Winograd Convolutions in MLIR

Harsh Menon (nod.ai)

Overview

- Motivation
- Winograd Convolution Algorithm
- MLIR Representation
- MLIR Code Generation Strategy
- Results
- Conclusions & Future Work

Motivation

- Deep learning has seen tremendous success in a wide range of applications from object detection, self-driving cars, natural language processing, generative art and more
- State of the art models tend to be extremely large containing trillions of parameters
- Training and serving these models is computationally intensive
- Matrix multiplication and convolutions are responsible for most of the compute
- Optimizing these operators can result in a significant improvement in performance
- Vendor libraries (MKL, cuDNN, etc.) provide optimized hand-written kernels for these operators

Optimizing Convolutions

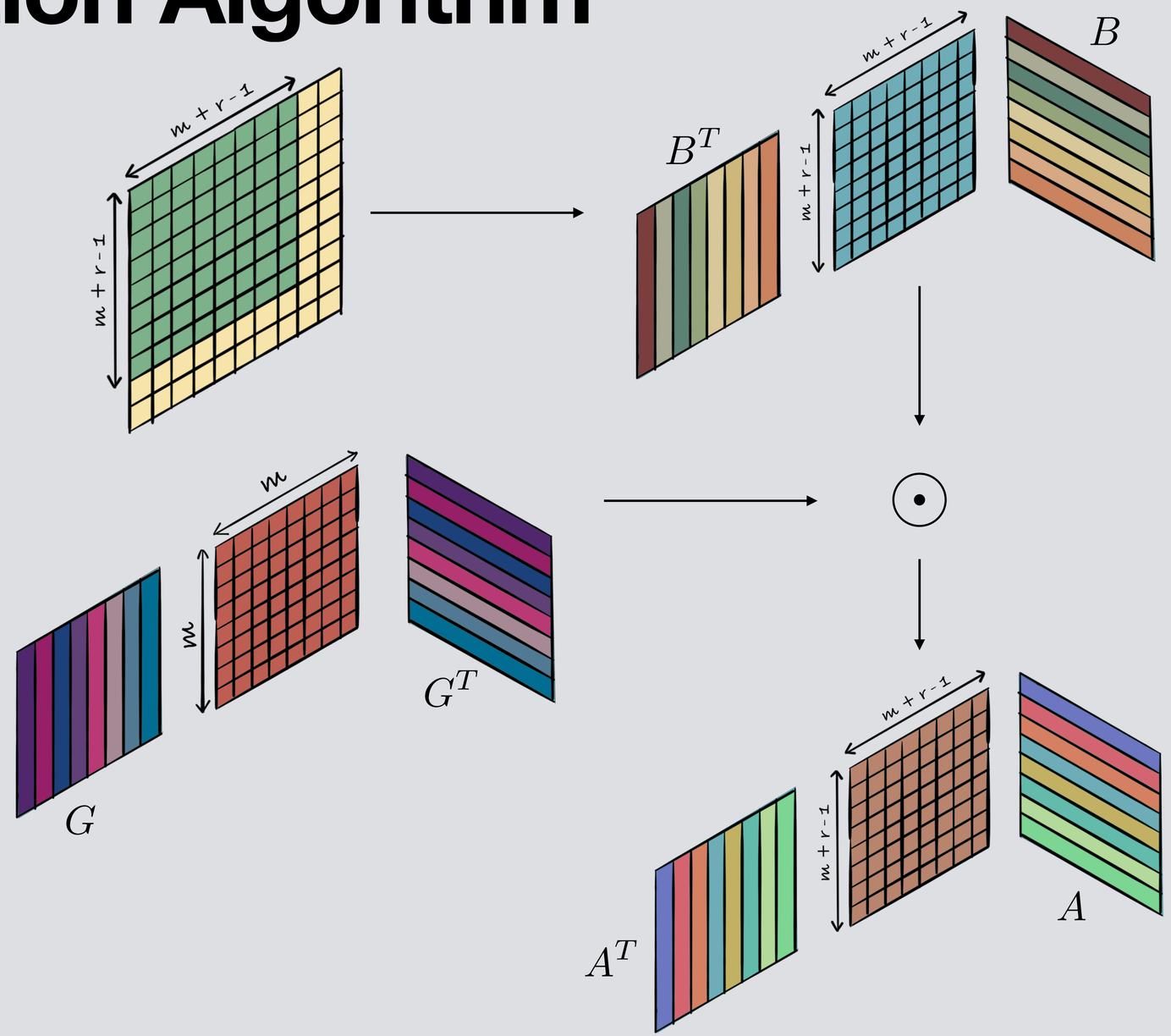
- Convolutions have been the dominant operator in image/video based models from the Resnet-era
- Continue to be used extensively even in state of the art generative models like StableDiffusion etc.
- Unlike matrix multiplication, several different ways to implement convolutions
  - Direct Convolution
  - Implicit GEMM (Generalized matrix-matrix product) (im2col)
  - Winograd Convolution
  - FFT(Fast Fourier Transform) Convolution
- Each approach has its advantages/disadvantages
  - No clear winner, so choice of algorithm depends on problem
- cuDNN has both Winograd and Implicit GEMM
- In this talk, we will focus on Winograd convolutions

Winograd Convolution

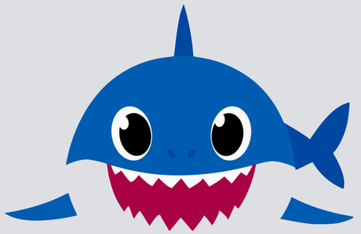
- Introduced by Schmuel Winograd to signal processing in 1980
- Based on Chinese Remainder Theorem
- Transforms kernel and input to the “Winograd domain” where convolution becomes element-wise multiplication, then transforms output back
- Applied to deep learning convolutions by Lavin et al. in 2016
- Computes a output tile of size $r \times r$ for a kernel of size $m \times m$ using a input tile of size $(m + r - 1) \times (m + r - 1)$
- This is referred to as $F(m \times m, r \times r)$
- Compared to the standard algorithm, this approach uses fewer floating point operations to compute the convolution (for example $F(3 \times 3, 2 \times 2)$ has 2.25X lower arithmetic complexity than the standard approach)
- Parameterized by 3 constant matrices B, G, A

Winograd Convolution Algorithm

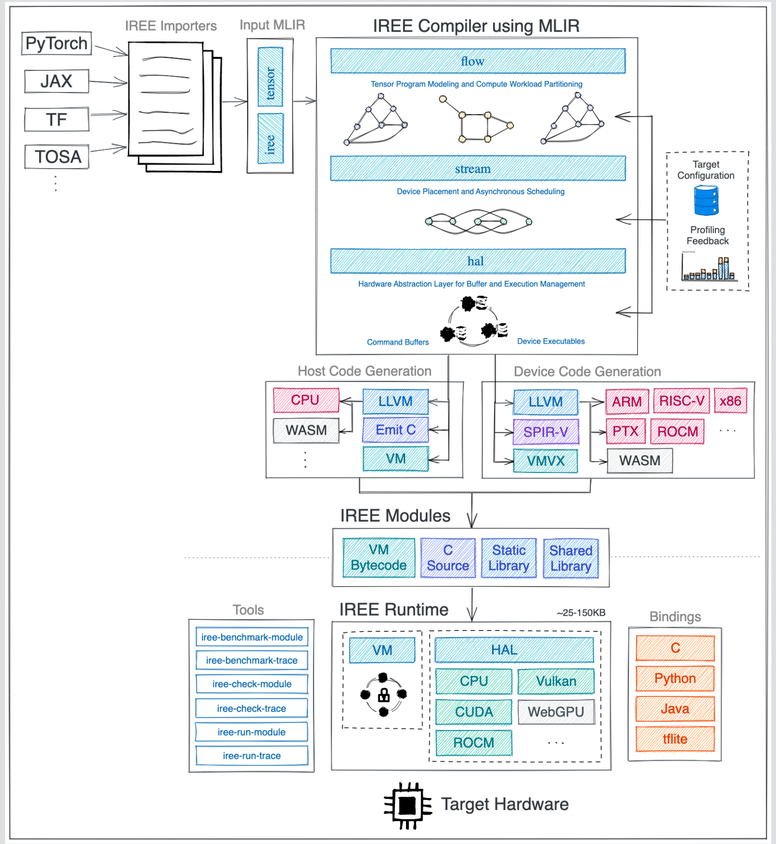
- Input Transformation
  - Compute: $B^T I B$
- Filter Transformation
  - Compute: $G F G^T$
- Element-wise Product
  - Compute: $(B^T I B) \odot (G F G^T)$
- Output Transformation
  - Compute: $A^T((B^T I B) \odot (G F G^T))A$



MLIR Implementation



- Implemented in IREE/SHARK
- IREE is an open-source MLIR-based end-to-end compiler and runtime that lowers ML models for datacenter and edge workloads
- SHARK builds on top of IREE, adds additional performance optimizations, backends for custom accelerators, and contains a fully validated set of hundreds of easy to deploy models
- Supports X86, NVIDIA, AMD, RISC-V, Vulkan and ARM
- Supports Tensorflow, JAX, TFLite, PyTorch
- Will be comparing Winograd approach to direct convolution approach implemented in IREE
- Code can be found here: https://github.com/nod-ai/SHARK-Runtime/tree/tiled_winograd



MLIR Representation

- Starting with a model expressed in PyTorch, we obtain an MLIR representation using torch-mlir
- IREE further decomposes the computation graph into subgraphs (dispatch regions)
- Consider a dispatch with convolution in it as shown in the IR below as our starting point
- We simulate the filter using the unfoldable constant operator in the Util Dialect for the inference use-case
- We assume the input is in NHWC format and the kernel is in HWCF format

```
module {
  func.func @conv(%arg0: tensor<2x10x10x1280xf32>) ->
tensor<2x8x8x1280xf32> {
    %cst = arith.constant 0.000000e+00 : f32
    %0 = tensor.empty() : tensor<2x8x8x1280xf32>
    %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<2x8x8x1280xf32>)
-> tensor<2x8x8x1280xf32>
    %2 = util.unfoldable_constant dense<1.000000e-01> :
tensor<3x3x1280x1280xf32>
    %3 = linalg.conv_2d_nhwc_hwcf {dilations = dense<1> :
tensor<2xi64>, strides = dense<1> : tensor<2xi64>} ins(%arg0, %2 :
tensor<2x10x10x1280xf32>, tensor<3x3x1280x1280xf32>) outs(%1 :
tensor<2x8x8x1280xf32>) -> tensor<2x8x8x1280xf32>
    return %3 : tensor<2x8x8x1280xf32>
  }
}
```

MLIR Representation

- We first transform this convolution into a series of operations corresponding to the Winograd transform
- We create custom ops to represent the input transform and output transform
- Element-wise multiplication gets converted to a batch matrix multiplication
- Filter transformation gets folded

```
func.func @conv(%arg0: !hal.buffer_view) -> !hal.buffer_view attributes {iree.abi.stub} {
  %cst = arith.constant 0.000000e+00 : f32
  %cst_0 = arith.constant dense_resource<__elided__> : tensor<64x1280x1280xf32>
  %0 = hal.tensor.import %arg0 : !hal.buffer_view -> tensor<2x10x10x1280xf32>
  %1 = flow.winograd.input_transform %0 : tensor<2x10x10x1280xf32> -> tensor<8x8x2x2x2x1280xf32>
  %2 = util.do_not_optimize(%cst_0) : tensor<64x1280x1280xf32>
  %collapsed = tensor.collapse_shape %1 [[0, 1], [2, 3, 4], [5]] : tensor<8x8x2x2x2x1280xf32> into tensor<64x8x1280xf32>
  %3 = tensor.empty() : tensor<64x8x1280xf32>
  %4 = linalg.fill ins(%cst : f32) outs(%3 : tensor<64x8x1280xf32>) -> tensor<64x8x1280xf32>
  %5 = linalg.batch_matmul ins(%collapsed, %2 : tensor<64x8x1280xf32>, tensor<64x1280x1280xf32>) outs(%4 : tensor<64x8x1280xf32>) -> tensor<64x8x1280xf32>
  %expanded = tensor.expand_shape %5 [[0, 1], [2, 3, 4], [5]] : tensor<64x8x1280xf32> into tensor<8x8x2x2x2x1280xf32>
  %6 = flow.winograd.output_transform %expanded : tensor<8x8x2x2x2x1280xf32> -> tensor<2x12x12x1280xf32>
  %extracted_slice = tensor.extract_slice %6[0, 0, 0, 0] [2, 8, 8, 1280] [1, 1, 1, 1] : tensor<2x12x12x1280xf32> to tensor<2x8x8x1280xf32>
  %7 = hal.tensor.export %extracted_slice : tensor<2x8x8x1280xf32> -> !hal.buffer_view
  return %7 : !hal.buffer_view
}
```

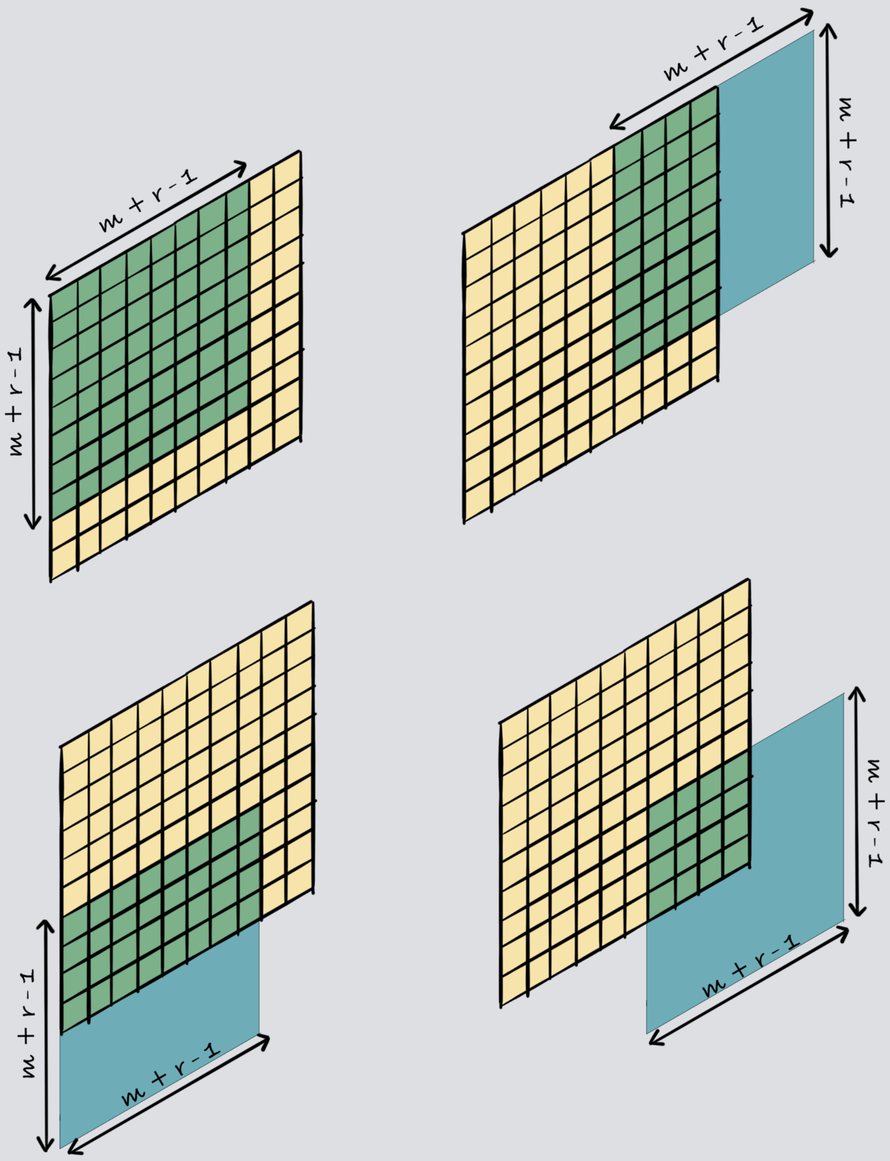
MLIR Representation

- Can we represent the winograd ops as linalg.generics?
- Yes! The double matrix-multiplication can be expressed concisely as $d_{lj} = d_{lj} + b_{il} i_{ik} b_{kj}$
- However, this representation is undesirable because this representation would require special handling due to the high dimensionality of the iteration space (compared to the matrix multiplication and convolution encountered in ML workloads)

```
%4 = linalg.generic {indexing_maps = [affine_map<(d0, d1, d2, d3, d4, d5, d6, d7) -> (d3, d4 * 6 + d6, d5 * 6 + d7, d2)>,
                                      affine_map<(d0, d1, d2, d3, d4, d5, d6, d7) -> (d1, d7)>,
                                      affine_map<(d0, d1, d2, d3, d4, d5, d6, d7) -> (d0, d6)>,
                                      affine_map<(d0, d1, d2, d3, d4, d5, d6, d7) -> (d0, d1, d2, d3, d4, d5)>],
                     iterator_types = ["parallel", "parallel", "parallel", "parallel", "parallel", "parallel", "reduction", "reduction"]}
ins(%padded, %cst, %cst : tensor<2x14x14x1280xf32>, tensor<8x8xf32>, tensor<8x8xf32>) outs(%3 : tensor<8x8x1280x2x2x2xf32>) {
^bb0(%in: f32, %in_4: f32, %in_5: f32, %out: f32):
  %17 = arith.mulf %in, %in_4 : f32
  %18 = arith.mulf %17, %in_5 : f32
  %19 = arith.addf %out, %18 : f32
  linalg.yield %19 : f32
} -> tensor<8x8x1280x2x2x2xf32>
```

MLIR Representation

- Instead, we start with an opaque op and only materialize the op when tiling and distributing to workgroups and threads
- Typically, we want to separate the algorithm from the schedule (one of the key lessons from Halide)
- However, in this case, the algorithm and schedule are intimately related
  - The matrices B, G, A are defined only for very specific tile sizes
- Next, let's look at the tiled version of the IR for the input and output transform



Winograd Input Transform

```
  %workgroup_id_x = hal.interface.workgroup.id[0] : index
  %workgroup_count_x = hal.interface.workgroup.count[0] : index
  %3 = affine.apply affine_map<()[s0] -> (s0 * 32)>()[%workgroup_id_x]
  %4 = affine.apply affine_map<()[s0] -> (s0 * 32)>()[%workgroup_count_x]
  scf.for %arg0 = %3 to %c1280 step %4 {
    %5 = flow.dispatch.tensor.load %1, offsets = [0, 0, 0, %arg0], sizes = [2, 10, 10, 32], strides = [1, 1, 1, 1] : !flow.dispatch.tensor<readonly:2x10x10x1280xf32> ->
tensor<2x10x10x32xf32>
    %6 = flow.dispatch.tensor.load %2, offsets = [0, 0, 0, 0, 0, %arg0], sizes = [8, 8, 2, 2, 2, 32], strides = [1, 1, 1, 1, 1, 1] : !flow.dispatch.tensor<writeonly:8x8x2x2x2x1280xf32> ->
tensor<8x8x2x2x2x32xf32>
    %7 = scf.for %arg1 = %c0 to %c10 step %c6 iter_args(%arg2 = %6) -> (tensor<8x8x2x2x2x32xf32>) {
      %8 = affine.min affine_map<(d0) -> (-d0 + 10, 8)>(%arg1)
      %9 = affine.apply affine_map<(d0) -> (d0 floordiv 6)>(%arg1)
      %10 = scf.for %arg3 = %c0 to %c10 step %c6 iter_args(%arg4 = %arg2) -> (tensor<8x8x2x2x2x32xf32>) {
        %11 = affine.min affine_map<(d0) -> (-d0 + 10, 8)>(%arg3)
        %12 = affine.apply affine_map<(d0) -> (d0 floordiv 6)>(%arg3)
        %13 = scf.for %arg5 = %c0 to %c32 step %c1 iter_args(%arg6 = %arg4) -> (tensor<8x8x2x2x2x32xf32>) {
          %14 = scf.for %arg7 = %c0 to %c2 step %c1 iter_args(%arg8 = %arg6) -> (tensor<8x8x2x2x2x32xf32>) {
            %extracted_slice = tensor.extract_slice %5[%arg7, %arg1, %arg3, %arg5] [1, %8, %11, 1] [1, 1, 1, 1] : tensor<2x10x10x32xf32> to tensor<?x?xf32>
            %15 = linalg.fill ins(%cst : f32) outs(%0 : tensor<8x8xf32>) -> tensor<8x8xf32>
            %inserted_slice = tensor.insert_slice %extracted_slice into %15[0, 0] [%8, %11] [1, 1] : tensor<?x?xf32> into tensor<8x8xf32>
            %extracted_slice_2 = tensor.extract_slice %arg8[0, 0, %arg7, %9, %12, %arg5] [8, 8, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1] : tensor<8x8x2x2x2x32xf32> to tensor<8x8xf32>
            %16 = linalg.fill ins(%cst : f32) outs(%extracted_slice_2 : tensor<8x8xf32>) -> tensor<8x8xf32>
            %17 = linalg.matmul ins(%inserted_slice, %cst_1 : tensor<8x8xf32>, tensor<8x8xf32>) outs(%16 : tensor<8x8xf32>) -> tensor<8x8xf32>
            %18 = linalg.fill ins(%cst : f32) outs(%extracted_slice_2 : tensor<8x8xf32>) -> tensor<8x8xf32>
            %19 = linalg.matmul ins(%cst_0, %17 : tensor<8x8xf32>, tensor<8x8xf32>) outs(%18 : tensor<8x8xf32>) -> tensor<8x8xf32>
            %inserted_slice_3 = tensor.insert_slice %19 into %arg8[0, 0, %arg7, %9, %12, %arg5] [8, 8, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1] : tensor<8x8xf32> into tensor<8x8x2x2x2x32xf32>
            scf.yield %inserted_slice_3 : tensor<8x8x2x2x2x32xf32>
          }
          scf.yield %14 : tensor<8x8x2x2x2x32xf32>
        } {iree.spirv.distribute_dim = 0 : index}
        scf.yield %13 : tensor<8x8x2x2x2x32xf32>
      } {iree.spirv.distribute_dim = 1 : index}
      scf.yield %10 : tensor<8x8x2x2x2x32xf32>
    } {iree.spirv.distribute_dim = 2 : index}
    flow.dispatch.tensor.store %7, %2, offsets = [0, 0, 0, 0, 0, %arg0], sizes = [8, 8, 2, 2, 2, 32], strides = [1, 1, 1, 1, 1, 1] : tensor<8x8x2x2x2x32xf32> -> !
flow.dispatch.tensor<writeonly:8x8x2x2x2x1280xf32>
  }
```

Distribute channels across workgroups

Distribute across threads

Input Transform

Winograd Output Transform

```
  %workgroup_id_x = hal.interface.workgroup.id[0] : index
  %workgroup_count_x = hal.interface.workgroup.count[0] : index
  %3 = affine.apply affine_map<()[s0] -> (s0 * 32)>()[%workgroup_id_x]
  %4 = affine.apply affine_map<()[s0] -> (s0 * 32)>()[%workgroup_count_x]
  scf.for %arg0 = %3 to %c1280 step %4 {
    %5 = flow.dispatch.tensor.load %1, offsets = [0, 0, 0, 0, 0, %arg0], sizes = [8, 8, 2, 2, 2, 32], strides = [1, 1, 1, 1, 1, 1] : !flow.dispatch.tensor<readonly:8x8x2x2x2x1280xf32> ->
tensor<8x8x2x2x2x32xf32>
    %6 = flow.dispatch.tensor.load %2, offsets = [0, 0, 0, %arg0], sizes = [2, 12, 12, 32], strides = [1, 1, 1, 1] : !flow.dispatch.tensor<writeonly:2x12x12x1280xf32> ->
tensor<2x12x12x32xf32>
    %7 = scf.for %arg1 = %c0 to %c2 step %c1 iter_args(%arg2 = %6) -> (tensor<2x12x12x32xf32>) {
      %8 = affine.apply affine_map<(d0) -> (d0 * 6)>(%arg1)
      %9 = scf.for %arg3 = %c0 to %c2 step %c1 iter_args(%arg4 = %arg2) -> (tensor<2x12x12x32xf32>) {
        %10 = affine.apply affine_map<(d0) -> (d0 * 6)>(%arg3)
        %11 = scf.for %arg5 = %c0 to %c32 step %c1 iter_args(%arg6 = %arg4) -> (tensor<2x12x12x32xf32>) {
          %12 = scf.for %arg7 = %c0 to %c2 step %c1 iter_args(%arg8 = %arg6) -> (tensor<2x12x12x32xf32>) {
            %extracted_slice = tensor.extract_slice %5[0, 0, %arg7, %arg1, %arg3, %arg5] [8, 8, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1] : tensor<8x8x2x2x2x32xf32> to tensor<8x8xf32>
            %extracted_slice_2 = tensor.extract_slice %arg8[%arg7, %8, %10, %arg5] [1, 6, 6, 1] [1, 1, 1, 1] : tensor<2x12x12x32xf32> to tensor<6x6xf32>
            %13 = linalg.fill ins(%cst : f32) outs(%0 : tensor<8x6xf32>) -> tensor<8x6xf32>
            %14 = linalg.matmul ins(%extracted_slice, %cst_1 : tensor<8x8xf32>, tensor<8x6xf32>) outs(%13 : tensor<8x6xf32>) -> tensor<8x6xf32>
            %15 = linalg.fill ins(%cst : f32) outs(%extracted_slice_2 : tensor<6x6xf32>) -> tensor<6x6xf32>
            %16 = linalg.matmul ins(%cst_0, %14 : tensor<6x8xf32>, tensor<8x6xf32>) outs(%15 : tensor<6x6xf32>) -> tensor<6x6xf32>
            %inserted_slice = tensor.insert_slice %16 into %arg8[%arg7, %8, %10, %arg5] [1, 6, 6, 1] [1, 1, 1, 1] : tensor<6x6xf32> into tensor<2x12x12x32xf32>
            scf.yield %inserted_slice : tensor<2x12x12x32xf32>
          }
          scf.yield %12 : tensor<2x12x12x32xf32>
        } {iree.spirv.distribute_dim = 0 : index}
        scf.yield %11 : tensor<2x12x12x32xf32>
      } {iree.spirv.distribute_dim = 1 : index}
      scf.yield %9 : tensor<2x12x12x32xf32>
    } {iree.spirv.distribute_dim = 2 : index}
    flow.dispatch.tensor.store %7, %2, offsets = [0, 0, 0, %arg0], sizes = [2, 12, 12, 32], strides = [1, 1, 1, 1] : tensor<2x12x12x32xf32> -> !
flow.dispatch.tensor<writeonly:2x12x12x1280xf32>
  }
```

Distribute channels across workgroups

Distribute across threads

Output Transform

MLIR Code Generation Strategy

- Next, we vectorize, bufferize and then convert to SPIR-V for targeting AMD GPU

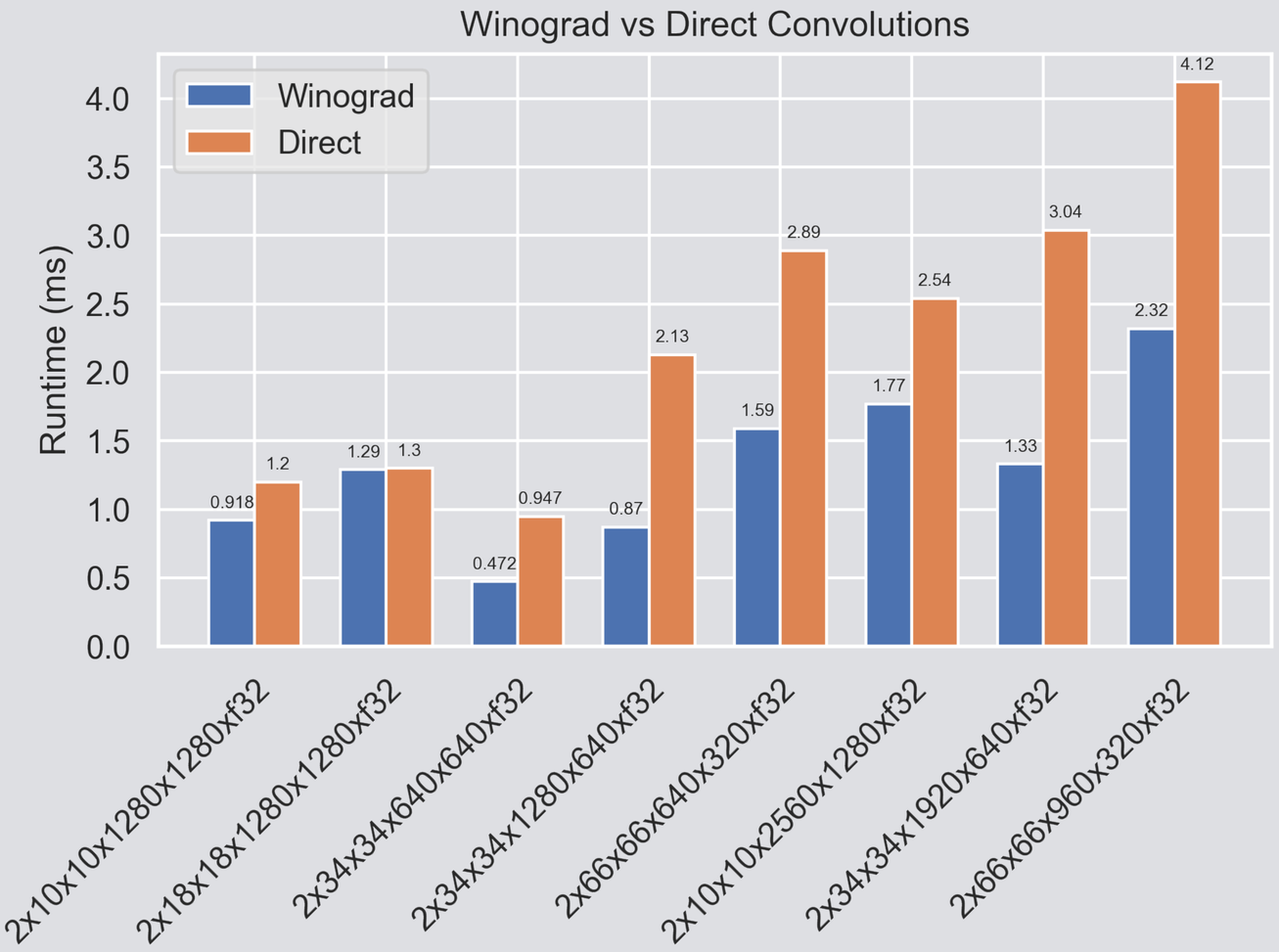
```
%workgroup_id_x = hal.interface.workgroup.id[0] : index
%subview = memref.subview %2[0, 0, 0, %4] [2, 10, 10, 32] [1, 1, 1, 1] : memref<2x10x10x1280xf32> to memref<2x10x10x32xf32, strided<[128000, 12800, 1280, 1], offset: ?>>
%subview_3 = memref.subview %3[0, 0, 0, 0, 0, %4] [8, 8, 2, 2, 2, 32] [1, 1, 1, 1, 1, 1] : memref<8x8x2x2x2x1280xf32> to memref<8x8x2x2x2x32xf32,
                                                                                                                          strided<[81920, 10240, 5120, 2560, 1280, 1], offset: ?>>
%5 = gpu.thread_id  z
%9 = gpu.thread_id  y
%13 = gpu.thread_id  x
...
scf.for %arg0 = %c0 to %c2 step %c1 {
  %subview_4 = memref.subview %subview[%arg0, %6, %10, %13] [1, %7, %11, 1] [1, 1, 1, 1] : memref<2x10x10x32xf32, strided<[128000, 12800, 1280, 1], offset: ?>> to memref<?x?xf32,
strided<[12800, 1280], offset: ?>>
  vector.transfer_write %cst_0, %alloca[%c0, %c0] {in_bounds = [true]} : vector<4xf32>, memref<8x8xf32, 6>
  vector.transfer_write %cst_0, %alloca[%c0, %c4] {in_bounds = [true]} : vector<4xf32>, memref<8x8xf32, 6>
  ...
  %81 = vector.extract %16[1] : vector<4xf32>
  %82 = vector.splat %81 : vector<4xf32>
  %83 = vector.fma %82, %32, %80 : vector<4xf32>
  %84 = vector.extract %16[2] : vector<4xf32>
  %85 = vector.splat %84 : vector<4xf32>
  %86 = vector.fma %85, %34, %83 : vector<4xf32>
  ...
  vector.transfer_write %341, %subview_6[%c0, %c0] {in_bounds = [true]} : vector<4xf32>, memref<8x8xf32, strided<[81920, 10240], offset: ?>>
  vector.transfer_write %349, %subview_6[%c0, %c4] {in_bounds = [true]} : vector<4xf32>, memref<8x8xf32, strided<[81920, 10240], offset: ?>>
```

MLIR Code Generation Strategy

- Batch Matrix Multiplication Operator goes down the existing SPIR-V matrix multiplication pipeline
- One/both of the operands on the RHS are promoted to shared memory for better performance and all memory copies to shared memory are vectorized
- Also applies GPU pipelining to further enhance performance

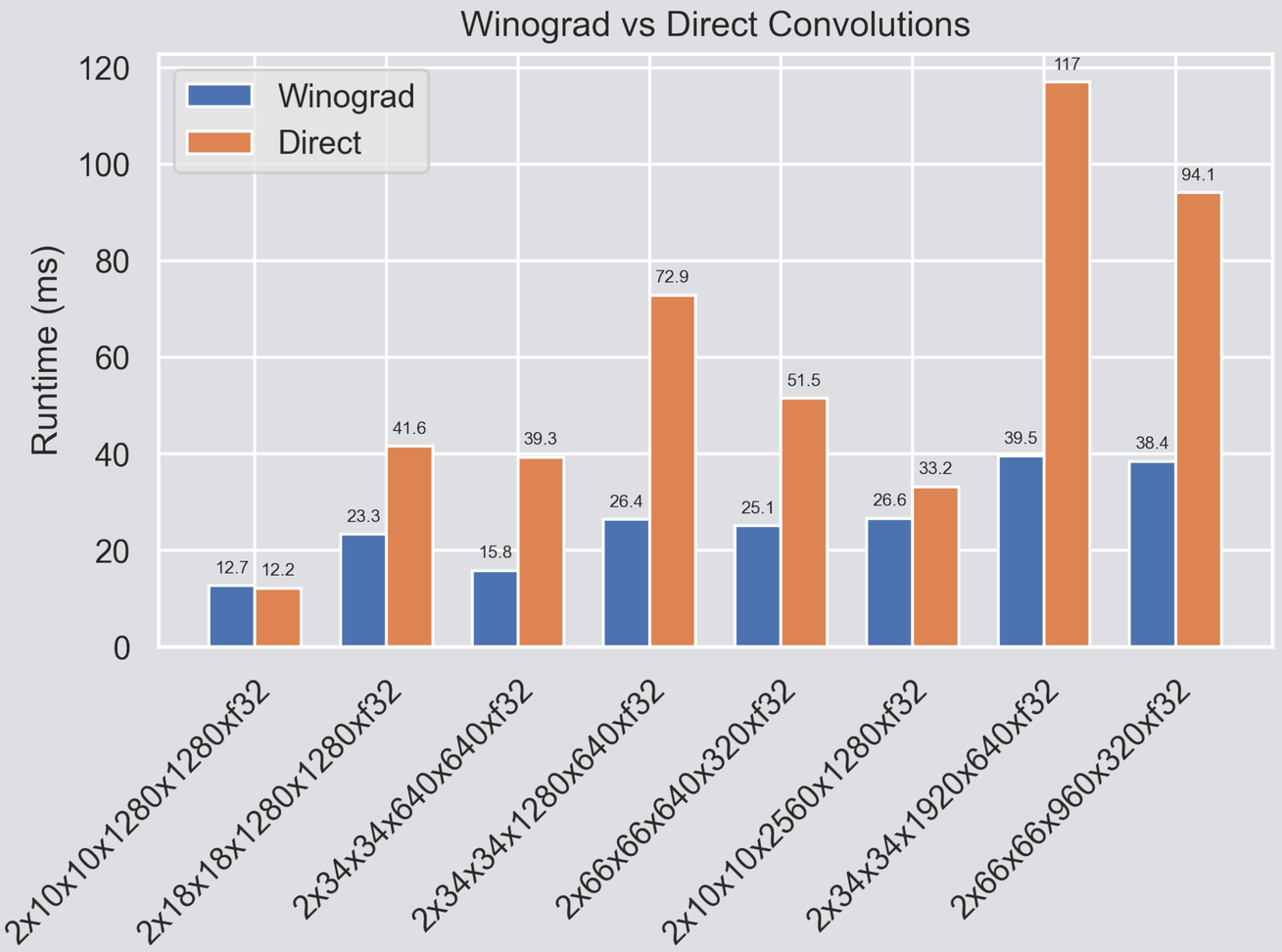
Results on GPU

- Tested on AMD Radeon 6900XT GPU on a variety of sizes found in ML workloads
- Speed-up of up to 2.5X
- Still more improvements possible by adding promotion to shared memory, pipelining etc.



Results on CPU

- Tested on Intel Xeon Platinum 8360Y on a variety of sizes found in ML workloads
- Speedups of up to 3.3X
- Still more improvements possible by addition levels of tiling for additional levels of cache etc.



Conclusions & Future Work

- Winograd algorithm is a viable option for convolutions on both CPU and GPU
- Downsides are kernel size and stride limitations, accuracy degradation with larger output tile size and with lower precisions
- Choice of sampling points used to construct B, G, A extremely important to manage loss of accuracy
- MLIR Representation required expressing algorithm in the tiled form and required mixing the algorithm description with the schedule
- Integrate this into LinalgExt dialect as a series of LinalgExt ops with their own backend specific pass pipeline
- Leverage tensor cores to get even better performance
- Take into account sparse nature of the constant matrices

References

- Liu, J. et al (2021) Optimizing Winograd-Based Convolution with Tensor Cores.
- Alam, S. et al (2022) Winograd Convolution for Deep Neural Networks: Efficient Point Selection
- Barabasz, B. (2022) Improving the Accuracy of the Winograd Convolution for Deep Neural Networks
- Lavin, A. et al (2015) Fast Algorithms for Convolutional Neural Networks
- Shi, F. et al (2018) Sparse Winograd Convolutional neural networks on small-scale systolic array
- Menon, H. (2022) MLIR Code Generation Tutorial