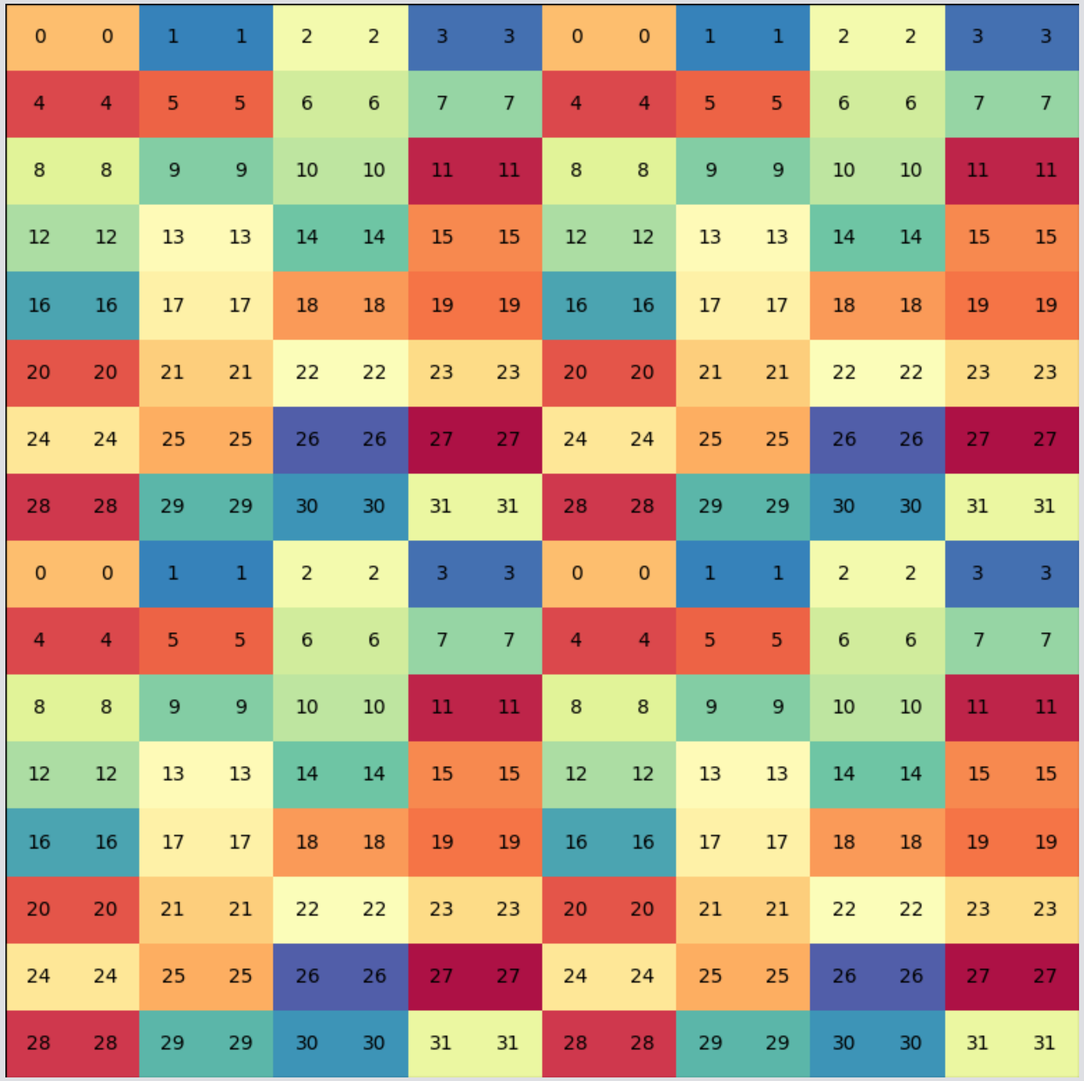
High-dimensional layout representations for vector distribution

Harsh Menon (nod.ai)

Motivation

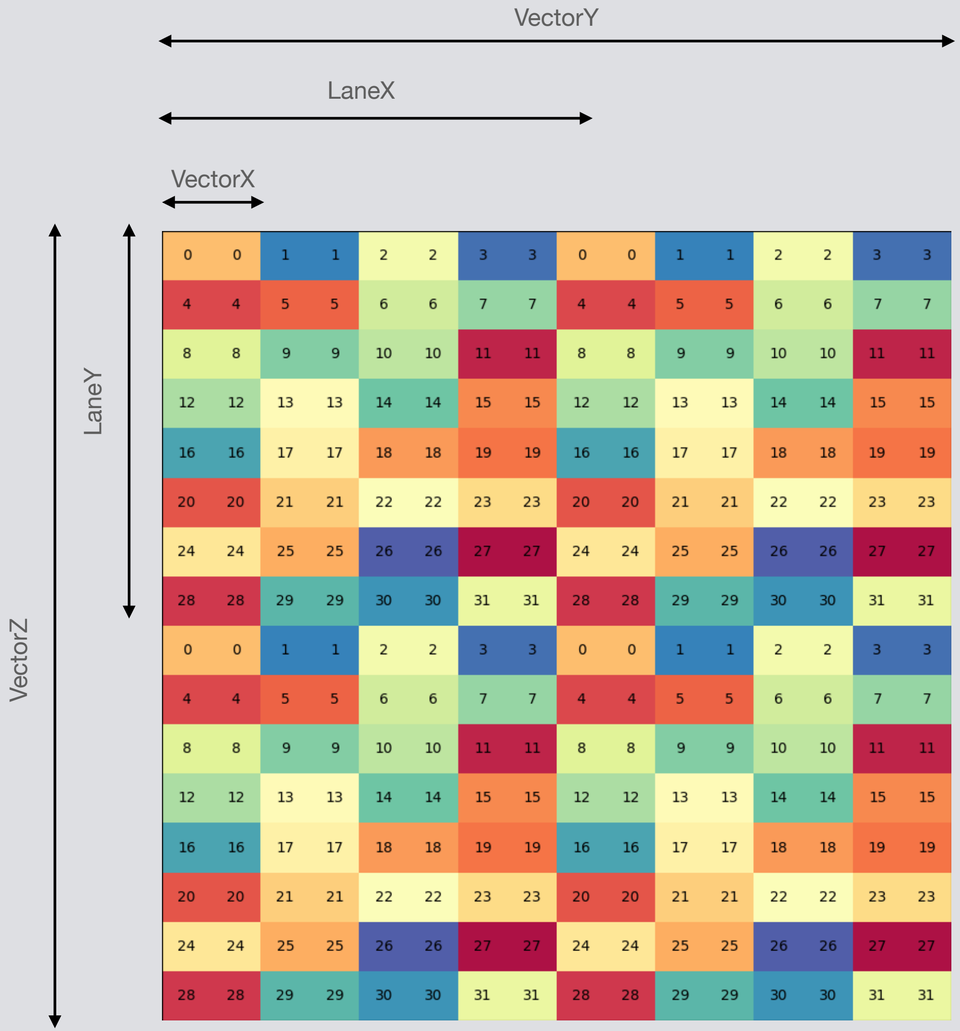
- Matrix multiplication is core computation unit of neural networks
- Specialized matrix-multiplication hardware such as tensor cores usually require the input/output data to be in a specific layout
- For NVIDIA GPUs, this layout specifies the mapping of threads to elements of the matrices (as shown 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?



Higher-Dimensional Layout Representations

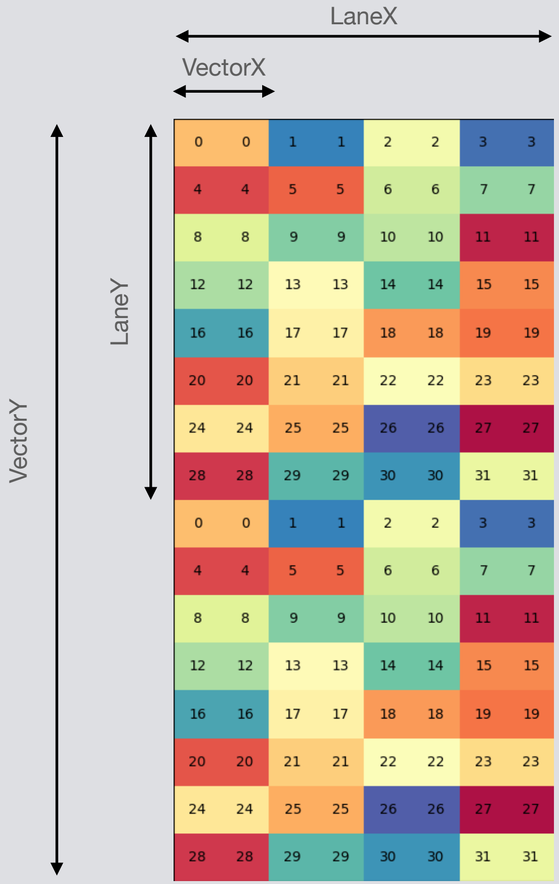
- Can we use a high dimensional layout representation?
- What might this high dimensional representation contain?
  - Vector dimensions (X, Y, Z)
  - Lane dimensions (X, Y, Z)
  - Batch dimensions (for row and column)
  - **Batch (row)** x **Batch (column)** x **LaneZ** x **LaneY** x **LaneX** x **VectorZ** x **VectorY** x **VectorX**
  - **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

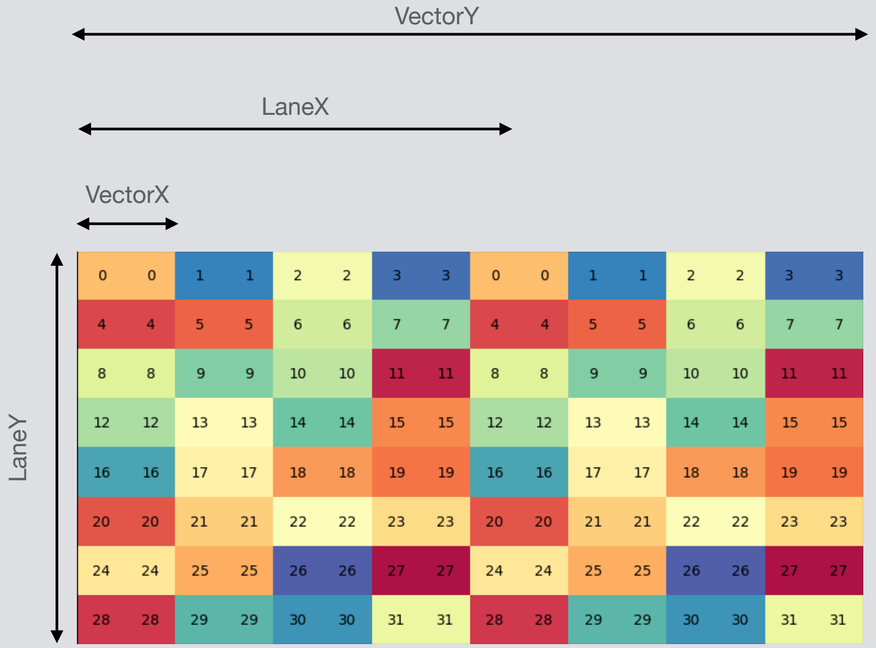


Higher-Dimensional Layout Representations

- 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

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)

$$\text{row} = l_y + 8v_z$$
$$\text{col} = v_x + 2l_x + 8v_y$$

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

```
%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

```
%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 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>
```



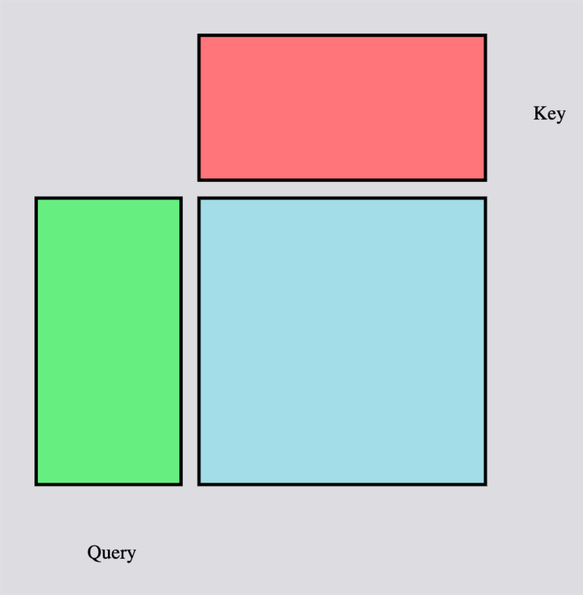
```
%shuffleResult_14, %valid_15 = gpu.shuffle …
%154 = vector.broadcast %shuffleResult_14 : i32 to vector<1xi32>
%155 = vector.bitcast %154 : vector<1xi32> to vector<2xf16>
%156 = arith.maxf %155, %151 : vector<2xf16>
...
%175 = vector.insert_strided_slice %170, %174
       {offsets = [0, 1, 1, 0], 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

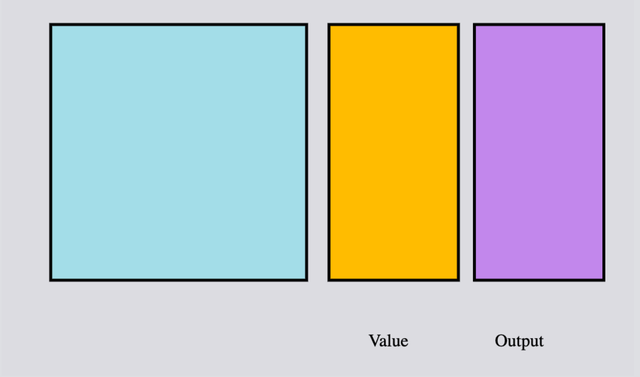
- **Inputs:** Query Matrix (Q), Key Matrix (K) and Value Matrix (V)
- Each of the inputs have shape (B x N x d) where B is the batch dimension, N is the sequence length and d is the head dimension
- Typically, sequence length is much larger than head dimension
- Usually, the operator includes additional 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



$$S = QK^T$$

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

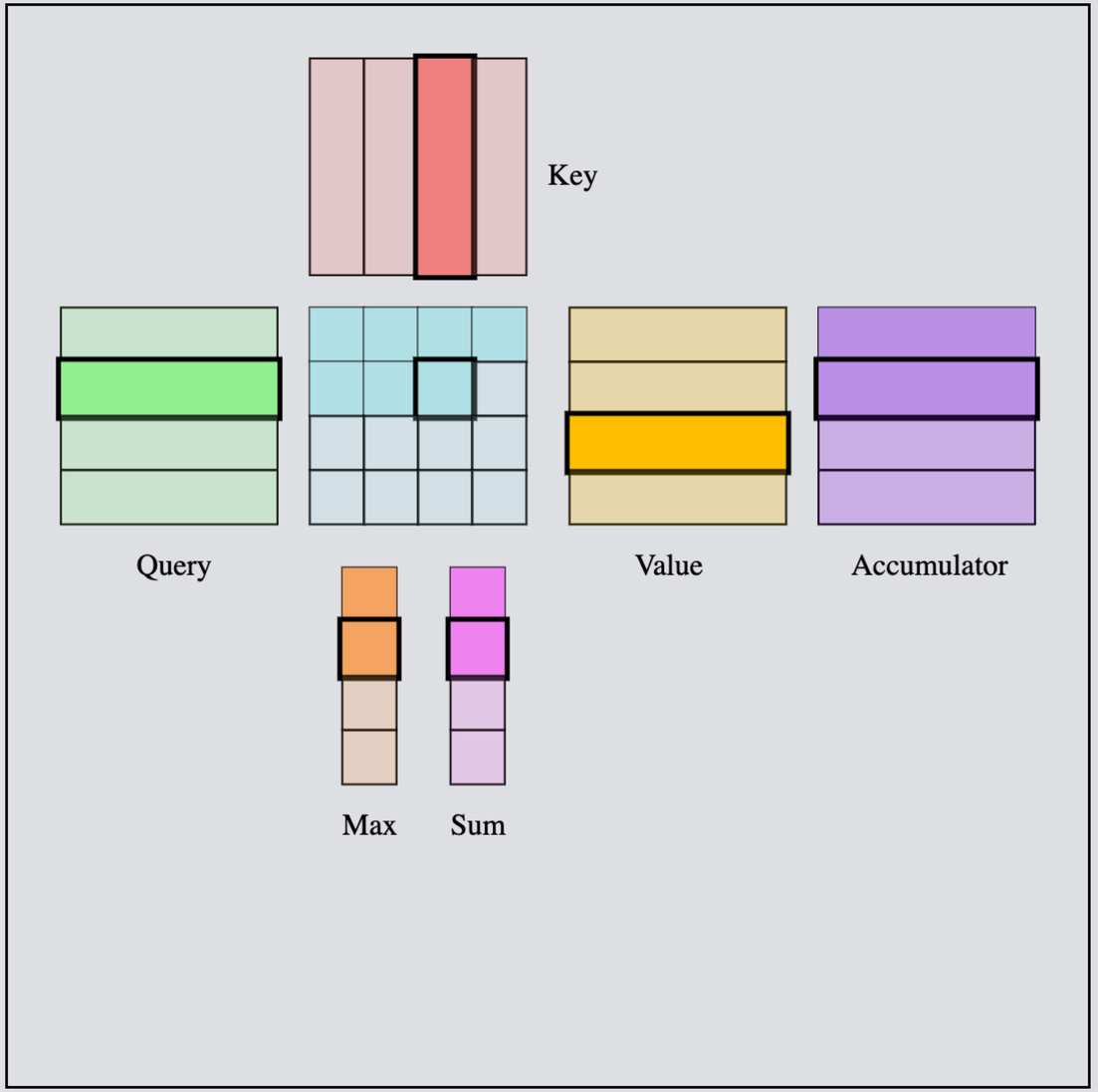
$$O = PV$$



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 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>
}
```

IR Before Layout Transformation

```
%9 = vector.transfer_read %alloc[%c0, %8, %c0], %cst_2 {in_bounds = [true, true]} : memref<1x128x64xf16, #gpu.address_space<workgroup>>, vector<32x64xf16>
%10 = vector.transfer_read %alloc_6[%8], %cst_2 {in_bounds = [true]} : memref<128xf16, #gpu.address_space<workgroup>>, vector<32xf16>
%11 = vector.transfer_read %alloc_5[%8], %cst_2 {in_bounds = [true]} : memref<128xf16, #gpu.address_space<workgroup>>, vector<32xf16>
%12 = vector.transfer_read %alloc_4[%c0, %8, %c0], %cst_2 {in_bounds = [true, true]} : memref<1x128x64xf16, #gpu.address_space<workgroup>>, vector<32x64xf16>
%13:3 = scf.for %arg0 = %c0 to %c1024 step %c128 iter_args(%arg1 = %10, %arg2 = %11, %arg3 = %12) -> (vector<32xf16>, vector<32xf16>, vector<32x64xf16>) {
  %14 = vector.transfer_read %alloc_9[%c0, %c0], %cst_2 {in_bounds = [true, true]} : memref<128x64xf16, #gpu.address_space<workgroup>>, vector<128x64xf16>
  %15 = vector.contract {indexing_maps = [#map5, #map6, #map7], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %9, %14, %cst_1 :
vector<32x64xf16>, vector<128x64xf16> into vector<32x128xf16>
  %16 = vector.multi_reduction <maxf>, %15, %arg1 [1] : vector<32x128xf16> to vector<32xf16>
  %17 = vector.broadcast %16 : vector<32xf16> to vector<128x32xf16>
  %18 = vector.transpose %17, [1, 0] : vector<128x32xf16> to vector<32x128xf16>
  %19 = arith.subf %15, %18 : vector<32x128xf16>
  %20 = math.exp %19 : vector<32x128xf16>
  %21 = arith.subf %arg1, %16 : vector<32xf16>
  %22 = math.exp %21 : vector<32xf16>
  %23 = arith.mulf %22, %arg2 : vector<32xf16>
  %24 = vector.multi_reduction <add>, %20, %23 [1] : vector<32x128xf16> to vector<32xf16>
  %25 = vector.broadcast %24 : vector<32xf16> to vector<128x32xf16>
  %26 = vector.transpose %25, [1, 0] : vector<128x32xf16> to vector<32x128xf16>
  %27 = arith.divf %20, %26 : vector<32x128xf16>
  %28 = vector.broadcast %23 : vector<32xf16> to vector<64x32xf16>
  %29 = vector.broadcast %24 : vector<32xf16> to vector<64x32xf16>
  %30 = vector.transpose %28, [1, 0] : vector<64x32xf16> to vector<32x64xf16>
  %31 = vector.transpose %29, [1, 0] : vector<64x32xf16> to vector<32x64xf16>
  %32 = arith.divf %30, %31 : vector<32x64xf16>
  %33 = arith.mulf %32, %arg3 : vector<32x64xf16>
  %34 = vector.transfer_read %alloc_10[%c0, %c0], %cst_2 {in_bounds = [true, true], permutation_map = #map8} : memref<128x64xf16, #gpu.address_space<workgroup>>,
vector<64x128xf16>
  %35 = vector.contract {indexing_maps = [#map5, #map6, #map7], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %27, %34, %33 :
vector<32x128xf16>, vector<64x128xf16> into vector<32x64xf16>
  scf.yield %16, %24, %35 : vector<32xf16>, vector<32xf16>, vector<32x64xf16>
}
vector.transfer_write %13#2, %alloc_4[%c0, %8, %c0] {in_bounds = [true, true]} : vector<32x64xf16>, memref<1x128x64xf16, #gpu.address_space<workgroup>>
vector.transfer_write %13#1, %alloc_5[%8] {in_bounds = [true]} : vector<32xf16>, memref<128xf16, #gpu.address_space<workgroup>>
vector.transfer_write %13#0, %alloc_6[%8] {in_bounds = [true]} : vector<32xf16>, memref<128xf16, #gpu.address_space<workgroup>>
```

Layout Propagation

Layout Propagation (Pseudo 2D)

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

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.f16.f16.f16.f16
    {%hh739, %hh740},
    {%hh1, %hh3, %hh2, %hh4},
    {%hh610, %hh611},
    {%hh96, %hh96};
mma.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16
    {%hh741, %hh742},
    {%hh5, %hh7, %hh6, %hh8},
    {%hh612, %hh613},
    {%hh739, %hh740};
...
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.f16.f16.f16.f16
    {%hh1991, %hh1992},
    {%hh1735, %hh1736, %hh1737, %hh1738},
    {%hh1863, %hh1864},
    {%hh1831, %hh1832};
mma.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16
    {%hh1993, %hh1994},
    {%hh1739, %hh1740, %hh1741, %hh1742},
    {%hh1865, %hh1866},
    {%hh1991, %hh1992}
```

Conclusions & Future Work

- Introduced a high dimensional layout for vector distribution that is expressive, self-contained and generalizable
- Evaluating performance of current approach and adding additional performance optimizations
- 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
- Apply to other hardware vendors
    - AMD GPUs using Vulkan, Custom accelerators, etc.

Acknowledgements

- Joint work with Thomas Raoux
- 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