Convolution in IREE

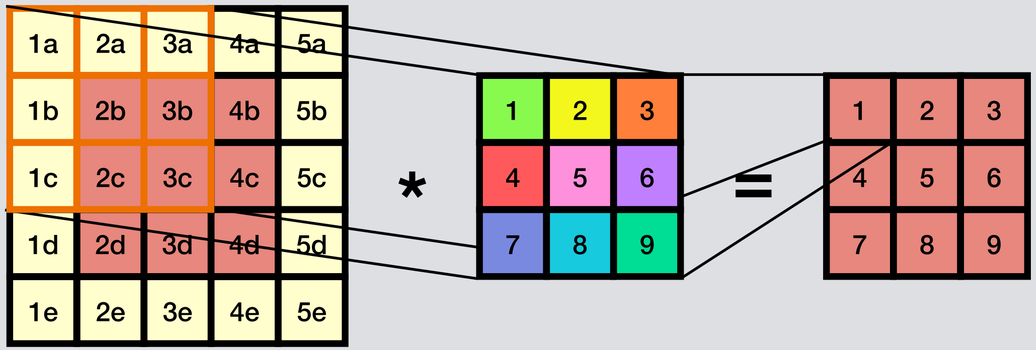
Quinn Dawkins (nod.ai)

Overview

- General Description of Convolution
- Img2Col and GEMM
- Implicit GEMM Code Generation
- Selecting a Data Layout
- Propagating Layout Transformations on Linalg ops in MLIR
- Conclusions & Future Work

Convolution

- Convolutions are a core operation in many ML/AI workloads, in particular when processing visual data
- A convolution applies a sliding **filter** to an **input** “image” to produce an **output** tensor
- There are a few different strategies for computing convolution
  - Direct Convolution
  - Winograd Convolution
  - FFT (Fast Fourier Transform) Convolution
  - **Implicit GEMM (Generalized Matrix Multiply)**
- cuDNN and CUTLASS (among others) implement Implicit GEMM to target GEMM based accelerators



General Description of Convolution

Dimension Types

- A general view of the loop-nested computation of a convolution allows us to classify each loop similar to a GEMM
  - **Batch** - parallel loop over shared dimension of the **input** and **output**
  - **Output Image** - parallel loop over a dimension of the **output** image shape
  - **Input Channel** - reduction over shared dimension of the **input** and **filter**
  - **Output Channel** - parallel loop over shared dimension of the **filter** and **output**
  - **Filter Loop** - reduction over one dimension of the sliding **filter**
  - **Depth*** - parallel loop over shared dimension of **input**, **filter**, and **output**
- **Stride** indicates the stride of the sliding filter along the **input image** dimensions
- **Dilation** (stride of the filter elements) and **Depth** are not covered here

General Description of Convolution

Dimension Types

- A general view of the loop-nested computation of a convolution allows us to classify each loop similar to a GEMM
  - Batch size **B = 2**
  - Output image sizes **OH** and **OW = 128**
  - Input channel size **IC = 64**
  - Output channel size **OC = 32**
  - Filter loop sizes **FH** and **FW = 3**
  - Input image sizes **IH** and **IW = 130**

```
linalg.conv_2d_nchw_fchw
    {dilations = [DH, DW]: tensor<2xi64>, strides = [SH, SW]: tensor<2xi64>}
    ins(%input, %filter : tensor<BxICxIHxIWxf32>, tensor<OCxICxFHxFWxf32>)
    outs(%output : tensor<BxOCxOHxOWxf32>)
```

```
linalg.conv_2d_nchw_fchw
    {dilations = dense<1>: tensor<2xi64>, strides = dense<1>: tensor<2xi64>}
    ins(%input, %filter : tensor<2x64x130x130xf32>, tensor<32x64x3x3xf32>)
    outs(%output : tensor<2x32x128x128xf32>)
```

- Strides **SH** and **SW = 1**
- nchw = batch x channel x image height x image width
- fchw = output channel x input channel x filter height x filter width

General Description of Convolution

Dimension Types

- A general view of the loop-nested computation of a convolution allows us to classify each loop similar to a GEMM
  - Batch size **B = 2**
  - Output image sizes **OH** and **OW = 128**
  - Input channel size **IC = 64**
  - Output channel size **OC = 32**
  - Filter loop sizes **FH** and **FW = 3**
  - Input image sizes **IH** and **IW = 130**
- Strides **SH** and **SW = 1**

```
#map = affine_map<(d0, d1, d2, d3, d4, d5, d6) -> (d0, d4, d2 + d5, d3 + d6)>
#map1 = affine_map<(d0, d1, d2, d3, d4, d5, d6) -> (d1, d4, d5, d6)>
#map2 = affine_map<(d0, d1, d2, d3, d4, d5, d6) -> (d0, d1, d2, d3)>
module {
  func.func @conv_2d_nchw_fchw(%in: tensor<2x64x130x130xf32>, %filter:
tensor<32x64x3x3xf32>, %out: tensor<2x32x128x128xf32>) -> tensor<2x32x128x128xf32> {
    %0 = linalg.generic {indexing_maps = [#map, #map1, #map2], iterator_types =
["parallel", "parallel", "parallel", "parallel", "reduction", "reduction",
"reduction"]} ins(%in, %filter : tensor<2x64x130x130xf32>, tensor<32x64x3x3xf32>)
outs(%out : tensor<2x32x128x128xf32>) {
    ^bb0(%in: f32, %in_0: f32, %out: f32):
      %1 = arith.mulf %in, %in_0 : f32
      %2 = arith.addf %out, %1 : f32
      linalg.yield %2 : f32
    } -> tensor<2x32x128x128xf32>
    return %0 : tensor<2x32x128x128xf32>
  }
}
```

- nchw = batch x channel x image height x image width
- fchw = output channel x input channel x filter height x filter width

General Description of Convolution

Loop representation

- A general view of the loop-nested computation of a convolution allows us to classify each loop similar to a GEMM
  - Batch size **B = 2**
  - Output image sizes **OH** and **OW = 128**
  - Input channel size **IC = 64**
  - Output channel size **OC = 32**
  - Filter loop sizes **FH** and **FW = 3**
  - Input image sizes **IH** and **IW = 130**
- Strides **SH** and **SW = 1**

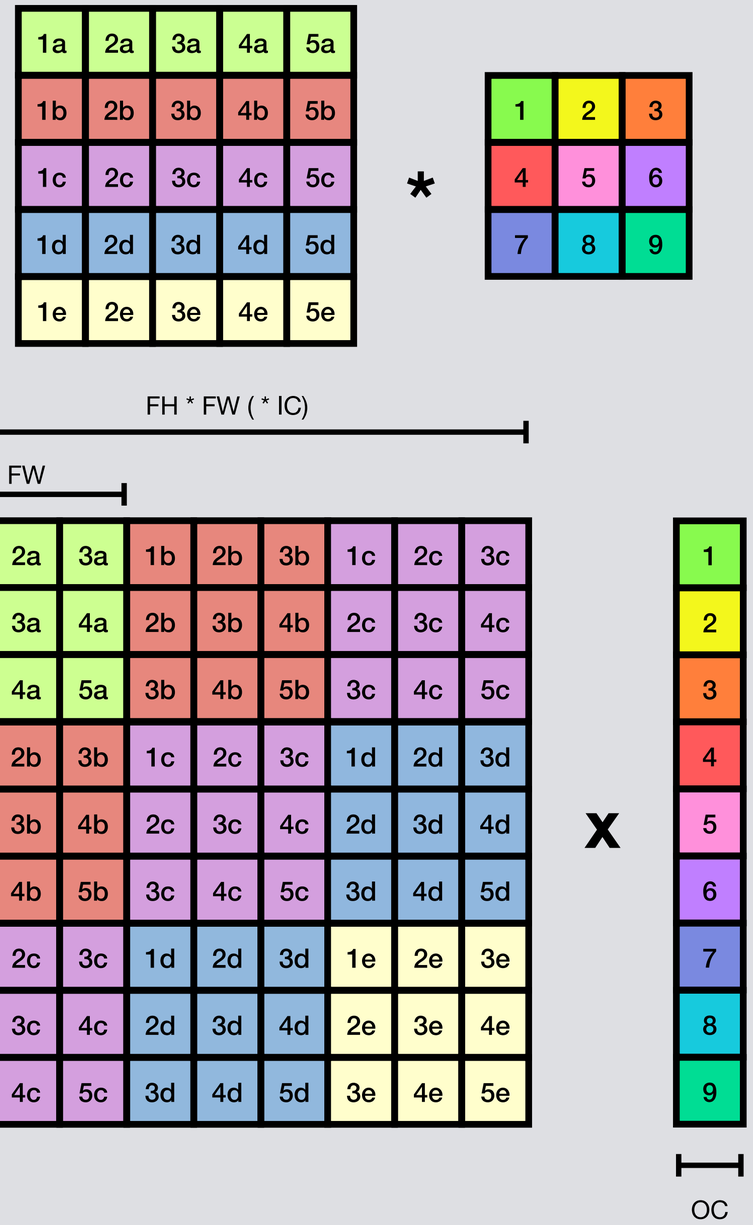
```
#map = affine_map<(d0, d1) -> (d0 + d1)>
module {
  func.func @conv_2d_nchw_fchw(%arg0: tensor<2x64x130x130xf32>, %arg1: tensor<32x64x3x3xf32>, %arg2:
tensor<2x32x128x128xf32>) -> tensor<2x32x128x128xf32> {
    %0 = bufferization.to_memref %arg1 : memref<32x64x3x3xf32>
    %1 = bufferization.to_memref %arg0 : memref<2x64x130x130xf32>
    %2 = bufferization.to_memref %arg2 : memref<2x32x128x128xf32>
    %alloc = memref.alloc() {alignment = 64 : i64} : memref<2x32x128x128xf32>
    memref.copy %2, %alloc : memref<2x32x128x128xf32> to memref<2x32x128x128xf32>
    scf.for %arg3 = %c0 to %c2 step %c1 {                  # Batch
      scf.for %arg4 = %c0 to %c32 step %c1 {               # Output Channel
        scf.for %arg5 = %c0 to %c128 step %c1 {            # Output Height
          scf.for %arg6 = %c0 to %c128 step %c1 {          # Output Width
            scf.for %arg7 = %c0 to %c64 step %c1 {         # Input Channel
              scf.for %arg8 = %c0 to %c3 step %c1 {        # Filter Height
                scf.for %arg9 = %c0 to %c3 step %c1 {      # Filter Width
                  %4 = affine.apply #map(%arg5, %arg8)
                  %5 = affine.apply #map(%arg6, %arg9)
                  %6 = memref.load %1[%arg3, %arg7, %4, %5] : memref<2x64x130x130xf32>
                  %7 = memref.load %0[%arg4, %arg7, %arg8, %arg9] : memref<32x64x3x3xf32>
                  %8 = memref.load %alloc[%arg3, %arg4, %arg5, %arg6] : memref<2x32x128x128xf32>
                  %9 = arith.mulf %6, %7 : f32
                  %10 = arith.addf %8, %9 : f32

                  // Depends only on B/OC/OH/OW -> IC/FH/FC are the contracting dimensions
                  memref.store %10, %alloc[%arg3, %arg4, %arg5, %arg6] : memref<2x32x128x128xf32>
                }
              }
            }
          }
        }
      }
    }
    %3 = bufferization.to_tensor %alloc : memref<2x32x128x128xf32>
    return %3 : tensor<2x32x128x128xf32>
  }
}
```

"Contracting" dimensions

Convolved indices

Img2Col and GEMM

- Convolution doesn't map directly to GEMM based accelerators such as NVidia tensor cores
  - The sliding window of the input is contracted with the filter rather than a full row/column
- Img2Col is a transformation to expand the input to a column matrix based on the shape of the sliding filter, mapping the convolution to a GEMM
- Maps to a GEMM with shape (NCHW convolution)
  - M = OC, N = OH * OW, K = IC * FH * FW
- NHWC convolution
  - M = OH * OW, N = OC, K = FH * FW * IC



Img2Col and GEMM

- Img2Col can be thought of as a gather on the input into a collapsed shape
- Input tensor with extents: B x IC x IH x IW
- Gather: B x [IC * FH * FW] x [OH * OW]
- **input shape**: tensor<2x64x130x130>
- **gather shape**: tensor<2x64x3x3x128x128>
- **collapsed shape**: tensor<2x576x16384>
- Equivalent to unrolling the collapsed iterators (IC * FH * FW)/(OH * OW) and then applying the indexing map for the original convolution to the unrolled indices

```
#map = affine_map<(d0, d1, d2) -> (d0, d1, d2)>
#map1 = affine_map<(d0) -> (d0 floordiv 9)>
#map2 = affine_map<(d0) -> (d0 mod 9)>
#map3 = affine_map<(d0) -> ((d0 mod 9) floordiv 3)>
#map4 = affine_map<(d0) -> (d0 mod 3)>
#map5 = affine_map<(d0) -> (d0 floordiv 128)>
#map6 = affine_map<(d0) -> (d0 mod 128)>
#map7 = affine_map<(d0, d1) -> (d0 floordiv 128 + (d1 mod 9) floordiv 3)>
#map8 = affine_map<(d0, d1) -> (d0 + d1 - (d0 floordiv 128) * 128 - (d1 floordiv 3) * 3)>
func.func @conv_2d_nchw_fchw() {
  %empty = tensor.empty() : tensor<2x576x16384xf32>
  %img2col_tensor = linalg.generic {indexing_maps = [#map], iterator_types = ["parallel", "parallel",
"parallel"]} outs(%empty : tensor<2x576x16384xf32>) {
  ^bb0(%out: f32):
    %10 = linalg.index 0 : index
    %11 = linalg.index 1 : index
    %12 = linalg.index 2 : index
    %c64 = arith.constant 64 : index
    %c3 = arith.constant 3 : index
    %c3_1 = arith.constant 3 : index
    %c9 = arith.constant 9 : index
    %13 = affine.apply #map1(%11)
    %14 = affine.apply #map2(%11)
    %15 = affine.apply #map3(%11)
    %16 = affine.apply #map4(%11)
    %c128 = arith.constant 128 : index
    %c128_2 = arith.constant 128 : index
    %17 = affine.apply #map5(%12)
    %18 = affine.apply #map6(%12)
    %19 = affine.apply #map7(%12, %11)
    %20 = affine.apply #map8(%12, %11)
    %extracted = tensor.extract %input[%10, %13, %19, %20] : tensor<2x64x130x130xf32>
    linalg.yield %extracted : f32
  } -> tensor<2x576x16384xf32>
}
```

Unroll **576** to **IC, FH, FW**

Unroll **16384** to **OH, OW**

Convolve **OH|OW * SH|SW + FH|FW**

Img2Col and GEMM

- After Img2Col, the convolution becomes a GEMM with a fused leading gather on the input
- Additionally filter and outputs are collapsed to match the shape of the column tensor
  - **Filter**: OC x (IC * FH * FW)
  - **Output**: B x OC x (OH * OW)

```
#map9 = affine_map<(d0, d1, d2, d3) -> (d1, d3)>
#map10 = affine_map<(d0, d1, d2, d3) -> (d0, d3, d2)>
#map11 = affine_map<(d0, d1, d2, d3) -> (d0, d1, d2)>
func.func @conv_2d_nchw_fchw() {
  %5 = tensor.empty() : tensor<2x32x128x128xf32>
  %6 = linalg.fill ins(%cst : f32) outs(%5 : tensor<2x32x128x128xf32>) -> tensor<2x32x128x128xf32>
  %collapsed = tensor.collapse_shape %4 [[0], [1, 2, 3]] : tensor<32x64x3x3xf32> into tensor<32x576xf32>
  %collapsed_0 = tensor.collapse_shape %6 [[0], [1], [2, 3]] : tensor<2x32x128x128xf32> into tensor<2x32x16384xf32>
  %8 = /*img2col*/ -> tensor<2x576x16384xf32>
  %9 = linalg.generic {indexing_maps = [#map9, #map10, #map11], iterator_types = ["parallel", "parallel", "parallel",
"reduction"]} ins(%collapsed, %8 : tensor<32x576xf32>, tensor<2x576x16384xf32>) outs(%collapsed_0 :
tensor<2x32x16384xf32>) {
  ^bb0(%in: f32, %in_1: f32, %out: f32):
    %10 = arith.mulf %in, %in_1 : f32
    %11 = arith.addf %10, %out : f32
    linalg.yield %11 : f32
  } -> tensor<2x32x16384xf32>
  %expanded = tensor.expand_shape %9 [[0], [1], [2, 3]] : tensor<2x32x16384xf32> into tensor<2x32x128x128xf32>
  return %expanded : tensor<2x32x128x128xf32>
}
```

GEMM indexing maps

Note that the filter (LHS for NCHW) does not share the batch dimension (**d0**)

GEMM with “broadcast” batch size = 2

M = OC = 32

N = OH * OW = 16384

K = IC * FH * FW = 576

Implicit GEMM

- After Img2Col, implicit GEMM codegen follows matmul codegen
- New/specialized code generation strategies in IREE through the transform dialect
  - Specifies a script describing the sequence of transformations to apply
  - Similar to a fixed pass pipeline but can be constructed on the fly

Transform script is a region containing transform operations

```
module {
  transform.sequence  failures(propagate) {
  ^bb0(%arg0: !pdl.operation):
    // Step 1. Convert to img2col
    transform.iree.register_match_callbacks
    %0:4 = transform.iree.match_callback failures(propagate) "convolution"(%arg0) : (!pdl.operation) -> (!
pdl.operation, !pdl.operation, !pdl.operation, !pdl.operation)
    %img2col_tensor, %transformed = transform.structured.convert_conv2d_to_img2col %0#2 : (!pdl.operation) -> (!
pdl.operation, !pdl.operation)
    %1 = get_producer_of_operand %transformed[0] : (!pdl.operation) -> !pdl.operation

    // Bubble collapse_shape ops to boundaries of kernel
    %2 = transform.structured.match ops{["func.func"]} in %arg0 : (!pdl.operation) -> !pdl.operation
    transform.iree.apply_patterns %2 {bubble_collapse} : (!pdl.operation) -> ()
    %first, %rest = transform.iree.take_first %0#3, %1 : (!pdl.operation, !pdl.operation) -> (!pdl.operation, !
pdl.operation)
    %forall_op, %tiled_op = transform.structured.tile_to_forall_op %first   num_threads [] tile_sizes [1, 32, 32]
(mapping = [#gpu.block<z>, #gpu.block<y>, #gpu.block<x>])
    transform.iree.apply_patterns %arg0 {canonicalization, cse, licm, tiling_canonicalization} : (!pdl.operation) ->
()
    %3 = transform.structured.fuse_into_containing_op %rest into %forall_op
    transform.iree.populate_workgroup_count_region_using_num_threads_slice %forall_op : (!pdl.operation) -> ()
    %4 = transform.structured.match ops{["linalg.fill"]} in %arg0 : (!pdl.operation) -> !pdl.operation
    %5 = transform.structured.fuse_into_containing_op %4 into %forall_op
    %6 = transform.structured.fuse_into_containing_op %img2col_tensor into %forall_op
    transform.iree.apply_patterns %arg0 {canonicalization, cse, licm, tiling_canonicalization} : (!pdl.operation) ->
()

    // Tile reduction loop
    %first_0, %rest_1 = transform.iree.take_first %3, %tiled_op : (!pdl.operation, !pdl.operation) -> (!pdl.operation,
!pdl.operation)
    %tiled_linalg_op, %loops = transform.structured.tile_to_scf_for %first_0[0, 0, 0, 32]
    %7 = transform.structured.fuse_into_containing_op %6 into %loops

    // Promote filter operand
    %matmul_padded_l2 = transform.structured.pad %tiled_linalg_op {
      padding_values = [0.0 : f32, 0.0 : f32, 0.0 : f32],
      padding_dimensions = [0, 1, 2],
      pack_paddings=[1, 0, 1]
    }
    ...
  }
}
```

```
hal.executable.variant public @vulkan_spirv_fb, target = <"vulkan", "vulkan-spirv-fb", {spirv.target_env = #spirv.target_env<#spirv.vce<v1.3, [Shader, GroupNonUniform], [SPV_KHR_storage_buffer_storage_class,
SPV_KHR_variable_pointers]>, api=Vulkan, #spirv.resource_limits<max_compute_workgroup_size = [128, 128, 64], subgroup_size = 64, cooperative_matrix_properties_nv = []>>}> {
  hal.executable.export public @conv_2d_nchw_fchw_dispatch_0_conv_2d_nchw_fchw_2x32x128x128x64x3x3_f32 ordinal(0) layout(#hal.pipeline.layout<push_constants = 0, sets = [<0, bindings = [<0, storage_buffer, ReadOnly>, <1,
storage_buffer, ReadOnly>, <2, storage_buffer>]>]>) attributes {translation_info = #iree_codegen.translation_info<TransformDialectCodegen>} {
  ^bb0(%arg0: !hal.device):
    %x, %y, %z = flow.dispatch.workgroup_count_from_slice
    hal.return %x, %y, %z : index, index, index
  }
  builtin.module {
    func.func @conv_2d_nchw_fchw_dispatch_0_conv_2d_nchw_fchw_2x32x128x128x64x3x3_f32() {
      %c0 = arith.constant 0 : index
      %cst = arith.constant 0.000000e+00 : f32
      %0 = hal.interface.binding.subspan set(0) binding(0) type(storage_buffer) alignment(64) offset(%c0) flags(ReadOnly) : !flow.dispatch.tensor<readonly:tensor<2x64x130x130xf32>>
      %1 = hal.interface.binding.subspan set(0) binding(1) type(storage_buffer) alignment(64) offset(%c0) flags(ReadOnly) : !flow.dispatch.tensor<readonly:tensor<32x64x3x3xf32>>
      %2 = hal.interface.binding.subspan set(0) binding(2) type(storage_buffer) alignment(64) offset(%c0) : !flow.dispatch.tensor<writeonly:tensor<2x32x128x128xf32>>
      %3 = flow.dispatch.tensor.load %0, offsets = [0, 0, 0, 0], sizes = [2, 64, 130, 130], strides = [1, 1, 1, 1] : !flow.dispatch.tensor<readonly:tensor<2x64x130x130xf32>> -> tensor<2x64x130x130xf32>
      %4 = flow.dispatch.tensor.load %1, offsets = [0, 0, 0, 0], sizes = [32, 64, 3, 3], strides = [1, 1, 1, 1] : !flow.dispatch.tensor<readonly:tensor<32x64x3x3xf32>> -> tensor<32x64x3x3xf32>
      %5 = tensor.empty() : tensor<2x32x128x128xf32>
      %6 = linalg.fill ins(%cst : f32) outs(%5 : tensor<2x32x128x128xf32>) -> tensor<2x32x128x128xf32>
      %7 = linalg.conv_2d_nchw_fchw {dilations = dense<1> : tensor<2xi64>, strides = dense<1> : tensor<2xi64>} ins(%3, %4 : tensor<2x64x130x130xf32>, tensor<32x64x3x3xf32>) outs(%6 : tensor<2x32x128x128xf32>) ->
tensor<2x32x128x128xf32>
      flow.dispatch.tensor.store %7, %2, offsets = [0, 0, 0, 0], sizes = [2, 32, 128, 128], strides = [1, 1, 1, 1] : tensor<2x32x128x128xf32> -> !flow.dispatch.tensor<writeonly:tensor<2x32x128x128xf32>>
      return
    }
  }
}
```

IREE can select a transform dialect codegen pipeline which will construct the transform ops on the fly using typical op building APIs which can then be interpreted and applied to the IR

Implicit GEMM

```
module {
  transform.sequence  failures(propagate) {
  ^bb0(%arg0: !pdl.operation):
    // Step 1. Convert to img2col
    transform.iree.register_match_callbacks
    %0:4 = transform.iree.match_callback failures(propagate) "convolution"(%arg0) : (!
pdl.operation) -> (!pdl.operation, !pdl.operation, !pdl.operation, !pdl.operation)
    %img2col_tensor, %transformed = transform.structured.convert_conv2d_to_img2col
%0#2 : (!pdl.operation) -> (!pdl.operation, !pdl.operation)
    %1 = get_producer_of_operand %transformed[0] : (!pdl.operation) -> !pdl.operation

    // Bubble collapse_shape ops to boundaries of kernel
    %2 = transform.structured.match ops{["func.func"]} in %arg0 : (!pdl.operation)
-> !pdl.operation
    transform.iree.apply_patterns %2 {bubble_collapse} : (!pdl.operation) -> ()
    %first, %rest = transform.iree.take_first %0#3, %1 : (!pdl.operation, !
pdl.operation) -> (!pdl.operation, !pdl.operation)

    // Tile and distribute to workgroups
    %forall_op, %tiled_op = transform.structured.tile_to_forall_op %first
num_threads [] tile_sizes [1, 32, 32](mapping = [#gpu.block<z>, #gpu.block<y>,
#gpu.block<x>])
    transform.iree.apply_patterns %arg0 {canonicalization, cse, licm,
tiling_canonicalization} : (!pdl.operation) -> ()
    %3 = transform.structured.fuse_into_containing_op %rest into %forall_op
    transform.iree.populate_workgroup_count_region_using_num_threads_slice
%forall_op : (!pdl.operation) -> ()
    %4 = transform.structured.match ops{["linalg.fill"]} in %arg0 : (!pdl.operation)
-> !pdl.operation
    %5 = transform.structured.fuse_into_containing_op %4 into %forall_op

    // Fuse img2col with workgroup forall
    %6 = transform.structured.fuse_into_containing_op %img2col_tensor into %forall_op
    transform.iree.apply_patterns %arg0 {canonicalization, cse, licm,
tiling_canonicalization} : (!pdl.operation) -> ()
  }
}
```

```
#map = affine_map<(d0) -> (d0 * 32)>
#map6 = affine_map<(d0, d1, d2, d3) -> (d1, d3)>
#map7 = affine_map<(d0, d1, d2, d3) -> (d0, d3, d2)>
#map8 = affine_map<(d0, d1, d2, d3) -> (d0, d1, d2)>
builtin.module {
  func.func @conv_2d_nchw_fchw_dispatch_0_conv_2d_nchw_fchw_2x32x128x128x64x3x3_f32() {
    %collapsed = tensor.collapse_shape %4 [[0], [1, 2, 3]] : tensor<32x64x3x3xf32> into
tensor<32x576xf32>
    %collapsed_0 = tensor.collapse_shape %5 [[0], [1], [2, 3]] : tensor<2x32x128x128xf32> into
tensor<2x32x16384xf32>
    %6 = tensor.empty() : tensor<2x576x16384xf32>
    %7 = scf.forall (%arg0, %arg1, %arg2) in (2, 1, 512) shared_outs(%arg3 = %collapsed_0) ->
(tensor<2x32x16384xf32>) {
      %8 = affine.apply #map(%arg1)
      %9 = affine.apply #map(%arg2)
      %extracted_slice = tensor.extract_slice %collapsed[%8, 0] [32, 576] [1, 1] : tensor<32x576xf32> to
tensor<32x576xf32>
      %extracted_slice_1 = tensor.extract_slice %6[%arg0, 0, %9] [1, 576, 32] [1, 1, 1] :
tensor<2x576x16384xf32> to tensor<1x576x32xf32>
      %10 = /*img2col*/ -> tensor<1x576x32xf32>
      %extracted_slice_2 = tensor.extract_slice %arg3[%arg0, %8, %9] [1, 32, 32] [1, 1, 1] :
tensor<2x32x16384xf32> to tensor<1x32x32xf32>
      %11 = linalg.fill ins(%cst : f32) outs(%extracted_slice_2 : tensor<1x32x32xf32>) ->
tensor<1x32x32xf32>
      %12 = linalg.generic {indexing_maps = [#map6, #map7, #map8], iterator_types = ["parallel",
"parallel", "parallel", "reduction"]} ins(%extracted_slice, %10 : tensor<32x576xf32>,
tensor<1x576x32xf32>) outs(%11 : tensor<1x32x32xf32>) {
      ^bb0(%in: f32, %in_3: f32, %out: f32):
        %13 = arith.mulf %in, %in_3 : f32
        %14 = arith.addf %13, %out : f32
        linalg.yield %14 : f32
      } -> tensor<1x32x32xf32>
      scf.forall.in_parallel {
        tensor.parallel_insert_slice %12 into %arg3[%arg0, %8, %9] [1, 32, 32] [1, 1, 1] :
tensor<1x32x32xf32> into tensor<2x32x16384xf32>
      }
    } {mapping = [#gpu.block<z>, #gpu.block<y>, #gpu.block<x>]}
    %expanded = tensor.expand_shape %7 [[0], [1], [2, 3]] : tensor<2x32x16384xf32> into
tensor<2x32x128x128xf32>
    return %expanded : tensor<2x32x128x128xf32>
  }
}
```

Implicit GEMM

```
module {
  transform.sequence  failures(propagate) {
  ^bb0(%arg0: !pdl.operation):

    // Step 2. Tile reduction loop
    %first_0, %rest_1 = transform.iree.take_first %3, %tiled_op : (!
pdl.operation, !pdl.operation) -> (!pdl.operation, !pdl.operation)
    %tiled_linalg_op, %loops = transform.structured.tile_to_scf_for %first_0[0,
0, 0, 32]
    %7 = transform.structured.fuse_into_containing_op %6 into %loops

    // Promote filter operand by padding on the output/filter
    %matmul_padded_l2 = transform.structured.pad %tiled_linalg_op {
      padding_values = [0.0 : f32, 0.0 : f32, 0.0 : f32],
      padding_dimensions = [0, 1, 2],
      pack_paddings=[1, 0, 1]
    }
    %pad_res = transform.get_producer_of_operand %matmul_padded_l2[2]
       : (!pdl.operation) -> !pdl.operation
    %pad_res_2 = transform.cast %pad_res : !pdl.operation to !
transform.op<“tensor.pad”>

    // Hoist pad on the output
    %pad_res_hoisted = transform.structured.hoist_pad %pad_res_2 by 1 loops
       : (!transform.op<"tensor.pad">) -> !pdl.operation
    transform.iree.apply_patterns %arg0
      {canonicalization, cse, licm, fold_tensor_subsets, tiling_canonicalization}
: (!pdl.operation) -> ()
    %insert_slice_back = transform.structured.match ops{["tensor.insert_slice"]}
in %arg0
      : (!pdl.operation) -> !pdl.operation
    %copy_back = transform.structured.insert_slice_to_copy %insert_slice_back
      : (!pdl.operation) -> !pdl.operation
    transform.iree.apply_patterns %arg0
      {canonicalization, cse, licm, tiling_canonicalization} : (!pdl.operation)
-> ()
  }
}
```

```
#map = affine_map<(d0) -> (d0 * 32)>
#map6 = affine_map<(d0, d1, d2, d3) -> (d1, d3)>
#map7 = affine_map<(d0, d1, d2, d3) -> (d0, d3, d2)>
#map8 = affine_map<(d0, d1, d2, d3) -> (d0, d1, d2)>
builtin.module {
  func.func @conv_2d_nchw_fchw_dispatch_0_conv_2d_nchw_fchw_2x32x128x128x64x3x3_f32() {
    %collapsed = tensor.collapse_shape %4 [[0], [1, 2, 3]] : tensor<32x64x3x3xf32> into tensor<32x576xf32>
    %collapsed_0 = tensor.collapse_shape %5 [[0], [1], [2, 3]] : tensor<2x32x128x128xf32> into
tensor<2x32x16384xf32>
    %6 = tensor.empty() : tensor<2x576x16384xf32>
    %7 = scf.forall (%arg0, %arg1, %arg2) in (2, 1, 512) shared_outs(%arg3 = %collapsed_0) ->
(tensor<2x32x16384xf32>) {
      %8 = tensor.empty() : tensor<1x32x32xf32>
      %9 = linalg.fill ins(%cst : f32) outs(%8 : tensor<1x32x32xf32>) -> tensor<1x32x32xf32>
      %10 = affine.apply #map(%arg1)
      %11 = affine.apply #map(%arg2)
      %12 = scf.for %arg4 = %c0 to %c576 step %c32 iter_args(%arg5 = %9) -> (tensor<1x32x32xf32>) {
        %extracted_slice = tensor.extract_slice %collapsed[%10, %arg4] [32, 32] [1, 1] : tensor<32x576xf32> to
tensor<32x32xf32>
        %extracted_slice_1 = tensor.extract_slice %6[%arg0, %arg4, %11] [1, 32, 32] [1, 1, 1] :
tensor<2x576x16384xf32> to tensor<1x32x32xf32>
        %13 = /*im2col*/ -> tensor<1x32x32xf32>
        %padded = tensor.pad %extracted_slice nofold low[%c0, %c0] high[%c0, %c0] {
        ^bb0(%arg6: index, %arg7: index):
          tensor.yield %cst : f32
        } : tensor<32x32xf32> to tensor<32x32xf32>
        %14 = linalg.generic {indexing_maps = [#map6, #map7, #map8], iterator_types = ["parallel", "parallel",
"parallel", "reduction"]} ins(%padded, %13 : tensor<32x32xf32>, tensor<1x32x32xf32>) outs(%arg5 :
tensor<1x32x32xf32>) {
        ^bb0(%in: f32, %in_2: f32, %out: f32):
          %15 = arith.mulf %in, %in_2 : f32
          %16 = arith.addf %15, %out : f32
          linalg.yield %16 : f32
        } -> tensor<1x32x32xf32>
        scf.yield %14 : tensor<1x32x32xf32>
      }
      scf.forall.in_parallel {
        tensor.parallel_insert_slice %12 into %arg3[%arg0, %10, %11] [1, 32, 32] [1, 1, 1] : tensor<1x32x32xf32>
into tensor<2x32x16384xf32>
      }
    } {mapping = [#gpu.block<z>, #gpu.block<y>, #gpu.block<x>]}
    %expanded = tensor.expand_shape %7 [[0], [1], [2, 3]] : tensor<2x32x16384xf32> into tensor<2x32x128x128xf32>
    return %expanded : tensor<2x32x128x128xf32>
  }
}
```

Implicit GEMM

```
module {
  transform.sequence  failures(propagate) {
  ^bb0(%arg0: !pdl.operation):
    // Tile to threads (SIMT)
    %forall_op_2, %tiled_op_3 = transform.structured.tile_to_forall_op %7
num_threads [0, 32, 8] tile_sizes [](mapping = [#gpu.linear<y>, #gpu.linear<x>])
    transform.iree.apply_patterns %arg0 {canonicalization, cse, licm,
tiling_canonicalization} : (!pdl.operation) -> ()
    %forall_op_4, %tiled_op_5 = transform.structured.tile_to_forall_op %rest_1
num_threads [0, 32, 8] tile_sizes [](mapping = [#gpu.linear<y>, #gpu.linear<x>])
    transform.iree.apply_patterns %arg0 {canonicalization, cse, licm,
tiling_canonicalization} : (!pdl.operation) -> ()

    %pad_lhs = transform.get_producer_of_operand %matmul_padded_l2[0]
       : (!pdl.operation) -> !pdl.operation
    %forall_pad_lhs, %tiled_pad_lhs =
      transform.structured.tile_to_forall_op %pad_lhs num_threads [32, 8]
        ( mapping = [#gpu.linear<y>, #gpu.linear<x>] )
    %if_lhs = transform.structured.match ops{["scf.if"]} in %forall_pad_lhs
      : (!pdl.operation) -> !transform.any_op
    transform.scf.take_assumed_branch %if_lhs take_else_branch
      : (!transform.any_op) -> ()

    %forall_copy_back, %tiled_copy_back =
    transform.structured.tile_to_forall_op %copy_back num_threads [0, 32, 8]
      ( mapping = [#gpu.linear<y>, #gpu.linear<x>] )

    transform.iree.apply_patterns %arg0
      {canonicalization, cse, licm} : (!pdl.operation) -> ()

    //// Masked vectorize on padding
    transform.structured.masked_vectorize %tiled_pad_lhs vector_sizes [1, 4]
    transform.structured.masked_vectorize %tiled_copy_back vector_sizes [1, 4]
  }
}
```

```
%13 = scf.for %arg4 = %c0 to %c576 step %c32 iter_args(%arg5 = %9) -> (tensor<1x32x32xf32>) {
  %extracted_slice = tensor.extract_slice %collapsed[%10, %arg4] [32, 32] [1, 1] :
tensor<32x576xf32> to tensor<32x32xf32>
  %extracted_slice_1 = tensor.extract_slice %6[%arg0, %arg4, %11] [1, 32, 32] [1, 1, 1] :
tensor<2x576x16384xf32> to tensor<1x32x32xf32>
  %14 = scf.forall (%arg6, %arg7) in (32, 8) shared_outs(%arg8 = %extracted_slice_1) ->
(tensor<1x32x32xf32>) {
    %17 = affine.apply #map1(%arg7)
    %extracted_slice_2 = tensor.extract_slice %arg8[0, %arg6, %17] [1, 1, 4] [1, 1, 1] :
tensor<1x32x32xf32> to tensor<1x1x4xf32>
    %18 = /*img2col*/ -> tensor<1x1x4xf32>
    scf.forall.in_parallel {
      tensor.parallel_insert_slice %18 into %arg8[0, %arg6, %17] [1, 1, 4] [1, 1, 1] :
tensor<1x1x4xf32> into tensor<1x32x32xf32>
    }
  } {mapping = [#gpu.linear<y>, #gpu.linear<x>]}
  %15 = scf.forall (%arg6, %arg7) in (32, 8) shared_outs(%arg8 = %12) -> (tensor<32x32xf32>) {
    /* setup pad mask */
    %25 = vector.create_mask %20, %23 : vector<1x4xi1>
    %26 = vector.mask %25 { vector.transfer_read %extracted_slice_2[%c0_3, %c0_3], %cst {in_bounds
= [true, true]} : tensor<?x?xf32>, vector<1x4xf32> } : vector<1x4xi1> -> vector<1x4xf32>
    %27 = vector.transfer_write %26, %24[%c0_3, %c0_3] {in_bounds = [true, true]} :
vector<1x4xf32>, tensor<1x4xf32>
    scf.forall.in_parallel {
      tensor.parallel_insert_slice %27 into %arg8[%arg6, %17] [1, 4] [1, 1] : tensor<1x4xf32> into
tensor<32x32xf32>
    }
  } {mapping = [#gpu.linear<y>, #gpu.linear<x>]}
  %16 = linalg.generic {indexing_maps = [#map12, #map13, #map14], iterator_types = ["parallel",
"parallel", "parallel", "reduction"]} ins(%15, %14 : tensor<32x32xf32>, tensor<1x32x32xf32>)
outs(%arg5 : tensor<1x32x32xf32>) {
  ^bb0(%in: f32, %in_2: f32, %out: f32):
    %17 = arith.mulf %in, %in_2 : f32
    %18 = arith.addf %17, %out : f32
    linalg.yield %18 : f32
  } -> tensor<1x32x32xf32>
  scf.yield %16 : tensor<1x32x32xf32>
}
```

Implicit GEMM

Tiling to warps for Cooperative Matrix/WMMA (not shown)

```
module {
  transform.sequence  failures(propagate) {
  ^bb0(%arg0: !pdl.operation):
    // Tile to warps (SIMD)
    %forall_op_8, %tiled_op_9 = transform.structured.tile_to_forall_op
%matmul_padded_l2   num_threads [0, 2, 2] tile_sizes [](mapping =
[#gpu.warp<y>, #gpu.warp<x>])
    %fill = transform.structured.match ops{["linalg.fill"]} in %arg0 : (!
pdl.operation) -> !pdl.operation
    %forall_op_6, %tiled_op_7 = transform.structured.tile_to_forall_op %fill
num_threads [0, 2, 2] tile_sizes [](mapping = [#gpu.warp<y>, #gpu.warp<x>])

    // Lower masks
    %func_v = transform.structured.match ops{["func.func"]} in %arg0
      : (!pdl.operation) -> !pdl.operation
    %func_v_2 = transform.vector.lower_masked_transfers %func_v
      : (!pdl.operation) -> !pdl.operation
    transform.iree.apply_patterns %func_v_2 { rank_reducing_linalg,
rank_reducing_vector }
      : (!pdl.operation) -> ()
    %func_v_3 = transform.structured.vectorize %func_v_2
{vectorize_nd_extract}
    transform.iree.apply_patterns %arg0 {canonicalization, cse, licm}
      : (!pdl.operation) -> ()
  }
}
```

Huge amount of index arithmetic for Img2Col :(

```
%13 = scf.forall (%arg6, %arg7) in (32, 8) shared_outs(%arg8 = %7) -> (tensor<1x32x32xf32>) {
  %16 = affine.apply #map2(%arg7)
  %extracted_slice_10 = tensor.extract_slice %arg8[0, %arg6, %16] [1, 1, 4] [1, 1, 1] : tensor<1x32x32xf32> to tensor<1x1x4xf32>
  %17 = arith.addi %arg4, %arg6 : index
  %18 = arith.cmpi slt, %17, %c0 : index
  %19 = arith.subi %c-1, %17 : index
  %20 = arith.select %18, %19, %17 : index
  %21 = arith.divsi %20, %c9 : index
  %22 = arith.subi %c-1, %21 : index
  %23 = arith.select %18, %22, %21 : index
  %24 = arith.muli %arg2, %c32 : index
  %25 = vector.broadcast %24 : index to vector<4xindex>
  %26 = arith.addi %25, %cst_0 : vector<4xindex>
  %27 = arith.muli %arg7, %c4 : index
  %28 = vector.broadcast %27 : index to vector<4xindex>
  %29 = arith.addi %26, %28 : vector<4xindex>
  %30 = arith.cmpi slt, %29, %cst_1 : vector<4xindex>
  %31 = arith.subi %cst_3, %29 : vector<4xindex>
  %32 = arith.select %30, %31, %29 : vector<4xi1>, vector<4xindex>
  %33 = arith.divsi %32, %cst_2 : vector<4xindex>
  %34 = arith.subi %cst_3, %33 : vector<4xindex>
  %35 = arith.select %30, %34, %33 : vector<4xi1>, vector<4xindex>
  %36 = arith.remsi %17, %c9 : index
  %37 = arith.cmpi slt, %36, %c0 : index
  %38 = arith.addi %36, %c9 : index
  %39 = arith.select %37, %38, %36 : index
  %40 = arith.cmpi slt, %39, %c0 : index
  %41 = arith.subi %c-1, %39 : index
  %42 = arith.select %40, %41, %39 : index
  %43 = arith.divsi %42, %c3 : index
  %44 = arith.subi %c-1, %43 : index
  %45 = arith.select %40, %44, %43 : index
  %46 = vector.broadcast %45 : index to vector<4xindex>
  %47 = arith.addi %35, %46 : vector<4xindex>
  %48 = arith.addi %24, %arg4 : index
  %49 = vector.broadcast %48 : index to vector<4xindex>
  %50 = arith.addi %49, %cst_0 : vector<4xindex>
  %51 = arith.addi %50, %28 : vector<4xindex>
  %52 = vector.broadcast %arg6 : index to vector<4xindex>
  %53 = arith.addi %51, %52 : vector<4xindex>
  %54 = arith.muli %35, %cst_4 : vector<4xindex>
  %55 = arith.addi %53, %54 : vector<4xindex>
  %56 = arith.divsi %20, %c3 : index
  %57 = arith.subi %c-1, %56 : index
  %58 = arith.select %18, %57, %56 : index
  %59 = arith.muli %58, %c-3 : index
  %60 = vector.broadcast %59 : index to vector<4xindex>
  %61 = arith.addi %55, %60 : vector<4xindex>
  %62 = vector.broadcast %arg0 : index to vector<4xindex>
  %63 = arith.muli %62, %cst_7 : vector<4xindex>
  %64 = vector.broadcast %23 : index to vector<4xindex>
  %65 = arith.addi %64, %63 : vector<4xindex>
  %66 = arith.muli %65, %cst_8 : vector<4xindex>
  %67 = arith.addi %47, %66 : vector<4xindex>
  %68 = arith.muli %67, %cst_8 : vector<4xindex>
  %69 = arith.addi %61, %68 : vector<4xindex>
  %70 = vector.gather %2[%c0, %c0, %c0, %c0] [%69], %cst_5, %cst_6 : tensor<2x64x130x130xf32>, vector<4xindex>, vector<4xi1>,
vector<4xf32> into vector<4xf32>
  %71 = vector.transfer_write %70, %extracted_slice_10[%c0, %c0, %c0] {in_bounds = [true]} : vector<4xf32>, tensor<1x1x4xf32>
  scf.forall.in_parallel {
    tensor.parallel_insert_slice %71 into %arg8[0, %arg6, %16] [1, 1, 4] [1, 1, 1] : tensor<1x1x4xf32> into tensor<1x32x32xf32>
  }
}
```

Implicit GEMM

- Reference matmul codegen script
  - https://github.com/iree-org/iree-samples/blob/main/transform_dialect/examples/cuda/fill_matmul_unaligned_wmma_vector_4_codegen_spec.mlir
  - Includes transforms for multi buffering, async copies, and software pipelining that are currently excluded for implicit GEMM
  - Follow on work needed to enable those optimizations with the leading Img2Col gather

```
%90 = gpu.subgroup_mma_load_matrix %subview_12[%c0, %c0] {leadDimension = 32 : index} :
memref<?x32xf32, strided<[32, 1], offset: ?>, #gpu.address_space<workgroup>> -> !
gpu.mma_matrix<16x16xf32, "AOp">
%91 = gpu.subgroup_mma_load_matrix %subview_12[%c0, %c16] {leadDimension = 32 : index} :
memref<?x32xf32, strided<[32, 1], offset: ?>, #gpu.address_space<workgroup>> -> !
gpu.mma_matrix<16x16xf32, "AOp">
%92 = gpu.subgroup_mma_load_matrix %subview_13[%c0, %c0, %c0] {leadDimension = 32 : index} :
memref<1x32x?xf32, strided<[1024, 32, 1], offset: ?>, #gpu.address_space<workgroup>> -> !
gpu.mma_matrix<16x16xf32, "BOp">
%93 = gpu.subgroup_mma_load_matrix %subview_13[%c0, %c16, %c0] {leadDimension = 32 : index} :
memref<1x32x?xf32, strided<[1024, 32, 1], offset: ?>, #gpu.address_space<workgroup>> -> !
gpu.mma_matrix<16x16xf32, "BOp">
%94 = gpu.subgroup_mma_compute %90, %92, %arg1 : !gpu.mma_matrix<16x16xf32, "AOp">, !
gpu.mma_matrix<16x16xf32, "BOp"> -> !gpu.mma_matrix<16x16xf32, "COp">
%95 = gpu.subgroup_mma_compute %91, %93, %94 : !gpu.mma_matrix<16x16xf32, "AOp">, !
gpu.mma_matrix<16x16xf32, "BOp"> -> !gpu.mma_matrix<16x16xf32, "COp">
```

```
module {
  transform.sequence  failures(propagate) {
  ^bb0(%arg0: !pdl.operation):
    // Bufferize
    transform.iree.apply_patterns %arg0 {canonicalization, cse, licm}
      : (!pdl.operation) -> ()
    transform.iree.eliminate_empty_tensors %arg0 : (!pdl.operation) -> ()
    %variant_op_3 = transform.iree.bufferize { target_gpu } %arg0
      : (!pdl.operation) -> (!pdl.operation)
    %func_m = transform.structured.match ops{["func.func"]} in %variant_op_3
      : (!pdl.operation) -> !pdl.operation
    transform.iree.erase_hal_descriptor_type_from_memref %func_m
      : (!pdl.operation) -> ()
    transform.iree.apply_buffer_optimizations %func_m : (!pdl.operation) -> ()


    // Map to workgroups and threads
    transform.iree.apply_patterns %variant_op_3
      {canonicalization, cse, licm, tiling_canonicalization}
      : (!pdl.operation) -> ()
    transform.iree.forall_to_workgroup %func_m : (!pdl.operation) -> ()
    transform.iree.map_nested_forall_to_gpu_threads %func_m
        workgroup_dims = [64, 4, 1] warp_dims = [2, 2, 1]
      : (!pdl.operation) -> ()
    transform.iree.apply_patterns %variant_op_3 {canonicalization, cse, licm}
      : (!pdl.operation) -> ()

    // Cleanup and unroll
    transform.iree.apply_patterns %func_m {canonicalization, cse, licm}
      : (!pdl.operation) -> ()
    transform.iree.hoist_static_alloc %func_m : (!pdl.operation) -> ()
    transform.iree.apply_patterns %func_m { fold_memref_aliases }
      : (!pdl.operation) -> ()
    transform.iree.apply_patterns %func_m { extract_address_computations }
      : (!pdl.operation) -> ()
    transform.iree.apply_patterns %func_m {canonicalization, cse, licm}
      : (!pdl.operation) -> ()
   transform.iree.apply_patterns %func_m { unroll_vectors_gpu_coop_mat }
      : (!pdl.operation) -> ()

    %func_m_2 = transform.structured.hoist_redundant_vector_transfers %func_m
      : (!pdl.operation) -> !pdl.operation

    transform.iree.apply_patterns %func_m_2 {canonicalization, cse, licm}
      : (!pdl.operation) -> ()
    transform.iree.apply_buffer_optimizations %func_m_2 : (!pdl.operation) -> ()

    // This must occur after bufferization, unrolling and hoisting because of the
    // fancy CUDA types.
    transform.iree.vector.vector_to_mma_conversion %func_m_2 { use_wmma }
      : (!pdl.operation) -> ()

  }
}
```

Vectorizing Img2Col

- Img2Col on NCHW does contiguous loads of size FW = 3
  - #map8 = affine_map<(d0, d1) -> (d0 + d1 - (d0 floordiv 128) * 128 - (d1 floordiv 3) * 3)>
    %11 = linalg.index 1 : index // %11 = 0 to 576
    %12 = linalg.index 2 : index // %12 = 0 to 16384
    %20 = affine.apply #map8(%12, %11)
    %extracted = tensor.extract %input[%10, %13, %19, %20] : tensor<2x64x130x130xf32>
- Channels inner most (NHWC) does contiguous loads of size IC = 64
  - #map6 = affine_map<(d0) -> (d0 mod 64)>
    %12 = linalg.index 2 : index
    %18 = affine.apply #map6(%12)
    %extracted = tensor.extract %3[%10, %19, %20, %18] : tensor<2x130x130x64xf32>

Selecting a Data Layout

- In general, only an inner tile size that matches the target vector size is required
  - For Cooperative Matrix on AMD RDNA3 that is 16 x 16
  - Idea: “pack” the convolution **input**, **filter**, and **output** and then “unpack” back to the original layout by tiling the channel dimensions with **tensor.pack** and **tensor.unpack**
- inner_dims_pos
  - Specifies which dimensions are tiled and the order of the resulting tiles
- inner_tiles
  - Specifies the sizes of the tiles
- outer_dims_pos
  - Specifies the order of the outer dimensions

```
operation ::= `tensor.pack` $source
              (`padding_value` `(` $padding_value^ `:` type($padding_value) `)`)?
              (`outer_dims_perm` `=` $outer_dims_perm^)?
              `inner_dims_pos` `=` $inner_dims_pos
              `inner_tiles` `=`
              custom<DynamicIndexList>($inner_tiles, $static_inner_tiles)
              `into` $dest attr-dict `:` type($source) `->` type($dest)
```

Selecting a Data Layout

- In general, only an inner tile size that matches the target vector size is required
  - Idea: “pack” the convolution **input**, **filter**, and **output** and then “unpack” back to the original layout by tiling the channel dimensions with **tensor.pack** and **tensor.unpack**
  - Leads to the NCHWc layout where **c** is the inner tile of the input channel dimension
    - IC = C * c, if c = C then this becomes N1HWC = NHWC

```
module {
  func.func @conv_2d_nchw_fchw(%arg0: tensor<2x64x130x130xf32>, %arg1: tensor<32x64x3x3xf32>) -> tensor<2x32x128x128xf32> {
    %cst = arith.constant 0.000000e+00 : f32
    %0 = tensor.empty() : tensor<2x1x130x130x64xf32>
    %pack = tensor.pack %arg0 inner_dims_pos = [1] inner_tiles = [64] into %0 : tensor<2x64x130x130xf32> -> tensor<2x1x130x130x64xf32>
    %collapsed = tensor.collapse_shape %pack [[0], [1, 2], [3], [4]] : tensor<2x1x130x130x64xf32> into tensor<2x130x130x64xf32>
    %1 = tensor.empty() : tensor<1x1x3x3x64x32xf32>
    %pack_0 = tensor.pack %arg1 inner_dims_pos = [1, 0] inner_tiles = [64, 32] into %1 : tensor<32x64x3x3xf32> -> tensor<1x1x3x3x64x32xf32>
    %collapsed_1 = tensor.collapse_shape %pack_0 [[0, 1, 2], [3], [4], [5]] : tensor<1x1x3x3x64x32xf32> into tensor<3x3x64x32xf32>
    %2 = tensor.empty() : tensor<2x128x128x32xf32>
    %3 = linalg.fill ins(%cst : f32) outs(%2 : tensor<2x128x128x32xf32>) -> tensor<2x128x128x32xf32>
    %4 = linalg.conv_2d_nhwc_hwcf {dilations = dense<1> : tensor<2xi64>, strides = dense<1> : tensor<2xi64>} ins(%collapsed, %collapsed_1 :
tensor<2x130x130x64xf32>, tensor<3x3x64x32xf32>) outs(%3 : tensor<2x128x128x32xf32>) -> tensor<2x128x128x32xf32>
    %expanded = tensor.expand_shape %4 [[0], [1, 2], [3], [4]] : tensor<2x128x128x32xf32> into tensor<2x1x128x128x32xf32>
    %5 = tensor.empty() : tensor<2x32x128x128xf32>
    %unpack = tensor.unpack %expanded inner_dims_pos = [1] inner_tiles = [32] into %5 {__unpack__} : tensor<2x1x128x128x32xf32> ->
tensor<2x32x128x128xf32>
    return %unpack : tensor<2x32x128x128xf32>
  }
}
```

NCHW -> NHWC

FCHW -> HWCF

NHWC -> NCHW

Propagating Pack/Unpack

- Packing the input and output incurs an additional cost
- Can be mitigated by propagating the **tensor.pack**/**tensor.unpack** ops through the graph
  - https://github.com/llvm/llvm-project/blob/main/mlir/lib/Dialect/Linalg/Transforms/DataLayoutPropagation.cpp
- **tensor.pack** converts a tensor from the original layout to a packed layout based on the tile positions/sizes
- **tensor.unpack** converts from the packed to the original layout, cancelling with a pack op with the same tile positions/sizes
- Pack/unpack propagation can then generally be thought of as transitioning ops to the packed space
  - **tensor.pack** propagates upward to its producers
  - **tensor.unpack** propagates downward to its consumers

Propagating Pack/Unpack

- An example from https://github.com/llvm/llvm-project/blob/main/mlir/test/Dialect/Linalg/data-layout-propagation.mlir

```
#map0 = affine_map<(d0, d1) -> (d0, d1)>
func.func @elem_pack_transpose_inner_and_outer_dims(%arg0:
tensor<128x256xi32>, %dest: tensor<16x4x16x32xi32>) ->
tensor<16x4x16x32xi32>{
  %init = tensor.empty() : tensor<128x256xi32>
  %elem = linalg.generic {indexing_maps = [#map0, #map0],
iterator_types = ["parallel", "parallel"]}
      ins(%arg0 : tensor<128x256xi32>)
      outs(%init : tensor<128x256xi32>) {
    ^bb0(%arg3: i32, %arg4: i32):
      %4 = arith.addi %arg3, %arg3 : i32
      linalg.yield %4 : i32
  } -> tensor<128x256xi32>
  %pack = tensor.pack %elem
    outer_dims_perm = [1, 0]
    inner_dims_pos = [1, 0]
    inner_tiles = [16, 32]
    into %dest : tensor<128x256xi32> -> tensor<16x4x16x32xi32>
  return %pack : tensor<16x4x16x32xi32>
}
```

```
#map = affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>
module {
  func.func @elem_pack_transpose_inner_and_outer_dims(%arg0:
tensor<128x256xi32>, %arg1: tensor<16x4x16x32xi32>) ->
tensor<16x4x16x32xi32> {
    %0 = tensor.empty() : tensor<16x4x16x32xi32>
    %pack = tensor.pack %arg0
       outer_dims_perm = [1, 0]
       inner_dims_pos = [1, 0]
       inner_tiles = [16, 32]
       into %0 : tensor<128x256xi32> -> tensor<16x4x16x32xi32>
    %1 = linalg.generic {indexing_maps = [#map, #map],
iterator_types = ["parallel", "parallel", "parallel", "parallel"]}
ins(%pack : tensor<16x4x16x32xi32>) outs(%arg1 :
tensor<16x4x16x32xi32>) {
    ^bb0(%in: i32, %out: i32):
      %2 = arith.addi %in, %in : i32
      linalg.yield %2 : i32
    } -> tensor<16x4x16x32xi32>
    return %1 : tensor<16x4x16x32xi32>
  }
}
```

Propagating Pack/Unpack

- WIP patterns for propagation through ops with gather semantics and tensor.insert_slice
- Gather ops (**tensor.extract/linalg.index**)
  - Gather ops compute the index with which to extract an element inside the body of the linalg op
  - Propagation is similar to standard element wise propagation, however the packing of the input linalg op depends on analysis of the **linalg.index** ops and their users to determine whether or not propagation is possible
- **tensor.insert_slice**
  - Requires any tiled dimensions to either be aligned on the tile size or to cover no more than a single tile
  - Otherwise multiple insert_slice ops would need to be generated when propagating

Conclusions & Future Work

- Implicit GEMM is an approach to convolution that maps the problem to a GEMM with leading gather
  - Particularly well motivated when targeting GEMM based accelerators
- Difficult to get performance for NCHW convolutions (default layout for PyTorch models)
  - Motivates applying a layout transformation to tile at least a 16 extent dimension from the input channels to the inner most dim (NCHWc)
  - Layout transformation achieved through pack/unpack propagation
- Catching implicit GEMM codegen up to matmul codegen in IREE still requires generalizations in pipelining patterns
- Improvements to Img2Col arithmetic
- Enable unaligned shapes for implicit GEMM