES – VULKAN EXAMPLE

Unpacked 32-bit

```
// 4 VGPR per result (GCN1/2)
// 3 ops per result (GCN1/2)
float s,t,u,v;
s = min(s, -u);
t = min(t, u);
v = s*t;
```

AoS 16-bit Packing

```
// 2 VGPR per result (GCN3/4/5)
// 3 ops per result (GCN3/4)
// 2 ops per result (GCN5)
f16vec2 st,uv;
st = min(st, f16vec2(-uv.x, uv.x));
uv.y = st.x * st.y;
```

SoA 16-bit Packing

```
// 2 VGPR per result (GCN3/4/5)
// 3 ops per result (GCN3/4)
// 1.5 ops per result (GCN5)
f16vec2 ss,tt,uu,vv;
ss = min(ss, -uu);
tt = min(tt, uu);
vv = ss * tt;
```

PACKED MATH GENERAL STRATEGIES – MIX AND MATCH

- Work in Array of Structures (AoS) form
  - Each lane does the work for only one instance of computation
  - Leverage packed operations for pair of similar components (like XY then ZW)
  - Leverage packing to get reduced VGPR count for better occupancy
  - Harder to get gains due to less chances to pack

- Work in Structure of Arrays (SoA) form
  - Each lane does work for two instances of shader in parallel
  - One instance works in the low 16-bit word (x), the other in the high 16-bit word (y)
  - Easier to get gains, trivial to pack, but can have some amount of transpose overhead
  - Leverage included swizzle to broadcast in cases where both instances source same data
  - Factor transpose into a prior store so data access is already in SoA form

VULKAN AND DIRECTX 12 PACKING OVERVIEW

- Both APIs
  - For constant buffers pre-pack 16-bit constants on CPU into uints
  - Manually typecast in-shader literal constants to 16-bit in high-level shading language
- Vulkan
  - Use native 16-bit types via AMD_gpu_shader_half_float
  - “Ship One Shader” to target any GPU (GCN3/4/5 or other vendor) with one shader
  - Windows ® 7 through Windows 10 support
- DirectX 12
  - Use AMD’s GPUOpen shader header to get functions to typecast uint to/from packed types
  - Do not use built-in dot() and normalize(), instead write out manually to avoid FXC issues
  - Use one shader permutation to target GCN3/4/5
  - Windows 10 support

GCN3/4/5 16-BIT OPERATION AND CONSTANTS

- Constants need to enter the shader already packed

  Otherwise there can be higher VGPR pressure and VALU overhead

  Constants are often single-use so really do not want extra VALU overhead per constant

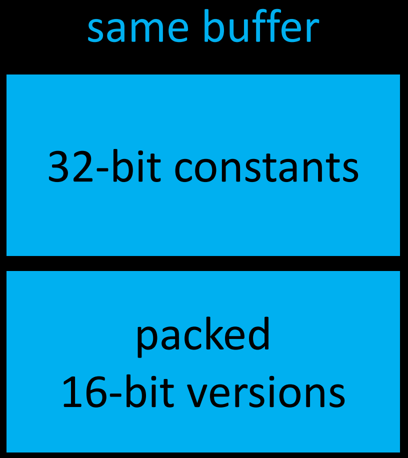
- Constants are loaded via scalar loads and stay in SGPRs

  Scalar ALU does not have floating point operations (nor conversions)

- Run-time packing of constants requires two VALU operations

  This moves packed constants into VGPRs

  Which is quite bad for register usage and can negates gains



- For low frequency constants

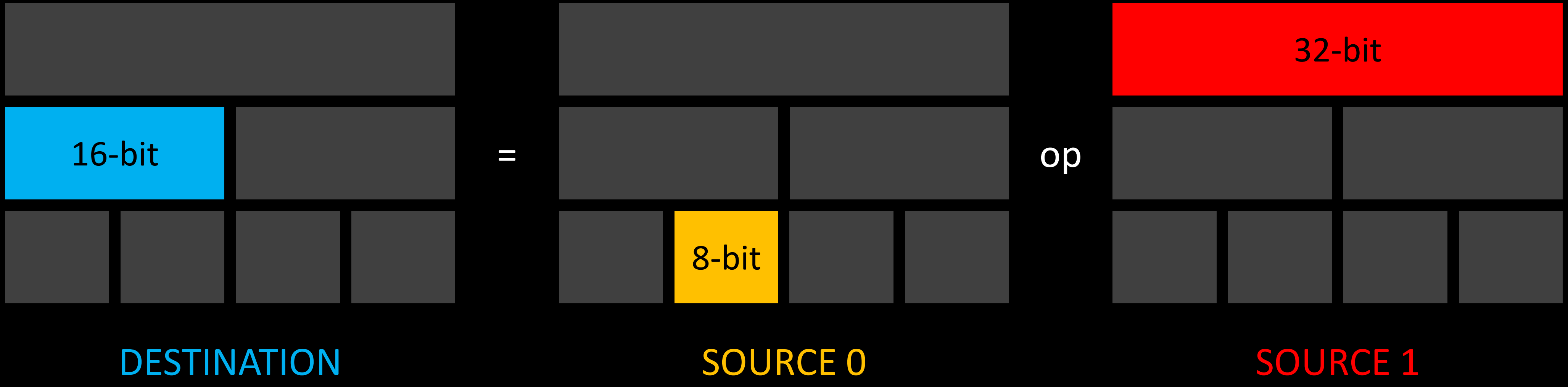
  Send packed and unpacked in same buffer, use Ship One Shader to adapt at load-time for GPU

DIVING INTO INSTRUCTION LEVEL DETAILS

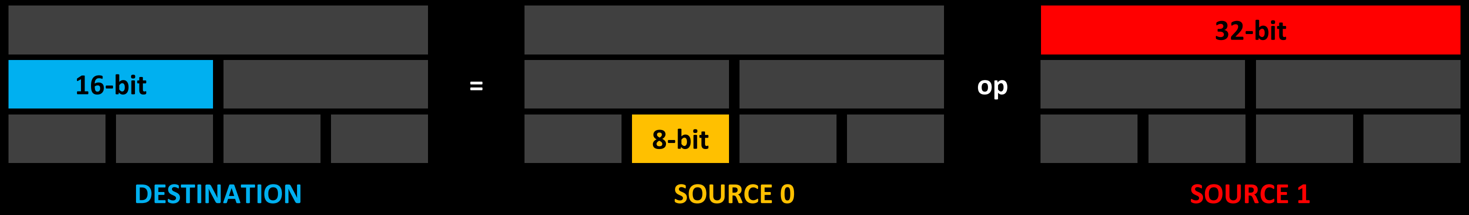
- Pack/unpack support starts on GCN3 with SDWA

SDWA – SUB DWORD ADDRESSING

- Added on GCN3, ability to unpack do an operation and repack in 1 clock



SDWA OPCODE MODIFIER



- Applicable to 32-bit VOP1/VOP2/VOPC instructions (2 source operand max)
- VOP_SDWA is a 32-bit opcode extension DWORD

[7:0] SRC0 – VGPR for src0

[10:8] DST_SEL – Select where to pack: byte {0,1,2,3} or word {0,1} or dword {0}

[12:11] DST_UNUSED – Select preserve or overwrite unused bits (with zero or sign extend)

[13] CLAMP – For floating point, optional clamp to [0.0, 1.0]

[18:16] SRC0_SEL – Select where to unpack: byte {0,1,2,3} or word {0,1} or dword {0}

[19] SRC0_SEXT – For integers, select zero or sign extension on unpack

[20] SRC0_NEG – For floating point, optional source negation

[21] SRC0_ABS – For floating point, optional source absolute value (before optional negation)

[26:24] SRC1_SEL – Select where to unpack: byte {0,1,2,3} or word {0,1} or dword {0}

[27] SRC1_SEXT – For integers, select zero or sign extension on unpack

[28] SRC1_NEG – For floating point, optional source negation

[29] SRC1_ABS – For floating point, optional source absolute value (before optional negation)

SDWA FOR INTEGERS

- For 8-bit and 16-bit integers both signed and unsigned
  - Provides bitfieldInsert() for any 8-bit or 16-bit aligned value in DST
  - Provides zero or sign-extend 8-bit or 16-bit output to 32-bit DST
  - Provides shift left by {8,16,24}-bits when 8-bit output extended to 32-bit DST
  - Provides shift left by 16-bits when 16-bit output extended to 32-bit DST
  - Provides bitfieldExtract() for any 8-bit or 16-bit aligned value in SRC0
  - Provides bitfieldExtract() for any 8-bit or 16-bit aligned value in SRC1

- Keep data packed until usage for 32-bit integer operations
- Repack intermediate data to save VGPRs

- Keep dynamically uniform values 32-bit (SALU does not have SDWA)

SDWA FOR INTEGERS – DISASSEMBLY

- Simple Vulkan shader example

```
#version 450
layout(set=0,binding=0,std430) buffer ssbo { int a[64]; int b[64]; int c[64]; };
layout(local_size_x=64,local_size_y=1) in;
void main() { c[gl_LocalInvocationID.x]=
        bitfieldExtract(a[gl_LocalInvocationID.x],8,8)+
        bitfieldExtract(b[gl_LocalInvocationID.x],0,16); }
```

- Disassembly fragment

```
BUFFER_LOAD_DWORD v1 v0 s[0:3] 0 offen
BUFFER_LOAD_DWORD v2 v0 s[0:3] 0 offen offset:256
S_WAITCNT vmcnt(0)
V_ADD_U32 v1 vcc sext(v1) sext(v2) src0_sel:BYTE_1 src1_sel:WORD_0
BUFFER_STORE_DWORD v1 v0 s[0:3] 0 offen offset:512
```

SDWA FOR FLOATING POINT

- Unlike integer, SDWA for 16-bit float needs to use 16-bit float operations

- Extends VOP1/VOP2/VOPC to have some VOP3 features
  - Provides optional saturate() on DST
  - Provides optional ABS then optional NEG on any combination of SRC0 and SRC1

GCN3/4 16-BIT OPERATION SUPPORT

- AMD_gpu_shader_half_float
- Single-rate 16-bit float support

  {ADD, CEIL, COS, CVT*, DIV, EXP, FLOOR, FMA, FRACT, INTERP, LDEXP, LOG, MAC, MADAK, MADMK, MAX, MIN, MUL, RCP, RNDNE, RSQ, SAD, SIN, SQRT, TRUNC, ... etc }
- Single-rate 16-bit integer support

  Specialized 16-bit integer instructions {ADD, MAD, MAX, MIN, SHL, SHR, SUB, ... etc }

  Plus using SDWA with 32-bit ops for others

- SDWA supplies the ability to swizzle source and destination

  d.y = a.x * b.y;

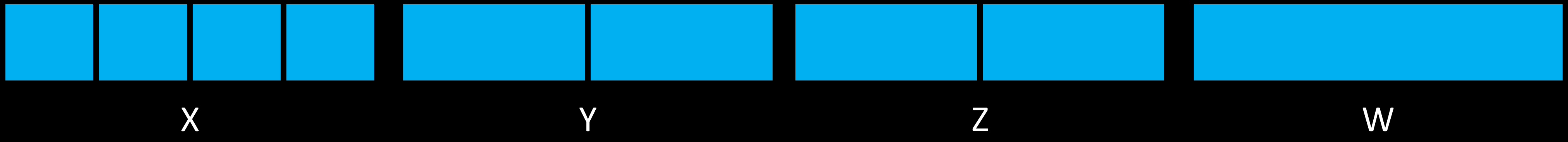
- Aim to provide register file savings

GCN3/4 VMEM AND PACKING

- GCN3/4 VMEM unit works with 32-bit addressing and 32-bit value return(s)
  - Can pack after return with one VALU op
    - V_CVT_PKRTZ_F16_F32 packed, source0, source1

- For non-filtered it is possible to freely alias 32-bit as packed pair of 16-bit
  - Or as a packed quad of 8-bit integer values

- 128-bit RGBA32U format can be an 8-channel format with aliasing
  - Or a 16-channel format for 8-bit integer values or mix and match across the 128-bits



GCN5 PACKED 16-BIT MATH – DOUBLE RATE OPERATIONS

- Signed and unsigned 16-bit: {ADD, MAD, MAX, MIN, SHR, SUB}
- 16-bit integer: {MUL, SHL}
- 16-bit float: {ADD, FMA, MAX, MIN, MUL}

  Non-double rate 16-bit operations work similar to GCN3/4 using SDWA

  SDWA is built into 16-bit VOP3 ops on GCN5

- Packed math includes optional source swizzle and optional floating point negate

```
d.xy = a.xx * b.xy + c.yx;
d.xy = min(a.yx, b.xx);
d.xy = f16vec2(-a.x,a.y) + f16vec2(-b.y,-b.y);
```

- Packed math includes optional clamp, aka saturate()

```
d.xy = clamp(a.xy * b.yy + c.xx, 0.0, 1.0);
```

GCN5 PACKED 16-BIT VECTOR BUFFER INSTRUCTIONS

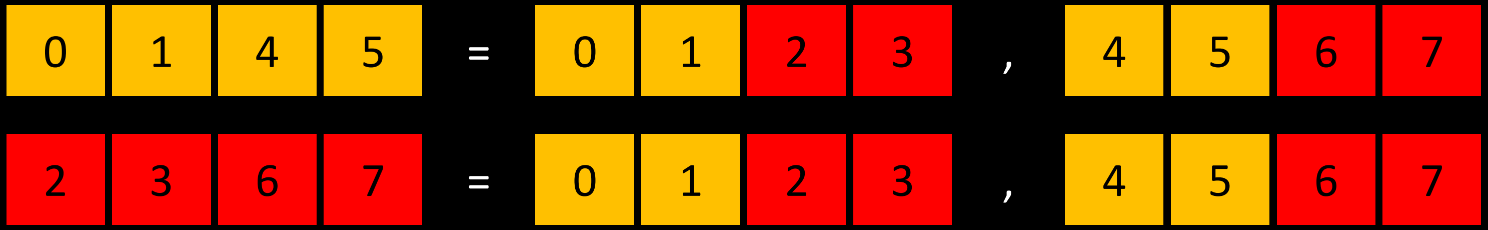
- Supported on signed/unsigned 8-bit and 16-bit memory accesses
  - Option to load into or store from the lower 16-bits of VGPR
  - Option to load into or store from the upper 16-bits of VGPR
  - These memory accesses use Structure of Arrays interface

- Supported on larger granularity {32,64,96,128}-bit accesses
  - Option to load into or store from packed values, two 16-bit values per VGPR
  - These memory accesses use Array of Structures interface

GCN5 VECTOR IMAGE INSTRUCTIONS

- Image instructions support both 32-bit and packed 16-bit float coordinates
  - In the 16-bit case one VGPR provides 2 coordinates
- Image instruction support both 32-bit and packed 16-bit data
  - In the 16-bit case one VGPR has 2 channels of data
- Can mix and match, for example 32-bit coordinates with packed 16-bit return

- This is an Array of Structures interface
  - AoS to SoA transpose can be done with two VALU operations total for the 4 16-bit values

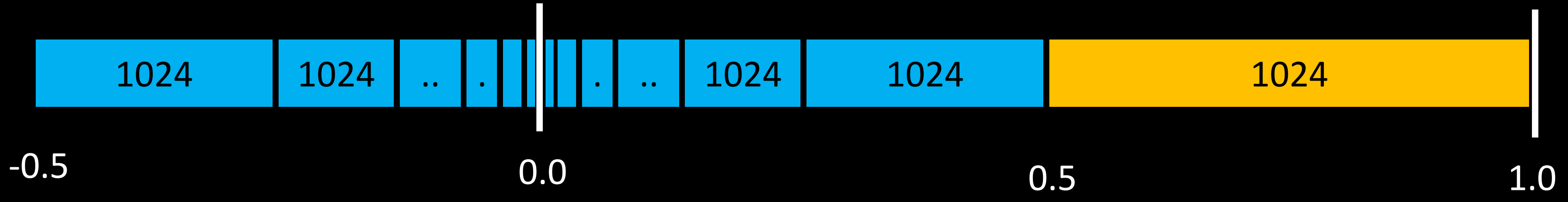


0 1 4 5 = 0 1 2 3 , 4 5 6 7

2 3 6 7 = 0 1 2 3 , 4 5 6 7

FP16 AS IMAGE COORDINATES

- Precision is better closer to zero
- The {0.5 to 1.0} range has 1024 values
- Using {0.0 to 1.0} can represent 2048 values with same precision
- Using {-0.5 to 0.5} can represent 4096 values with same precision (leverage wrap)
  - Highest precision option to represent coordinates
  - 256x256 image has 1/16 sub-texel worst case precision
  - 512x512 image has 1/8 sub-texel worst case precision



PACKED 16-BIT FLOATS IN HLSL

- HLSL challenges
  - The ‘min16float’ type is 16-bit but has 32-bit alignment (useless for constants)
  - The ‘half’ type is actually 32-bit
  - HLSL also does not have bitfieldExtract(), etc
- HLSL workarounds
  - Manual CPU-side pack/unpack into ‘uint’ for constant and buffer data
  - GPU-side unpack compiler pattern matches complex pattern and transform into NOP

```
min16float2 UnpackFP16(uint a) { return min16float2(f16tof32(uint2(a & 0xFFFF, a >> 16))); }
```

  - 16-bit literals need to be manually typecasted before being used (FXC issue)

```
a.xy = sqrt(1.0 - a.xy*a.xy); // do not use this, FXC promotes to 32-bit
a.xy = sqrt(min16float2(1.0) - a.xy*a.xy); // use this instead
```

  - Avoid built-in dot() and normalize(), write manual version

WAITING FOR OTHER PLAYERS

- Special thanks to

  AMD for allowing me to disclose and talk hardware details

  AMD Vulkan team for exposing the hardware

  Axel, Billy, Jean, and Tiago at idSoftware for pushing the hardware

  Graham Wihlidal for showing people how to use all the GCN things

  And many others who have been a great source of inspiration over the years

  . . .

- Follow-up at Timothy.Lottes@amd.com

DISCLAIMER & ATTRIBUTION

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. 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.

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 MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY 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.

ATTRIBUTION