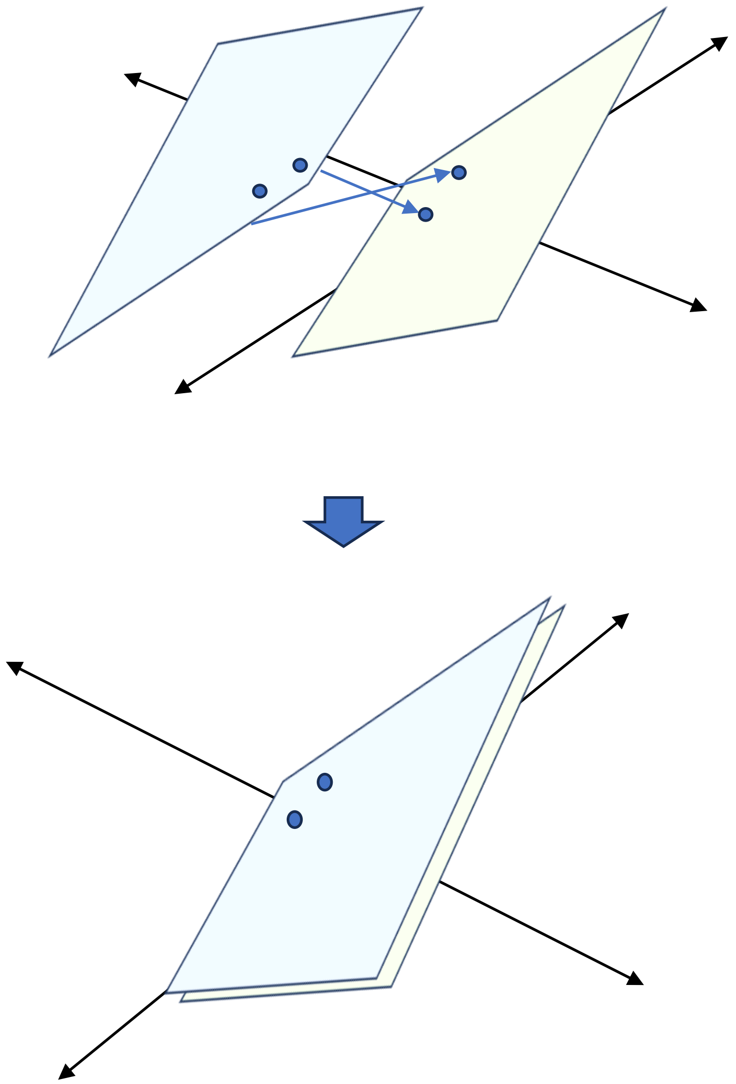
Qualcomm

MLIR Linalg Op Fusion - Theory & Practice

Javed Absar, Principal Engineer, Qualcomm Technologies International, Ltd.
Muthu M. Baskaran, Principal Engineer, Qualcomm Technologies, Inc.

Contents

- Linalg Dialect & Ops — LINALG OP
- What is Op Fusion? Why? — FUSION
- Op Fusion in Linalg — THEORY
- Fusion in ML Kernels — AND PRACTICE
- Conclusion

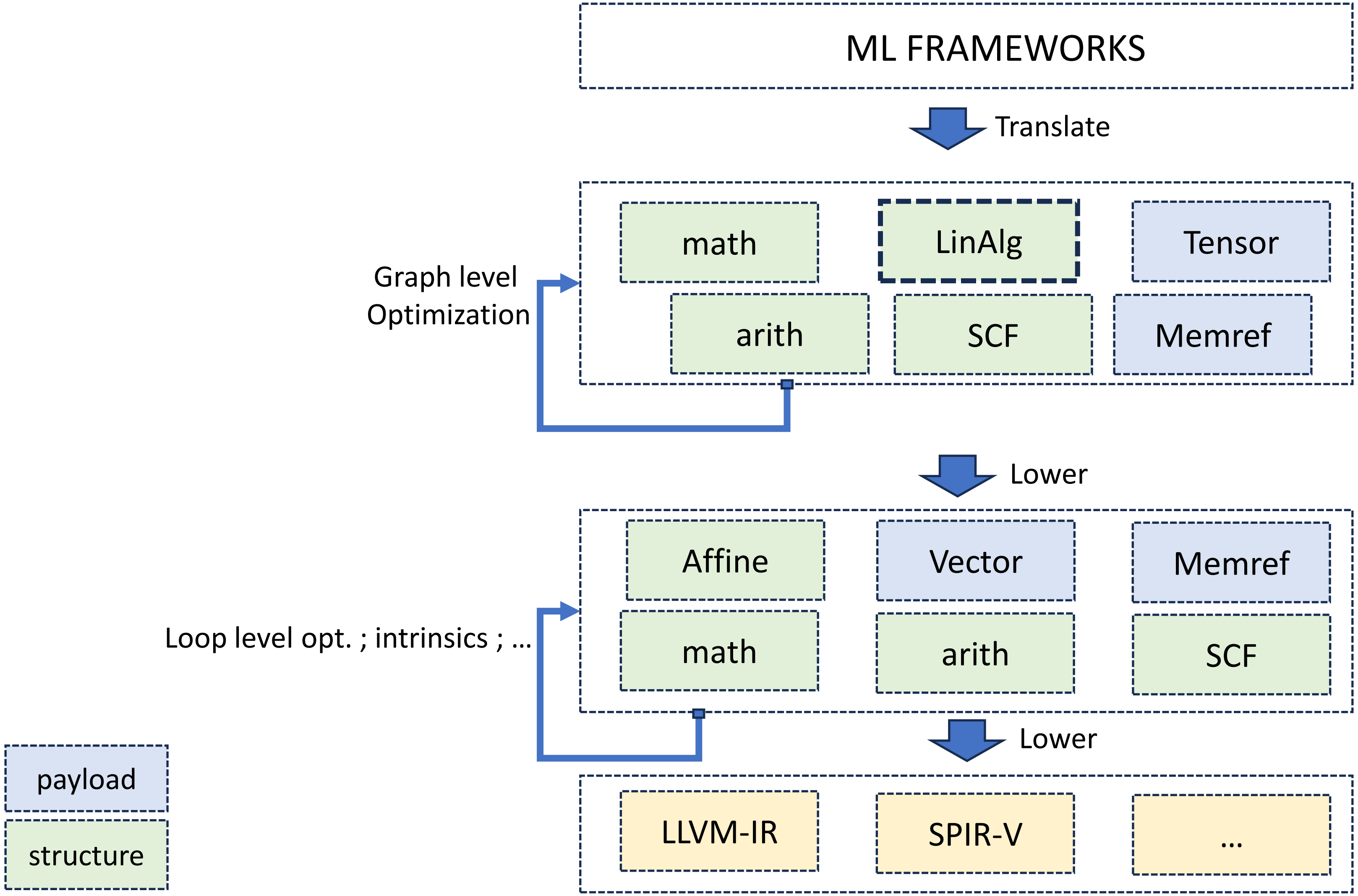


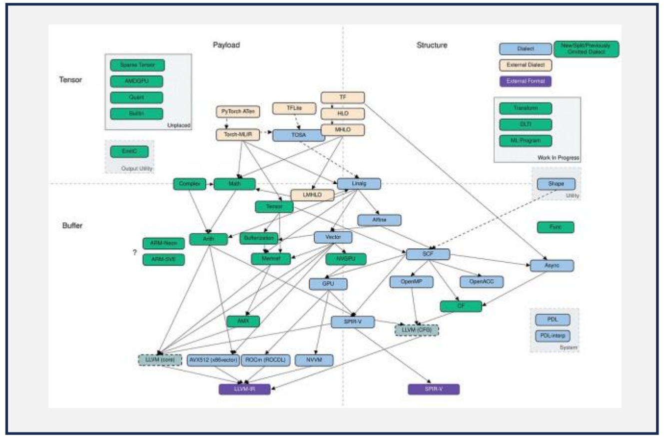
`git credit *linalg_fusion*`

- Aart Bik
- Albert Cohen
- Alexander Belyaev
- **Alex Zinenko**
- Amir Bishara
- Aviad Cohen
- Chris Lattner
- Geoffrey Martin Noble
- Guray Ozen
- **Hanhan Wang**
- Ivan Butygin
- Javed Absar
- Jakub Kuderski
- Julian Cross
- Jacques Pienaar
- Lei Zhang
- …

- Lorenzo Chelini
- **Mahesh Ravishankar**
- Matthias Springer
- Mehdi Amini
- Michelle Scuttari
- **Nicholas Vasilache**
- Nirved
- Oleg Shyshkov
- Quinn Dawkins
- River Riddle
- Stephan Herhut
- Sean Silva
- Thomas Raoux
- Tres Popp
- Tobias Gysi
- Tim Harvey
- …

LinAlg Dialect & Ops

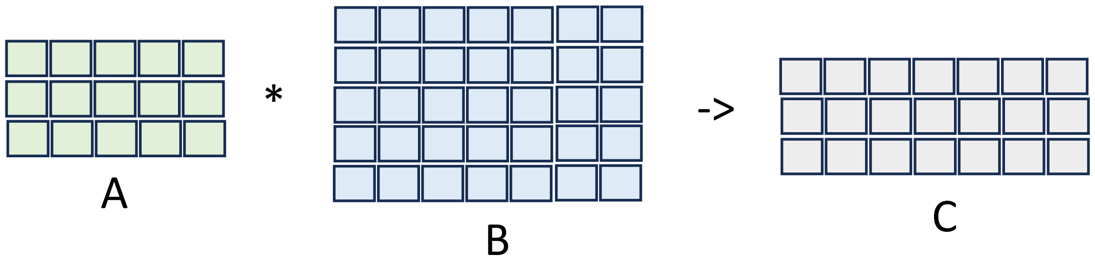




Linalg Dialect & Ops

NAMED-OPS

```
%res = linalg.matmul
           ins(%A, %B : tensor<3x5xf32>,
                        tensor<5x7xf32>)
           outs(%C: tensor<3x7xf32>)
           -> tensor<3x7xf32>
```

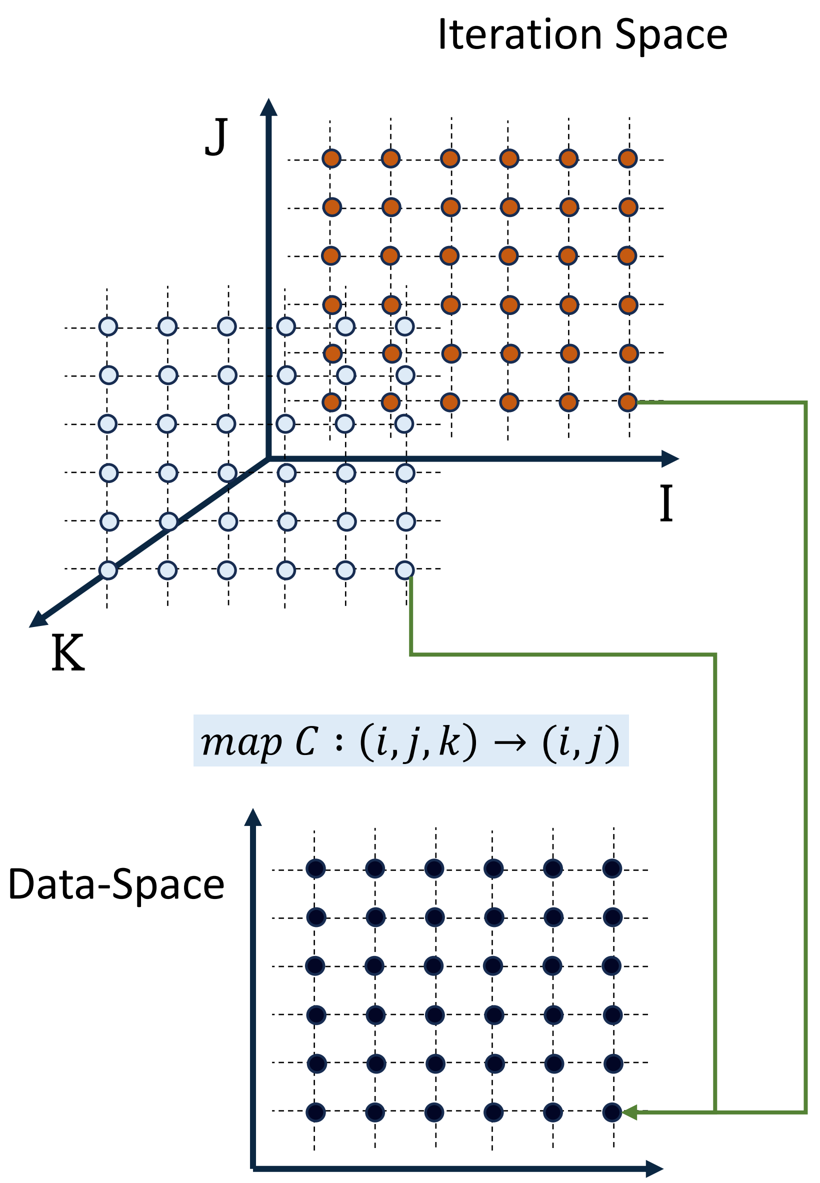


Linalg Generic

mlir-opt -linalg-generalize-named-ops

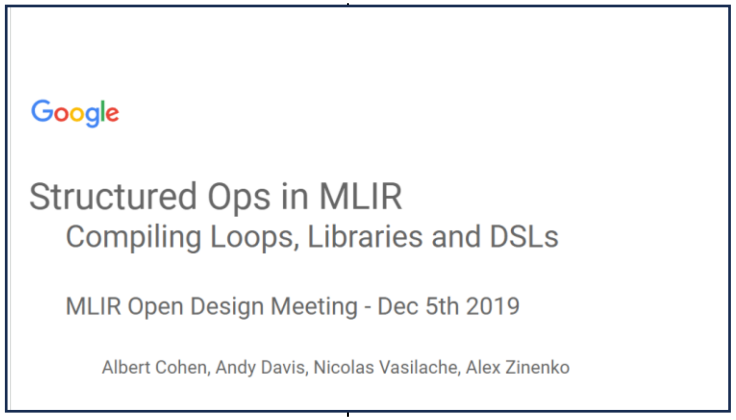
```
#map = affine_map<(i, j, k) -> (i, k)>
#map1 = affine_map<(i, j, k) -> (k, j)>
#map2 = affine_map<(i, j, k) -> (i, j)>
...
.. = linalg.generic
         {indexing_maps = [#map, #map1, #map2],
         iterator_types = ["parallel", "parallel", "reduction"]}
         ins(%A, %B : tensor<3x5xf32>, tensor<5x7xf32>)
         outs(%C : tensor<3x7xf32>) {
    ^bb0(%a: f32, %b: f32, %c_in: f32):
      %a_times_b = arith.mulf %a, %b : f32
      %c_out = arith.addf %c_in, %a_times_b : f32
      linalg.yield %c_out : f32
    } -> tensor<3x7xf32>
```

- Structured Op; i.e. structured data + structured iterators as a cohorent unit
- Iterator : – implicit perfectly nested - parallel, reduction from op-name
- Affine Map : iteration space inferred from input, output sizes
- block args at each iteration point
- Outs – initial value, shape, destination passing
- Trait attributes – doc, index map, library call, iterator types



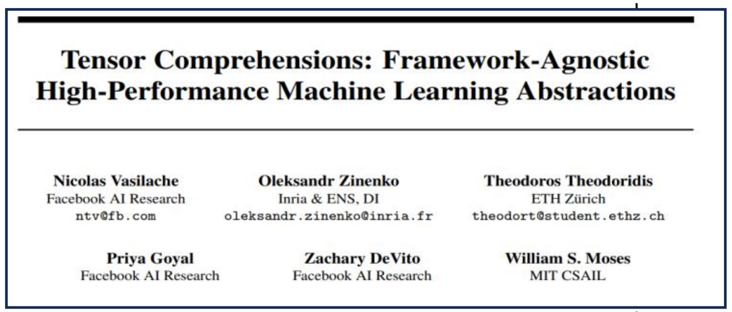
Linalg Generic – Lower to Loops

```
%c0 = arith.constant 0 : index
%c3 = arith.constant 3 : index
%c1 = arith.constant 1 : index
%c7 = arith.constant 7 : index
%c5 = arith.constant 5 : index
scf.for %arg3 = %c0 to %c3 step %c1 {
  scf.for %arg4 = %c0 to %c7 step %c1 {
    scf.for %arg5 = %c0 to %c5 step %c1 {
      %0 = memref.load %A[%arg3, %arg5]
            : memref<3x5xf32, strided<[?, ?], offset: ?>>
      %1 = memref.load %B[%arg5, %arg4]
            : memref<5x7xf32, strided<[?, ?], offset: ?>>
      %2 = memref.load %C[%arg3, %arg4]
            : memref<3x7xf32, strided<[?, ?], offset: ?>>
      %3 = arith.mulf %0, %1 : f32
      %4 = arith.addf %2, %3 : f32
      memref.store %4, %C[%arg3, %arg4]
           : memref<3x7xf32, strided<[?, ?], offset: ?>>
    }
  }
}
```



Linalg Named Ops

```python
@linalg_structured_op
def matmul(
    A=TensorDef(T1, S.M, S.K),
    B=TensorDef(T2, S.K, S.N),
    C=TensorDef(U, S.M, S.N, output=True),
    cast=TypeFnAttrDef(default=TypeFn.cast_signed),
):
    """Performs a matrix multiplication of two 2D inputs.

    Numeric casting is performed on the operands to the inner multiply, promoting
    them to the same data type as the accumulator/output.
    """
    domain(D.m, D.n, D.k)
    implements(ContractionOpInterface)
    C[D.m, D.n] += cast(U, A[D.m, D.k]) * cast(U, B[D.k, D.n])
```



Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions

Nicolas Vasilache, Facebook AI Research, ntv@fb.com
Oleksandr Zinenko, Inria & ENS, DI, oleksandr.zinenko@inria.fr
Theodoros Theodoridis, ETH Zürich, theodort@student.ethz.ch
Priya Goyal, Facebook AI Research
Zachary DeVito, Facebook AI Research
William S. Moses, MIT CSAIL

linalg_opdsl : Python based DSL for authoring Linalg op definitions.
Inspired by Tensor Comprehensions but adapted to represent linalg structured ops.

```
copy, elemwise_unary, exp, log, abs, ceil, floor, negf, elemwise_binary, add, sub, mul, div, div_unsigned, max, matmul, matmul_unsigned,
quantized_matmul,  matmul_transpose_a, matmul_transpose_b, mmt4d, batch_mmt4d, batch_matmul, batch_matmul_transpose_a, batch_matmul_transpose_b,
quantized_batch_matmul, batch_reduce_matmul, matvec, vecmat,  batch_matvec, batch_vecmat, dot, conv_1d, conv_2d, conv_3d, conv_1d_nwc_wcf,
 conv_1d_ncw_fcw, conv_2d_nhwc_hwcf, conv_2d_nhwc_fhwc, conv_2d_nhwc_hwcf_q, conv_2d_nhwc_fhwc_q, conv_2d_nchw_fchw, conv_2d_ngchw_fgchw,
conv_2d_ngchw_gfchw, conv_3d_ndhwc_dhwcf, conv_3d_ndhwc_dhwcf_q, conv_3d_ncdhw_fcdhw, depthwise_conv_1d_nwc_wc, depthwise_conv_1d_ncw_cw,
depthwise_conv_1d_nwc_wcm, depthwise_conv_2d_nhwc_hwc, depthwise_conv_2d_nchw_chw, depthwise_conv_2d_nhwc_hwc_q, depthwise_conv_2d_nhwc_hwcm,
depthwise_conv_2d_nhwc_hwcm_q, depthwise_conv_3d_ndhwc_dhwc, depthwise_conv_3d_ncdhw_cdhw, depthwise_conv_3d_ndhwc_dhwcm, pooling_nhwc_sum,
pooling_nchw_sum, pooling_nhwc_max, pooling_nhwc_max_unsigned, pooling_nchw_max, pooling_nhwc_min, pooling_nhwc_min_unsigned, pooling_nwc_sum,
pooling_ncw_sum, pooling_nwc_max, pooling_nwc_max_unsigned, pooling_ncw_max, pooling_nwc_min, pooling_nwc_min_unsigned, pooling_ndhwc_sum,
pooling_ndhwc_max, pooling_ndhwc_min, fill, fill_rng_2d, …
```

What and Why : Op Fusion?

- What ?
  - Operator fusion ~ kernel fusion ~ loop fusion ?
- Why ?
  - Series of Linalg Ops after translation
  - Improves efficiency of DNN
    - Eliminate materialization of intermediate results (write to mem/read bac)
    - Reduce unnecessary scan of inputs
    - Eliminate unnecessary broadcast
    - Enable other optimizations
- But then, what about …?
  - Larger kernel ? Re-computation? vector register pressure? false dependence? Reduce parallelism? Always works and is great?

Linalg Op-Fusion (Producer-Consumer)

```
#map = affine_map<(d0, d1) -> (d0, d1)>
func.func @foo(%X : tensor<?x?xf32>, %Y : tensor<?x?xf32>,
               %Z: tensor<?x?xf32>) -> tensor<?x?xf32> {
  %0 = linalg.generic {
      indexing_maps = [#map, #map],
      iterator_types = ["parallel", "parallel"]}
      ins(%X : tensor<?x?xf32>) outs(%Z : tensor<?x?xf32>) {
    ^bb0(%in: f32, %out: f32):
      %res = arith.mulf %in, %in : f32
      linalg.yield %res : f32
    } -> (tensor<?x?xf32>)

  %1 = linalg.generic {
      indexing_maps = [#map, #map, #map],
      iterator_types = ["parallel", "parallel"]}
      ins(%0, %Y : tensor<?x?xf32>, tensor<?x?xf32>)
      outs(%Z  : tensor<?x?xf32>) {
    ^bb0(%x2: f32, %y: f32, %out: f32):
      %4 = arith.addf %x2, %y : f32
      linalg.yield %4 : f32
    } -> tensor<?x?xf32>
  return %1 : tensor<?x?xf32>
}
```

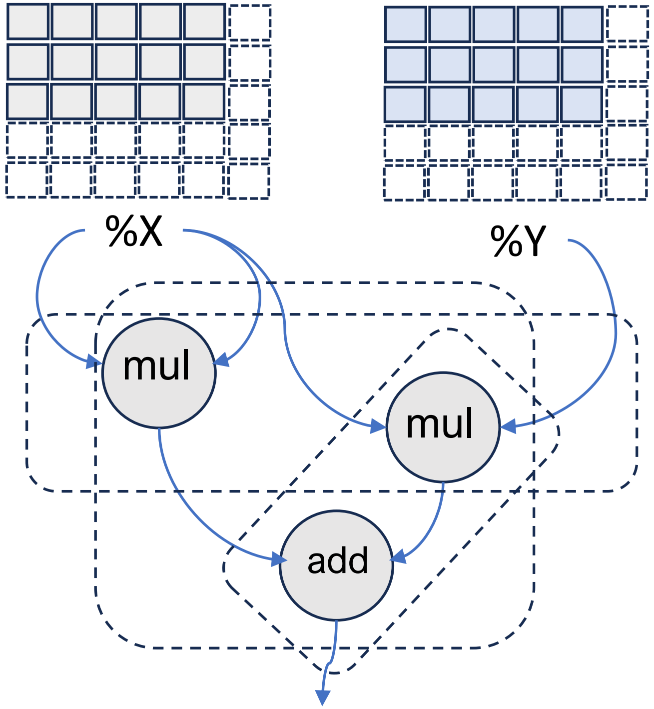
```
#map = affine_map<(d0, d1) -> (d0, d1)>
module {
  func.func @foo(%X: tensor<?x?xf32>, %Y: tensor<?x?xf32>,
                 %Z: tensor<?x?xf32>) -> tensor<?x?xf32> {
    %0 = linalg.generic
           {indexing_maps = [#map, #map, #map],
            iterator_types = ["parallel", "parallel"]}
      ins(%X, %Y : tensor<?x?xf32>, tensor<?x?xf32>)
      outs(%Z : tensor<?x?xf32>) {
    ^bb0(%in: f32, %in_0: f32, %out: f32):
      %1 = arith.mulf %in, %in : f32
      %2 = arith.addf %1, %in_0 : f32
      linalg.yield %2 : f32
    } -> tensor<?x?xf32>
    return %0 : tensor<?x?xf32>
  }
}
```

mlir-opt -test-linalg-elementwise-fusion-patterns=fuse-multiuse-producer

Evaluate: (X*X) +Y

Linalg Op-Fusion (Sibling, Producer-Consumer)

```
func.func @foo(%X: tensor<?x?xf32>, %Y: tensor<?x?xf32>,
               %Out: tensor<?x?xf32>) -> tensor<?x?xf32> {

  %xx = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
          ins(%X, %X : tensor<?x?xf32>, tensor<?x?xf32>)
          outs(%Out : tensor<?x?xf32>) -> tensor<?x?xf32>

  %xy = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
          ins(%X, %Y : tensor<?x?xf32>, tensor<?x?xf32>)
          outs(%Out : tensor<?x?xf32>) -> tensor<?x?xf32>

  %plus = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
          ins(%xx, %xy : tensor<?x?xf32>, tensor<?x?xf32>)
          outs(%Out: tensor<?x?xf32>) -> tensor<?x?xf32>

  return %plus : tensor<?x?xf32>
}
```



Evaluate: X*X + X*Y

Linalg Op-Fusion (Sibling, Producer-Consumer)

```
%xx = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
%xy = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
%plus = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
          ins(%xx, %xy : tensor<?x?xf32>, tensor<?x?xf32>)
```

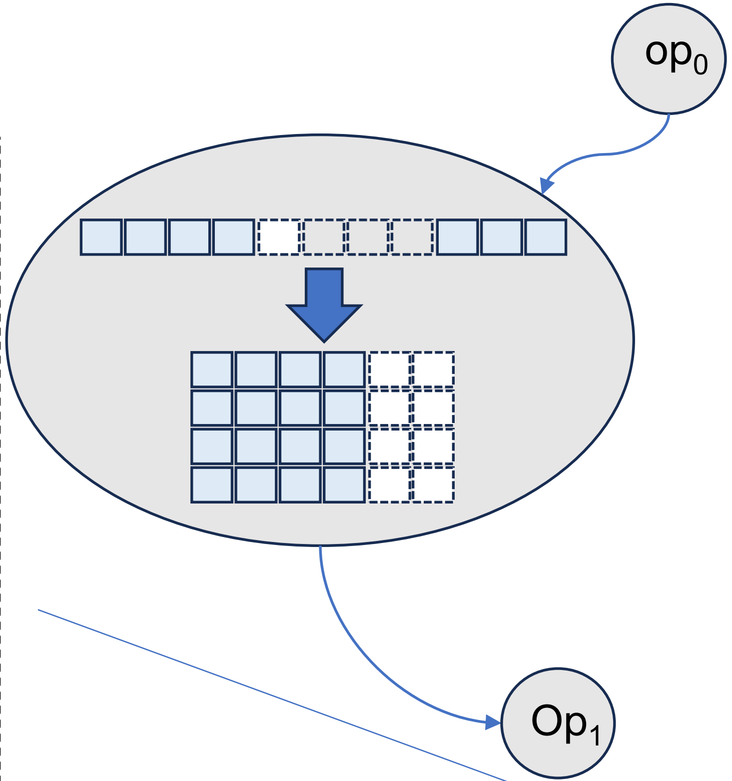
```
module attributes {transform.with_named_sequence} {
  transform.named_sequence @__transform_main(%fun: !transform.any_op {transform.readonly}) {
    %match = transform.structured.match ops{["linalg.elemwise_binary"]} in %fun
              : (!transform.any_op) -> !transform.any_op
    %xx, %xy, %plus = transform.split_handle %match : (!transform.any_op)
                        -> (!transform.op<"linalg.elemwise_binary">,
                            !transform.op<"linalg.elemwise_binary">,
                            !transform.op<"linalg.elemwise_binary">)

   transform.debug.emit_remark_at %xx, “xx op:"
      :  !transform.op<"linalg.elemwise_binary">

   %tiled_op, %loops:2 = transform.structured.tile_using_for %plus [1, 1]
        : (!transform.op<"linalg.elemwise_binary">) -> (!transform.any_op, !transform.any_op, !transform.any_op)

    %fused, %for = transform.structured.fuse_into_containing_op %xx into %loops#1
        : (!transform.op<"linalg.elemwise_binary">, !transform.any_op) -> (!transform.any_op, !transform.any_op)

    %fused2, %for2 = transform.structured.fuse_into_containing_op %xy into %for
        : (!transform.op<"linalg.elemwise_binary">, !transform.any_op) -> (!transform.any_op, !transform.any_op)
    transform.yield
  }
}
```

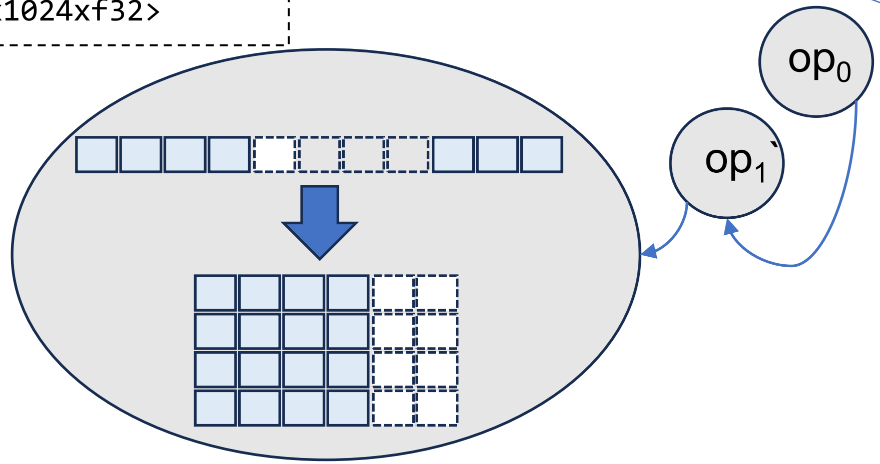
Linalg Op-Fusion (Sibling-Producer-Consumer)

```
  %0 = scf.for %arg3 = %c0 to %dim step %c1 iter_args(%arg4 = %arg2) ...{
       %1 = scf.for %arg5 = %c0 to %dim_0 step %c1 iter_args(%arg6 = %arg4) ...{
         ...
         %2 = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
                 ins(%expanded, %expanded : tensor<1x1xf32>, tensor<1x1xf32>)
                 outs(%expanded_2 : tensor<1x1xf32>) -> tensor<1x1xf32>
         …
         …
         %3 = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
                 ins(%expanded, %expanded_4 : tensor<1x1xf32>, tensor<1x1xf32>)
                 outs(%expanded_2 : tensor<1x1xf32>) -> tensor<1x1xf32>
         …
         …
         %4 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
                ins(%2, %3 : tensor<1x1xf32>, tensor<1x1xf32>)
                outs(%expanded_6 : tensor<1x1xf32>) -> tensor<1x1xf32>
         %collapsed = tensor.collapse_shape %4 [] : tensor<1x1xf32> into tensor<f32>
         …
         scf.yield %inserted_slice : tensor<?x?xf32>
       }
       scf.yield %1 : tensor<?x?xf32>
     }
...
```

Expand Dimension

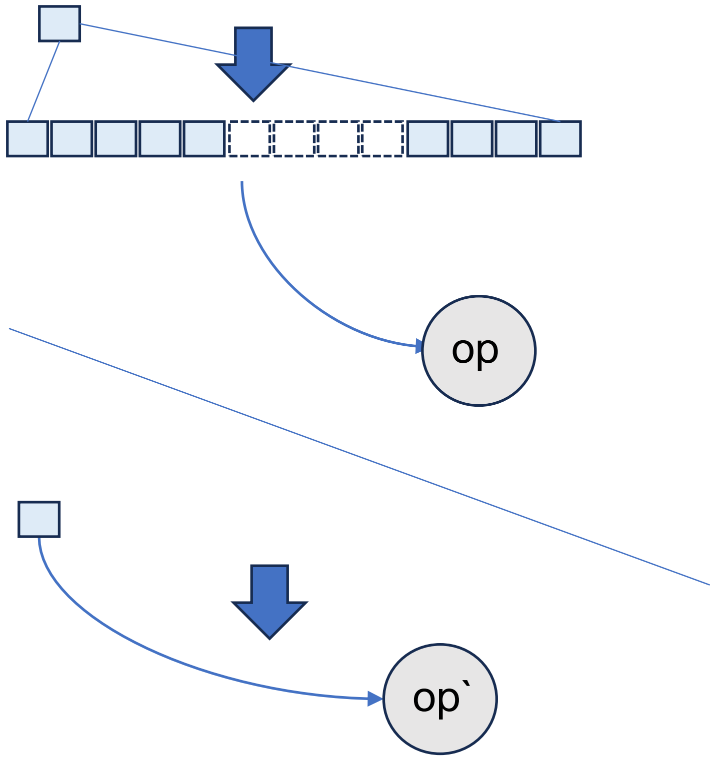
```
// Before
#map = affine_map<(d0, d1) -> (d0, d1)>
%expand_X = tensor.expand_shape %X [[0, 1]] : tensor<?xf32> into tensor<?x1024xf32>
%empty_tensor = tensor.empty [..] : tensor<?x1024xf32>
%result = linalg.generic {
     indexing_maps = [#map, #map],
     iterator_types = ["parallel" ,"parallel"]}
     ins(%expand_X : tensor<?x1024xf32>) outs(%empty_tensor : tensor<?x1024xf32>) {.. }


// After
#map = affine_map<(d0) -> (d0)>
%empty_tensor = tensor.empty [..] : tensor<?xf32>
%tmp= linalg.generic {
    indexing_maps = [#map, #map],
     iterator_types = ["parallel"]}
     ins(%X : tensor<?xf32>) outs(%empty_tensor : tensor<?xf32>) {.. }
%result = tensor.expand_shape %tmp[[0, 1]] : tensor<?xf32> into tensor<?x1024xf32>
```





Folding Fill

```
// Before
#map0 = affine_map<(d0) -> (d0)>
func.func @foldFill(%arg0: tensor<?xf16>) -> (tensor<?xf16>) {
  %c0 = arith.constant 0 : index
  %cst = arith.constant 7.0 : f32
  %0 = tensor.dim %arg0, %c0 : tensor<?xf16>
  %1 = tensor.empty(%0) : tensor<?xf16>
  %2 = linalg.fill ins(%cst : f32) outs(%1 : tensor<?xf16>) -> tensor<?xf16>
  %3 = tensor.empty(%0) : tensor<?xf16>
  %4 = linalg.generic
        {indexing_maps = [#map0, #map0, #map0], iterator_types=["parallel"]}
        ins(%arg0, %2 : tensor<?xf16>, tensor<?xf16>) outs (%3:tensor<?xf16>) {
        ^bb0(%arg1: f16, %arg2: f16, %arg3: f16):
          %5 = arith.addf  %arg1, %arg2 : f16
          linalg.yield %5 : f16
        } -> tensor<?xf16>

// After
  %cst = arith.constant 7.000000e+00 : f16
  %dim = tensor.dim %arg0, %c0 : tensor<?xf16>
  %0 = tensor.empty(%dim) : tensor<?xf16>
  %1 = linalg.generic
        {indexing_maps = [#map, #map], iterator_types = ["parallel"]}
        ins(%arg0 : tensor<?xf16>) outs(%0 : tensor<?xf16>) {
        ^bb0(%in: f16, %out: f16):
           %2 = arith.addf %in, %cst : f16
          linalg.yield %2 : f16
        } -> tensor<?xf16>>
```



Miscellaneous

```
^bb( … ):
    %idx0 = linalg.index 0 : index
    %idx1 = linalg.index 1 : index
    %4 = arith.index_cast %idx0 : index to i32
     …
```

- **Elementary** - two linalg.generic op linalg.generic_1 (producer) and linalg.generic_2 (consumer) both have one or more 'parallel' loops and linalg.generic_1 output tensor result is input to linalg.generic_2. The input and output tensors are n-D > 1.
- **Scalar + Tensor** - fusion can be performed also where there is a mix of scalars and tensor inputs to the region-body of the linalg.generic and the elementwise computation involves both scalars and tensors i.e. one of the indexing is like #map1 = affine_map<(d0, d1) -> ()> .
- **Transpose** - The linalg.generics affine map may imply transpose for some of the inputs. The fusion scheme then has to work out the new affine maps to align producer-consumer.
- **Broadcast** - linalg.generic_1 takes one or more scalars and produces n-D output tensors that form input to linalg.generic_2. We expect result fused linalg.generic to directly use the scalars.
- **Indexed Consumer:** In this scenario the consumer linalg.generic_2 yields tensors containing some function of index variables. The output of linalg.generic_1 is then used just for dimension information and so linalg.generic_1 could be totally removed after fusion and the original inputs of linalg.generic_1 are passed directly to linalg.generic_2.
- **Indexed Producer**: Similar to scenario above but in this case the producer yields tensor elements which are function of index variables. After fusion the indexing computation of producer is absorbed into consumer. The tensor contents of ins of linalg.generic_1 is still passed as arg to fused linalg.generic but as one can guess it does not have a 'use' in region-body of fused but only needed for perhaps dim calculation.
- **Fold Constant** - In this scenario there is just one linalg.generic but one of its ins is a constant tensor DenseElementsAttr. After fusion the constant tensor is demoted to scalar constant ins to fused linalg.generic.
- **Fold Fill** - In this case a 'linalg.fill' creates a tensor of constant and the created tensor is one of the args to linalg.generic. This is quite a common case. The fusion can then use just the scalar instead of 'filled tensor'.

Fusion in a Pass

```
struct  MyPlayCompilerFusionPass : public .. {
  auto funcOp = getOperation();
  auto context = &getContext();
  RewritePatternSet fusionPatterns(context);

  linalg::ControlFusionFn fuseElementwiseOpsControlFn =
        [&](OpOperand *fusedOperand) {
          Operation *producer = fusedOperand->get().getDefiningOp();
          Operation *consumer = fusedOperand->getOwner();
        // decide
        return shouldIBotherFusing(…);
  }

  linalg::populateElementwiseOpsFusionPatterns(fusionPatterns,
                                                fuseElementwiseOpsControlFn);

  linalg::ControlFusionFn fuseByExpansionControlFn =
      [](OpOperand *fusedOperand) {
        Operation *producer = fusedOperand->get().getDefiningOp();
        return producer->hasOneUse();
      };
  linalg::populateFoldReshapeOpsByExpansionPatterns(..);
  ..
  linalg::populateConstantFoldLinalgOperations(..);
 ..
 ..applyPatternsAndFoldGreedily(funcOp, std::move(fusionPatterns), ..);
}
```

Fusion Algorithms

- Extensive literature – loop-fusion, polyhedral analysis, kernel fusion.
- Kernel fusion
  - improve temporal locality (reduce communication with global memory)
  - increase opt. opportunity (CSE, CP)
  - Reduce local buffer
- Kennedy and McKinley, “maximizing data locality by loop fusion is NP-hard”.
- Pairwise greedy fusion, expanding fusion scope while maintaining profitability
- Greedy algorithm – fusing along the heaviest edge – cost function
- Disjoint Fusion Partition Groups; Fusible kernel list e.g. (p,q) ^ (q,r) -> {p,q,r}
- Stoer-Wagner mi-cut algorithm
- Multi-user and re-computation trade-offs (external dependence to fusible list)

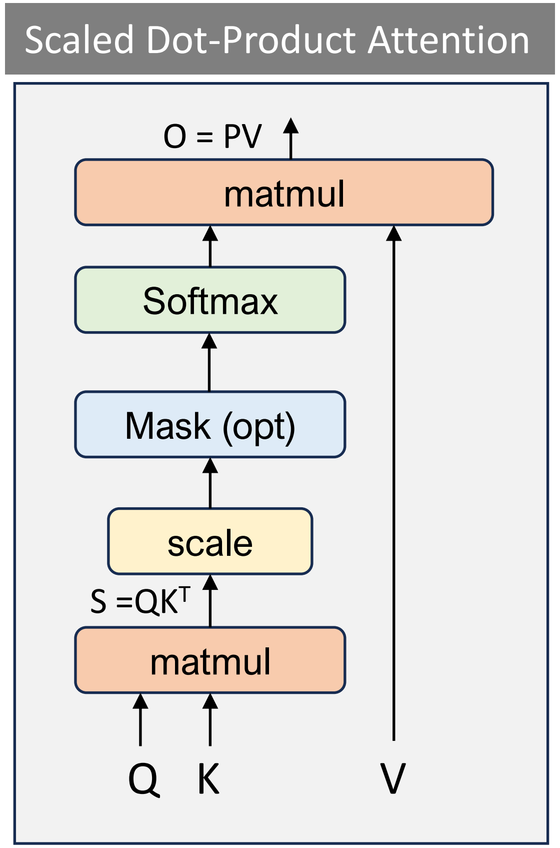
Fusion in DNN

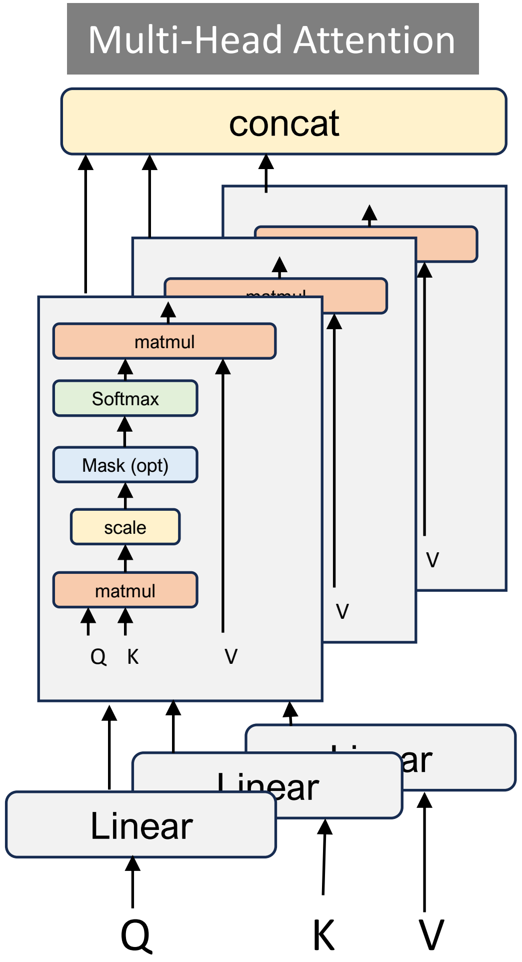
"**Attention Is All You Need**" is a landmark[1][2] 2017 research paper by Google.[3] Authored by eight scientists, … is considered by some to be a founding document for modern artificial intelligence, as transformers became the main architecture of large language models"

$$Q, K, V \in \mathbb{R}^{N\times d}$$

$$S = QK^T \in \mathbb{R}^{N\times N};\ P = softmax(S) \in \mathbb{R}^{N\times N};$$
$$O = PV \in \mathbb{R}^{N\times d}$$

$$softmax(x) = \frac{e^{x_i - \max(x_i)}}{\sum e^{x_i - \max(x_i)}}$$



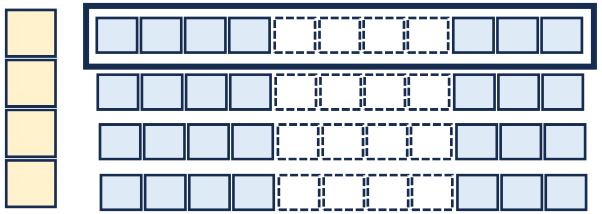


$$S = QK^T \in \mathbb{R}^{N\times N}; P = softmax(S); O = PV$$

ML Example - Attention

$$softmax(x) = \frac{e^{x_i - \max(x_i)}}{\sum e^{x_i - \max(x_i)}}$$

```
func.func @attention(%Q : memref<?x?xf32>, %K: memref<?x?xf32>, // Nxd
                     %V: memref<?x?xf32>, %out: memref<?x?xf32>) {
  ...
    %k_transpose = linalg.transpose ...

    %QKT = linalg.matmul
                ins(%q, %k_transpose : tensor<?x?xf32>, tensor<?x?xf32>)
                outs(%empty_NxN : tensor<?x?xf32>) -> tensor<?x?xf32>

     %t_minf = linalg.fill ins(%cst_minus_inf : f32) outs(%empty_N : tensor<?xf32>) -> tensor<?xf32>
     %max = linalg.reduce ins(%QKT : tensor<?x?xf32>) ...  %m = arith.maximumf %in, %init : f32

     %maxb = linalg.broadcast ins(%max: tensor<?xf32>) outs(%empty_NxN : tensor<?x?xf32>) dimensions = [1]
     %sub = linalg.elemwise_binary {fun = #linalg.binary_fn<sub>} ...

     %exp = linalg.elemwise_unary {fun = #linalg.unary_fn<exp>} ...

     %t_zeros = linalg.fill ins(%c0f : f32) outs(%empty_N : tensor<?xf32>) -> tensor<?xf32>
     %sum = linalg.reduce ...        %s = arith.addf %in, %init : f32 ...

     %sums = linalg.broadcast ...
     %p = linalg.elemwise_binary {fun = #linalg.binary_fn<div>}
           ins(%exp, %sums : tensor<?x?xf32>, tensor<?x?xf32>)...

     %o = linalg.matmul
              ins(%p, %v : tensor<?x?xf32>, tensor<?x?xf32>)...
```



$$S = QK^T \in \mathbb{R}^{N\times N}; P = softmax(S); O = PV$$

ML Example - Attention

$$softmax(x) = \frac{e^{x_i - \max(x_i)}}{\sum e^{x_i - \max(x_i)}}$$

```
 func.func @attention(%Q : memref<?x?xf32>, %K: memref<?x?xf32>, // Nxd
                     %V: memref<?x?xf32>, %out: memref<?x?xf32>) {
    ...
    %k_transpose = linalg.transpose ins(%k : tensor<?x?xf32>) outs(%empty_dxN : tensor<?x?xf32>)
permutation = [1, 0]
    %QKT = linalg.matmul  ins(%q, %k_transpose : tensor<?x?xf32>, tensor<?x?xf32>) ..

    ..
```

```
mlir-opt attention.linalg -linalg-generalize-named-ops -linalg-fuse-elementwise-ops -one-shot-bufferize -convert-linalg-to-loops
```

TRANSPOSE FOLDED

```
..
   %dim_4 = memref.dim %arg0, %c0 : memref<?x?xf32>
   %dim_5 = memref.dim %arg0, %c1 : memref<?x?xf32>
   %dim_6 = memref.dim %arg1, %c0 : memref<?x?xf32>
   scf.for %arg4 = %c0 to %dim_4 step %c1 {
     scf.for %arg5 = %c0 to %dim_6 step %c1 {
       scf.for %arg6 = %c0 to %dim_5 step %c1 {
         %0 = memref.load %arg0[%I, %K] : memref<?x?xf32>
         %1 = memref.load %arg1[%J, %K] : memref<?x?xf32>
         %2 = memref.load %alloc[%I, %J] : memref<?x?xf32>
         %3 = arith.mulf %0, %1 : f32
         %4 = arith.addf %2, %3 : f32
         memref.store %4, %alloc[%I, %J] : memref<?x?xf32>
       }
     }
   }
```

$$S = QK^T \in \mathbb{R}^{N\times N}; P = softmax(S); O = PV$$

ML Examples - Attention

$$softmax(x) = \frac{e^{x_i - \max(x_i)}}{\sum e^{x_i - \max(x_i)}}$$

```
...
%max = linalg.reduce ins(%QKT : tensor<?x?xf32>) outs(%t_minf : tensor<?xf32>) dimensions = [1]
  (%in: f32, %init: f32) {
    %m = arith.maximumf %in, %init : f32
    linalg.yield %m : f32
  }

%maxb = linalg.broadcast ins(%max: tensor<?xf32>) outs(%empty_NxN : tensor<?x?xf32>) dimensions = [1]

%sub = linalg.elemwise_binary {fun = #linalg.binary_fn<sub>}
      ins(%QKT, %maxb : tensor<?x?xf32>, tensor<?x?xf32>)
      outs(%empty_NxN: tensor<?x?xf32>) -> tensor<?x?xf32>
```

BROADCAST FOLDED

```
%8 = linalg.generic ...  %13 = arith.maximumf %in, %out : f32 ...

%9 = linalg.generic
       {indexing_maps = [#map4, #map5, #map4], iterator_types = ["parallel", "parallel"]}
       ins(%6, %8 : tensor<?x?xf32>, tensor<?xf32>)
       outs(%3 : tensor<?x?xf32>) {
   ^bb0(%in: f32, %in_2: f32, %out: f32):
     %13 = arith.subf %in, %in_2 : f32
     %14 = math.exp %13 : f32
     linalg.yield %14 : f32
   } -> tensor<?x?xf32>
```

$$S = QK^T \in \mathbb{R}^{N \times N}; P = softmax(S); O = PV$$

ML Examples - Attention

$$softmax(x) = \frac{e^{x_i - \max(x_i)}}{\sum e^{x_i - \max(x_i)}}$$

```
func.func @attention(%Q : memref<?x?xf32>, %K: memref<?x?xf32>, // Nxd
                     %V: memref<?x?xf32>, %out: memref<?x?xf32>) {
 ...
   %k_transpose = linalg.transpose ...

   %QKT = linalg.matmul
                ins(%q, %k_transpose : tensor<?x?xf32>, tensor<?x?xf32>)
                outs(%empty_NxN : tensor<?x?xf32>) -> tensor<?x?xf32>

    %t_minf = linalg.fill ins(%cst_minus_inf : f32) outs(%empty_N : tensor<?xf32>) -> tensor<?xf32>
    %max = linalg.reduce ins(%QKT : tensor<?x?xf32>) ...  %m = arith.maximumf %in, %init : f32

    %maxb = linalg.broadcast ins(%max: tensor<?xf32>) outs(%empty_NxN : tensor<?x?xf32>) dimensions = [1]
    %sub = linalg.elemwise_binary {fun = #linalg.binary_fn<sub>} ...

    %exp = linalg.elemwise_unary {fun = #linalg.unary_fn<exp>} ...

    %t_zeros = linalg.fill ins(%c0f : f32) outs(%empty_N : tensor<?xf32>) -> tensor<?xf32>
    %sum = linalg.reduce ...        %s = arith.addf %in, %init : f32 ...

    %sums = linalg.broadcast ...
    %p = linalg.elemwise_binary {fun = #linalg.binary_fn<div>}
          ins(%exp, %sums : tensor<?x?xf32>, tensor<?x?xf32>)...

    %o = linalg.matmul
             ins(%p, %v : tensor<?x?xf32>, tensor<?x?xf32>)...
```

$$S = QK^T \in \mathbb{R}^{N \times N}; P = softmax(S); O = PV$$

ML Examples - Attention

$$softmax(x) = \frac{e^{x_i - \max(x_i)}}{\sum e^{x_i - \max(x_i)}}$$

```
%38 = linalg.generic
        {indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>, affine_map<(d0, d1) -> (d0)>, affine_map<(d0, d1) -> (d0)>],
        iterator_types = ["parallel", "reduction"]}
        ins(%extracted_slice_2, %35 : tensor<?x?xf32>, tensor<?xf32>) outs(%37 : tensor<?xf32>)
        attrs =  {.. = [[32, 0], [8, 0], [0, 1], [0, 0]]>} {
    ^bb0(%in: f32, %in_5: f32, %out: f32):
FUSED  %40 = arith.subf %in, %in_5 : f32
       %41 = math.exp %40 : f32
       %42 = arith.addf %41, %out : f32
       linalg.yield %42 : f32
    } -> tensor<?xf32>
    %extracted_slice_4 = tensor.extract_slice %arg2[%arg1, 0] [%29, %20] [1, 1] : tensor<?x?xf32> to tensor<?x?xf32>
%39 = linalg.generic
         {indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>,
                           affine_map<(d0, d1) -> (d0)>, affine_map<(d0, d1) -> (d0)>,
                            affine_map<(d0, d1) -> (d0, d1)>],
         iterator_types = ["parallel", "parallel"]}
         ins(%extracted_slice, %32, %38 : tensor<?x?xf32>, tensor<?xf32>, tensor<?xf32>)
         outs(%extracted_slice_4 : tensor<?x?xf32>)
         attrs =  {.. = [[32, 0], [8, 0], [0, 0], [0, 32]]>} {
    ^bb0(%in: f32, %in_5: f32, %in_6: f32, %out: f32):
FUSED  %40 = arith.subf %in, %in_5 : f32
       %41 = math.exp %40 : f32
       %42 = arith.divf %41, %in_6 : f32
       linalg.yield %42 : f32
    } -> tensor<?x?xf32>
```

Is this sufficient?

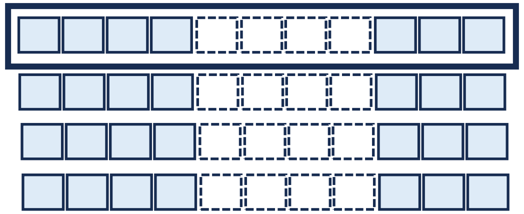
What do we want ?

More Patterns

When do we want it ?

NOW!

Std Attention – Fusion/Tiling



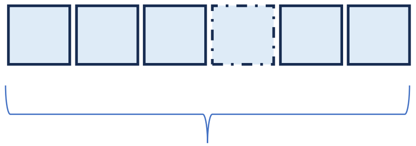
```
func.func @attention(%Q : memref<?x?xf32>, %K: memref<?x?xf32>, // Nxd
                     %V: memref<?x?xf32>, %out: memref<?x?xf32>) {
 ...
   %k_transpose = linalg.transpose ...

   %QKT = linalg.matmul
                 ins(%q, %k_transpose : tensor<?x?xf32>, tensor<?x?xf32>)
                 outs(%empty_NxN : tensor<?x?xf32>) -> tensor<?x?xf32>

    %t_minf = linalg.fill ins(%cst_minus_inf : f32) outs(%empty_N : tensor<?xf32>) -> tensor<?xf32>
    %max = linalg.reduce ins(%QKT : tensor<?x?xf32>) ...  %m = arith.maximumf %in, %init : f32
    …
    %maxb = linalg.broadcast ins(%max: tensor<?xf32>) outs(%empty_NxN : tensor<?x?xf32>) dimensions = [1]
    %sub = linalg.elemwise_binary {fun = #linalg.binary_fn<sub>} ...
    …
    %exp = linalg.elemwise_unary {fun = #linalg.unary_fn<exp>} ...
    …
    %t_zeros = linalg.fill ins(%c0f : f32) outs(%empty_N : tensor<?xf32>) -> tensor<?xf32>
    %sum = linalg.reduce ...        %s = arith.addf %in, %init : f32 ...

    %sums = linalg.broadcast ...
    %p = linalg.elemwise_binary {fun = #linalg.binary_fn<div>}
          ins(%exp, %sums : tensor<?x?xf32>, tensor<?x?xf32>)...

    %o = linalg.matmul
             ins(%p, %v : tensor<?x?xf32>, tensor<?x?xf32>)...
```

Obstacle 1

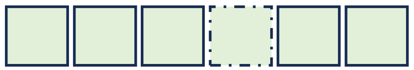
Obstacle 2

Flash Attention

$$l(x) = \sum_{i=1}^{N} e^{x_i - \max(x)}$$



$$l(x^a) = \sum_{i \in a} e^{x_i - \max(x^a)}$$



$$l(x^b) = \sum_{i \in b} e^{x_i - \max(x^b)}$$

$$\max(x) = \max(xa, xb)$$

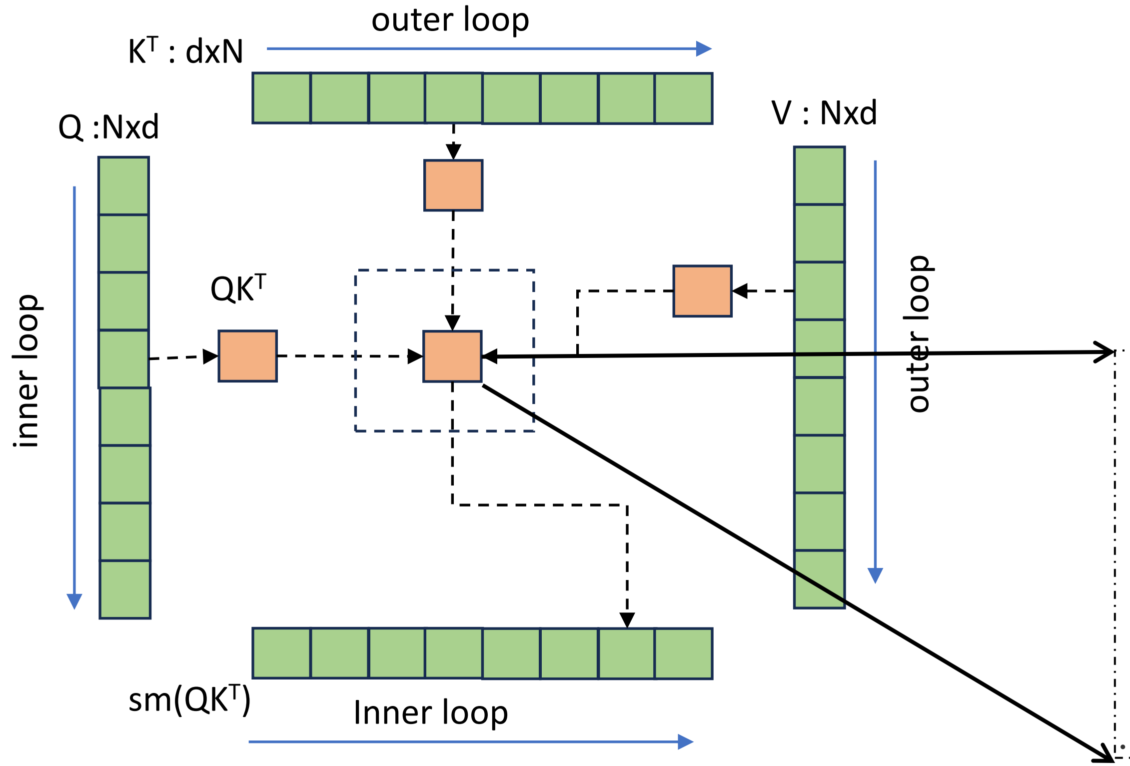
$$l(x) = l([x^a \quad x^b]) = e^{\max(x^a) - \max(x)} \left(l(x^a) = \sum_{i \in a} e^{x_i - \max(x^a)} \right) + e^{\max(x^b) - \max(x)} \left(l(x^b) = \sum_{i \in b} e^{x_i - \max(x^b)} \right)$$

$$l(x) = l([x^a \quad x^b]) = e^{\cancel{\max(x^a)} - \max(x)} \left(l(x^a) = \sum_{i \in a} e^{x_i - \cancel{\max(x^a)}} \right) + e^{\cancel{\max(x^b)} - \max(x)} \left(l(x^b) = \sum_{i \in b} e^{x_i - \cancel{\max(x^b)}} \right)$$

Flash Attention

$$S = QK^T \in \mathbb{R}^{N \times N};\ P = softmax(S) \in \mathbb{R}^{N \times N};$$
$$O = PV \in \mathbb{R}^{N \times d}$$

$$softmax(x) = \frac{e^{x_i - \max(x_i)}}{\sum e^{x_i - \max(x_i)}}$$




```
  ..= linalg.generic
         {indexing_maps = ..
         iterator_types = ["parallel", "parallel"]}
         ins..%expanded_9, %expanded>) {
   ^bb0(%in: f32, %in_14: ..):
     %58 = arith.subf %in_15, %in_16 : f32
     %59 = math.exp2 %58 : f32
     %60 = arith.mulf %in_14, %59 : f32
     %61 = arith.addf %in, %60 : f32
     linalg.yield %61 : f32
 } -> tensor<..xf32>
..
```

```
//carry and update statistics
for (auto k = 0; k < Br; ++k)
  mi_new[k] = std::max(mi[k], mij[k]);


for (auto k = 0; k < Br; ++k)
   li_new[k] = (std::exp(mi[k] - mi_new[k]) * li[k])   +   std::exp(mij[k] - mi_new[k])*lij[k];
```

Conclusion

- Linalg – a useful dialect for ML graph.
- Fusion in Linalg.
- Rewrite patterns and applications of patterns.
- In practice, algebraic/algorithmic insight useful.

Thank you

Follow us on:

For more information, visit us at:

qualcomm.com & qualcomm.com/blog