Mitigating GPU Bank Conflicts with Swizzling

Keshav Vinayak Jha
Sr. SDE @ AMD

Refs: [1], [6], [9]

Talk Map

A 45-minute path from hardware constraints to compiler mitigation

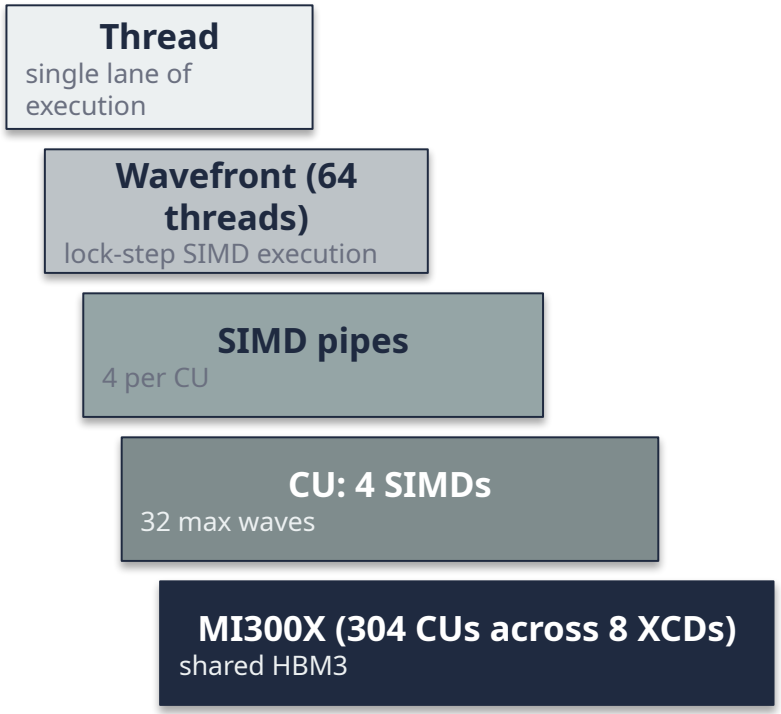
0-5 min	**GPUs – Compute, Memory, Smem**
5-18 min	**LDS, Bank Conflicts**
18-27 min	**Padding vs. XOR swizzle** Two ways to perturb the address pattern, with different costs.
27-38 min	**IREE swizzle case study** Where the swizzle hint enters the pipeline, how it lowers, and where it must fall back.
38-45 min	**Results and limits** What improved, what tied, and what constraints remain.

Thesis: bank conflicts are often address-layout bugs. The compiler can mitigate them if it preserves the program layout and changes only the physical LDS address mapping.

Refs: [1], [2], [3], [6], [9]

GPU Compute Hierarchy (AMD MI300X)

Threads -> wavefronts -> 4 SIMD pipes/CU -> 304 CUs/GPU.



- Wavefront = the unit of execution. 64 threads run the same instruction in lock-step.
- Each SIMD has 8 wave slots on gfx942: 4 x 8 = 32 max waves/CU.
- Each CU has one LDS and L1/vector cache shared by its 4 SIMDs.
- Inter-CU communication only via L2 / 256 MiB Infinity Cache / HBM (slow).

Refs: [1], [2], [3], [12]

Launch Grid: Many Workgroups

A kernel dispatch creates independent workgroups; each workgroup contains wavefronts.

Kernel dispatch / grid

WG0	WG1	WG2	WG3	WG4	WG5	WG6
WG7	WG8	**WG9**	WG10	WG11	WG12	WG13
WG14	WG15	WG16	WG17	WG18	WG19	WG20
WG21	WG22	WG23	WG24	WG25	WG26	WG27

One workgroup

256 threads = 4 wavefronts

- wavefront 0: 64 lanes
- wavefront 1: 64 lanes
- wavefront 2: 64 lanes
- wavefront 3: 64 lanes

HIP launch terms

```
dim3 block(256);        // WG size
dim3 grid(ceilDiv(N, 256)); // # WGs
hipLaunchKernelGGL(kernel, grid, block,
                   0, stream, N);

int wg  = blockIdx.x;   // workgroup id
int tid = threadIdx.x;  // thread in WG
```

Grid = all workgroups in this kernel dispatch

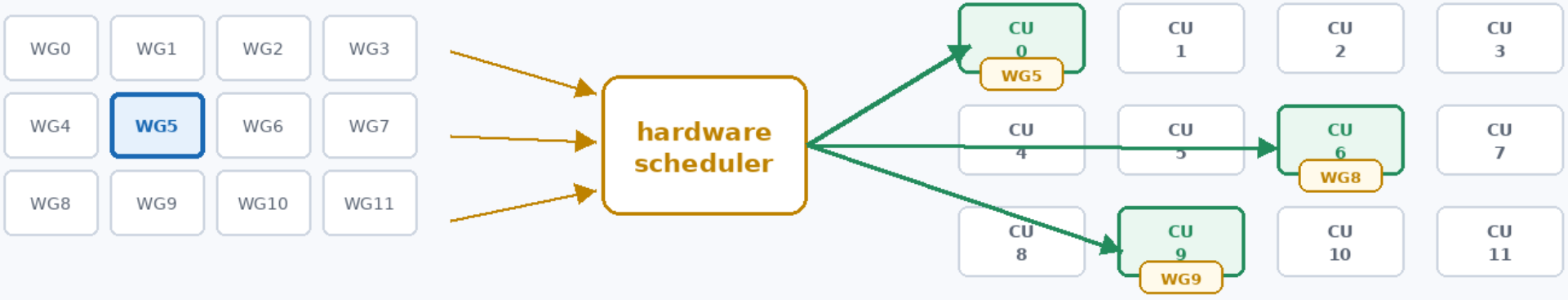
Workgroups are independent unless the kernel uses global memory side effects.

Grid → CU assignment → One CU → LDS slices → LDS instruction

Refs: [2]

Scheduling: Workgroups Land on CUs

The scheduler assigns whole workgroups to CUs; many CUs execute different workgroups in parallel.



Workgroup placement is whole-workgroup placement.

A workgroup runs on one CU; it is not split across multiple CUs.

A large grid gives the scheduler enough independent workgroups to keep many CUs busy.

Workgroup tile mapping

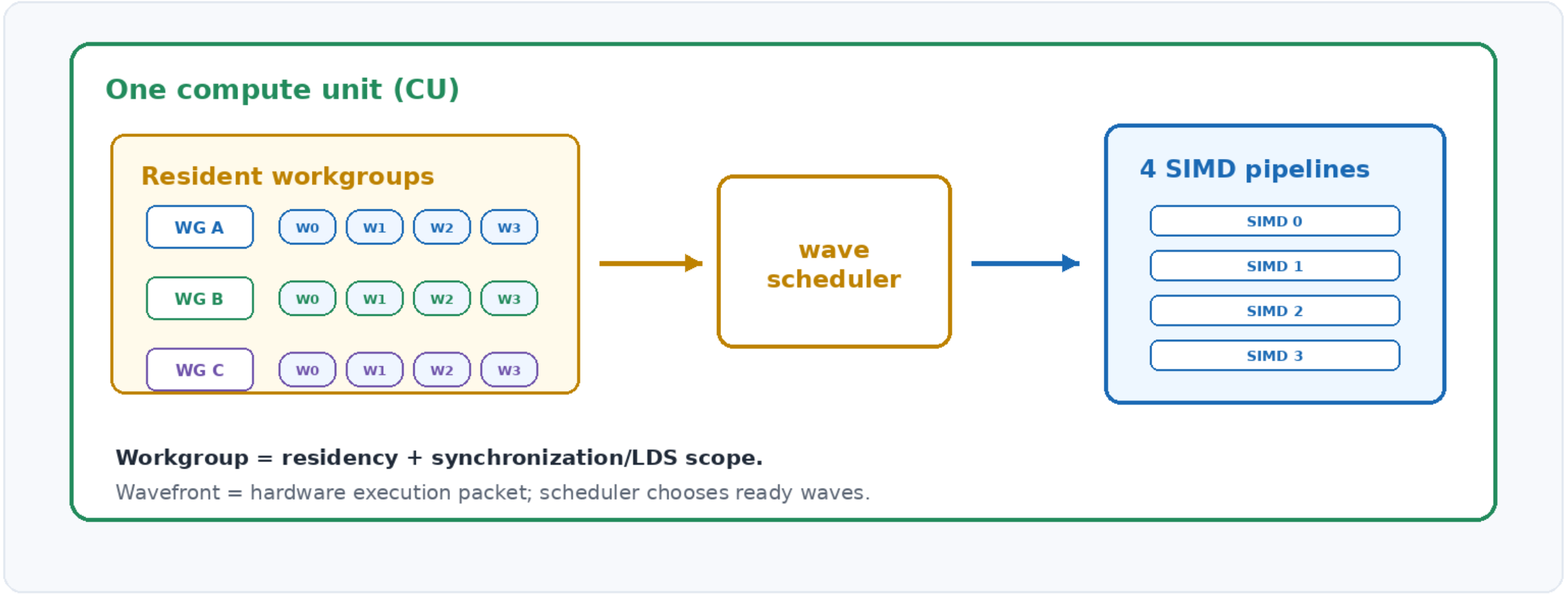
```
constexpr int WG = 16; // tile dim
__global__ void kernel(...) {
  int tid = threadIdx.x; // 0..255
  int row = blockIdx.y*WG + tid/WG;
  int col = blockIdx.x*WG + tid%WG;
} // scheduler chooses CU
```

Grid → **CU assignment** → One CU → LDS slices → LDS instruction

Refs: [2], [4]

Inside One CU: Resident Waves

Workgroups are resident on a CU; the CU schedules their wavefronts onto shared SIMD pipelines.

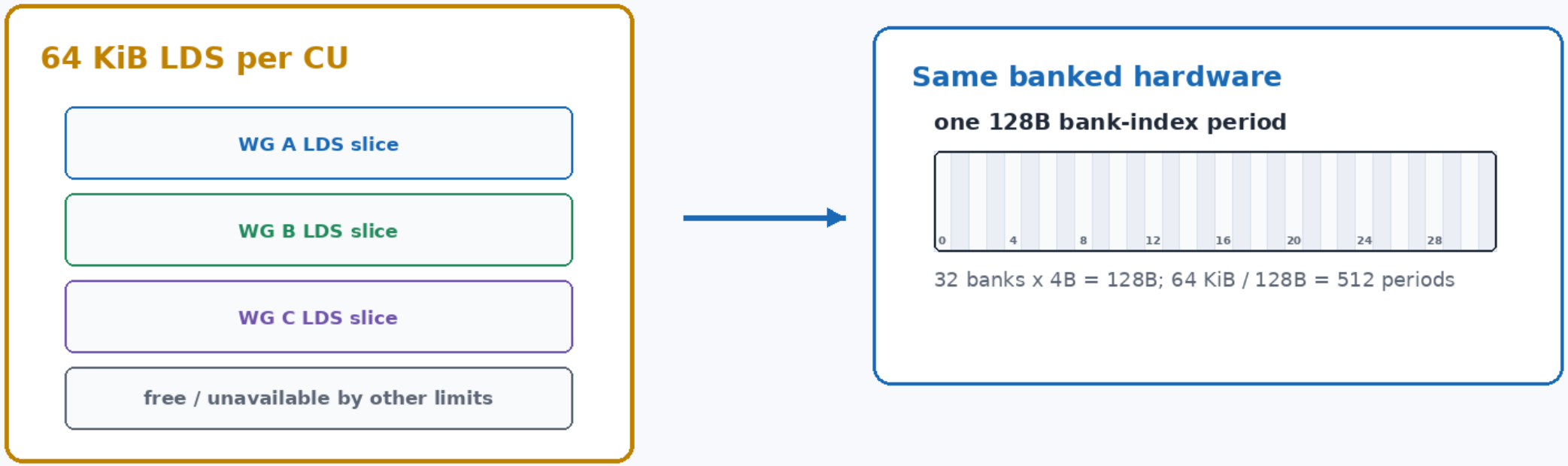


Grid → CU assignment → **One CU** → LDS slices → LDS instruction

Refs: [2], [3], [4]

LDS Allocation: Slices on One Banked Memory

Resident workgroups get separate LDS slices, but those slices all use the same 32-bank LDS hardware.



Capacity is partitioned by resident workgroup.

The bank mapping repeats across every slice because the banks are CU-local hardware.

Grid → CU assignment → One CU → **LDS slices** → LDS instruction

Refs: [1], [3], [4]

One LDS Instruction: Where Conflicts Happen

A regular stride can alias the low address bits, making many lanes hit the same LDS bank.

Bank mapping rule

bank = (byte_addr / 4) % 32

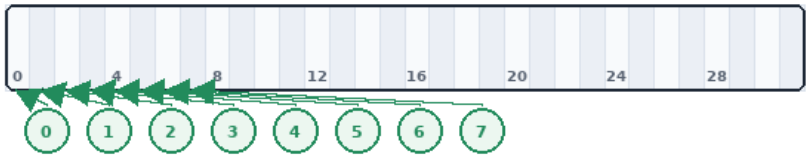
32 banks x 4B => bank pattern repeats every 128B

Contiguous row access

tile[row][lane]

addr = base + (row*LD + lane)*4

bank = lane % 32



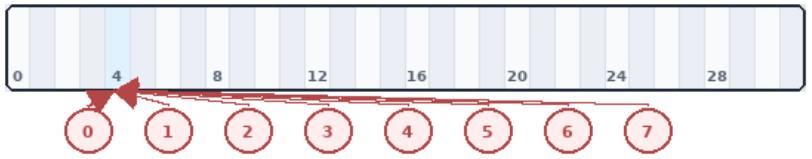
same instruction

Column / stride access

tile[lane][col] // LD = 32 f32

addr = base + (lane*32 + col)*4

bank = col



Why: bank advance per lane = (stride_bytes / 4) % 32.

Shared factors with 32 use fewer banks; advance 0 aliases one bank.

Here: stride_bytes = 32 f32 x 4B = 128B.

128B is one bank period, so bank advance is 0.

Grid → CU assignment → One CU → LDS slices → **LDS instruction**

Refs: [3], [4], [5]

GPU Memory Hierarchy

Simplified view: CU-local storage first; chip-level caches and HBM farther out.

Tier	Scope	Size (MI300X)	Latency	Notes
VGPR/SGPR files	per-CU backing	512 KiB VGPR; 12.5 KiB SGPR	very low	Compiler-allocated; spills go to scratch.
LDS (shared memory)	workgroup slice on CU	64 KiB per CU	low latency	Programmer-managed scratchpad. Bank-parallel.
L1 vector cache	per CU	32 KiB per CU	moderate latency	Global-memory vector cache.
L2 cache	per-XCD slice	32 MiB total (4/XCD)	higher latency	Shared by CUs in each XCD.
HBM3 (global)	GPU-wide	192 GiB HBM3	highest latency	5.3 TB/s; behind a 256 MiB Infinity Cache (L3).

LDS is the only programmer-controlled fast memory.

It exists for one job: stage data so that multiple wavefronts can re-read it without going to HBM.

Critical for tile-based algorithms (matmul, attention) where every element of a tile is read many times.

Refs: [1], [3], [4]

Why LDS Matters for Matmul/Attention

MFMA reuses each tile element many times. HBM can't keep up.

MFMA (Matrix Fused Multiply-Add): hardware intrinsic that does a matrix multiply in a few cycles.

- e.g. MFMA_F32_16x16x16_F16 multiplies a 16×16 tile per instruction.
- Each element of the input tiles is reused 16 times across the operation.

Attention's $QK^T(=P)$ and $P \cdot V$ both decompose into a chain of MFMAs.

Loading every tile element from HBM repeatedly would spend most of the kernel on memory latency.

Solution: stage the tile in LDS once, then have the wavefront read it many times.

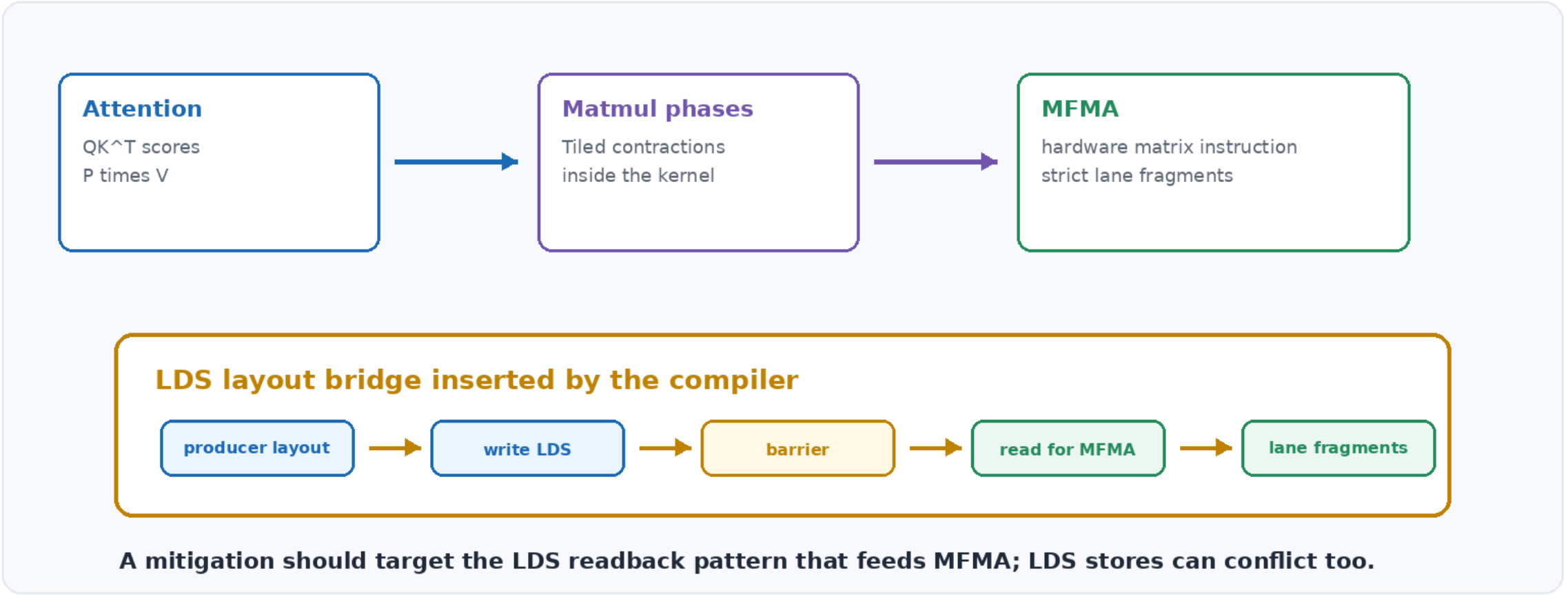
- HBM read: high latency, global-memory bandwidth budget.
- LDS read: much lower latency, bank-parallel service.

```
// Materialized LDS promotion.
%lds = memref.alloc() : memref<64x128xf16, #gpu.address_space<workgroup>>
%g   = vector.transfer_read  %global[%g0, %g1], %zero : vector<4xf16>
vector.transfer_write %g, %lds[%r, %c] {in_bounds = [true]}
gpu.barrier memfence [#gpu.address_space<workgroup>]
// Per-lane fragments; 64 lanes collectively form the MFMA tile.
%lhs_lane = vector.transfer_read %lds[%r, %k], %zero : vector<4xf16>
%rhs_lane = vector.transfer_read %lds[%k, %n], %zero : vector<4xf16>
%out_lane = iree_codegen.inner_tiled ins(%lhs_lane, %rhs_lane) outs(%acc_lane)
       {kind = #iree_gpu.mma_layout<MFMA_F32_16x16x16_F16>,
        semantics = #iree_gpu.mma_semantics<distributed = true>}
```

Refs: [3], [6], [11], [13]

MFMA Creates the LDS Readback

Attention uses matrix instructions; LDS bridges producer layout and MFMA fragments.



Refs: [3], [6], [11]

MFMA Readback Causes the Conflict

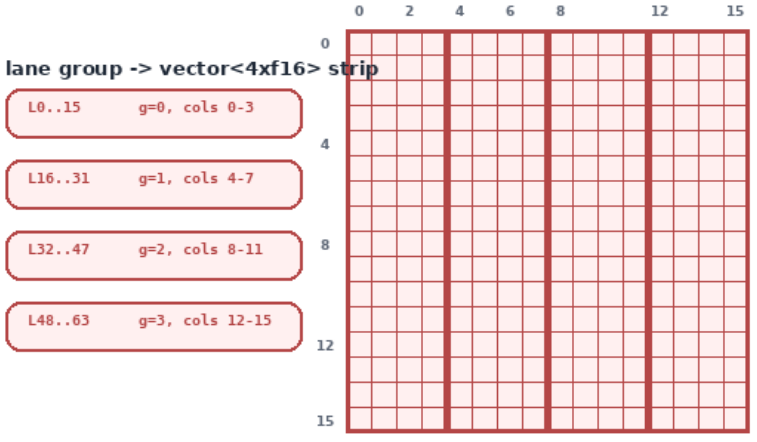
The wave reads correct fragments, but the LDS bank formula maps them together.

MFMA_F32_16x16x16_F16

operand tile = 16 x 16 = 256 f16; 64 lanes carry 4 f16 each

example LDS row pitch LD = 128 f16; each lane reads vector<4xf16>

16x16 MFMA window: four 16x4 groups



each group spans rows 0..15; each lane loads 4 contiguous f16 values

Full vector<4xf16> read

```
g = lane // 16, row = lane % 16, j = 0..3
col = 4*g + j
bank(j) = floor(((row * 128 + col) * 2) / 4) % 32
        = (row * 64 + 2*g + floor(j/2)) % 32
        = 2*g + floor(j/2)
```

lanes 0..15: cols 0..3 -> banks 0,1
lanes 16..31: cols 4..7 -> banks 2,3
lanes 32..47: cols 8..11 -> banks 4,5
lanes 48..63: cols 12..15 -> banks 6,7

Bank view

two banks per group; 16 lanes per bank-word

Refs: [3], [5], [6]

Conflict Arithmetic

A small modulo calculation predicts the common cross-row conflict pattern.

GCD predicts the conflict from slide 12

```
bank = floor(byte_addr / 4) % 32
```

1. Bank advance per row

```
LD = 128 f16
row_stride = 128 * 2B = 256B
advance = (256B / 4B) % 32
        = 64 % 32
        = 0
```

2. GCD gives the period

```
d = gcd(advance, 32)
  = gcd(0, 32) = 32
repeat_period = 32 / d = 1
=> 1 distinct bank-word

16 rows / 1 bank-word
=> 16-way conflict
```

3. Apply to vector<4xf16>

```
4 f16 = 8B = two bank-words
```

For lanes 0..15:

```
cols 0,1 -> bank 0: 16 lanes
cols 2,3 -> bank 1: 16 lanes
```

Same advance=0 logic repeats for the other groups.

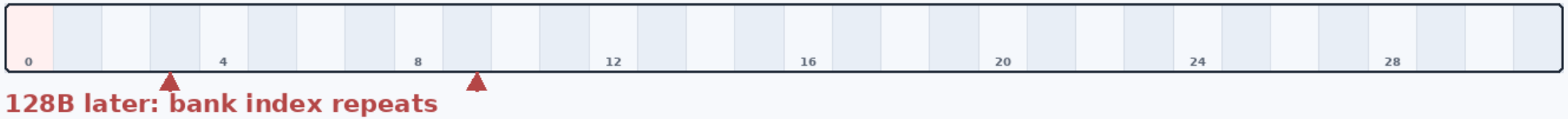
Takeaway: advance 0 means the row term never changes the bank. **A 16-row group therefore creates a 16-way collision per bank-word.**

Refs: [3], [5]

Why Common Tiles Collide

Power-of-two tile widths often line up with the 128-byte LDS bank period.

Power-of-two row strides often equal whole bank periods



f32 LD=32	f16 LD=64	f16 LD=128	f32 LD=128
32 x 4B	64 x 2B	128 x 2B	128 x 4B
128B	**128B**	**256B**	**512B**
advance 0	advance 0	advance 0	advance 0

This is why simple unmitigated LDS layouts are often conflict-prone for MFMA tiles.

Refs: [3], [5]

Padding: Shift Row Starts

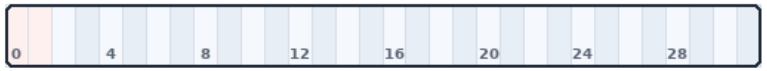
The existing mitigation changes the row stride so rows start on different banks.

Before padding

physical LD = 128 f16

```
lane row = lane % 16, j = 0..3

elem = row * 128 + j
bank(j) = floor((elem * 2) / 4) % 32
        = (64*row + floor(j/2)) % 32
        = floor(j/2)
```




```
lanes 0..15, cols 0,1 -> bank 0
lanes 0..15, cols 2,3 -> bank 1
```

two 16-way conflicts

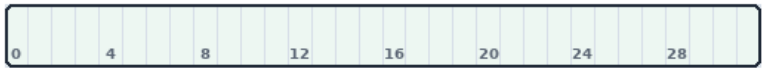
add 4 scalar f16 / row

After padding

physical LD = 132 f16

same logical columns; wider LDS row pitch

```
elem = row * 132 + j
bank(j) = floor((elem * 2) / 4) % 32
        = (66*row + floor(j/2)) % 32
        = (2*row + floor(j/2)) % 32
```




```
lane 0  row 0  -> banks 0,1
lane 1  row 1  -> banks 2,3
lane 2  row 2  -> banks 4,5
...     ...    -> ...
lane 15 row 15 -> banks 30,31
```

one lane per bank-word for this 16-row group

Padding physically widens LDS rows; the address formula changes because the row stride changed.

Refs: [3], [7]

Where Padding Falls Short

Padding works, but it pays in LDS capacity and address arithmetic.

Spends LDS

+3.1% for 128->132
+6.25% for 64->68
can affect occupancy

Targets one shape

mitigates row-stride aliasing
not every vector load/store
pattern

Address math

extra integer
address work

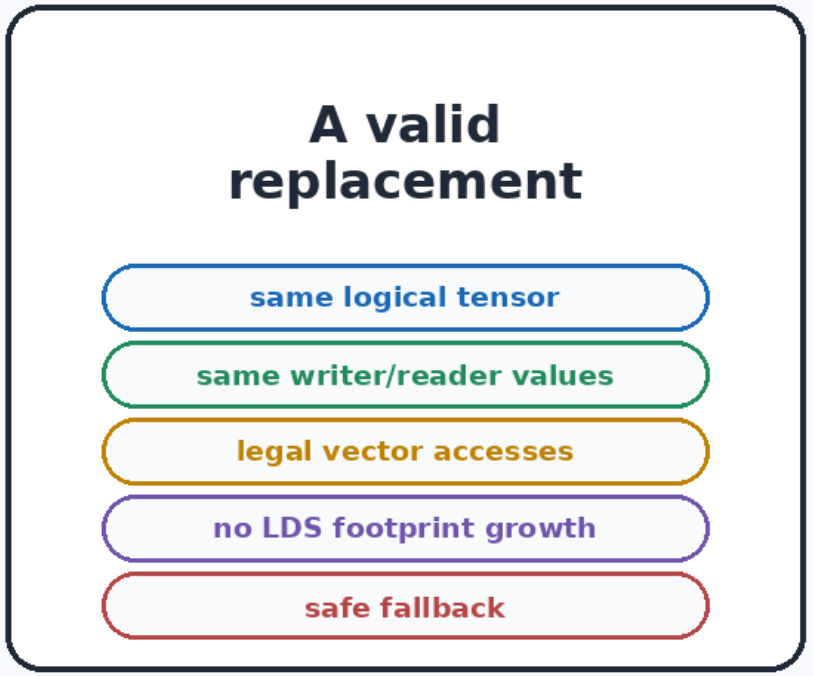
Padding is a strong fallback, but it is not free.

Ideally, we keep the mitigation while avoiding those costs when it is legal.

Refs: [7], [9], [10]

What the Replacement Must Preserve

Before changing the LDS layout, the compiler has to keep these contracts.



If the compiler cannot prove these, it should keep the existing path.

Refs: [6], [7]

Swizzle: Permute Columns Per Row

The XOR shuffle has four parameters; two define the row and access-group size.

Attribute shape

```
#iree_codegen.xor_shuffle<
  row_width,
  access_width,
  row_stride,
  per_phase>
```

running example:

```
#iree_codegen.xor_shuffle<128, 4, 128, 1>
```

What each parameter means

- **row_width:** columns/elements in one LDS row; 128 f16 here
- **access_width:** contiguous columns moved together; 4 -> 32 column groups
- **row_stride:** element distance between logical row starts (128 here)
- **per_phase:** # rows sharing the same column-group permutation

Derived:

```
groups_per_row = row_width / access_width
phase = (row / per_phase) % groups_per_row
physical_group = logical_group xor phase
```

A swizzle does not change tile[row][col]; it changes the physical LDS column group.

Refs: [8]

Swizzle: One Address Example

Apply the formula once, then map the resulting LDS address to a bank.

```
#iree_codegen.xor_shuffle</*row_width=*/128, /*access_width=*/4, /*row_stride=*/128, /*per_phase=*/1>
```

One logical access (`tile[3][8]`)

```
groups= row_width / access_width = 128 / 4 = 32
logical_col_group = floor(col / access_width) = floor(8 / 4) = 2
col_in_group = col % access_width = 8 % 4 = 0
Phase = floor(row / per_phase) % groups = floor(3 / 1) % 32 = 3
physical_col_group = logical_col_group xor phase = 2 xor 3 = 1
physical_col = physical_col_group*access_width + col_in_group = 4
```

original offset = row*row_stride + col = 3*128 + 8 = 392 elems

swizzled offset = row*row_stride + physical_col = 3*128 + 4 = 388 elems

```
byte offset f16 = 388 * 2 = 776B
```

Address -> bank

byte offset f16 = 776B


```
bank = (byte_offset / 4) % 32
     = (776 / 4) % 32
     = 194 % 32 = 2
```

The hardware bank function is fixed; swizzle changes the address fed into it.

Refs: [3], [8]

What Swizzle Does to the Read

The same MFMA access now maps its 16 rows across LDS banks.

Same MFMA read: row = lane % 16, j = 0..3, col = 4*(lane//16) + j

Start with the same MFMA readback. Swizzle only changes the physical LDS column before bank mapping.

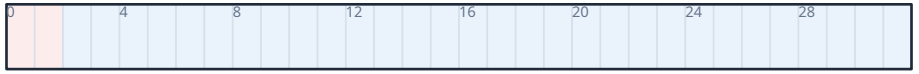
Without swizzle

```
row = lane % 16,  j = 0..3
col = 4*(lane//16) + j
bank = (64*row + 2*(lane//16) + floor(j/2)) % 32
```

```
= (2*(lane//16) + floor(j/2)) % 32
  // row term drops out
```

For lanes 0..15 (lane//16 = 0):

row 0..15 all hit bank pair {0,1}

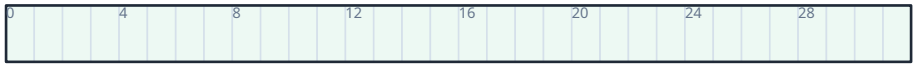


same two bank-words repeat across rows

With XOR swizzle

```
row = lane % 16,  j = 0..3
col = 4*(lane//16) + j
logical_group = floor(col/4) = lane//16
in_group = col % 4 = j
phase = row  (per_phase = 1)
physical_group = logical_group xor phase
phys_col = 4*physical_group + in_group
bank = (64*row + 2*physical_group
        + floor(j/2)) % 32
     = (2*((lane//16) xor row)
        + floor(j/2)) % 32
```

```
lanes 0..15: row 0->{0,1}, row 1->{2,3}
             ... row 15->{30,31}
```



The MFMA-facing logical read is unchanged; the physical column breaks the row-stride alias.

Refs: [8]

IREE Implementation Flow

State flow, larger IR snapshots, and why padding remains the fallback.

State flow

GPUVectorAlloc
derive writer/reader layouts

↓

Bufferize + flatten
tensor -> flat memref

↓

GPUReduceBankConflictsPass
skip hinted; pad fallback

↓

ResolveSwizzleHints
rewrite final LDS offsets

Before/after IR snapshots

```
// before GPUVectorAlloc
%v = ... : vector<16x128xf16>
%to = iree_vector_ext.to_layout %v
      to layout(#reader)
      {shared_memory_conversion =
        #iree_gpu.derived_thread_config}

// after GPUVectorAlloc: vector -> LDS -> vector
%a = bufferization.alloc_tensor()
     : tensor<16x128xf16, #wg>
%h = iree_codegen.swizzle_hint %a
     #xor<128,4,128,1>
%w = vector.transfer_write %v, %h[...]
%b = iree_gpu.value_barrier %w
%r = vector.transfer_read %b[...]
%to = iree_vector_ext.to_layout %r
      to layout(#reader)

// after bufferize + flatten
%f = memref.alloc() : memref<2048xf16,#wg>
%h2 = iree_codegen.swizzle_hint %f
      #xor<128,4,128,1>
%view = memref.expand_shape %h2
      : memref<2048xf16> -> memref<16x128xf16>

// ResolveSwizzleHints
%phys = swizzleOffset(%logical)
%r = vector.load %f[%phys]
```

Why fallback exists

Layout:

```
#reader =
  #iree_vector_ext.nested_layout<
    subgroup_tile = [1, 1],
    batch_tile = [2, 4],
    outer_tile = [4, 1],
    thread_tile = [4, 2],
    element_tile = [1, 4],
    subgroup_strides = [0, 0],
    thread_strides = [1, 4]
  >
%y = iree_vector_ext.to_layout %x
      to layout(#reader)
      : vector<32x32xf16>
```

Skip XOR if:

- layout missing/unsupported
- empty element_tile or rank-0
- invalid row/access groups
- f16/bf16 access mismatch
- rows already spread across banks

No swizzle_hint => padding may apply.

Refs: [6], [7], [8]

PR #23778: The Compiler Change

The PR adds a conservative decision point, then reuses the existing swizzle machinery.

1. Decide at vector allocation

GPUVectorAlloc.cpp computes a legal XOR shuffle from writer/reader layouts.

Only emitted when the flag is on and the logical tensor is still visible.

v

2. Represent without changing shape

iree_codegen.swizzle_hint wraps the LDS allocation.

#iree_codegen.xor_shuffle records row_width, access_width, row_stride, per_phase.

v

3. Resolve at concrete accesses

ResolveSwizzleHints verifies users and rewrites LDS offsets.

Stores and loads use the same physical address mapping.

Refs: [6], [7], [8]

IR View: Same Logical Tensor

The algorithm sees the same tensor; only the LDS address calculation changes.

Before: plain LDS address

```
%m = memref.alloc() : memref<...>
%w = vector.transfer_write %tile,
     %m[%row, %col]
...
%r = vector.transfer_read %m[%row, %col]
// physical address follows logical offsets
```

After: same logical tensor + hint

```
%h = iree_codegen.swizzle_hint %m
     #iree_codegen.xor_shuffle<
       row_width=128, access_width=4,
       row_stride=128, per_phase=1>
%w = vector.transfer_write %tile, %h[%row,%col]
%r = vector.transfer_read  %h[%row,%col]
```

Correctness rule

Both write and read must use the same physical-address scheme, or the reader observes a different value.

Refs: [6], [8]

Safety Rules from Review

The swizzle path must prove legality or fall back.

One classifier

One analysis decides whether a swizzle can be emitted and later resolved.

Vector boundary

Do not split vector accesses unless the layout proves the split is legal.

Fallback

Unsupported cases remain unhinted, so the padding path can still handle them.

Flag isolation

Flag off keeps the existing padding path and old lowering behavior.

Correctness first

A hint that cannot be lowered safely is not emitted. That is why the PR needed legality rules, not just an XOR formula.

Refs: [7], [8]

Evaluation Setup

Measure runtime and hardware counters across the same 110 attention shapes.

Configurations

baseline: no padding/no swizzle
padding: existing mitigation
swizzle: PR path

Sweep

MI300X / gfx942
110 attention shapes
f16 and f32

Counters

SQ_LDS_BANK_CONFLICT
LdsLatency
SQ_WAIT_INST_ANY
instruction mix

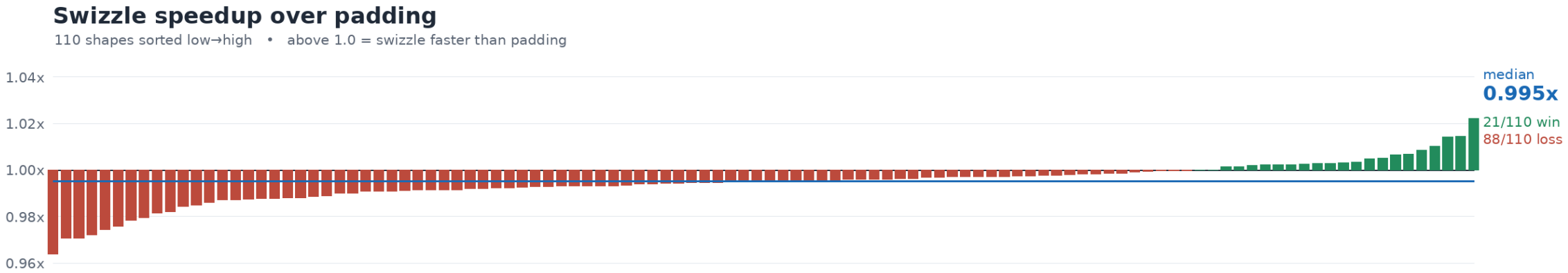
Reading the plots

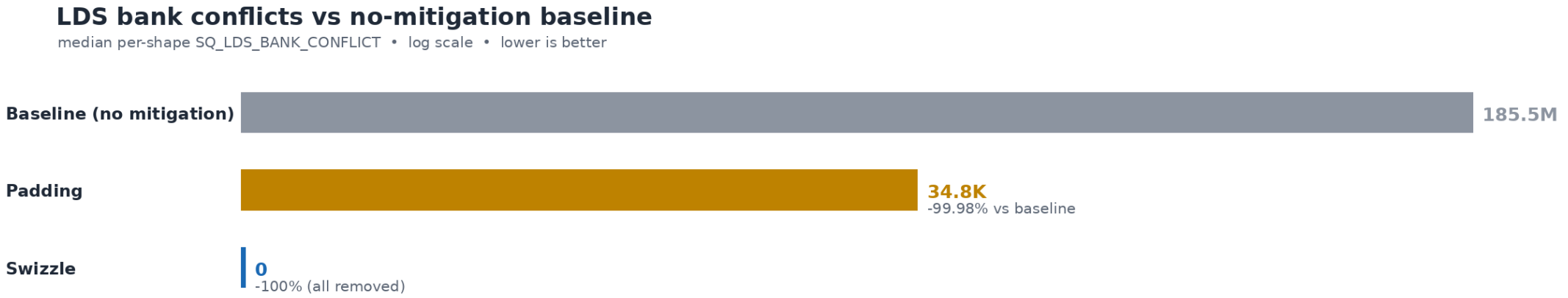
Every point is normalized against the no-mitigation baseline, so higher is better for both runtime and bank-conflict reduction.

Refs: [5], [9], [10]

Evaluation: f16 Attention Sweep

MI300X · 110 attention shapes. Top: swizzle vs padding (>1.0 = swizzle wins). Bottom: bank conflicts vs the no-mitigation baseline.





Swizzle clears every conflict (0 on 110/110 shapes); padding clears them on 55/110.

1.485x	1.477x	21/110	110/110	4.7%
median padding perf	median swizzle perf	swizzle faster vs padding	swizzle BC zero	median LDS saved

Swizzle eliminates every LDS bank conflict (0 on 110/110 shapes); padding clears them on 55/110. Runtime ties padding because these waits were not the f16 bottleneck — see next slide.

Refs: [9]

f16: Conflicts Eliminated, Runtime Ties Padding

Runtime ties padding (median 0.995×), but swizzle removes every conflict and trims LDS — a correctness/occupancy win.

Benefit: bank conflicts disappear
SQ_LDS_BANK_CONFLICT: 8.52M → 0 • BC=0 on 110/110 shapes (padding: 55/110)

v

MFMA / VMEM
unchanged

Instructions
+0.45%

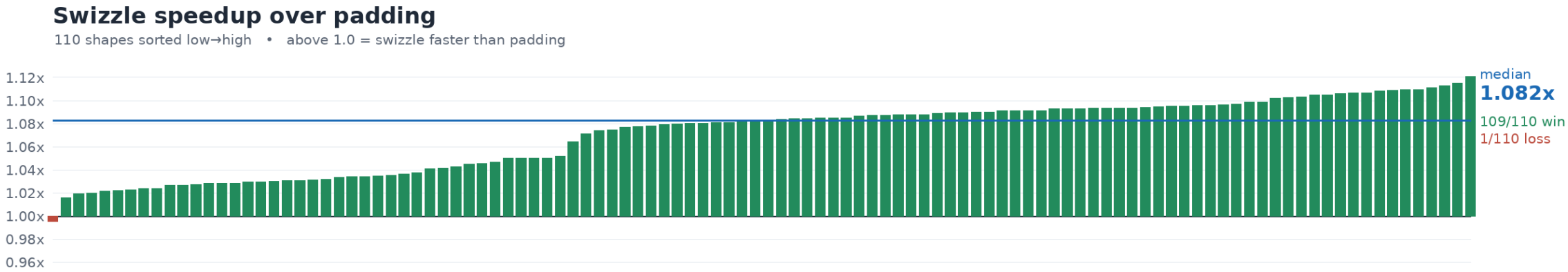
SQ_WAVE_CYCLES
+0.9%

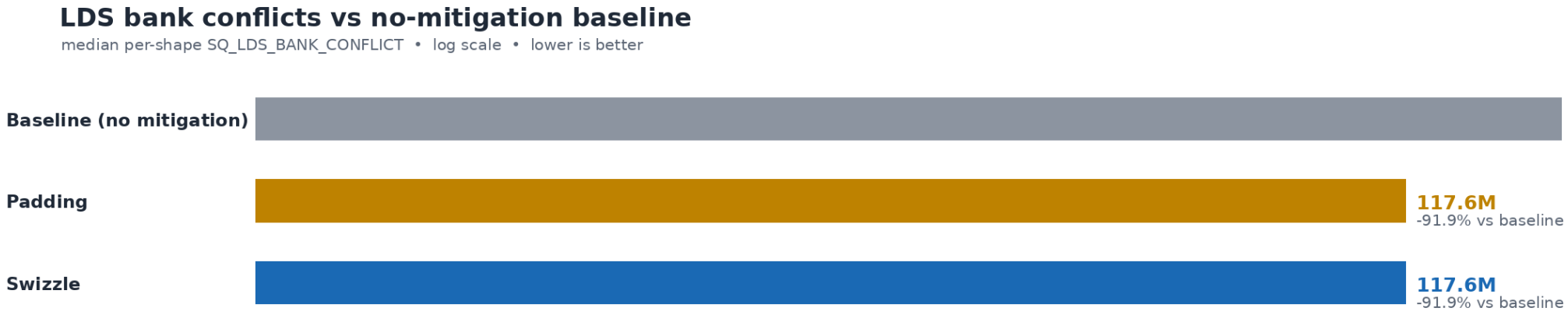
Interpretation
Swizzle is strictly cleaner: zero bank conflicts on 110/110 shapes vs padding's 55/110, and ~4.7% less LDS. The clock ties only because LDS waits were not the f16 bottleneck in this run.

Refs: [5], [9], [10]

Evaluation: f32 Attention Sweep

MI300X · 110 attention shapes. Top: swizzle vs padding (>1.0 = swizzle wins). Bottom: bank conflicts vs the no-mitigation baseline.





Padding and swizzle hit the SAME residual floor — so the f32 win is not from fewer conflicts (next slide).

1.724x	1.830x	109/110	110/110	4.1%
median padding perf	median swizzle perf	swizzle faster vs padding	same BC count	median LDS saved

Padding and swizzle reach the same residual conflict floor (identical counts), so the f32 speedup does not come from removing conflicts — the next slide shows where it comes from.

Refs: [9]

Why f32 Speeds Up Even When BC Matches

Same total bank conflicts, but spread across more, smaller LDS reads — so the wave waits less before MFMA.

Padding trace: compact LDS, full drain

```
ds_read2_b32 v[86:87], v63, ...
s_waitcnt lgkmcnt(0)
v_mfma_f32_16x16x4_f32 ...
```

ds_read2_b32: two 32-bit LDS values per lane in one instruction.

lgkmcnt(0): wait until no LDS/scalar-memory/message ops remain.

Swizzle trace: split LDS, partial wait

```
ds_read_b32 v125, v68
ds_read_b32 v130, v99, offset:496
s_waitcnt lgkmcnt(1)
v_mfma... consumes v125
```

ds_read_b32: one 32-bit LDS value per lane.

lgkmcnt(1): one younger LDS op may remain while MFMA consumes the ready operand.

Measured counters (f32 1×1024×128)

SQ_LDS_BANK_CONFLICT: 532K → 532K (unchanged)

SQ_INSTS_LDS: 298K → 514K (+72%)

BC per LDS inst: 1.78 → 1.04 (-42%)

LdsLatency: 48.2 → 43.7 (-9.5%)

SQ_WAIT_INST_ANY: 2.04M → 1.62M (-21%)

SQ_WAVE_CYCLES: 7.88M → 7.67M (-2.7%)

Same total BC, spread over more LDS reads → less waiting.

Speaker line

f32 swizzle does not remove the residual bank-conflict floor (same SQ_LDS_BANK_CONFLICT). It spreads those conflicts across more, smaller LDS reads, so total LDS wait (SQ_WAIT_INST_ANY) drops 21% and the kernel finishes in fewer wave-cycles.

Refs: [5], [10]

Where Swizzle Hits Its Limits

The hardware bank model still sets the floor: gcd aliasing, element width, and wave64 phasing.

Element width

f32 is one full 4-byte bank word; f16 packs two values per bank word.

Wave64 phasing

LDS issues a wave64 access as four 16-lane phases; even a perfect spread costs 4 phases, and conflicts serialize within each.

Access asymmetry

A wide writer and narrow reader can force fallback when one XOR scheme cannot serve both.

Stride aliasing

Power-of-two row strides can still hit the 128B bank-address period.

Limit

Swizzle improves the layout, but it still operates inside the hardware bank model. Padding remains useful for unsupported cases.

Refs: [3], [5], [8]

Take-aways

The whole story in five steps.

1
LDS is banked
32 banks x 4B

2
MFMA readback
lanes may walk columns

3
Conflict pattern
row stride aliases bank

4
Mitigations
padding or XOR swizzle

5
Measured nuance
f16 flat; f32 faster

Main claim
The compiler can fix bank conflicts only when it preserves program layout and legal vector accesses.

Refs: [1], [3], [6], [7], [8], [9], [10]

Citations: Hardware and Programming Model

[1] ROCm GPU hardware specifications. MI300X/gfx942 table: 304 CUs, wavefront size 64, 64 KiB LDS, L1/L2/L3, VGPR/SGPR sizes. https://rocm.docs.amd.com/en/latest/reference/gpu-arch-specs.html
[2] HIP programming model. Threads, blocks/workgroups, grids, shared memory, and SIMT hierarchy. https://rocm.docs.amd.com/projects/HIP/en/latest/understand/programming_model.html
[12] HIP hardware implementation. CDNA3 compute-unit diagram and workgroup-to-CU scheduling/resource model. https://rocmdocs.amd.com/projects/HIP/en/develop/understand/hardware_implementation.html
[3] ROCm Compute Profiler pipeline descriptions. CDNA VALU/SIMD model, wave slots, LDS 32 banks x 4B, bank-conflict definition, scheduler. https://rocm.docs.amd.com/projects/rocprofiler-compute/en/latest/conceptual/pipeline-descriptions.html
[4] LLVM AMDGPU backend user guide, GFX942 memory model. Workgroup wavefronts on one CU, multiple SIMDs, single CU-local LDS, LDS operations. https://rocm.docs.amd.com/projects/llvm-project/en/latest/LLVM/llvm/html/AMDGPUUsage.html
[5] ROCm Compute Profiler LDS metrics. Bank conflict, bank conflicts/access, LDS instructions, LDS latency definitions. https://rocm.docs.amd.com/projects/rocprofiler-compute/en/docs-7.1.0/conceptual/local-data-share.html

Citations: IREE Swizzle Implementation

[6] IREE PR #23778 local source @ 2bee973: GPUVectorAlloc.cpp. Opt-in flag, layout-derived XOR swizzle, shared-memory roundtrip, value_barrier. /home/keshavvinayakjha/Desktop/code/iree-worktrees/gpu-swizzle/src/compiler/src/iree/compiler/Codegen/Common/GPU/GPUVectorAlloc.cpp
[7] IREE PR #23778 local source @ 2bee973: GPUReduceBankConflicts.cpp and LLVMGPU/Passes.cpp. 64-bit padding, padding skip for swizzled allocs, pass ordering. /home/keshavvinayakjha/Desktop/code/iree-worktrees/gpu-swizzle/src/compiler/src/iree/compiler/Codegen/Common/GPU/GPUReduceBankConflicts.cpp
[8] IREE PR #23778 local source @ 2bee973: ResolveSwizzleHints.cpp and IREECodegenAttrs.{td,cpp}. XORShuffleAttr formula, access-width legality, flat-contiguous verification, vector load/store offset rewriting. /home/keshavvinayakjha/Desktop/code/iree-worktrees/gpu-swizzle/src/compiler/src/iree/compiler/Codegen/Common/ResolveSwizzleHints.cpp
[11] AMD Instinct MI300 CDNA3 ISA reference and AMD GPUOpen Matrix Cores note. MFMA/matrix-core behavior and instruction naming. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-mi300-cdna3-instruction-set-architecture.pdf ; https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-matrix-cores-readme/
[13] IREE PR #24408. VectorDistribute LDS operand promotion rework; promotion types propagate to transfer_read/gather and GPUVectorAlloc materializes LDS promotion. https://github.com/iree-org/iree/pull/24408

Citations: Evaluation Data

[9] Local 110-shape sweep data and chart generator. swizzle_vs_padding_f16_latest.csv, swizzle_vs_padding_f32_latest.csv, nopad_baseline_f16.csv, nopad_baseline_f32.csv, update_swizzle_deck.py. /home/keshavvinayakjha/Desktop/code/swizzle_talk_assets/

[10] Local counter evidence summary. f32 PR counter table from trace_f32_2bee973 and f16 focused MI300X rerun trace_f16_talk_fresh_20260523. /home/keshavvinayakjha/Desktop/code/swizzle_talk_assets/counter_evidence_summary.csv

[12] Local ROCprofiler-SDK Advanced Thread Trace capture for f32 1x1024x128, padding vs swizzle. Decoded stats CSVs, code object disassembly, per-wave JSON, and raw .att files under /home/keshavvinayakjha/Desktop/code/swizzle_talk_assets/att_f32_1x1024x128/