High-dimensional layouts for vector distribution

Harsh Menon (nod.ai)

Overview

- Motivation
- High-dimensional Layout Representation
- Vector Distribution Building Blocks
  - Reads/Writes
  - Contractions
  - Reductions
  - For loops
  - 1-D operators
- Flash Attention
  - Algorithm
  - Applying the layout representation to flash attention
- Results
- Conclusions & Future Work
- Acknowledgements

Motivation

- Deep learning has seen explosive growth in recent years with more mainstream applications such as NLP-based web search assistants (Chat-GPT), generative art (StableDiffusion), drug discovery and more
- Highly accurate models require enormous amounts of curated data as well as large models with trillions of parameters
- Matrix multiplications are the core component of computation in these models
- Most hardware vendors have specialized hardware for matrix multiplication
- One of the fundamental tasks for ML compilers is making efficient use of these specialized units

Images from nod.ai SHARK Discord channel

Motivation

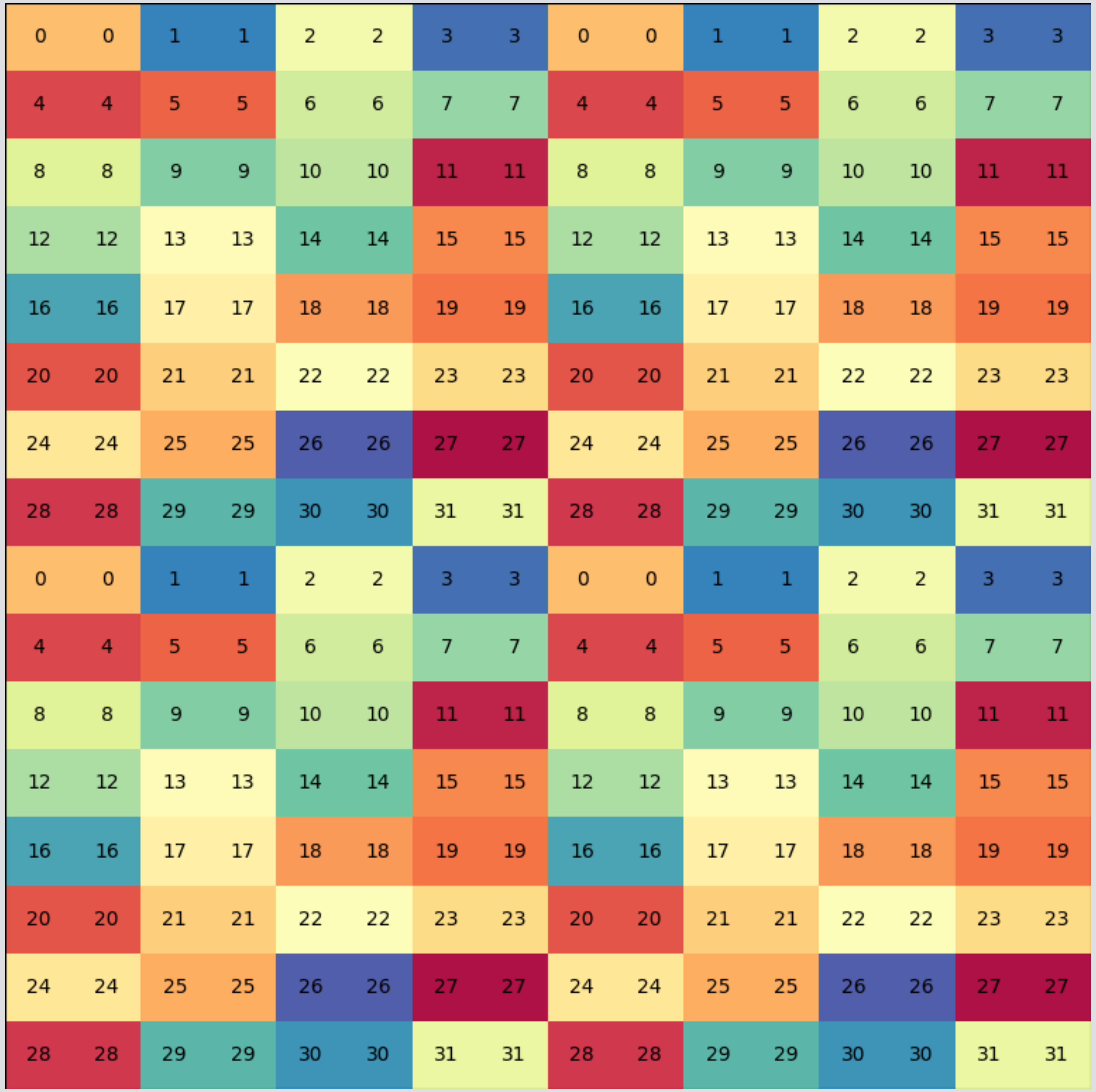
- Specialized matrix-multiplication hardware such as tensor cores usually require the input/output data to be in a specific layout

- For NVIDIA GPUs, this layout specifies the mapping of threads to elements of the matrices (as shown on the right)

- Since the majority of compute centers around matrix multiplication, we would like to reuse the data that is already present in registers for subsequent operations

- Implies that we need to be aware of the data layout and reuse that layout as much as possible (as this would keep all the data in registers and avoid any additional data movement)

- How do we represent this layout?



Layout Representations

- Desirable properties of a layout representation
    - **Expressive**: Capable of capturing the layout for a variety of tensor types, operators etc.
    - **Self-contained**: Vector distribution should be derivable completely from the layout
    - **Generalizable**: Extends to more than one architecture

Layout Representations

- Returning to mma.sync.m16n8k16 example
  - How can we represent this layout?
  - One option is using hardcoded functions provided by the PTX ISA, shown below

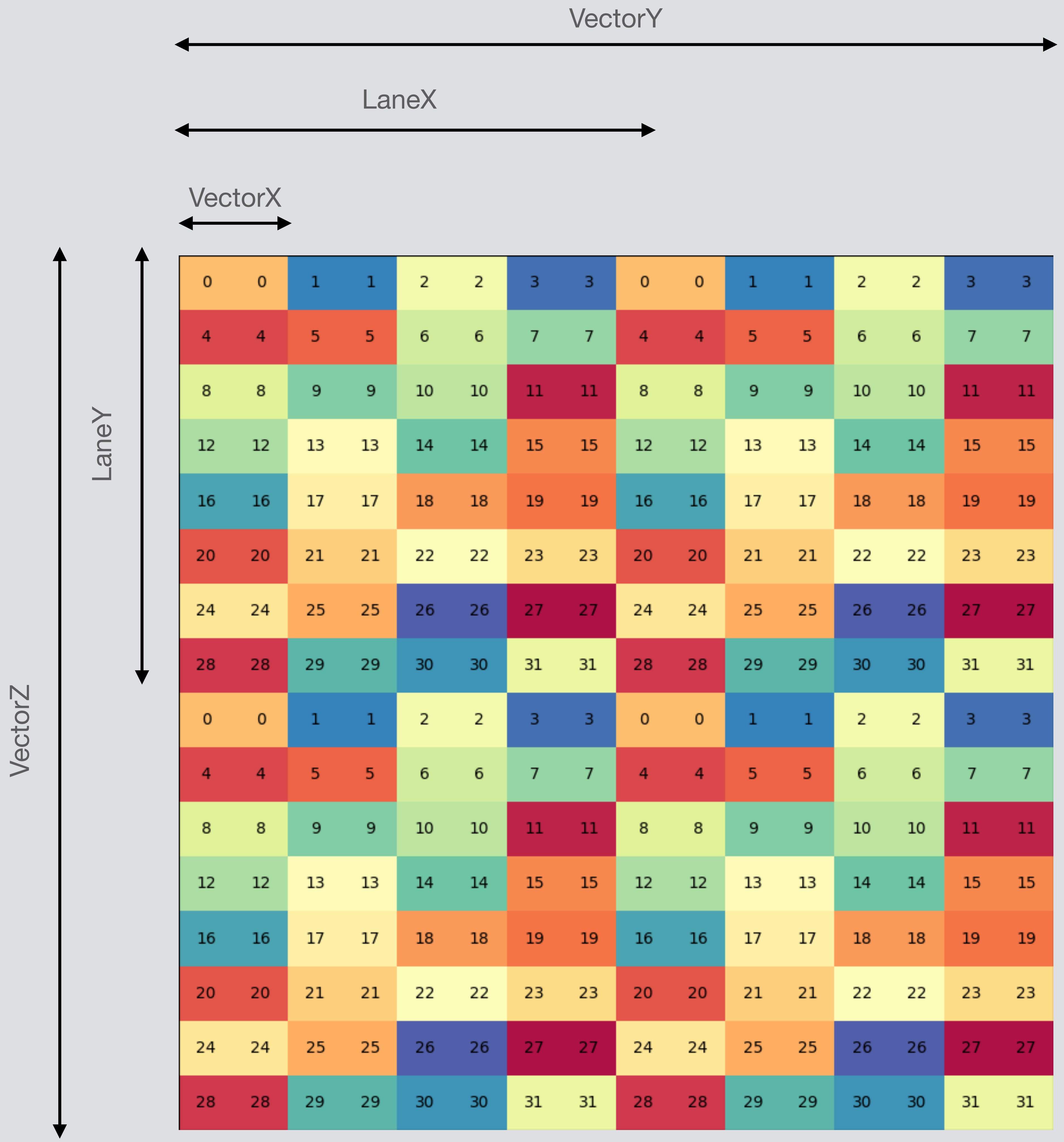
```
group_id, thread_id_in_group = get_ids(lane_id)
if (i >= 0 and i < 2) or (i >= 4 and i < 6):
    row = group_id
else:
    row = group_id + 8

if i < 4:
    col = thread_id_in_group * 2 + (i & 0x1)
else:
    col = thread_id_in_group * 2 + (i & 0x1) + 8
```

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

Higher-Dimensional Layout Representations

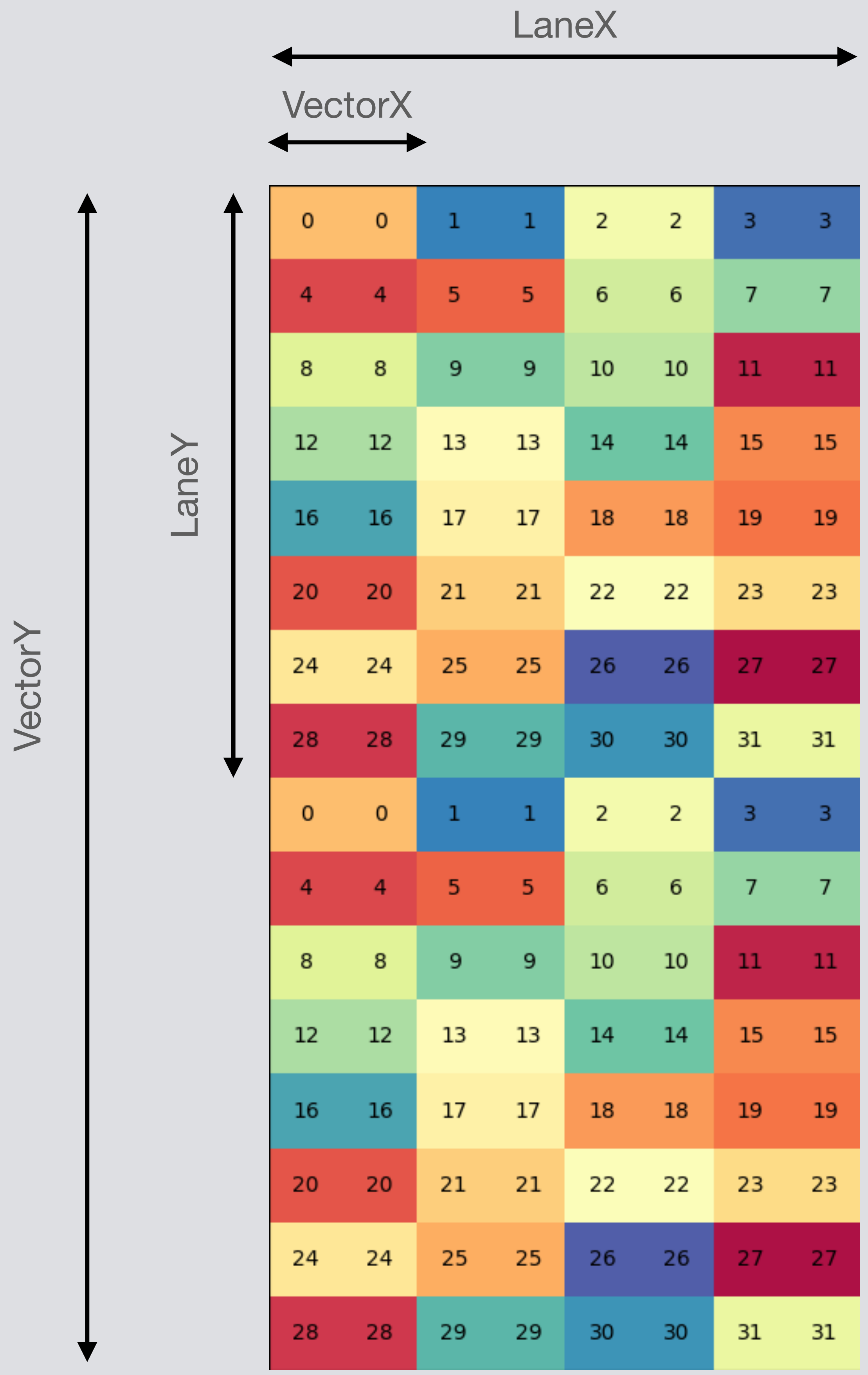
- Could we use a higher dimensional layout representation instead?
- What might this high dimensional representation contain?
    - Vector dimensions (X, Y, Z)
    - Lane dimensions (X, Y, Z)
    - Batch dimensions (for row and column)
    - **Batch (row)** x **Batch (column)** x **LaneZ** x **LaneY** x **LaneX** x **VectorZ** x **VectorY** x **VectorX**
    - **Row**: VectorX = 2, LaneX = 4, VectorY = 2
    - **Column**: 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

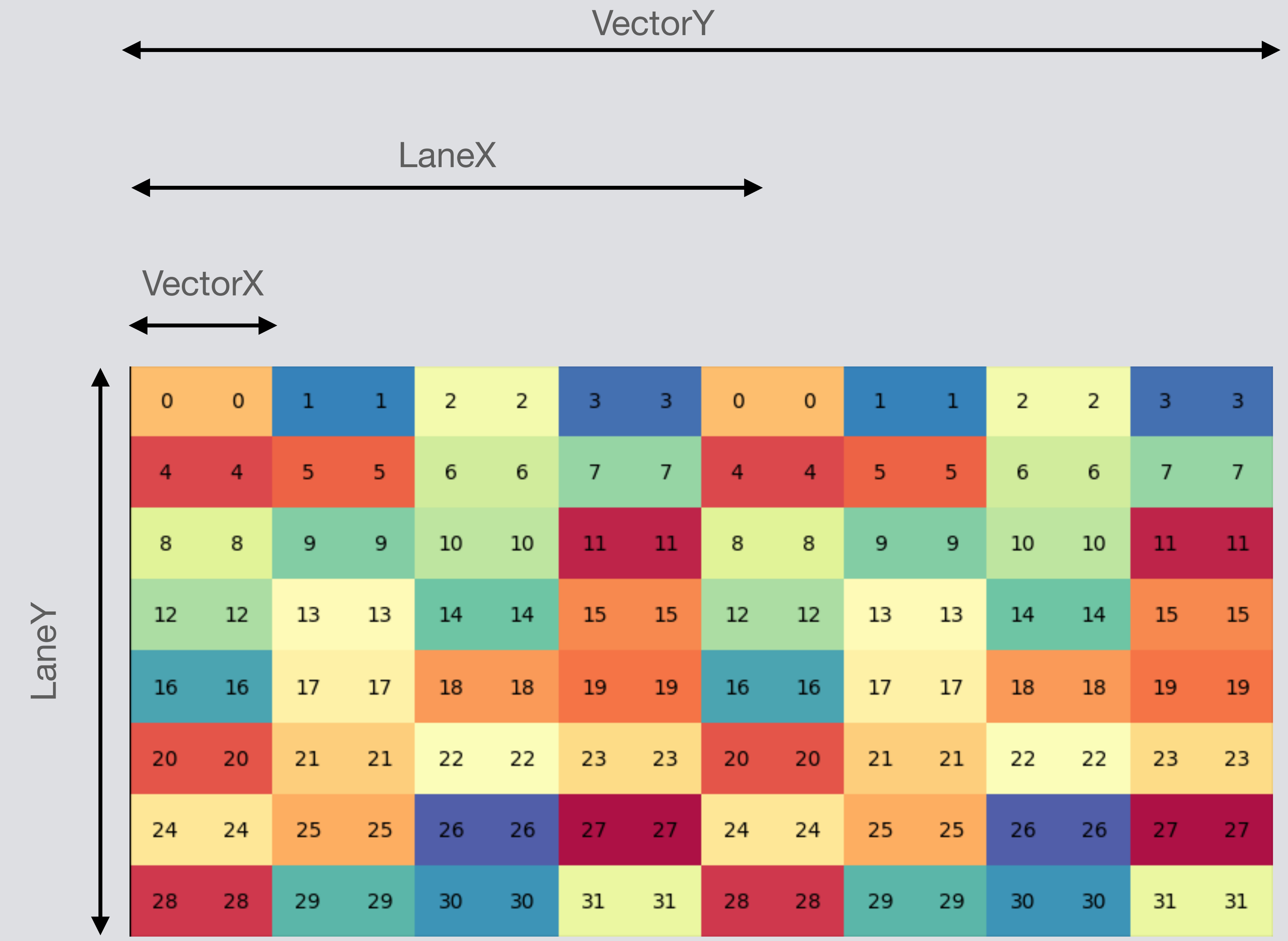


Higher-Dimensional Layout Representations

- 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

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
- We can compute that from our layout representation as shown below (for batch = 1)

$$\mathrm{row} = l_y + 8v_z$$

$$\mathrm{col} = v_x + 2l_x + 8v_y$$

- Can be extended to load more than one element at a time (ldmatrix.4)

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

↓

```
#map = affine_map<(d0, d1, d2) -> (d1 + d2 * 16)>
#map1 = affine_map<(d0, d1, d2) -> (d0 * 2)>
%3 = gpu.thread_id  x
%4 = gpu.thread_id  y
%5 = affine.apply #map(%3, %4, %c0)
%6 = affine.apply #map1(%3, %4, %c0)
%9 = memref.load %0[%5, %6] : memref<16x16xf16>
%10 = vector.broadcast %9 : f16 to vector<1xf16>
%11 = vector.insert_strided_slice %10, %cst_1 {offsets =
[0, 0, 0, 0], strides = [1]} : vector<1xf16> into
vector<1x1x4x2xf16>
```

Distributing contractions

- Contractions can be mapped directly to the mma.sync operation
- We standardize on one form of the vector contraction $D = C + AB^T$
- Need to emit multiple mma.sync ops for the batch dimensions and accumulate the result across the reduction dimensions
- Can also handle mixed mode mma.sync where input operands are in FP16 but accumulators are in FP32

```
%5 = vector.contract {indexing_maps = [affine_map<(d0, d1,
d2) -> (d0, d2)>, affine_map<(d0, d1, d2) -> (d1, d2)>,
affine_map<(d0, d1, d2) -> (d0, d1)>], iterator_types =
["parallel", "parallel", "reduction"], kind =
#vector.kind<add>} %3, %4, %cst : vector<16x16xf16>,
vector<8x16xf16> into vector<16x8xf16>
```

↓

```
%55 = vector.extract %cst[0, 0] : vector<1x1x2x2xf16>
%56 = vector.extract %40[0, 0] : vector<1x1x4x2xf16>
%57 = vector.extract %54[0, 0] : vector<1x1x2x2xf16>
%58 = nvgpu.mma.sync(%56, %57, %55) {mmaShape = [16, 8, 16]} :
(vector<4x2xf16>, vector<2x2xf16>, vector<2x2xf16>)
-> vector<2x2xf16>
%59 = vector.insert %58, %cst [0, 0] :
vector<2x2xf16> into vector<1x1x2x2xf16>
```

Distributing reductions

- Reductions followed by broadcast (and transpose) to original shape allows us to only consider the final result and assign to it the same layout as the source

- Reduction along the columns

Batch	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

- Based on the layout, we can deduce that 4 lanes are involved in the reduction

```
%5 = vector.contract {indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d2)>, affine_map<(d0, d1, d2) -> (d1, d2)>, affine_map<(d0, d1, d2) -> (d0, d1)>], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %3, %4, %cst : vector<16x16xf16>, vector<8x16xf16> into vector<16x8xf16>
%6 = vector.multi_reduction <maxf>, %5, %init [1] : vector<16x8xf16> to vector<16xf16>
%7 = vector.broadcast %6 : vector<16xf16> to vector<8x16xf16>
%8 = vector.transpose %7, [1, 0] : vector<8x16xf16> to vector<16x8xf16>
```

↓

```
%61 = vector.bitcast %60 : vector<2xf16> to vector<1xi32>
%62 = vector.extract %61[0] : vector<1xi32>
%c1_i32 = arith.constant 1 : i32
%c32_i32 = arith.constant 32 : i32
%shuffleResult, %valid = gpu.shuffle  xor %62,
    %c1_i32, %c32_i32 : i32
%63 = vector.broadcast %shuffleResult : i32 to vector<1xi32>
%64 = vector.bitcast %63 : vector<1xi32> to vector<2xf16>
%65 = arith.maxf %64, %60 : vector<2xf16>
```

Distributing for loops

- Can support for loops by replacing the loop carried variables and yield variables with their corresponding SIMT values
- Also need to propagate the layout through the for loops to any reads/writes outside the for loop

```
%7 = scf.for %arg0 = %c0 to %c4 step %c1 iter_args(%arg1 = %6)
     -> (vector<16x8xf16>) {
      %11 = vector.transfer_read %0[%c0, %10], %cst
{in_bounds = [true, true]} : memref<16x64xf16>, vector<16x16xf16>
      %13 = vector.transfer_read %1[%c0, %10], %cst
{in_bounds = [true, true]} : memref<8x64xf16>, vector<8x16xf16>
      %14 = vector.contract
{indexing_maps = [#map4, #map5, #map6],
iterator_types = ["parallel", "parallel", "reduction"],
kind = #vector.kind<add>} %11, %13, %arg1 :
vector<16x16xf16>, vector<8x16xf16> into vector<16x8xf16>
      scf.yield %14 : vector<16x8xf16>
```

↓

```
%28 = scf.for %arg0 = %c0 to %c4 step %c1 iter_args(%arg1 = %27)
      -> (vector<1x1x2x2xf16>) {
   ...
   scf.yield  %85 : vector<1x1x2x2xf16>
}
```

Distributing operators with 1D vectors

- So far we avoided dealing with 1D vectors by considering reductions as reductions followed by broadcasts
- We can deal with operations on 1D vectors using the layout but this would introduce extra complexity as only some lanes would be participating in the operation
- An alternative is to tread 1D vectors as pseudo-2D vectors (broadcasted along the reduction dimension) if the 1D vectors are used as intermediate values in the computation
- Allows reusing the layout of the mma operations

```
%12 = vector.multi_reduction <maxf>, %10, %11 [1] :
      vector<16x16xf16> to vector<16xf16>
%13 = vector.transfer_read %3[%7], %cst_1 {in_bounds = [true]}
      : memref<16xf16>, vector<16xf16>
%14 = arith.subf %11, %12 : vector<16xf16>
%15 = math.exp %14 : vector<16xf16>
%16 = arith.mulf %15, %13 : vector<16xf16>
```

↓

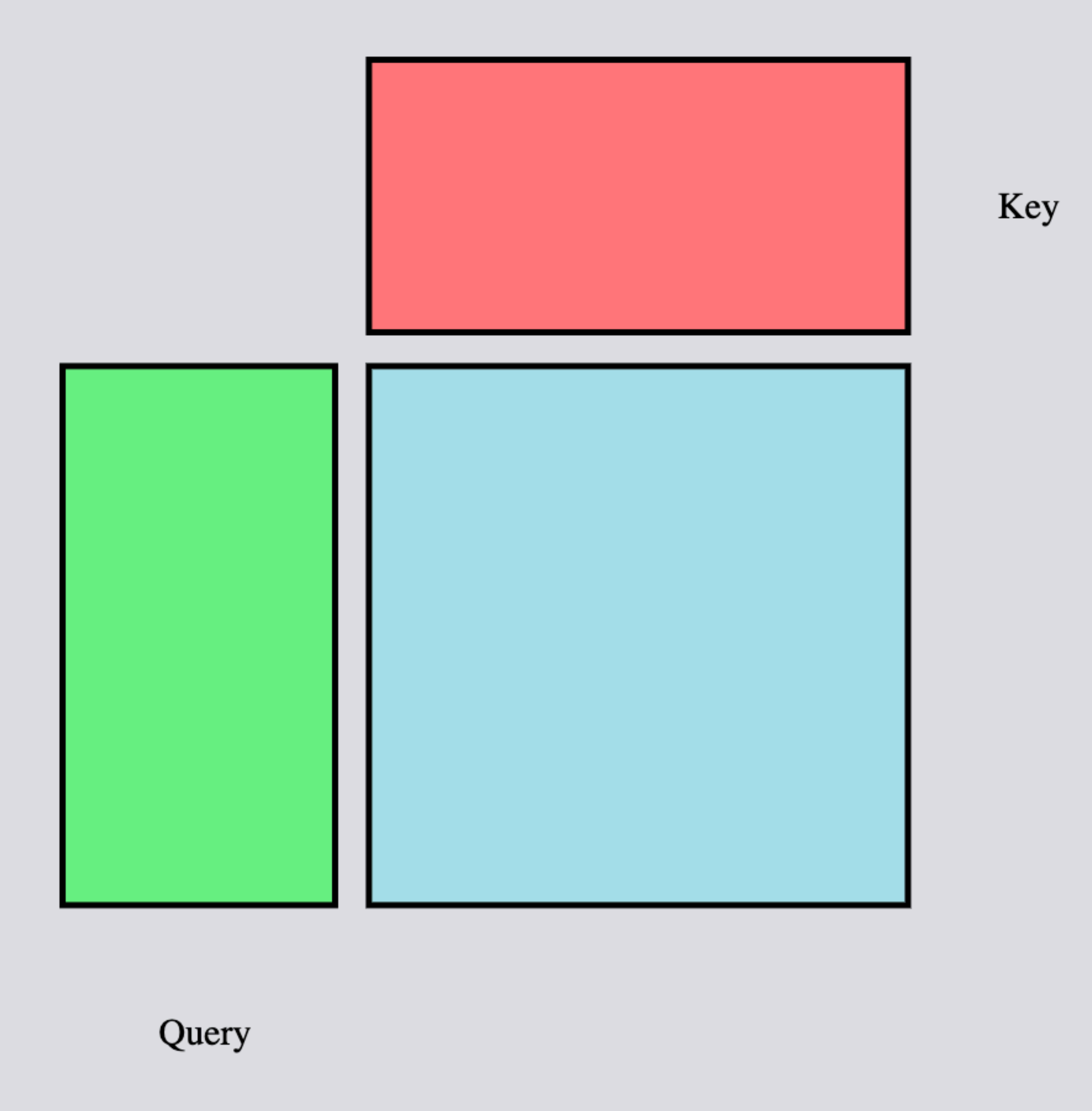
```
%shuffleResult_14, %valid_15 = gpu.shuffle …
%154 = vector.broadcast %shuffleResult_14 : i32 to vector<1xi32>
%155 = vector.bitcast %154 : vector<1xi32> to vector<2xf16>
%156 = arith.maxf %155, %151 : vector<2xf16>
...
%175 = vector.insert_strided_slice %170, %174
       {offsets = [0, 1, 1, 0], strides = [1]} :
       vector<1xf16> into vector<1x2x2x2xf16>
%176 = vector.insert_strided_slice %170, %175
       {offsets = [0, 1, 1, 1], strides = [1]} :
       vector<1xf16> into vector<1x2x2x2xf16>
...
%177 = arith.subf %94, %164 : vector<1x2x2x2xf16>
%178 = math.exp %177 : vector<1x2x2x2xf16>
%179 = arith.mulf %178, %176 : vector<1x2x2x2xf16
```

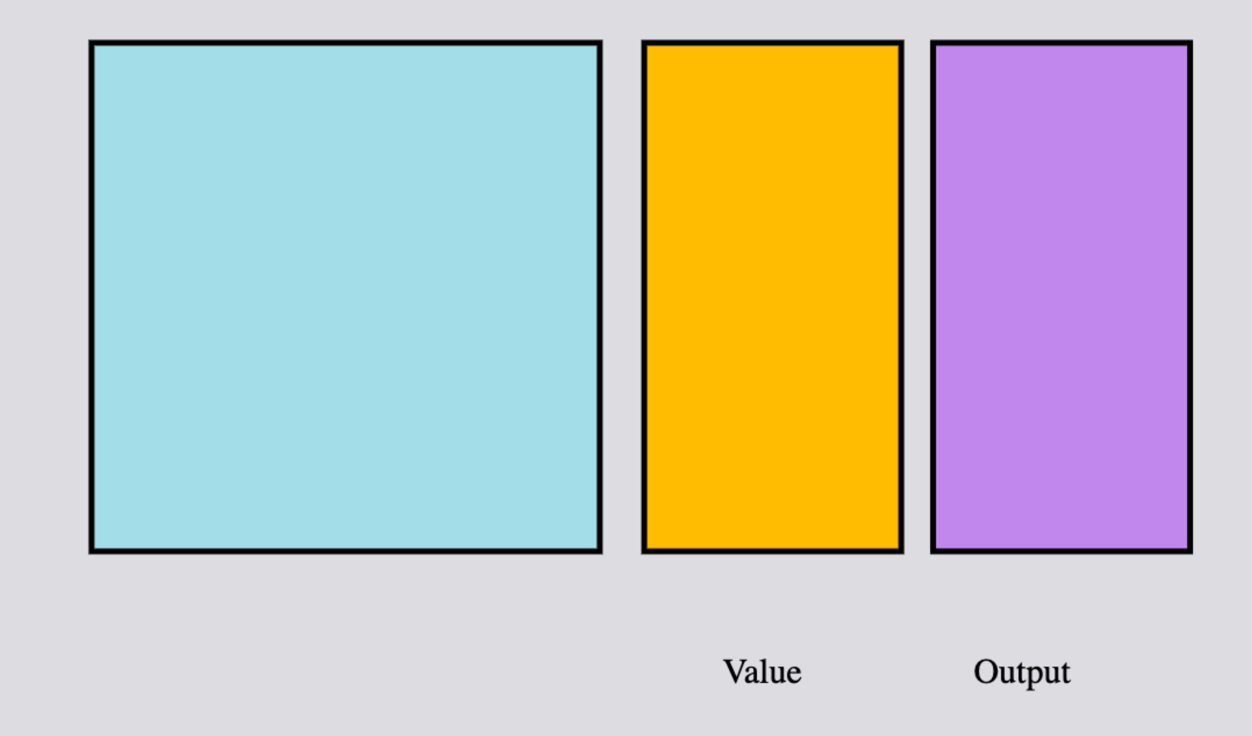
Flash Attention

Naive Attention

Overview

- **Inputs:** Query Matrix (Q), Key Matrix (K) and Value Matrix (V)
- Each of the inputs have shape (B x N x d) where B is the batch dimension, N is the sequence length and d is the head dimension
- Typically, sequence length is much larger than head dimension
- Usually, the operator includes additional steps such as masking (causal attention), dropout, scaling etc., but we ignore these for now
- Downsides to this approach are the need to materialize a potentially large N x N matrix





$$S = QK^T$$

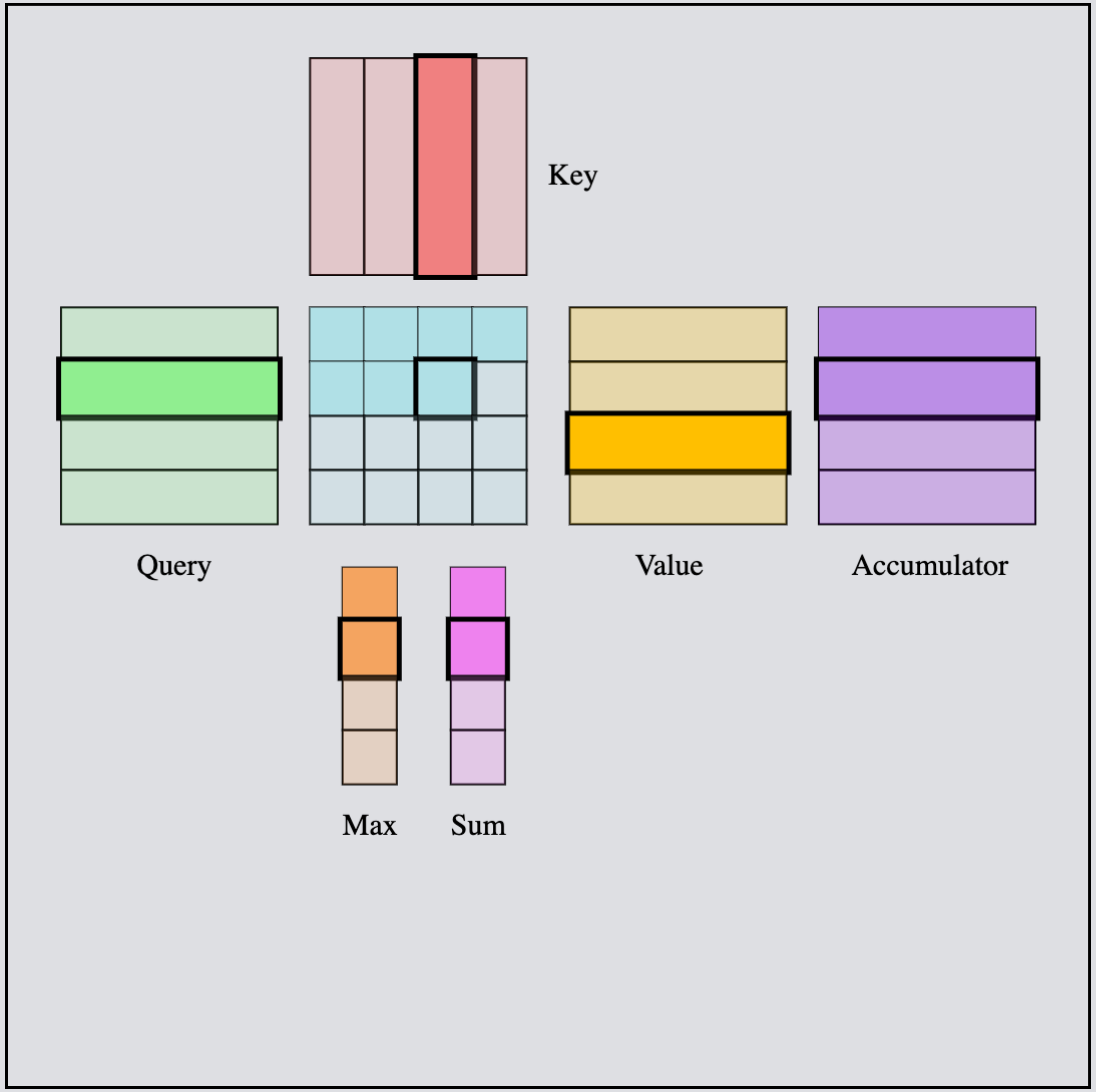
$$P = \mathrm{softmax}(S)$$

$$O = PV$$

Flash Attention Algorithm

Overview

- Three core tenets to this approach:
  - Fusion: Combine all 3 operators into a single dispatch regions
  - Tiling: Tile the operators so that, we perform the matmul, softmax and matmul only on one tile at a time
  - Aggregation: Apply fixups to the softmax and output after processing each tile

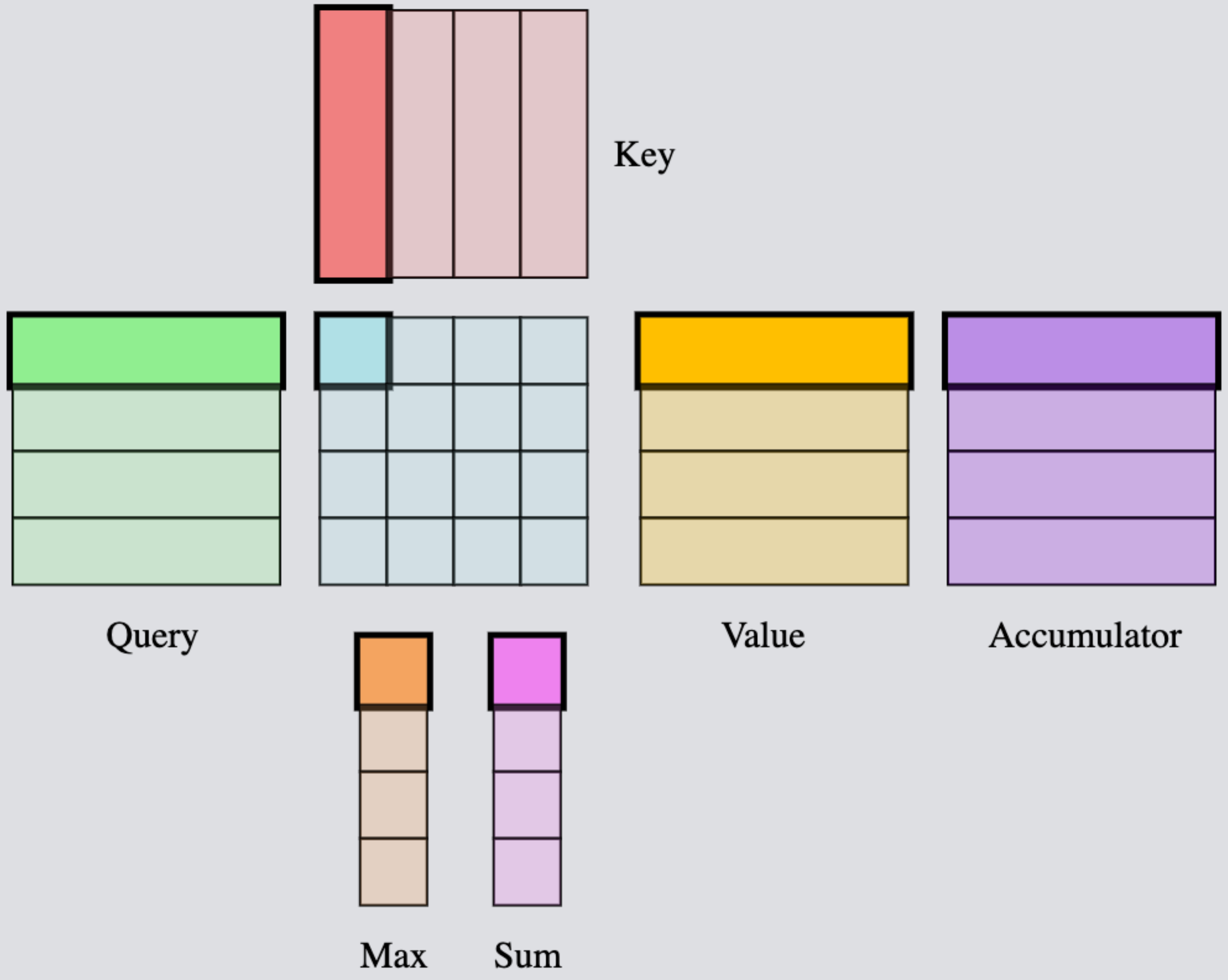


Flash Attention Algorithm

Softmax Algebraic Aggregation

- Compute softmax one block at a time
- Maintain a running max (m_i) and sum (l_i)
- For each block S_{ij} , we compute

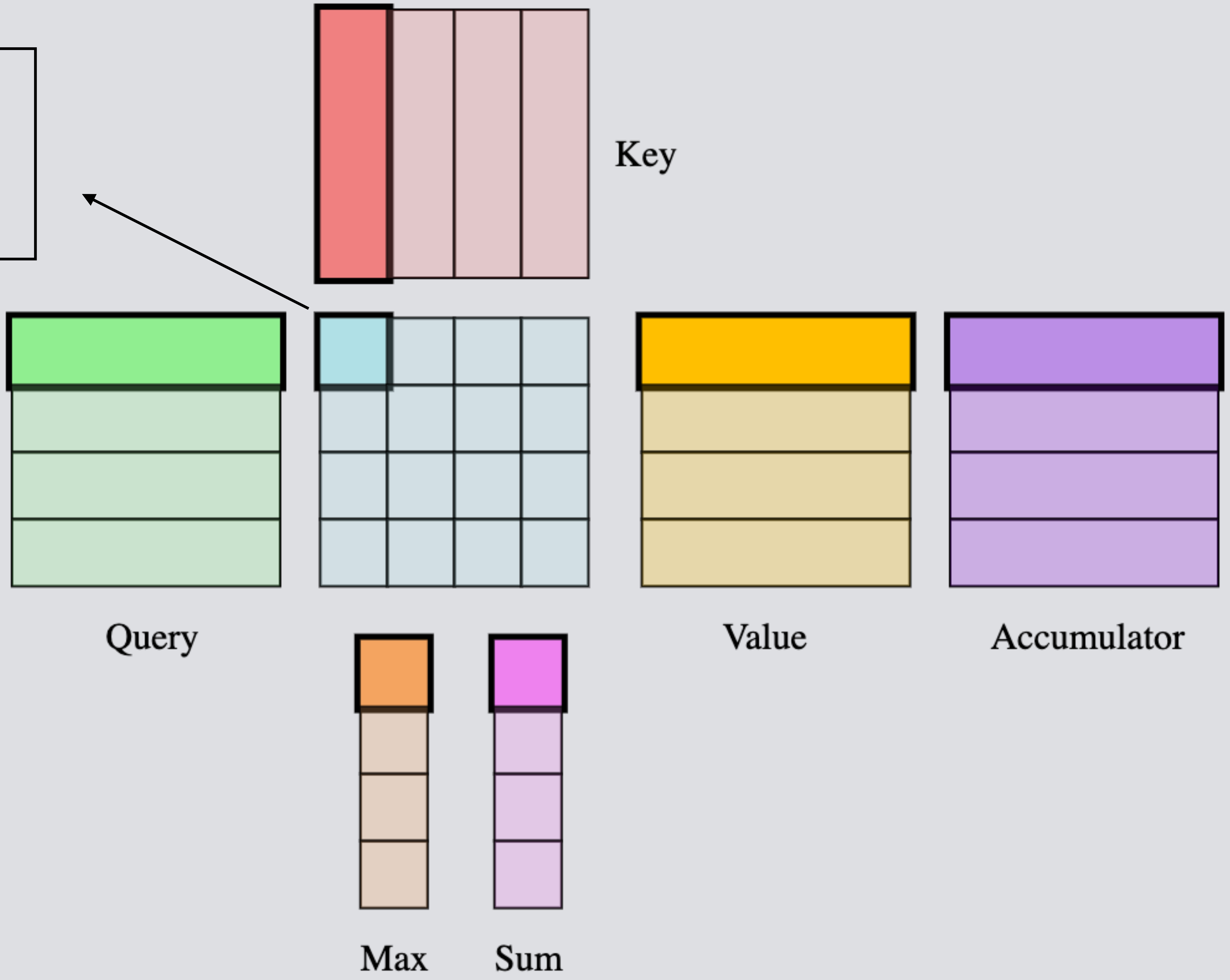
$$
\begin{aligned}
\tilde{m}_{ij} &= \text{rowmax}(S_{ij}) \\
m_i^{new} &= \max(\tilde{m}_{ij}, m_i) \\
\tilde{P}_{ij} &= \exp(S_{ij} - m_i^{new}) \\
\tilde{l}_{ij} &= \text{rowsum}(\tilde{P}_{ij}) \\
l_i^{new} &= \tilde{l}_{ij} + \exp(m_i - m_i^{new}) l_i \\
P_{ij} &= \tilde{P}_{ij} / l_i^{new} \\
A_{ij} &= \tilde{A}_{ij} \exp(m_i - m_i^{new}) l_i / l_i^{new}
\end{aligned}
$$



Flash Attention Algorithm

Softmax Algebraic Aggregation

$$\frac{\exp(x_0 - m_0^{new})}{l_0^{new}}$$

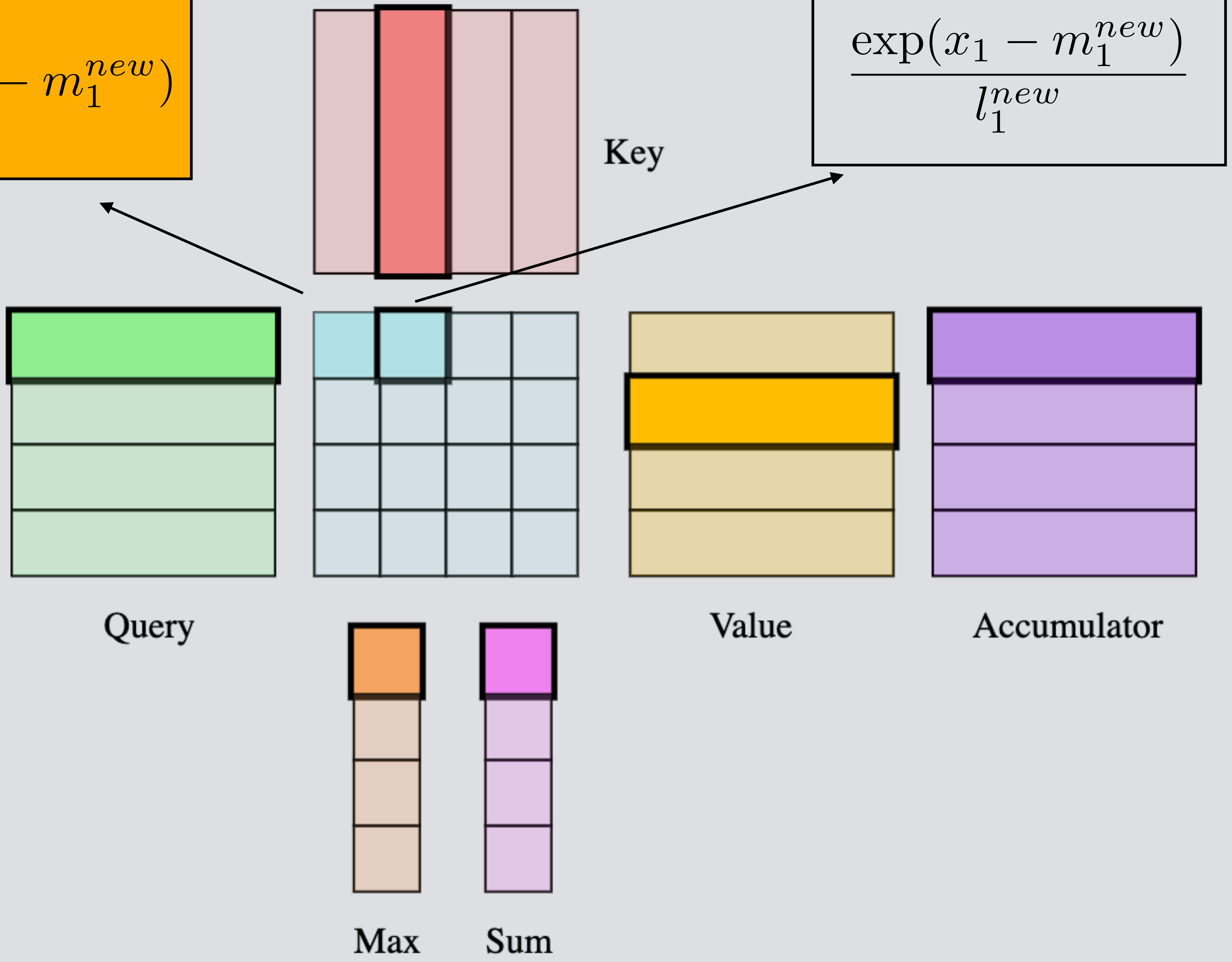


Flash Attention Algorithm

Softmax Algebraic Aggregation

$$\boxed{\frac{\exp(x_0 - m_0^{new})}{l_0^{new}} \times} \boxed{\left(\frac{l_0^{new}}{l_1^{new}}\right) \exp(m_0^{new} - m_1^{new})}$$

$$\boxed{\frac{\exp(x_1 - m_1^{new})}{l_1^{new}}}$$



Flash Attention in MLIR

- Start with a LinalgExt representation of the operator where we only expose the first two dimensions to tiling
- Decomposes into a two matrix multiplications and a sequence of linalg generics that implement the softmax operator
- Expose tiling and decomposition as a transform dialect operator so that it can compose with the rest of the utilities in the transform dialect

```
func.func @attention(%query: tensor<20x1024x64xf16>, %key:
tensor<20x1024x64xf16>, %value: tensor<20x1024x64xf16>) ->
tensor<20x1024x64xf16> {
  %0 = tensor.empty() : tensor<20x1024x64xf16>
  %1 = iree_linalg_ext.attention ins(%query, %key,
%value : tensor<20x1024x64xf16>, tensor<20x1024x64xf16>,
tensor<20x1024x64xf16>) outs(%0 : tensor<20x1024x64xf16>)
-> tensor<20x1024x64xf16>
  return %1 : tensor<20x1024x64xf16>
}
```

Tiled & Decomposed Attention

```
%17 = linalg.matmul_transpose_b ins(… tensor<32x64xf16>, tensor<32x64xf16>) outs(…
tensor<32x32xf32>)
```

```
%18 = linalg.generic ins(%17) outs(%extracted_slice_8) {
        ^bb0(%in: f32, %out: f32):
           %25 = arith.maxf %in, %out : f32
           linalg.yield %25 : f32} -> tensor<32xf32>
```

$$\tilde{m}_{ij} = \text{rowmax}(S_{ij})$$
$$m_i^{new} = \max(\tilde{m}_{ij}, m_i)$$

```
%19 = linalg.generic {ins(%18 : tensor<32xf32>) outs(%17) {
        ^bb0(%in: f32, %out: f32):
          %25 = arith.subf %out, %in : f32
          %26 = math.exp2 %25 : f32
          linalg.yield %26 : f32} -> tensor<32x32xf32>
```

$$\tilde{P}_{ij} = \exp(S_{ij} - m_i^{new})$$

```
%20 = linalg.generic ins(%extracted_slice_8, %18) outs(%extracted_slice_9)
        ^bb0(%in: f32, %in_12: f32, %out: f32):
          %25 = arith.subf %in, %in_12 : f32
          %26 = math.exp2 %25 : f32
          %27 = arith.mulf %26, %out : f32
          linalg.yield %27 : f32} -> tensor<32xf32>
```

$$\exp(m_i - m_i^{new})l_i$$

```
%21 = linalg.generic ins(%19 : tensor<32x32xf32>) outs(%20 : tensor<32xf32>) {
        ^bb0(%in: f32, %out: f32):
           %25 = arith.addf %in, %out : f32
           linalg.yield %25 : f32} -> tensor<32xf32>
```

$$\tilde{l}_{ij} = \text{rowsum}(\tilde{P}_{ij})$$
$$l_i^{new} = \tilde{l}_{ij} + \exp(m_i - m_i^{new})l_i$$

Tiled & Decomposed Attention

```
%22 = linalg.generic outs(%21 : tensor<32xf32>) {
        ^bb0(%out: f32):
            %cst_24 = arith.constant 1.000000e+00 : f32
            %41 = arith.divf %cst_24, %out : f32
            linalg.yield %41 : f32} -> tensor<32xf32>
```

```
%23 = linalg.generic ins(%22 : tensor<32xf32>) outs(%19 : tensor<32x32xf32>) {
        ^bb0(%in: f32, %out: f32):
           %25 = arith.mulf %out, %in : f32
           linalg.yield %25 : f32} -> tensor<32x32xf32>
```

$$P_{ij} = \tilde{P}_{ij}/l_i^{new}$$

```
%24 = linalg.generic ins(%23 : tensor<32x32xf32>) outs(%36 : tensor<32x32xf16>) {
        ^bb0(%in: f32, %out: f16):
           %41 = arith.truncf %in : f32 to f16
           linalg.yield %41 : f16} -> tensor<32x32xf16>
```

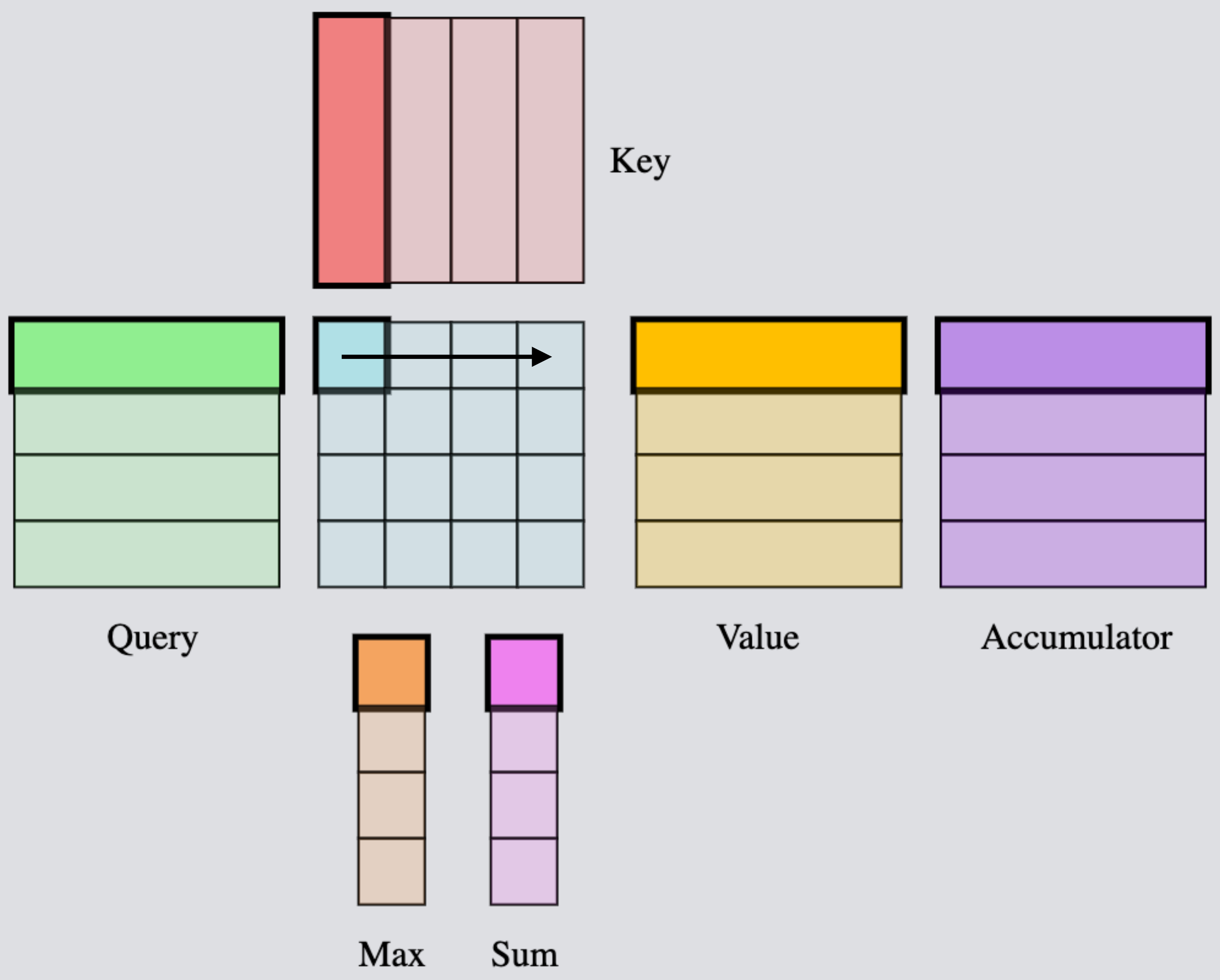
```
%25 = linalg.generic ins(%extracted_slice_7, %20, %23 : tensor<32x64xf32>, tensor<32xf32>,
tensor<32xf32>) outs(%14 : tensor<32x64xf32>) {
        ^bb0(%in: f32, %in_12: f32, %in_13: f32, %out: f32):
           %25 = arith.mulf %in_12, %in_13 : f32
           %26 = arith.mulf %25, %in : f32
           linalg.yield %26 : f32} -> tensor<32x64xf32>
```

$$A_{ij} = \tilde{A}_{ij}\exp(m_i - m_i^{new})l_i/l_i^{new}$$

```
%26 = linalg.matmul ins(%24, %extracted_slice_6 : tensor<32x32xf16>, tensor<32x64xf16>)
outs(%23 : tensor<32x64xf32>) -> tensor<32x64xf32>
```

Tiled & Decomposed Attention

- Add a sequential loop that corresponds to computing the attention along a rows
- Only load the key and value matrices in this loop and reuse the query and accumulator matrices
- Further distribute the query dimension among warps inside this sequential loop
- Promote operands to shared memory



Code generation using Transform Dialect

- High-level strategy:
  - Tile and distribute attention
  - Vectorize
  - Bufferize
  - Preprocess contractions to get to the right form
  - Hoist transfer reads/writes and additional optimizations
  - Apply layout transformation

Code generation using Transform Dialect

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

    // Get attention op
    // ==========================================
    %attention = transform.structured.match ops{["iree_linalg_ext.attention"]} in %variant_op : (!pdl.operation) -> !pdl.operation

    // Tile and distribute to workgroups
    // ==========================================
    %forall_grid, %tiled_attention =
    transform.iree.tile_to_forall_and_workgroup_count_region %attention tile_sizes [1, 256]
      ( mapping = [#gpu.block<x>, #gpu.block<y>] )

    // Tile and decompose attention
    // ==========================================
    %attention2 = transform.structured.match ops{["iree_linalg_ext.attention"]} in %variant_op : (!pdl.operation) -> !pdl.operation
    %outer_loop, %max_fill, %sum_fill, %mid_loop, %inner_loop, %fill_op, %first_matmul, %reduce_max, %partial_softmax, %reduce_sum, %update,
    %softmax, %scale_acc, %second_matmul = tile_and_decompose_attention %attention2 :
       (!pdl.operation) -> (!pdl.operation, !pdl.operation, !pdl.operation, !pdl.operation, !pdl.operation, !pdl.operation, !pdl.operation, !pdl.operation, !pdl.operat
pdl.operation, !pdl.operation, !pdl.operation, !pdl.operation, !pdl.operation)

    // Vectorize function
    // ==========================================
    %func = transform.structured.match ops{["func.func"]} in %variant_op : (!pdl.operation) -> !pdl.operation
    transform.iree.apply_patterns %func {  rank_reducing_linalg, rank_reducing_vector } : (!pdl.operation) -> ()
    %func_3 = transform.structured.vectorize %func

    // Bufferization
    // ==========================================
    transform.iree.apply_patterns %func_3
      { fold_reassociative_reshapes, canonicalization, tiling_canonicalization, cse } : (!pdl.operation) -> ()
    transform.iree.eliminate_empty_tensors %variant_op : (!pdl.operation) -> ()
    transform.iree.apply_patterns %func_3 { erase_unnecessary_tensor_operands } : (!pdl.operation) -> ()
    %variant_op_3 = transform.iree.bufferize { target_gpu } %variant_op : (!pdl.operation) -> (!pdl.operation)
    %memref_func = transform.structured.match ops{["func.func"]} in %variant_op_3 : (!pdl.operation) -> !pdl.operation
    transform.iree.erase_hal_descriptor_type_from_memref %memref_func : (!pdl.operation) -> ()
```

Code generation using Transform Dialect

```
    // Post-bufferization vector distribution
    // ===========================================================================
    %func_7 = transform.structured.match ops{["func.func"]} in %variant_op_3 : (!pdl.operation) -> !pdl.operation
    transform.iree.forall_to_workgroup %func_7 : (!pdl.operation) -> ()
    transform.iree.map_nested_forall_to_gpu_threads %func_7 workgroup_dims = [4, 8, 2] warp_dims = [2, 1, 1] : (!pdl.operation) -> ()

    %func_8 = transform.structured.hoist_redundant_vector_transfers %memref_func
    : (!pdl.operation) -> !pdl.operation
    transform.iree.apply_patterns %func_8 { fold_memref_aliases } : (!pdl.operation) -> ()
    transform.iree.apply_patterns %func_8 { cse } : (!pdl.operation) -> ()
    transform.iree.apply_patterns %func_8 { canonicalization } : (!pdl.operation) -> ()
    transform.iree.apply_buffer_optimizations %func_8 : (!pdl.operation) -> ()

    // Do layout analysis and lower to mma
    // ===========================================================================
    %func_10 = transform.structured.match ops{["func.func"]} in %variant_op_3 : (!pdl.operation) -> !pdl.operation
    %reordered_func = transform.iree.reorder_transpose %func_10 : (!pdl.operation) -> !pdl.operation
    transform.iree.apply_patterns %reordered_func { cse } : (!pdl.operation) -> ()
    %func_11 = transform.iree.layout_analysis_and_distribution %reordered_func : (!pdl.operation) -> (!pdl.operation)
}
```

IR Before Layout Transformation

```
%9 = vector.transfer_read %alloc[%c0, %8, %c0], %cst_4 {in_bounds = [true, true]} : memref<1x128x64xf16, #gpu.address_space<workgroup>>, vector<32x64xf16>
%10 = arith.extf %9 : vector<32x64xf16> to vector<32x64xf32>
%11:3 = scf.for %arg0 = %c0 to %c4096 step %c128 iter_args(%arg1 = %cst_0, %arg2 = %cst_1, %arg3 = %cst) -> (vector<32xf32>, vector<32xf32>, vector<32x64xf32>) {
  %13 = vector.transfer_read %alloc_11[%c0, %c0], %cst_4 {in_bounds = [true, true]} : memref<128x64xf16, #gpu.address_space<workgroup>>, vector<128x64xf16>
  %14 = arith.extf %13 : vector<128x64xf16> to vector<128x64xf32>
  %15 = vector.contract {indexing_maps = [#map4, #map5, #map6], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %10, %14, %cst_2 :
vector<32x64xf32>, vector<128x64xf32> into vector<32x128xf32>
  %16 = vector.multi_reduction <maxf>, %15, %arg1 [1] : vector<32x128xf32> to vector<32xf32>
  %17 = vector.broadcast %16 : vector<32xf32> to vector<128x32xf32>
  %18 = vector.transpose %17, [1, 0] : vector<128x32xf32> to vector<32x128xf32>
  %19 = arith.subf %15, %18 : vector<32x128xf32>
  %20 = math.exp2 %19 : vector<32x128xf32>
  %21 = arith.subf %arg1, %16 : vector<32xf32>
  %22 = math.exp2 %21 : vector<32xf32>
  %23 = arith.mulf %22, %arg2 : vector<32xf32>
  %24 = vector.multi_reduction <add>, %20, %23 [1] : vector<32x128xf32> to vector<32xf32>
  %25 = arith.divf %cst_3, %24 : vector<32xf32>
  %26 = vector.broadcast %25 : vector<32xf32> to vector<128x32xf32>
  %27 = vector.transpose %26, [1, 0] : vector<128x32xf32> to vector<32x128xf32>
  %28 = arith.mulf %20, %27 : vector<32x128xf32>
  %29 = arith.truncf %28 : vector<32x128xf32> to vector<32x128xf16>
  %30 = vector.broadcast %23 : vector<32xf32> to vector<64x32xf32>
  %31 = vector.broadcast %25 : vector<32xf32> to vector<64x32xf32>
  %32 = vector.transpose %30, [1, 0] : vector<64x32xf32> to vector<32x64xf32>
  %33 = vector.transpose %31, [1, 0] : vector<64x32xf32> to vector<32x64xf32>
  %34 = arith.mulf %32, %33 : vector<32x64xf32>
  %35 = arith.mulf %34, %arg3 : vector<32x64xf32>
  %36 = arith.extf %29 : vector<32x128xf16> to vector<32x128xf32>
  %37 = vector.transfer_read %alloc_12[%c0, %c0], %cst_4 {in_bounds = [true, true], permutation_map = #map7} : memref<128x64xf16, #gpu.address_space<workgroup>>,
vector<64x128xf16>
  %38 = arith.extf %37 : vector<64x128xf16> to vector<64x128xf32>
  %39 = vector.contract {indexing_maps = [#map4, #map5, #map6], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %36, %38, %35 :
vector<32x128xf32>, vector<64x128xf32> into vector<32x64xf32>
  scf.yield %16, %24, %39 : vector<32xf32>, vector<32xf32>, vector<32x64xf32>
}
%12 = arith.truncf %11#2 : vector<32x64xf32> to vector<32x64xf16>
vector.transfer_write %12, %alloc_7[%c0, %8, %c0] {in_bounds = [true, true]} : vector<32x64xf16>, memref<1x128x64xf16, #gpu.address_space<workgroup>>
```

Layout Propagation

```
%9 = vector.transfer_read %alloc[%c0, %8, %c0], %cst_4 {in_bounds = [true, true]} : memref<1x128x64xf16, #gpu.address_space<workgroup>>, vector<32x64xf16>
%10 = arith.extf %9 : vector<32x64xf16> to vector<32x64xf32>
%11:3 = scf.for %arg0 = %c0 to %c4096 step %c128 iter_args(%arg1 = %cst_0, %arg2 = %cst_1, %arg3 = %cst) -> (vector<32xf32>, vector<32xf32>, vector<32x64xf32>) {
  %13 = vector.transfer_read %alloc_11[%c0, %c0], %cst_4 {in_bounds = [true, true]} : memref<128x64xf16, #gpu.address_space<workgroup>>, vector<128x64xf16>
  %14 = arith.extf %13 : vector<128x64xf16> to vector<128x64xf32>
  %15 = vector.contract {indexing_maps = [#map4, #map5, #map6], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %10, %14, %cst_2 :
vector<32x64xf32>, vector<128x64xf32> into vector<32x128xf32>
  %16 = vector.multi_reduction <maxf>, %15, %arg1 [1] : vector<32x128xf32> to vector<32xf32>
  %17 = vector.broadcast %16 : vector<32xf32> to vector<128x32xf32>
  %18 = vector.transpose %17, [1, 0] : vector<128x32xf32> to vector<32x128xf32>
  %19 = arith.subf %15, %18 : vector<32x128xf32>
  %20 = math.exp2 %19 : vector<32x128xf32>
  %21 = arith.subf %arg1, %16 : vector<32xf32>
  %22 = math.exp2 %21 : vector<32xf32>
  %23 = arith.mulf %22, %arg2 : vector<32xf32>
  %24 = vector.multi_reduction <add>, %20, %23 [1] : vector<32x128xf32> to vector<32xf32>
  %25 = arith.divf %cst_3, %24 : vector<32xf32>
  %26 = vector.broadcast %25 : vector<32xf32> to vector<128x32xf32>
  %27 = vector.transpose %26, [1, 0] : vector<128x32xf32> to vector<32x128xf32>
  %28 = arith.mulf %20, %27 : vector<32x128xf32>
  %29 = arith.truncf %28 : vector<  Layout Conflict #1  <32x128xf16>
  %30 = vector.broadcast %25 : vec                      <64x32xf32>
  %31 = vector.broadcast %25 : vector<32xf32> to vector<64x32xf32>
  %32 = vector.transpose %30, [1, 0] : vector<64x32xf32> to vector<32x64xf32>
  %33 = vector.transpose %31, [1, 0] : vector<64x32xf32> to vector<32x64xf32>
  %34 = arith.mulf %32, %33 : vector<32x64xf32>
  %35 = arith.mulf %34, %arg3 : vector<32x64xf32>
  %36 = arith.extf %29 : vector<32x128xf16> to vector<32x128xf32>
  %37 = vector.transfer_read %alloc_12[%c0, %c0], %cst_4 {in_bounds = [true, true], permutation_map = #map7} : memref<128x64xf16, #gpu.address_space<workgroup>>,
vector<64x128xf16>
  %38 = arith.extf %37 : vector<64x128xf16> to vector<64x128xf32>
  %39 = vector.contract {indexing_maps = [#map4, #map5, #map6], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %36, %38, %35 :
vector<32x128xf32>, vector<64x128xf32> into vector<32x64xf32>
  scf.yield %16, %24, %39 : vector<32xf32>, vector<32xf32>, vector<32x64xf32>
}
%12 = arith.truncf %11#2 : vector<32x64xf32> to vector<32x64xf16>
vector.transfer_write %12, %alloc_7[%c0, %8, %c0] {in_bounds = [true, true]} : vector<32x64xf16>, memref<1x128x64xf16, #gpu.address_space<workgroup>>
```

Layout Propagation (Pseudo 2D)

```
%9 = vector.transfer_read %alloc[%c0, %8, %c0], %cst_4 {in_bounds = [true, true]} : memref<1x128x64xf16, #gpu.address_space<workgroup>>, vector<32x64xf16>
%10 = arith.extf %9 : vector<32x64xf16> to vector<32x64xf32>
%11:3 = scf.for %arg0 = %c0 to %c4096 step %c128 iter_args(%arg1 = %cst_0, %arg2 = %cst_1, %arg3 = %cst) -> (vector<32xf32>, vector<32xf32>, vector<32x64xf32>) {
  %13 = vector.transfer_read %alloc_11[%c0, %c0], %cst_4 {in_bounds = [true, true]} : memref<128x64xf16, #gpu.address_space<workgroup>>, vector<128x64xf16>
  %14 = arith.extf %13 : vector<128x64xf16> to vector<128x64xf32>
  %15 = vector.contract {indexing_maps = [#map4, #map5, #map6], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %10, %14, %cst_2 :
vector<32x64xf32>, vector<128x64xf32> into vector<32x128xf32>
  %16 = vector.multi_reduction <maxf>, %15, %arg1 [1] : vector<32x128xf32> to vector<32xf32>
  %17 = vector.broadcast %16 : vector<32xf32> to vector<128x32xf32>
  %18 = vector.transpose %17, [1, 0] : vector<128x32xf32> to vector<32x128xf32>
  %19 = arith.subf %15, %18 : vector<32x128xf32>
  %20 = math.exp2 %19 : vector<32x128xf32>
  %21 = arith.subf %arg1, %16 : vector<32xf32>
  %22 = math.exp2 %21 : vector<32xf32>
  %23 = arith.mulf %22, %arg2 : vector<32xf32>
  %24 = vector.multi_reduction <add>, %20, %23 [1] : vector<32x128xf32> to vector<32xf32>
  %25 = arith.divf %cst_3, %24 : vector<32xf32>
  %26 = vector.broadcast %25 : vector<32xf32> to vector<128x32xf32>
  %27 = vector.transpose %26, [1, 0] : vector<128x32xf32> to vector<32x128xf32>
  %28 = arith.mulf %20, %27 : vector<32x128xf32>
  %29 = arith.truncf %28 : vector<32x128xf32> to vector<32x128xf16>
  %30 = vector.broadcast %23 : vector<32xf32> to vector<64x32xf32>
  %31 = vector.broadcast %25 : vector<32xf32> to vector<64x32xf32>
  %32 = vector.transpose %30, [1, 0] : vector<64x32xf32> to vector<32x64xf32>
  %33 = vector.transpose %31, [1, 0] : vector<64x32xf32> to vector<32x64xf32>
  %34 = arith.mulf %32, %33 : vector<32x64xf32>
  %35 = arith.mulf %34, %arg3 : vect
  %36 = arith.extf %29 : vector<32x128xf16> to vector<32x128xf32>
  %37 = vector.transfer_read %alloc_12[%c0, %c0], %cst_4 {in_bounds = [true, true], permutation_map = #map7} : memref<128x64xf16, #gpu.address_space<workgroup>>,
vector<64x128xf16>
  %38 = arith.extf %37 : vector<64x128xf16> to vector<64x128xf32>
  %39 = vector.contract {indexing_maps = [#map4, #map5, #map6], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %36, %38, %35 :
vector<32x128xf32>, vector<64x128xf32> into vector<32x64xf32>
  scf.yield %16, %24, %39 : vector<32xf32>, vector<32xf32>, vector<32x64xf32>
}
%12 = arith.truncf %11#2 : vector<32x64xf32> to vector<32x64xf16>
vector.transfer_write %12, %alloc_7[%c0, %8, %c0] {in_bounds = [true, true]} : vector<32x64xf16>, memref<1x128x64xf16, #gpu.address_space<workgroup>>
```

Layout Conflict #2

Resolving Layout Conflicts

- Different Types of Layout Conflicts
  - Lane conflicts
    - Require trip to shared memory
  - Vector and/or Batch conflicts
    - Can be resolved by appropriate broadcasting / extract operations
  - Layout Conflict #1 is a batch and vector conflict
  - Layout Conflict #2 is a batch conflict

Resolving Layout Conflicts

- Layout Conflict #1

- We have a vector<2x16x2x2xf16>, but we need a vector<2x8x4x2xf16>

```
%7084 = vector.extract %7083[0, 0] : vector<2x16x2x2xf16>
%7085 = vector.insert_strided_slice %7084, %cst_1100 {offsets = [0, 0, 0, 0], strides = [1, 1]} : vector<2x2xf16> into
vector<2x8x4x2xf16>
%7086 = vector.extract %7083[0, 1] : vector<2x16x2x2xf16>
%7087 = vector.insert_strided_slice %7086, %7085 {offsets = [0, 0, 2, 0], strides = [1, 1]} : vector<2x2xf16> into
vector<2x8x4x2xf16>
%7088 = vector.extract %7083[0, 2] : vector<2x16x2x2xf16
```

Resolving Layout Conflicts

- Layout Conflict #2
- We have a vector<2x16x2x2xf16>, but we need a vector<2x8x2x2xf16>
- Here we can extract the relevant slice since it is a pseudo 2D vector

```
%7149 = vector.extract_strided_slice %7148 {offsets = [0, 0, 0, 0], sizes = [2, 8, 2, 2], strides = [1, 1, 1, 1]} :
vector<2x16x2x2xf16> to vector<2x8x2x2xf16>
```

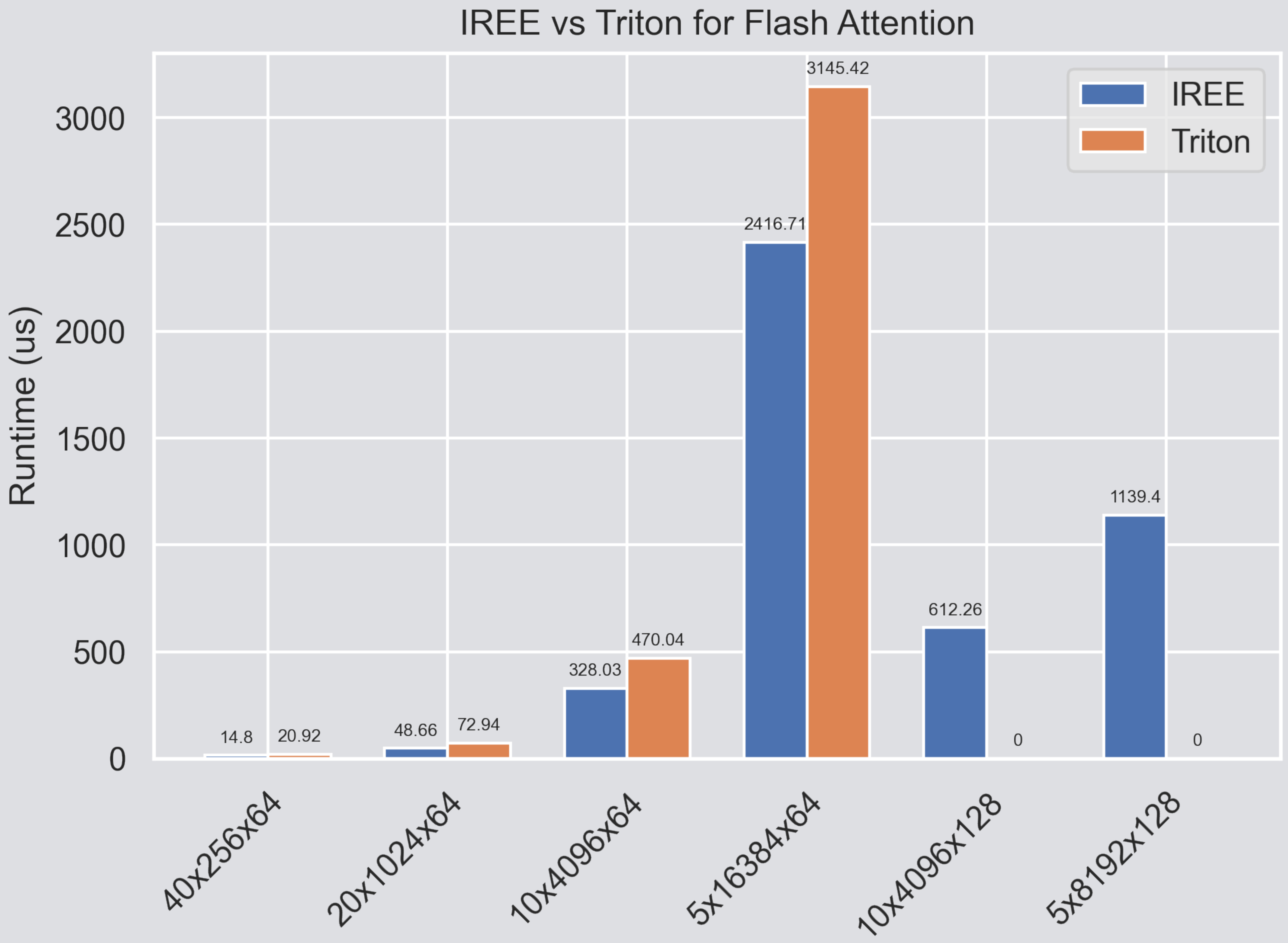
PTX Code generation

- After layout propagation, we apply the vector distribution rules we showed earlier
- Can then be lowered through existing IREE pipeline to generate PTX code

```
ldmatrix.sync.aligned.m8n8.x4.shared.b16 {%r453, %r454, %r455, %r456}, [%rd64];
mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
        {%f322, %f323, %f324, %f325},
        {%hh1, %hh3, %hh2, %hh4},
        {%hh33, %hh34},
        {%f255, %f255, %f255, %f255};

...
shfl.sync.bfly.b32  %r537|%p272, %r536, 2, 31, -1;
...
ldmatrix.sync.aligned.m8n8.x4.trans.shared.b16 {%r1478, %r1479, %r1480, %r1481}, [%rd96];
 mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
        {%f828, %f829, %f830, %f831},
        {%hh85, %hh86, %hh87, %hh88},
        {%hh147, %hh148},
        {%f824, %f825, %f826, %f827};
```

Results

- We test against Triton on an NVIDIA A100 on a range of sequence lengths going up to 16k
- Disable causal masking and scaling
- Benchmarking results captured using nsys



Conclusions & Future Work

- Introduced a high dimensional layout for vector distribution that is expressive, self-contained and generalizable
- Generalize this approach to represent the mapping as a series of composable transformations instead of a fixed vector
- Move from implicit to explicit IR representation
- Add other variations of attention (causal attention, backward, etc.)
- Apply to other hardware vendors
    - AMD GPUs using Vulkan, Custom accelerators, etc.

Acknowledgements

- Joint work with Thomas Raoux and Nicolas Vasilache
- Thanks to nod.ai and IREE team for support

References

- Dao, T. et al (2022) FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness
- Menon. H. (2023) Decomposable operators in IREE: Winograd Convolutions and Flash Attention
- Menon, H. (2022) MLIR Code Generation Tutorial