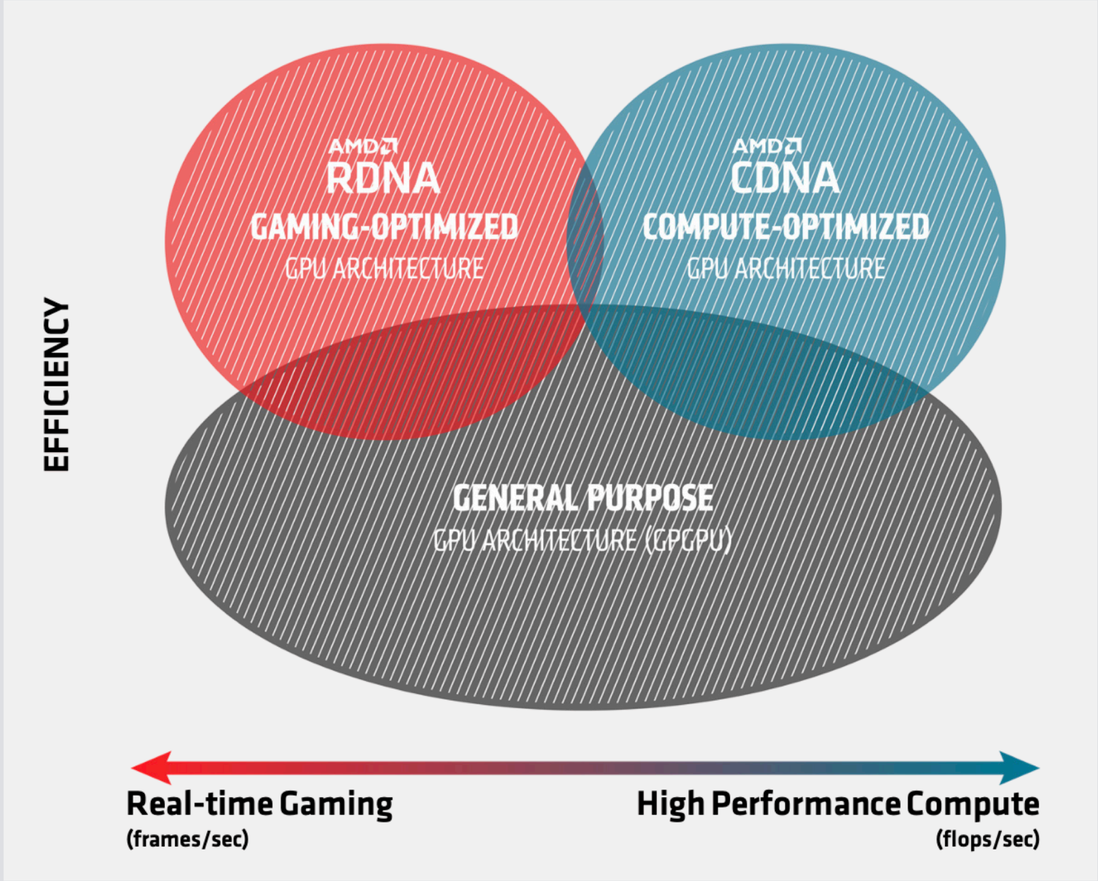
High-performance matrix multiplication on RDNA3 GPUs

Harsh Menon (nod.ai)

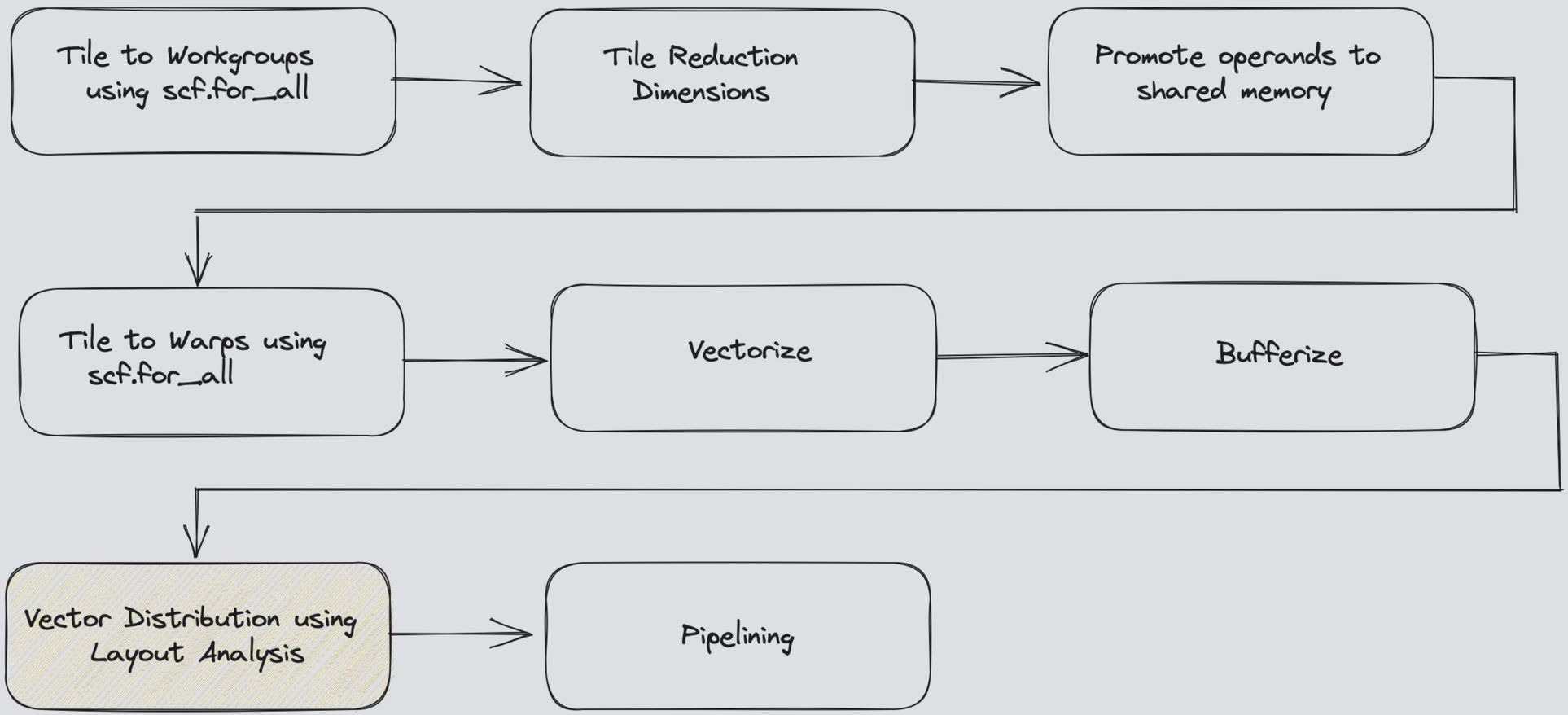
Motivation

- Matrix multiplication is the core computational unit of neural networks
- Specialized matrix-multiplication instructions exist on various GPUs (mma.sync on NVIDIA GPUs)
- AMD GPUs come in 2 different flavors - RDNA (Radeon GPUs) and CDNA (MI-series)
- RDNA3 GPUs have wmma instructions while CDNA GPUs have mfma instructions that target matrix multiplications
- Goal is to bring up the ROCM backend in IREE by starting with a high performance matrix multiplication on RDNA series GPUs targeting wmma instructions



https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf

Matrix Multiplication Pipeline



(Hardware specific)

Matrix Multiplication Pipeline using Transform Dialect

- Implement the pipeline using the transform dialect
- Exposes building blocks that can be used to build your own codegen pipeline
- Match the matrix multiplication operator (linalg.matmul, linalg.matmul_transpose_b, linalg.matmul_transpose_a)
- Create the pipeline using a series of existing and new codegen primitives such as:
  - Tiling: transform.structured.tile_to_forall_op, transform.structured.tile
  - Vectorize: transform.structured.vectorize
  - Bufferization: transform.iree.bufferize

```
transform.sequence failures(propagate) {
  ^bb0(%variant_op: !transform.any_op):

  // Get matmul op
  // ==========================================
  %matmul = transform.structured.match
ops{["linalg.matmul_transpose_b"]} in %variant_op : (!transform.any_op)
-> !transform.any_op

  // Tile and distribute to workgroups
  // ==========================================
  %forall_grid, %tiled_matmul =
  transform.structured.tile_to_forall_op %matmul tile_sizes [128, 64]
    ( mapping = [#gpu.block<x>, #gpu.block<y>] ) : (!transform.any_op)
-> (!transform.any_op, !transform.any_op)

  ...

  // Vectorize function
  // ==========================================
  %func = transform.structured.match ops{["func.func"]} in
%variant_op : (!transform.any_op) -> !transform.any_op
  transform.apply_patterns to %func {
    ...
    transform.apply_patterns.linalg.fold_unit_extent_dims_via_slices
    transform.apply_patterns.vector.cast_away_vector_leading_one_dim
  } : !transform.any_op
  %func_3 = transform.structured.vectorize %func : (!transform.any_op)
-> (!transform.any_op
```

IR before vector distribution

```
%11 = scf.for %arg0 = %c0 to %c1024 step %c32 iter_args(%arg1 = %cst) -> (vector<32x32xf16>) {
  %subview = memref.subview %0[%7, %arg0] [128, 32] [1, 1] : memref<128x1024xf16> to memref<128x32xf16, strided<[1024, 1], offset: ?>>
  %subview_1 = memref.subview %1[%8, %arg0] [64, 32] [1, 1] : memref<1280x1024xf16> to memref<64x32xf16, strided<[1024, 1], offset: ?>>
  %alloc = memref.alloc() {alignment = 64 : i64} : memref<128x32xf16, #gpu.address_space<workgroup>>
  gpu.barrier
  linalg.generic {indexing_maps = [#map6, #map6], iterator_types = ["parallel", "parallel"]} ins(%subview : memref<128x32xf16, strided<[1024, 1], offset: ?>>)
outs(%alloc : memref<128x32xf16, #gpu.address_space<workgroup>>) {
  ^bb0(%in: f16, %out: f16):
    linalg.yield %in : f16
  }
  gpu.barrier
  %alloc_2 = memref.alloc() {alignment = 64 : i64} : memref<64x32xf16, #gpu.address_space<workgroup>>
  gpu.barrier
  linalg.generic {indexing_maps = [#map6, #map6], iterator_types = ["parallel", "parallel"]} ins(%subview_1 : memref<64x32xf16, strided<[1024, 1], offset: ?>>)
outs(%alloc_2 : memref<64x32xf16, #gpu.address_space<workgroup>>) {
  ^bb0(%in: f16, %out: f16):
    linalg.yield %in : f16
  }
  gpu.barrier
  %12 = vector.transfer_read %alloc[%9, %c0], %cst_0 {in_bounds = [true, true]} : memref<128x32xf16, #gpu.address_space<workgroup>>, vector<32x32xf16>
  %13 = vector.transfer_read %alloc_2[%10, %c0], %cst_0 {in_bounds = [true, true]} : memref<64x32xf16, #gpu.address_space<workgroup>>, vector<32x32xf16>
  %14 = vector.contract {indexing_maps = [#map7, #map8, #map9], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %12, %13,
%arg1 : vector<32x32xf16>, vector<32x32xf16> into vector<32x32xf16>
  gpu.barrier
  memref.dealloc %alloc : memref<128x32xf16, #gpu.address_space<workgroup>>
  memref.dealloc %alloc_2 : memref<64x32xf16, #gpu.address_space<workgroup>>
  scf.yield %14 : vector<32x32xf16>
}
vector.transfer_write %11, %2[%5, %6] {in_bounds = [true, true]} : vector<32x32xf16>, memref<128x1280xf16>
```

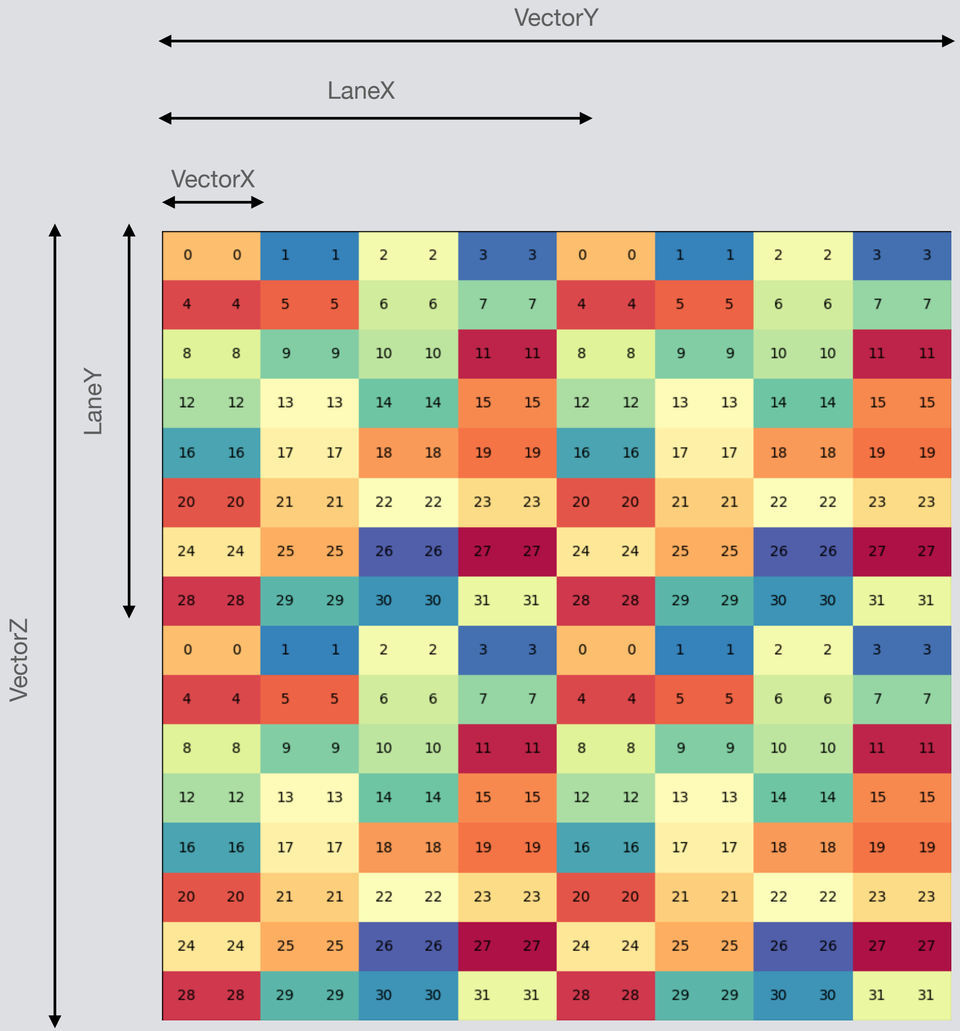
Memory Promotion

MMA operation

Higher-Dimensional Layout Representations

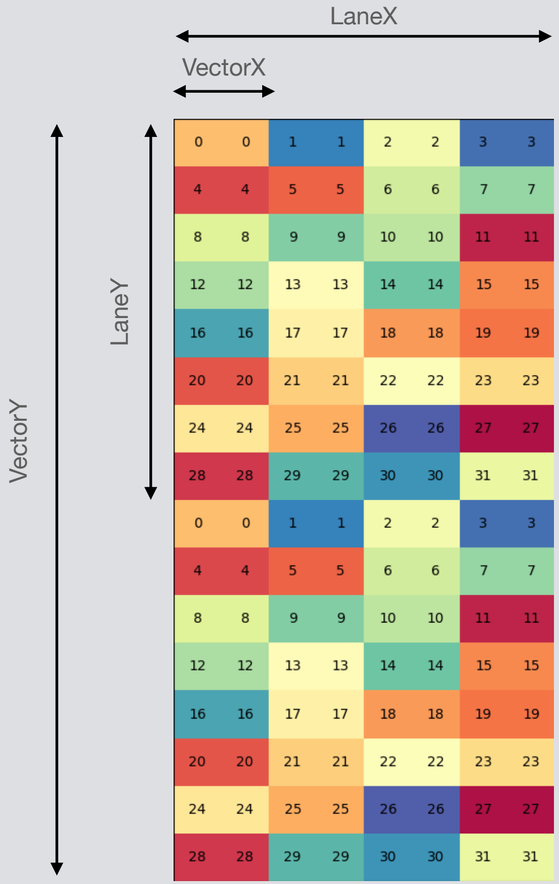
- High-dimensional layout representation for mma.sync instruction on NVIDIA GPUs
- N-dimensional generic representation of a mapping from matrix space to batch-lane-vector space
  - Vector dimensions (X, Y, Z)
  - Lane dimensions (X, Y, Z)
  - Batch dimensions (for row and column)
  - **Batch (row)** x **Batch (column)** x **LaneZ** x **LaneY** x **LaneX** x **VectorZ** x **VectorY** x **VectorX**
  - **Column**: VectorX = 2, LaneX = 4, VectorY = 2
  - **Row**: LaneY = 8, VectorZ = 2

Batch (row)	Batch (col)	Lane Z	Lane Y (0)	Lane X (1)	Vec Z (1)	Vec Y (2)	Vec X (0)
1	1	1	8	4	2	2	2

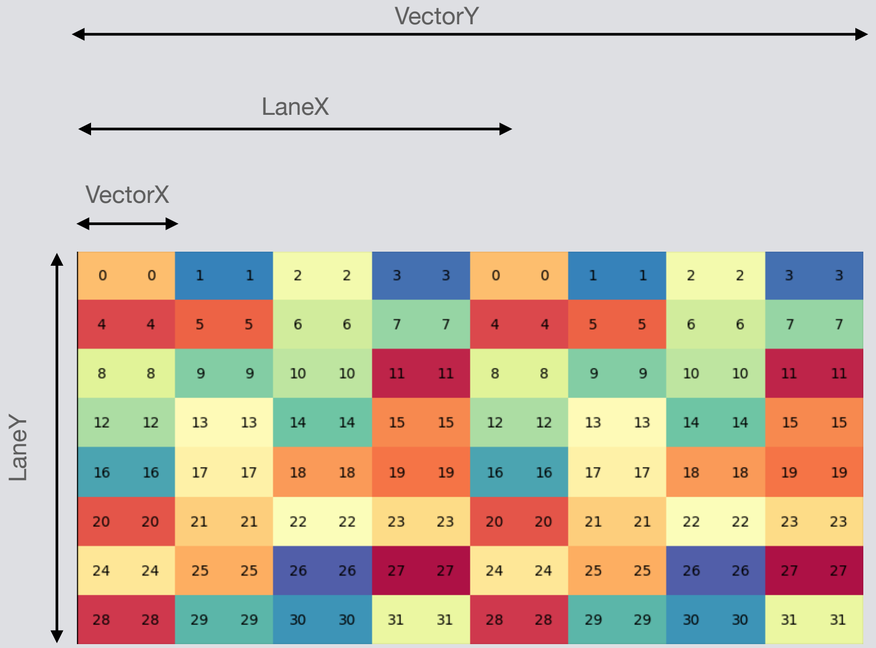


Higher-Dimensional Layout Representations

- Extends to B and C matrices as well



Batch (row)	Batch (col)	Lane Z	Lane Y (0)	Lane X (1)	Vec Z	Vec Y (1)	Vec X (0)
1	1	1	8	4	1	2	2



Batch (row)	Batch (col)	Lane Z	Lane Y (0)	Lane X (1)	Vec Z	Vec Y (2)	Vec X (0)
1	1	1	8	4	1	2	2

WMMA Instruction Layout - A Matrix

A Matrix: Matrix View

K

M

A0_0	A0_1	A0_2	A0_3	A0_4	A0_5	A0_6	A0_7	A0_8	A0_9	A0_10	A0_11	A0_12	A0_13	A0_14	A0_15
A1_0	A1_1	A1_2	A1_3	A1_4	A1_5	A1_6	A1_7	A1_8	A1_9	A1_10	A1_11	A1_12	A1_13	A1_14	A1_15
A2_0	A2_1	A2_2	A2_3	A2_4	A2_5	A2_6	A2_7	A2_8	A2_9	A2_10	A2_11	A2_12	A2_13	A2_14	A2_15
A14_0	A14_1	A14_2	A14_3	A14_4	A14_5	A14_6	A14_7	A14_8	A14_9	A14_10	A14_11	A14_12	A14_13	A14_14	A14_15
A15_0	A15_1	A15_2	A15_3	A15_4	A15_5	A15_6	A15_7	A15_8	A15_9	A15_10	A15_11	A15_12	A15_13	A15_14	A15_15

A Matrix: VGPR View for Wave32 – Column Major

	Lane 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V0 [15:0]	A0_0	A1_0	A2_0	A3_0	A4_0	A5_0	A6_0	A7_0	A8_0	A9_0	A10_0	A11_0	A12_0	A13_0	A14_0	A15_0
V0 [31:16]	A0_1	A1_1	A2_1	A3_1	A4_1	A5_1	A6_1	A7_1	A8_1	A9_1	A10_1	A11_1	A12_1	A13_1	A14_1	A15_1
V1 [15:0]	A0_2	A1_2	A2_2	A3_2	A4_2	A5_2	A6_2	A7_2	A8_2	A9_2	A10_2	A11_2	A12_2	A13_2	A14_2	A15_2
V1 [31:16]	Lanes 16-31 replicate lanes 0-15 Wave64: lanes 32-63 replicate lanes 0-31 (A matrix is transposed from normal view)															
V7 [15:0]	A0_14	A1_14	A2_14	A3_14	A4_14	A5_14	A6_14	A7_14	A8_14	A9_14	A10_14	A11_14	A12_14	A13_14	A14_14	A15_14
V7 [31:16]	A0_15	A1_15	A2_15	A3_15	A4_15	A5_15	A6_15	A7_15	A8_15	A9_15	A10_15	A11_15	A12_15	A13_15	A14_15	A15_15

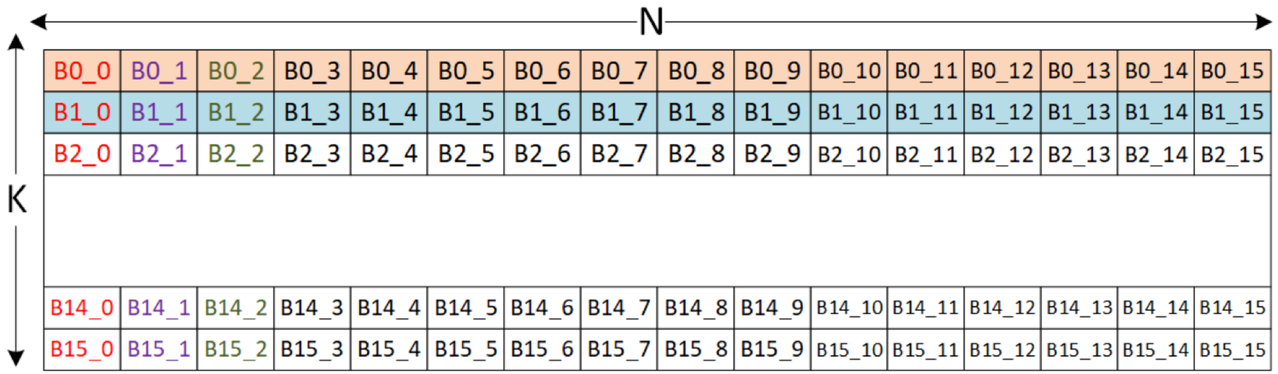
K

M

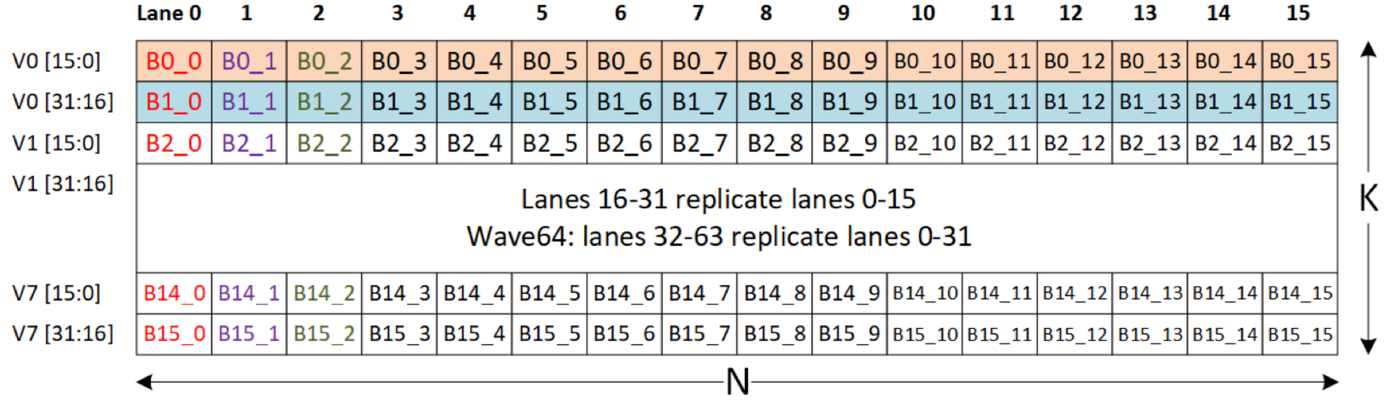
Batch (row)	Batch (col)	Lane X	Vec X
1	1	16	16

WMMA Instruction Layout - B Matrix

B Matrix: Matrix View



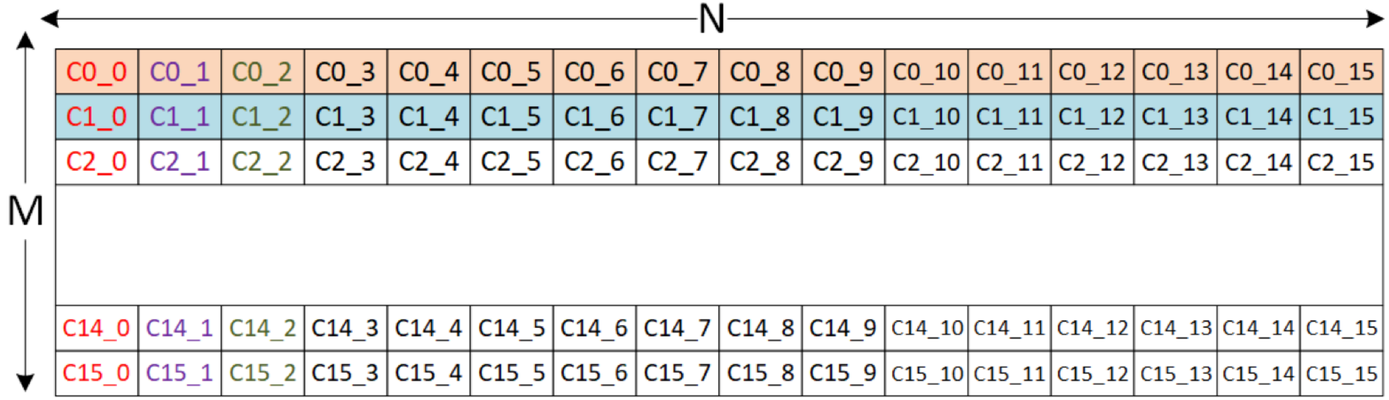
B Matrix: VGPR View for Wave32 – Row Major



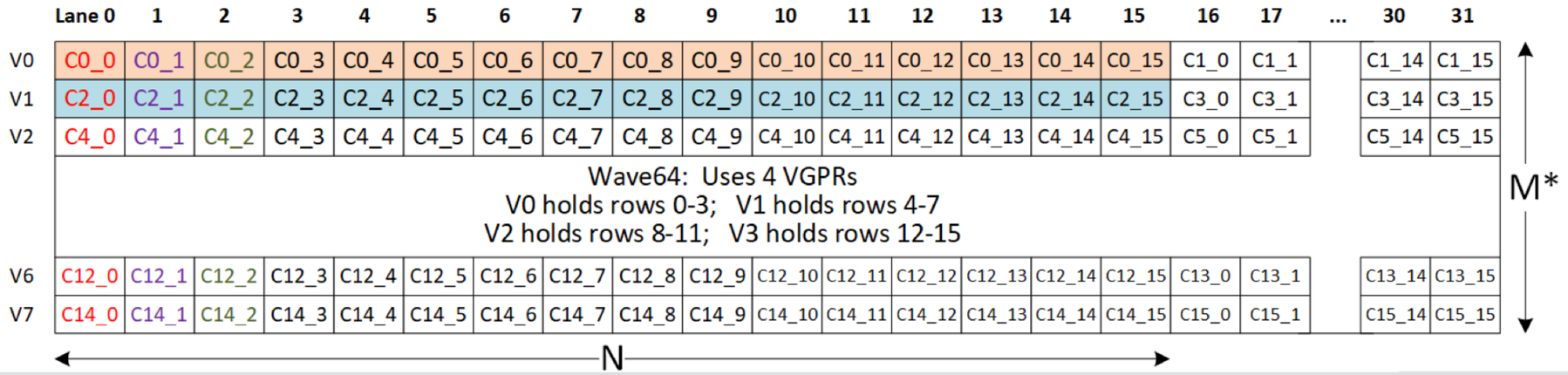
Batch (row)	Batch (col)	Lane X	Vec X
1	1	16	16

WMMA Instruction Layout - C Matrix

C & D Matrix: Matrix View



C & D Matrix: VGPR View for Wave32 – Row Major



Batch (row)	Batch (col)	Lane X	Vec X (1)	LaneY (0)
1	1	16	8	2

WMMA Instruction : Additional Details

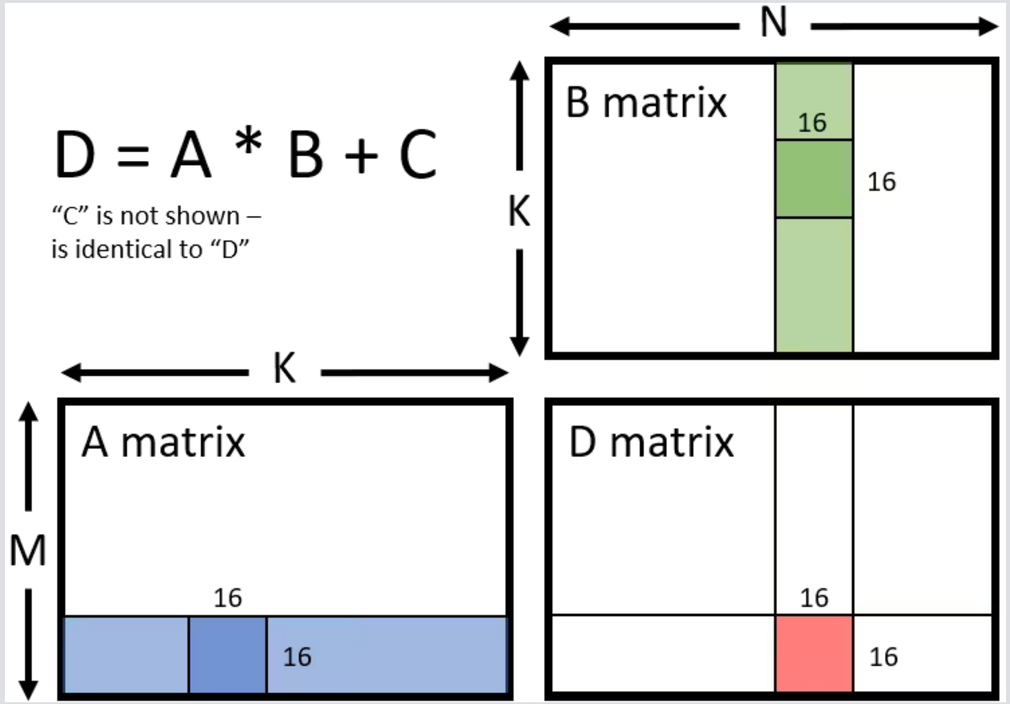
- Comes in different types

Instruction	Matrix A	Matrix B	Matrix C	Result Matrix
V_WMMA_F32_16X16X16_F16	16x16 F16	16x16 F16	16x16 F32	16x16 F32
V_WMMA_F32_16X16X16_BF16	16x16 BF16	16x16 BF16	16x16 F32	16x16 F32
V_WMMA_F16_16X16X16_F16	16x16 F16	16x16 F16	16x16 F16	16x16 F16
V_WMMA_BF16_16X16X16_BF16	16x16 BF16	16x16 BF16	16x16 BF16	16x16 BF16
V_WMMA_I32_16X16X16_IU8	16x16 IU8	16x16 IU8	16x16 I32	16x16 I32
V_WMMA_I32_16X16X16_IU4	16x16 IU4	16x16 IU4	16x16 I32	16x16 I32

- Currently, we focus on FP16 A, B, C and D matrices
- Represented in the AMDGPU Dialect as the WMMAOp

```
%170 = amdgpu.wmma %142 * %114 + %169 : vector<16xf16>, vector<16xf16>,
                                        vector<16xf16>
```

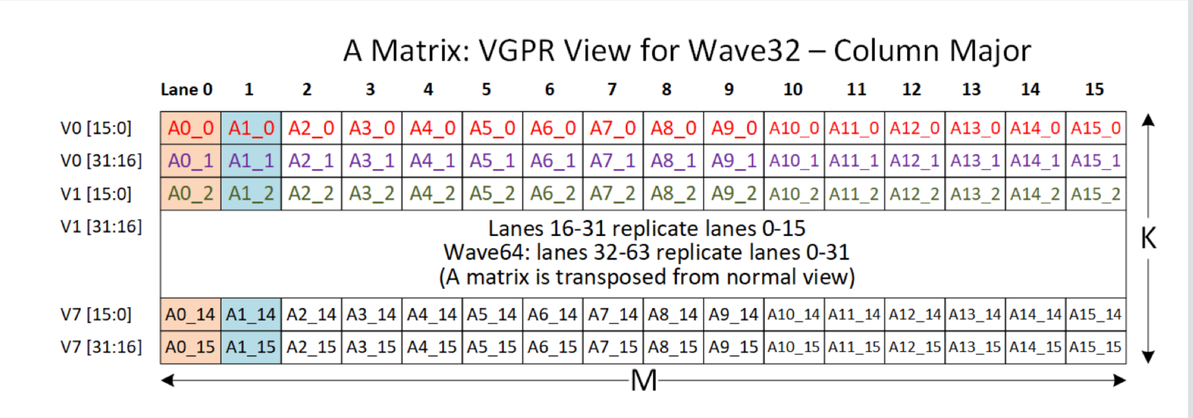
  - Even though the output is a vector with 16 elements, only 8 elements are valid
  - If subwordOffset is 0, the output is stored at 0, 2, 4, …
  - If subwordOffset is 1, the output is stored at 1, 3, 5, …

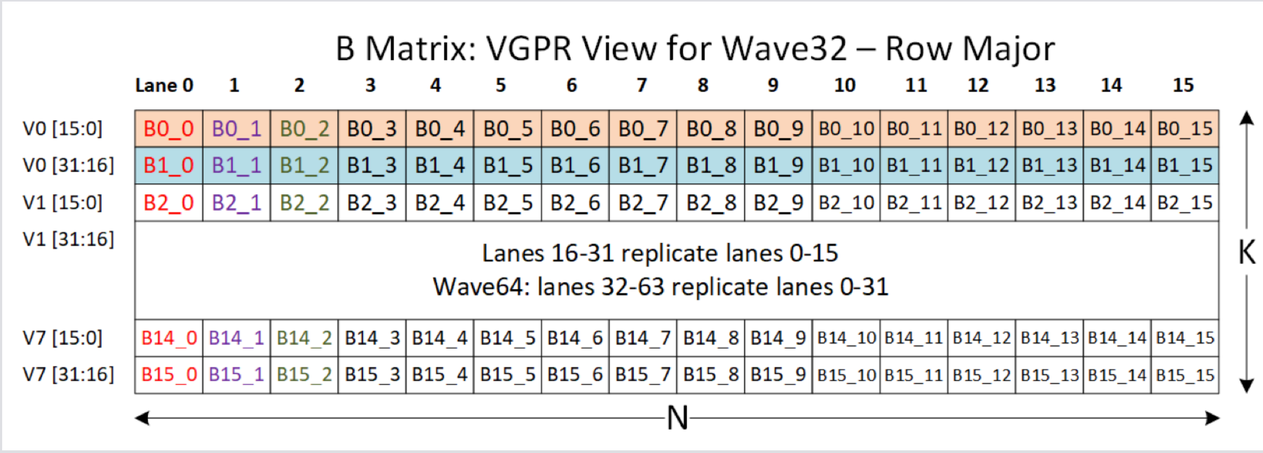


https://gpuopen.com/learn/wmma_on_rdna3/

WMMA Instruction : Contraction Form

- Transpose B
  - Allows loading A and B matrices 8 elements at a time (_loadb128 instructions)
- Transpose A
  - Can only load A and B matrices 1 element at a time
- Other variations are some combination of the above
- Can try these different variants to see which is more performant by switching the layout of row / column





Distributing reads/writes

- How can we use this to distribute reads/writes?
- We need to compute the indices of which row and column of the matrix each thread needs to load and this comes directly from the layout
- For NVIDIA, we extended it to the ldmatrix instruction
- For RDNA3, we consider two cases
  - Load/Store from global memory to registers
    - memref.load/store and vector.load/store produce flat instructions (which issue instructions simultaneous global and shared memory at the same time and hurt performance)
    - So instead we use raw_buffer_load/store instructions in the amdgpu dialect
  - Load/Store from shared memory (LDS) to registers
    - memref.load/store and vector.load/store produce the appropriate ds_load/store instructions

```
%3 = vector.transfer_read %0[%c0, %c0], %cst_0 {in_bounds
= [true, true]} : memref<16x16xf16>, vector<16x16xf16>
```



```
%427 = amdgpu.raw_buffer_load {boundsCheck = false,
indexOffset = 0 : i32} %3[%425, %426] sgprOffset %c0_i32 :
memref<128x1024xf16>, i32, i32 -> vector<8xf16>
```

```
%34 = vector.load %alloc[%29#1, %31, %c8] :
 memref<2x128x40xf16, #gpu.address_space<workgroup>>,
 vector<8xf16>
```

Distributing contractions

- For NVIDIA, we mapped contractions to mma.sync
- Here, we map to the amdgpu wmma op
- Need to emit multiple mma.sync ops for the batch dimensions and accumulate the result across the reduction dimensions
- Also need to extract the valid part of the result before storing to global memory

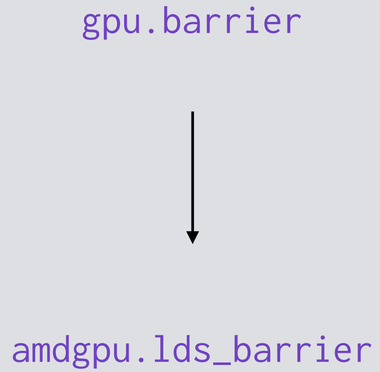
```
%14 = vector.contract {indexing_maps = [#map7, #map8,
#map9], iterator_types = ["parallel", "parallel",
"reduction"], kind = #vector.kind<add>} %12, %13, %arg1 :
vector<32x32xf16>, vector<32x32xf16> into
vector<32x32xf16>
```



```
%117 = amdgpu.wmma %87 * %116 + %115 : vector<16xf16>,
   vector<16xf16>, vector<16xf16>
%118 = vector.extract %117[0] : vector<16xf16>
%119 = vector.extract %117[2] : vector<16xf16>
%120 = vector.extract %117[4] : vector<16xf16>
%121 = vector.extract %117[6] : vector<16xf16>
%122 = vector.extract %117[8] : vector<16xf16>
%123 = vector.extract %117[10] : vector<16xf16>
%124 = vector.extract %117[12] : vector<16xf16>
%125 = vector.extract %117[14] : vector<16xf16>
```

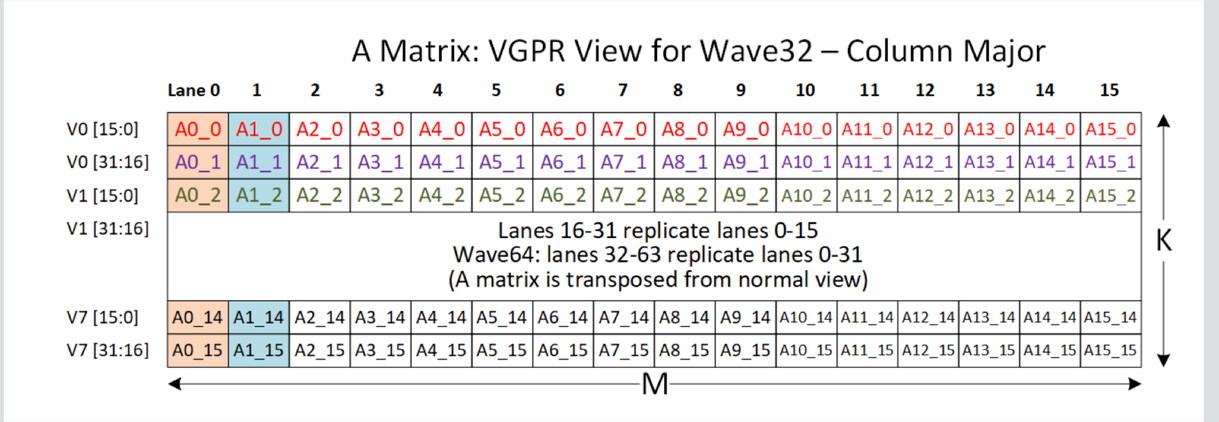
Barrier Ops

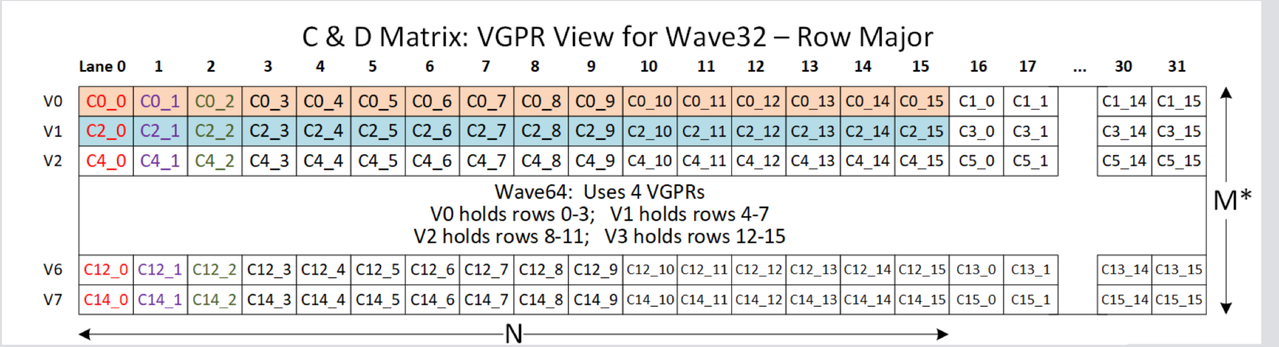
- Currently, we use gpu BarrierOp to insert barriers for loads/stores from/to LDS
- When lowering this op using the AMDGPU backend, we end up generating buffer_gl0_inv ops that invalidate the shader L0 cache associated with the warp
- Instead, we use amdgpu LdsBarrierOp which eliminates those ops and produces just the s_barrier ops that we need



Flash Attention Considerations

- In the implementation of Flash Attention using the high dimensional layout, we often ran into “layout conflicts”
- Situations where the C matrix from a previous matrix multiplication was used as the A matrix of a subsequent matrix multiplication
- In the mma.sync case, we were able to resolve these conflicts within each thread (by reordering the elements of each vector)
- For RDNA3 and WMMA ops, due to the significant layout differences between the C and A matrix, the trip to LDS / some sort of shuffle between threads will be unavoidable





Conclusions & Future Work

- Extended high dimension layout to AMD RDNA architectures
- Evaluating performance of current approach and adding additional performance optimizations
  - Added padding to avoid shared memory bank conflicts
  - Adding shared memory swizzle
- Move from implicit to explicit IR representation
- Apply same approach to mfma instructions for CDNA GPUs

Acknowledgements

- Thanks to Aaryaman Vasishta & Lucas Neves for their support on the details of the WMMA instruction and suggestions on how to improve performance

- Thanks to Thomas Raoux for help and support

- Thanks to nod.ai and IREE team for support and guidance

References

- RDNA3 Instruction Set Architecture Reference Guide (Aug 2023)
- Vasishta, A. (2023) How to accelerate AI applications on RDNA3 using WMMA
- Menon. H. (2023) Decomposable operators in IREE: Winograd Convolutions and Flash Attention
- Menon, H. (2022) MLIR Code Generation Tutorial