Decomposable Operators in IREE

Harsh Menon (nod.ai)

Overview

- Motivation
- Decomposable Operators
- Case Study #1 : Flash Attention
  - Algorithm Description
  - Code generation Strategy & Results
- Case Study #2 : Winograd Convolutions
  - Algorithm Description
  - Code generation Strategy & Results
- Conclusions & Future Work
- Acknowledgements

Motivation

- Deep learning has seen explosive growth in recent years with more mainstream applications such as NLP-based web search assistants (Chat-GPT), generative art (StableDiffusion), drug discovery and more
- Highly accurate models require enormous amounts of curated data as well as large models with trillions of parameters
- As a result, MLSys problems center around these two pillars of machine learning: data and computation
- Computation is represented as a graph generated using Python-based machine learning framework (PyTorch, JAX, TF etc.)
- Goal is to execute the computation as efficiently as possible given the constraints of the provided hardware
- ML Compilers provide a principled way to achieve this goal

Images from nod.ai SHARK Discord channel

Singular Focus

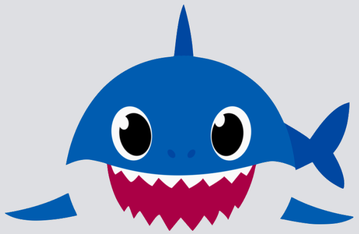
- Majority of the computation in these graphs occurs in matrix multiplications or convolutions
- Most of the effort in ML compilers is spent on optimizing these specific operators using tiling, vectorization, bufferization, fusion with bias adds and activations etc.
- As we approach theoretical peak performance on these operators, a natural question to ask is where are the next set of performance improvements going to come from?
- One possible answer is through decomposable operators, operators that decompose into a sequence of operators

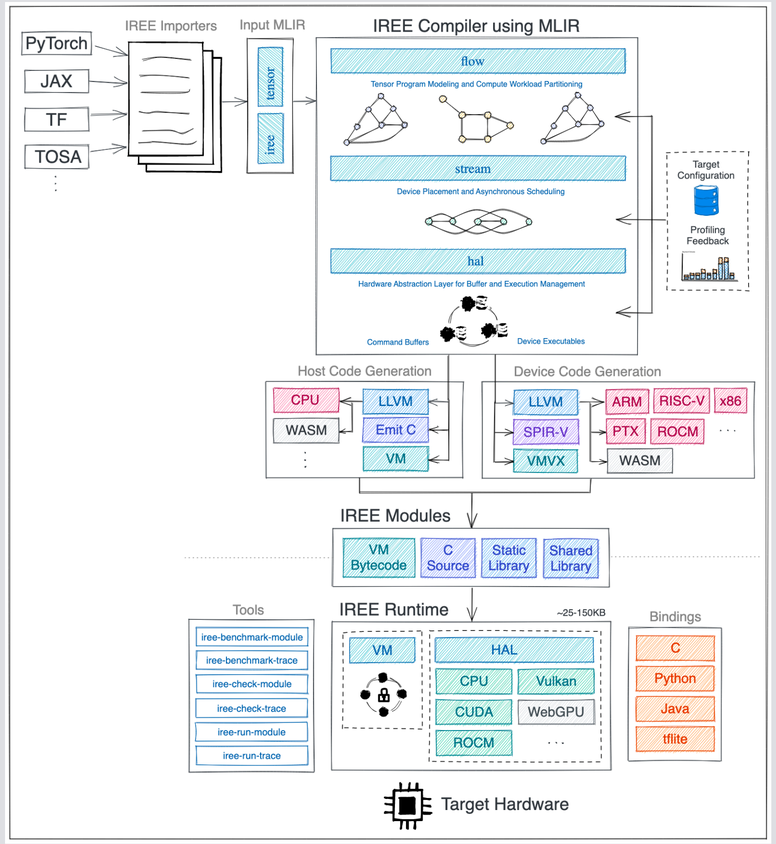
Decomposable Operators

- Decomposable operators are operators that can be broken down into a sequence of operators
- They satisfy the following properties
  - They implement the TilingInterface, though they may expose only a subset of their iteration space to the tiling interface
  - For the remainder of the iteration space, the operators implement TilingConstraints which just enforces a loop-based body with certain constraints
  - The sequence of operators may or may not be fused within a single dispatch region. The choice of whether to do so is based off a cost model
  - They may be parameterized by additional hyperparameters
- Decomposable operators are not new, some have been around for a while, but they have always been treated as a special-case
- Generalization of fusion as there are looser constraints on iteration spaces/iterator types of the constituent operators, but key difference is the need for specific tiling constraints
- Motivate the need for a generalized framework for these operators so that we can leverage code generation approaches across these diverse sets of operators

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



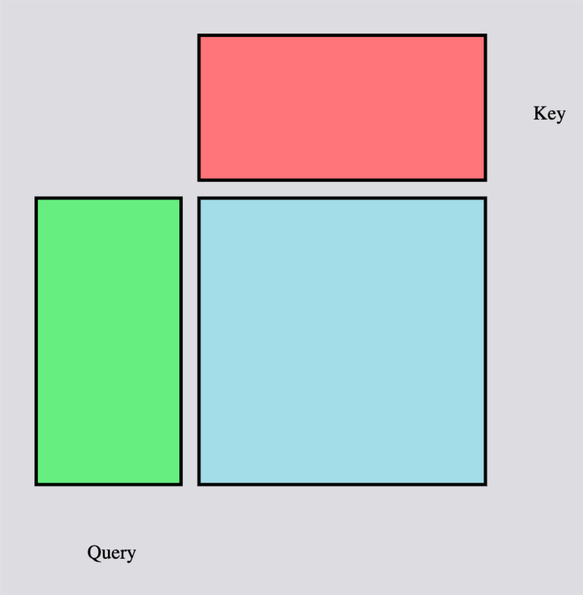


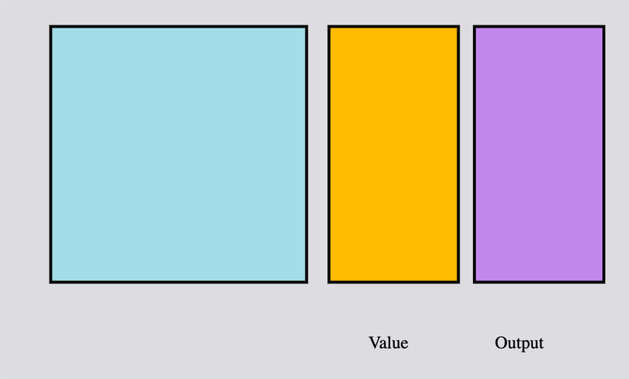
Flash Attention

Naive Attention

Overview

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





$$S = QK^T$$

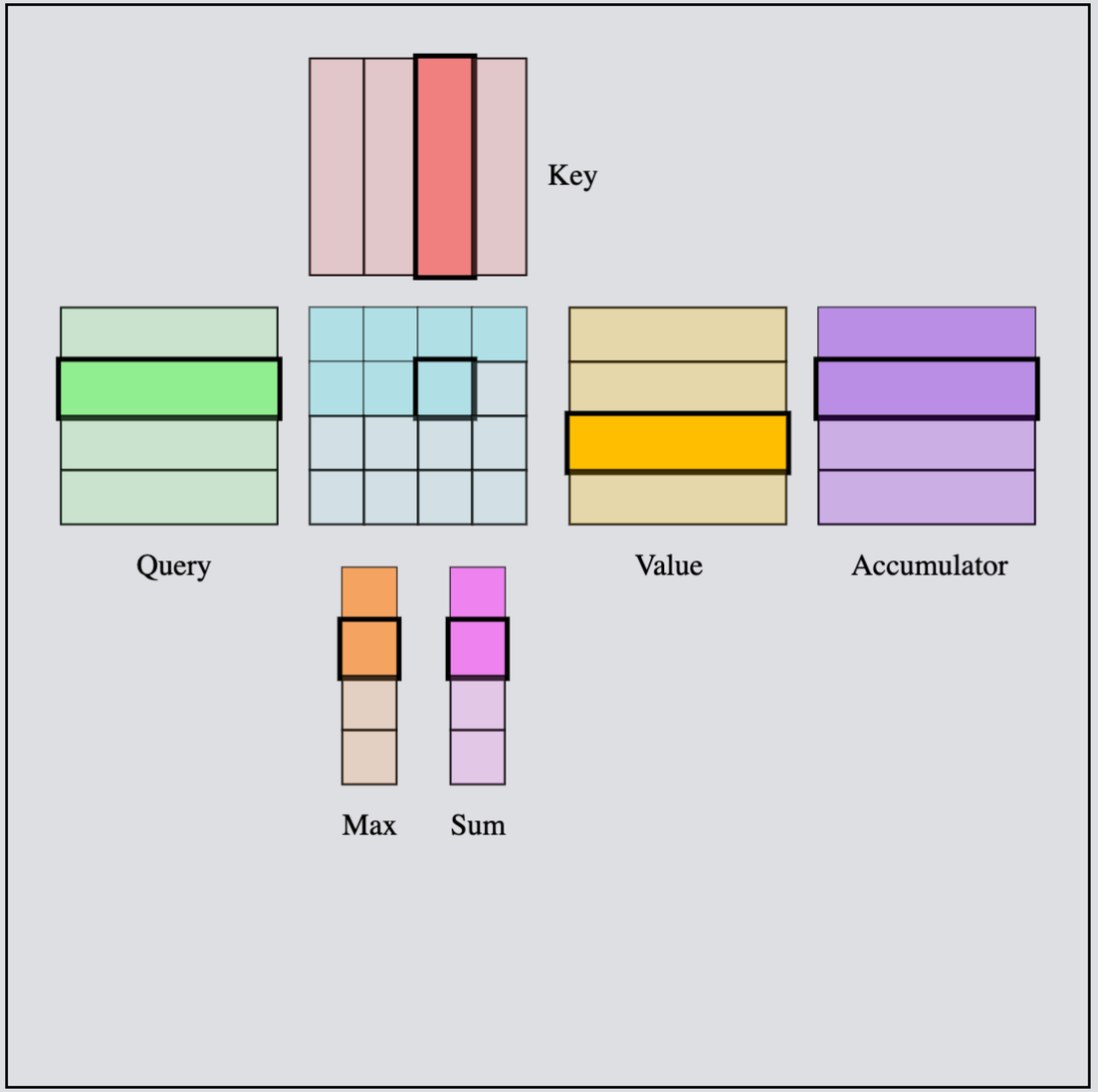
$$P = \mathrm{softmax}(S)$$

$$O = PV$$

Flash Attention Algorithm

Overview

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

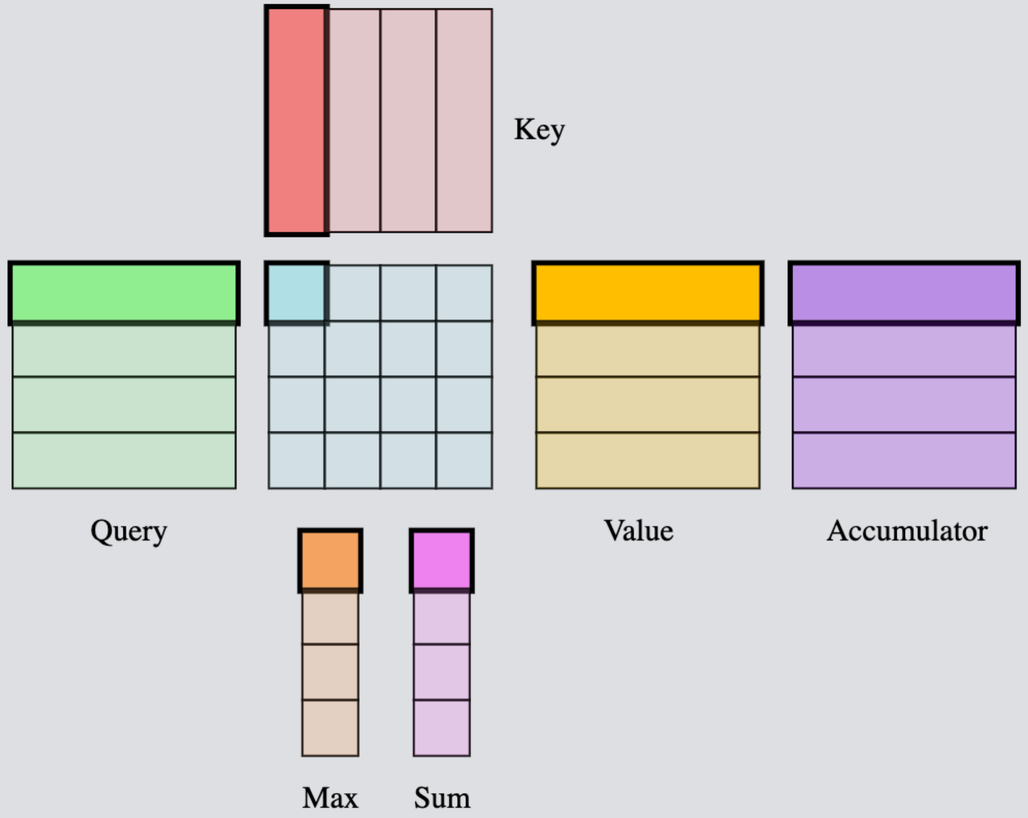


Flash Attention Algorithm

Softmax Algebraic Aggregation

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

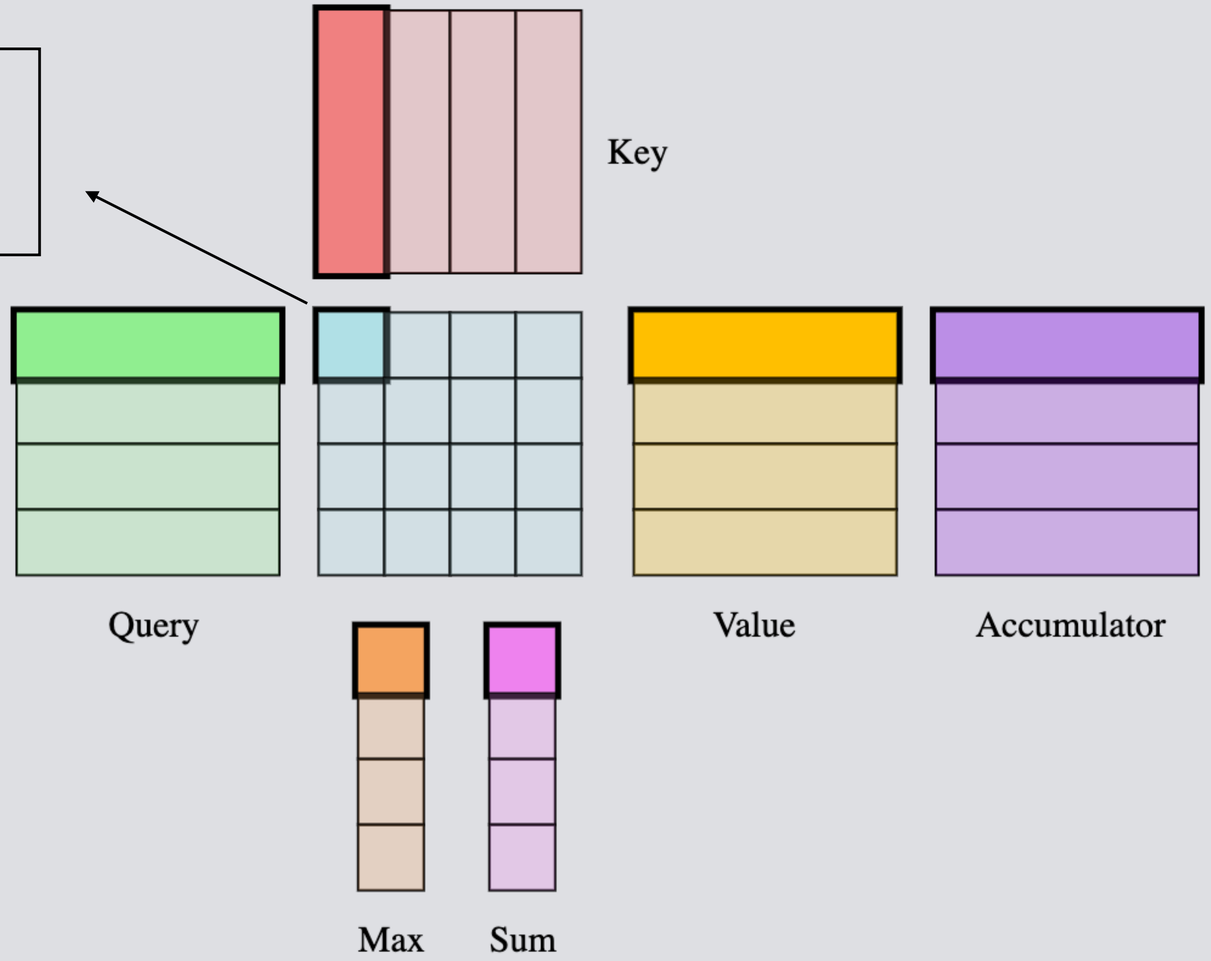
$$
\begin{aligned}
\tilde{m}_{ij} &= \mathrm{rowmax}(S_{ij}) \\
m_i^{new} &= \max(\tilde{m}_{ij}, m_i) \\
\tilde{P}_{ij} &= \exp(S_{ij} - m_i^{new}) \\
\tilde{l}_{ij} &= \mathrm{rowsum}(\tilde{P}_{ij}) \\
l_i^{new} &= \tilde{l}_{ij} + \exp(m_i - m_i^{new}) l_i \\
P_{ij} &= \tilde{P}_{ij} / l_i^{new}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



Flash Attention Algorithm

Softmax Algebraic Aggregation

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

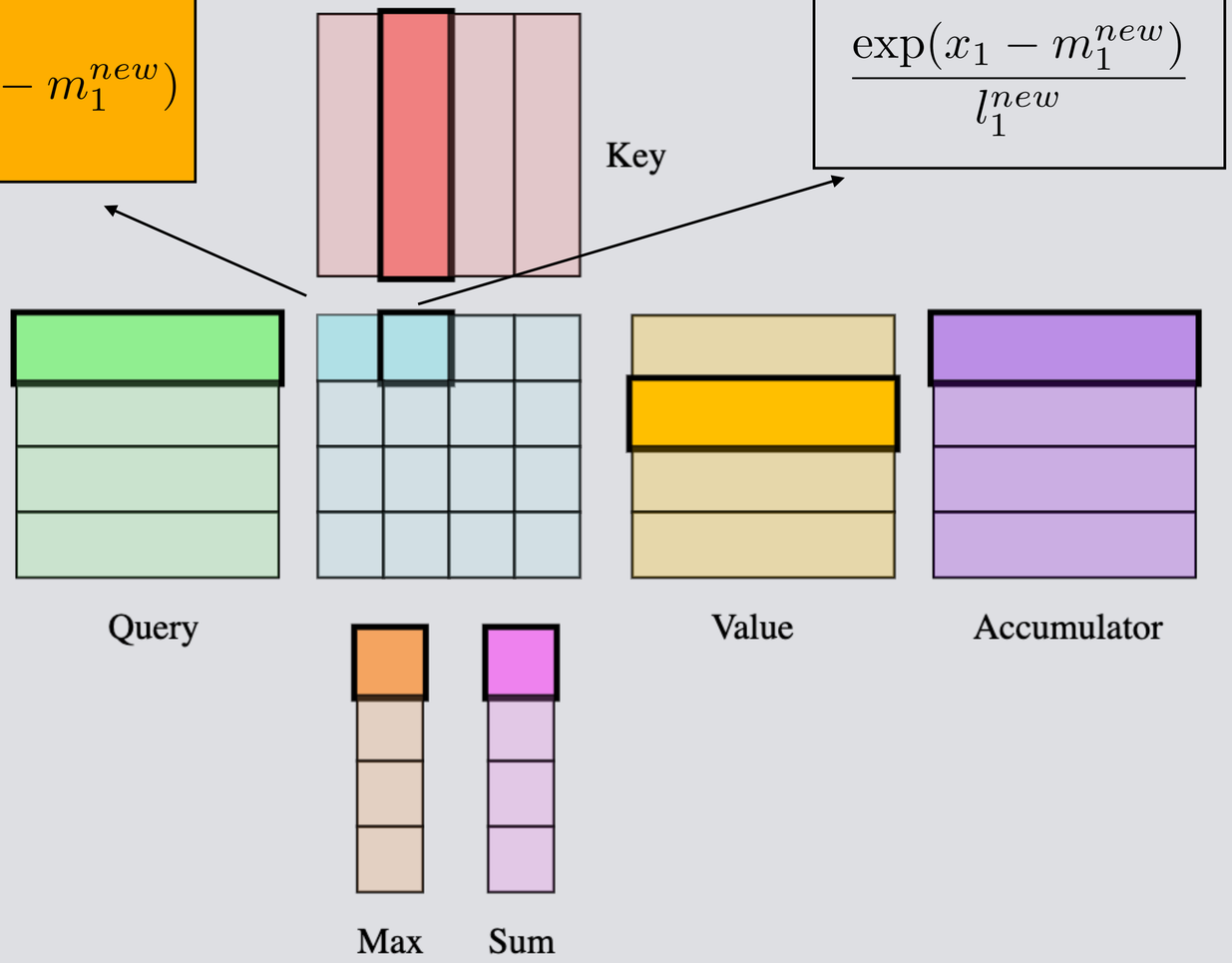


Flash Attention Algorithm

Softmax Algebraic Aggregation

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

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



Flash Attention in MLIR

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

```
func.func @attention(%query: tensor<192x1024x64xf32>,
                     %key: tensor<192x1024x64xf32>,
                     %value: tensor<192x1024x64xf32>) ->
tensor<192x1024x64xf32> {
  %0 = tensor.empty() : tensor<192x1024x64xf32>
  %1 = iree_linalg_ext.attention ins(%query, %key, %value :
tensor<192x1024x64xf32>, tensor<192x1024x64xf32>,
tensor<192x1024x64xf32>) outs(%0 : tensor<192x1024x64xf32>) ->
tensor<192x1024x64xf32>
  return %1 : tensor<192x1024x64xf32>
}
```

Tiled & Decomposed Attention

```
%17 = linalg.matmul_transpose_b ins(… tensor<32x64xf32>, tensor<32x64xf32>) outs(…
tensor<32x32xf32>)
```

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

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

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

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

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

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

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

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

Tiled & Decomposed Attention

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

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

```
%23 = linalg.generic ins(%extracted_slice_7, %20, %21 : tensor<32x64xf32>, tensor<32xf32>,
tensor<32xf32>) outs(%14 : tensor<32x64xf32>) {
        ^bb0(%in: f32, %in_12: f32, %in_13: f32, %out: f32):
          %25 = arith.divf %in_12, %in_13 : f32
          %26 = arith.mulf %25, %in : f32
          linalg.yield %26 : f32} -> tensor<32x64xf32>
```

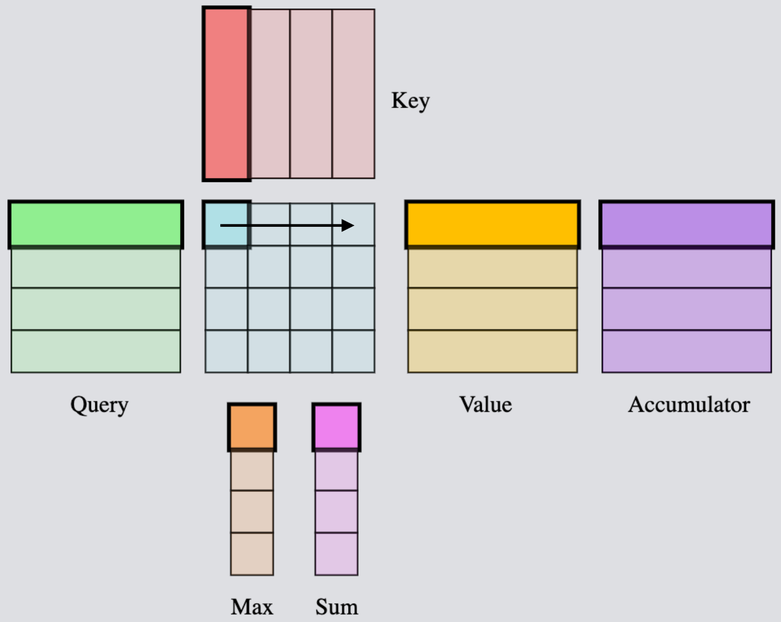
$$A_{ij} = \tilde{A}_{ij} \exp(m_i - m_i^{new}) l_i / l_i^{new}$$

```
%24 = linalg.matmul ins(%22, %extracted_slice_6 : tensor<32x32xf32>, tensor<32x64xf32>)
outs(%23 : tensor<32x64xf32>) -> tensor<32x64xf32>
```

Tiled & Decomposed Attention

Tiling Constraints

- Add a sequential loop that corresponds to computing the attention along a row
- Only load the key and value matrices in this loop and reuse the query and accumulator matrices




```
%15:3 = scf.for %arg3 = %c0 to %c1024 step %c32 iter_args(%arg4 = %extracted_slice_3, %arg5 = %11, %arg6 = %12)
         -> (tensor<1x32x64xf32>, tensor<1x32xf32>, tensor<1x32xf32>) {
            %extracted_slice_5 = tensor.extract_slice %extracted_slice_1[0, %arg3, 0] [1, 32, 64] [1, 1, 1]
            %extracted_slice_6 = tensor.extract_slice %extracted_slice_2[0, %arg3, 0] [1, 32, 64] [1, 1, 1]
            %extracted_slice_7 = tensor.extract_slice %arg4[0, 0, 0] [1, 32, 64] [1, 1, 1] : tensor<1x32x64xf32> to tensor<32x64xf32>
            %extracted_slice_8 = tensor.extract_slice %arg5[0, 0] [1, 32] [1, 1] : tensor<1x32xf32> to tensor<32xf32>
            %extracted_slice_9 = tensor.extract_slice %arg6[0, 0] [1, 32] [1, 1] : tensor<1x32xf32> to tensor<32xf32>
```

Code generation using Transform Dialect

- High-level strategy:
  - Convert both the matrix multiplications to corresponding wmma ops (distribute among warps)
  - Distribute the generics among the threads
  - Distribute all copies to shared memory
  - Hoist any allocations out of the innermost loop
  - Fuse fills with matrix multiplications

Code generation using Transform Dialect

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

    // Get attention op
    // ==========================================
    %attention = transform.structured.match ops{["iree_linalg_ext.attention"]} in %variant_op

    // Tile and distribute to workgroups
    // ==========================================
    %foreach_thread_grid, %tiled_attention =
    transform.iree.tile_to_foreach_thread_and_workgroup_count_region %attention tile_sizes [1, 32]
      ( mapping = [#gpu.block<x>, #gpu.block<y>] )

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

    // Tile and distribute generics
    // ==========================================
    transform.structured.tile_to_foreach_thread_op %update num_threads [32] (mapping = [#gpu.thread<x>])
    transform.structured.tile_to_foreach_thread_op %reduce_sum num_threads [32] (mapping = [#gpu.thread<x>])
    transform.structured.tile_to_foreach_thread_op %partial_softmax num_threads [32] (mapping = [#gpu.thread<x>])
    transform.structured.tile_to_foreach_thread_op %softmax num_threads [32] (mapping = [#gpu.thread<x>])
    transform.structured.tile_to_foreach_thread_op %reduce_max num_threads [32] (mapping = [#gpu.thread<x>])
    transform.structured.tile_to_foreach_thread_op %scale_acc num_threads [32] (mapping = [#gpu.thread<x>])
```

Code generation using Transform Dialect

```
    // Tile first matmul + Fuse fill
    // ==========================================
    %foreach_thread_3, %tiled_matmul = transform.structured.tile_to_foreach_thread_op %first_matmul num_threads [1] ( mapping = [#gpu.warp<x>] )
    transform.structured.fuse_into_containing_op %fill_op into %foreach_thread_3

    // Tile second matmul
    // ==========================================
    %foreach_thread_4, %tiled_matmul_2 = transform.structured.tile_to_foreach_thread_op %second_matmul num_threads [1] ( mapping = [#gpu.warp<x>] )

    // Vectorize function
    // ==========================================
    %func = transform.structured.match ops{["func.func"]} in %variant_op
    %funcx = transform.iree.apply_patterns %func {  rank_reducing_linalg, rank_reducing_vector }
    transform.structured.vectorize %funcx

    // Bufferization
    // ==========================================
    %variant_op_2 = transform.iree.eliminate_empty_tensors %variant_op
    %variant_op_3 = transform.iree.bufferize { target_gpu } %variant_op_2
    %memref_func = transform.structured.match ops{["func.func"]} in %variant_op_3
    transform.iree.erase_hal_descriptor_type_from_memref %memref_func

    // Convert vector to mma
    // ==========================================
    %func2 = transform.structured.match ops{["func.func"]} in %variant_op_3
    transform.iree.vector.vector_to_mma_conversion %func2

    // Map to GPU thread blocks
    // ==========================================
    %func3 = transform.structured.match ops{["func.func"]} in %variant_op_3
    %func4 = transform.iree.foreach_thread_to_workgroup %func3
    %func5 = transform.iree.map_nested_foreach_thread_to_gpu_threads %func4 {workgroup_size = [32, 1, 1]}
}
```

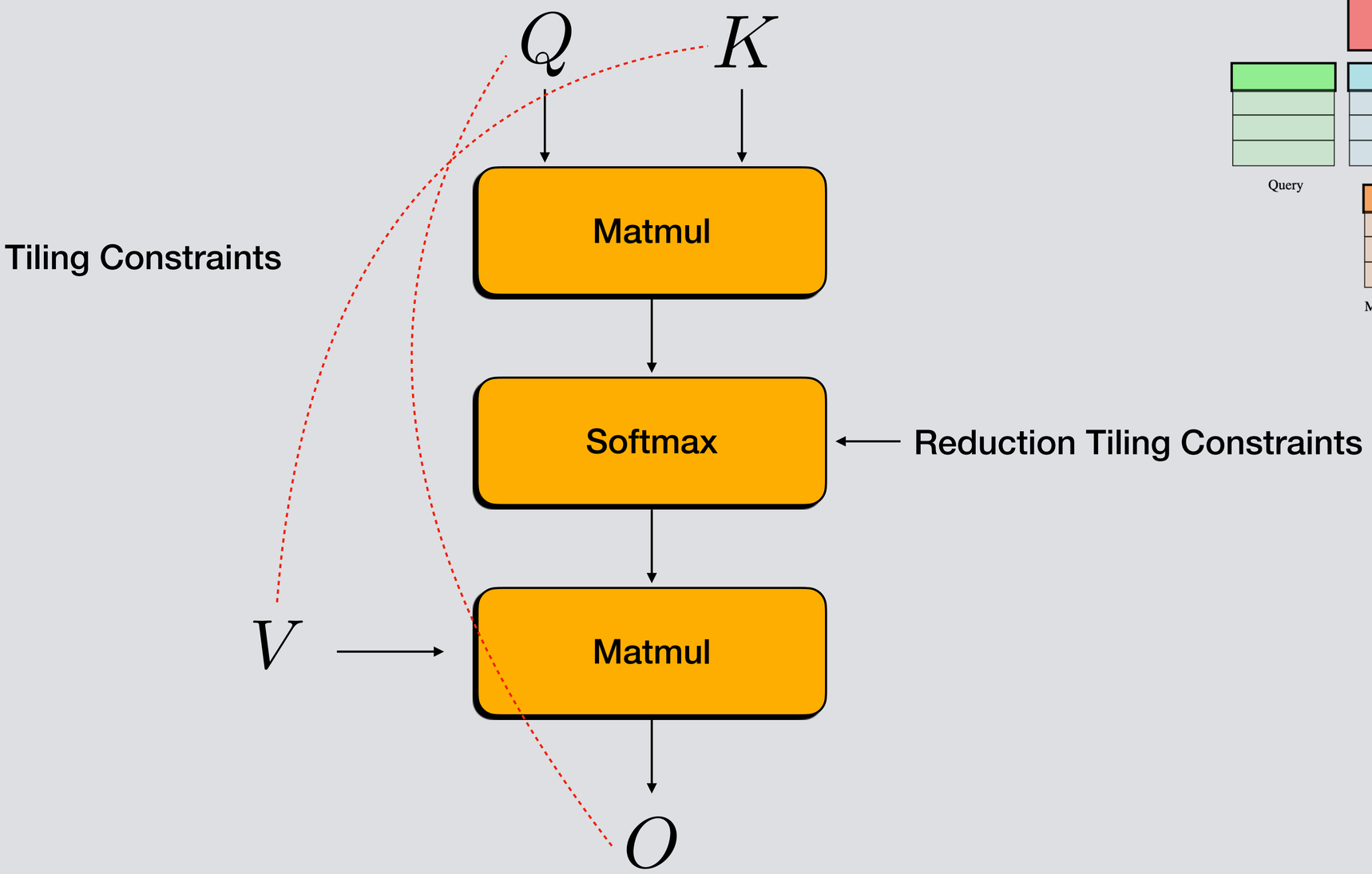
Code generation using Transform Dialect

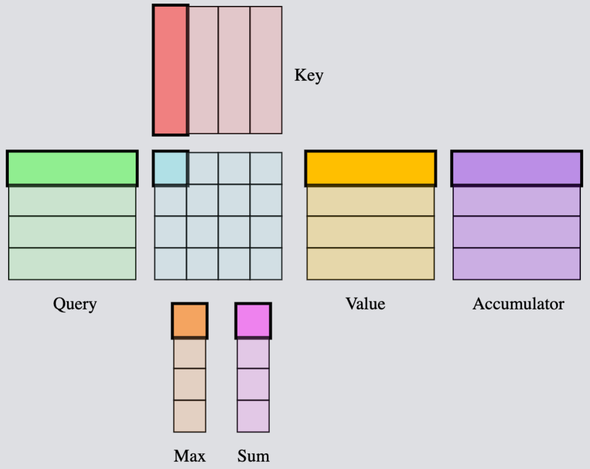
```
scf.for %arg0 = %c0 to %c1024 step %c32 {
      %26 = gpu.subgroup_mma_compute %8, %25, %0 : !gpu.mma_matrix<16x8xf32, "AOp">, !gpu.mma_matrix<8x16xf32, "BOp"> -> !gpu.mma_matrix<16x16xf32, "COp">
      …
      gpu.barrier
      %76 = vector.multi_reduction <maxf>, %73, %75 [0] : vector<32xf32> to f32
      %77 = vector.broadcast %76 : f32 to vector<f32>
      vector.transfer_write %77, %subview_15[] : vector<f32>, memref<f32, strided<[], offset: ?>, #gpu.address_space<workgroup>>
      gpu.barrier
      …
      %81 = arith.subf %80, %79 : vector<32xf32>
      %82 = math.exp %81 : vector<32xf32>
      …
      gpu.barrier
      …
      %89 = arith.subf %84, %86 : f32
      %90 = math.exp %89 : f32
      %91 = arith.mulf %90, %88 : f32
      …
      gpu.barrier
      …
      %96 = vector.multi_reduction <add>, %93, %95 [0] : vector<32xf32> to f32
      …
      gpu.barrier
      …
      %101 = arith.divf %100, %99 : vector<32xf32>
      …
      gpu.barrier

      %107 = arith.divf %104, %106 : vector<64xf32>
      %108 = arith.mulf %107, %102 : vector<64xf32>

      gpu.barrier
      …
      %143 = gpu.subgroup_mma_compute %109, %117, %135 : !gpu.mma_matrix<16x8xf32, "AOp">, !gpu.mma_matrix<8x16xf32, "BOp"> -> !gpu.mma_matrix<16x16xf32, "COp">
```

Decomposable Operators





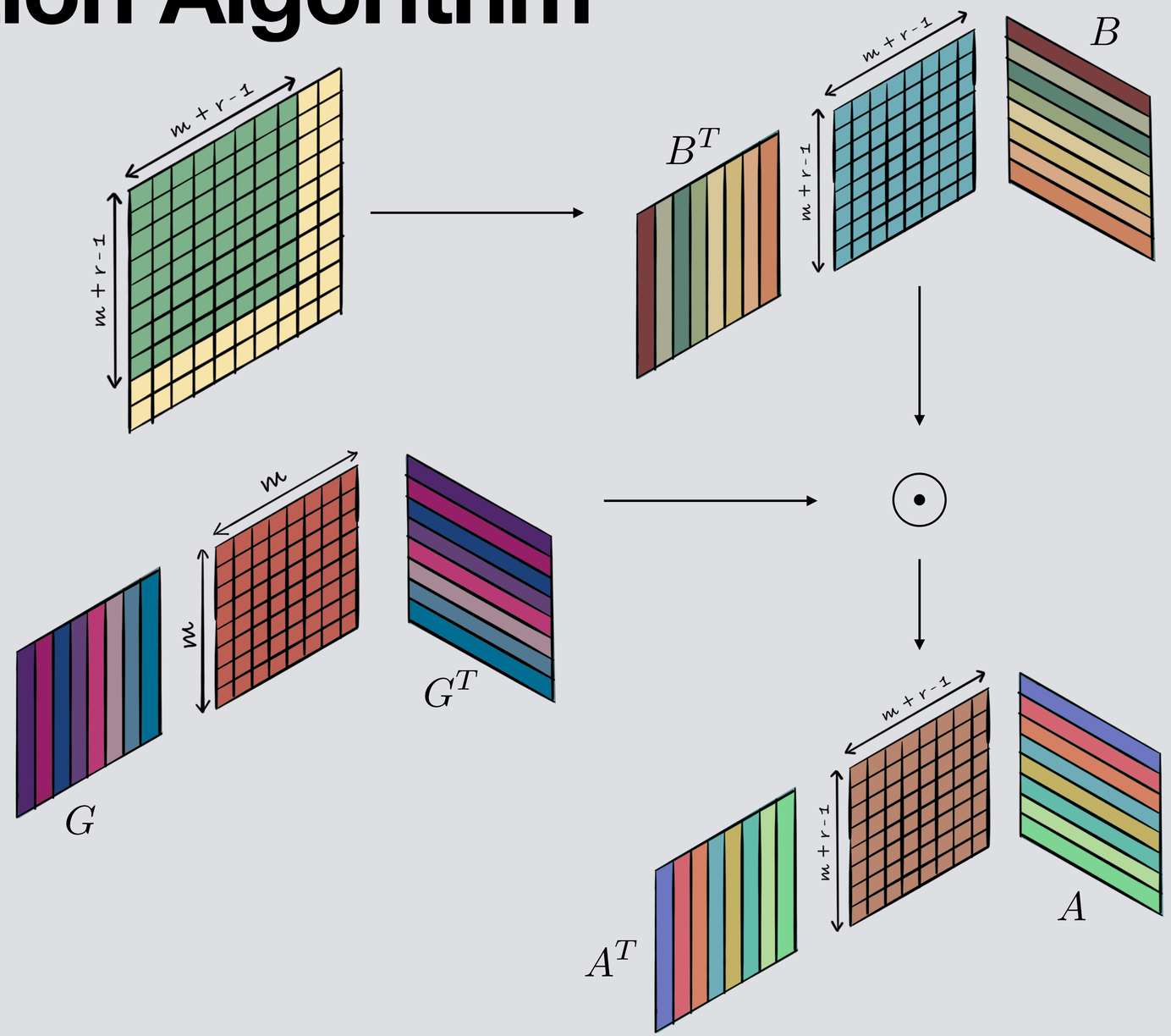
Winograd Convolutions

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 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 assume the input is in NHWC format and the kernel is in HWCF format

```
func.func @conv(%arg0: tensor<2x10x10x1280xf32>) ->
tensor<2x8x8x1280xf32> {
    %cst = arith.constant dense<1.000000e-01> :
tensor<3x3x1280x1280xf32>
    %cst_0 = arith.constant 0.000000e+00 : f32
    %0 = tensor.empty() : tensor<2x8x8x1280xf32>
    %1 = linalg.fill ins(%cst_0 : f32) outs(%0 :
tensor<2x8x8x1280xf32>) -> tensor<2x8x8x1280xf32>
    %2 = linalg.conv_2d_nhwc_hwcf {dilations = dense<1> :
tensor<2xi64>, strides = dense<1> : tensor<2xi64>} ins(%arg0, %cst :
tensor<2x10x10x1280xf32>, tensor<3x3x1280x1280xf32>) outs(%1 :
tensor<2x8x8x1280xf32>) -> tensor<2x8x8x1280xf32>
    return %2 : tensor<2x8x8x1280xf32>
}
```

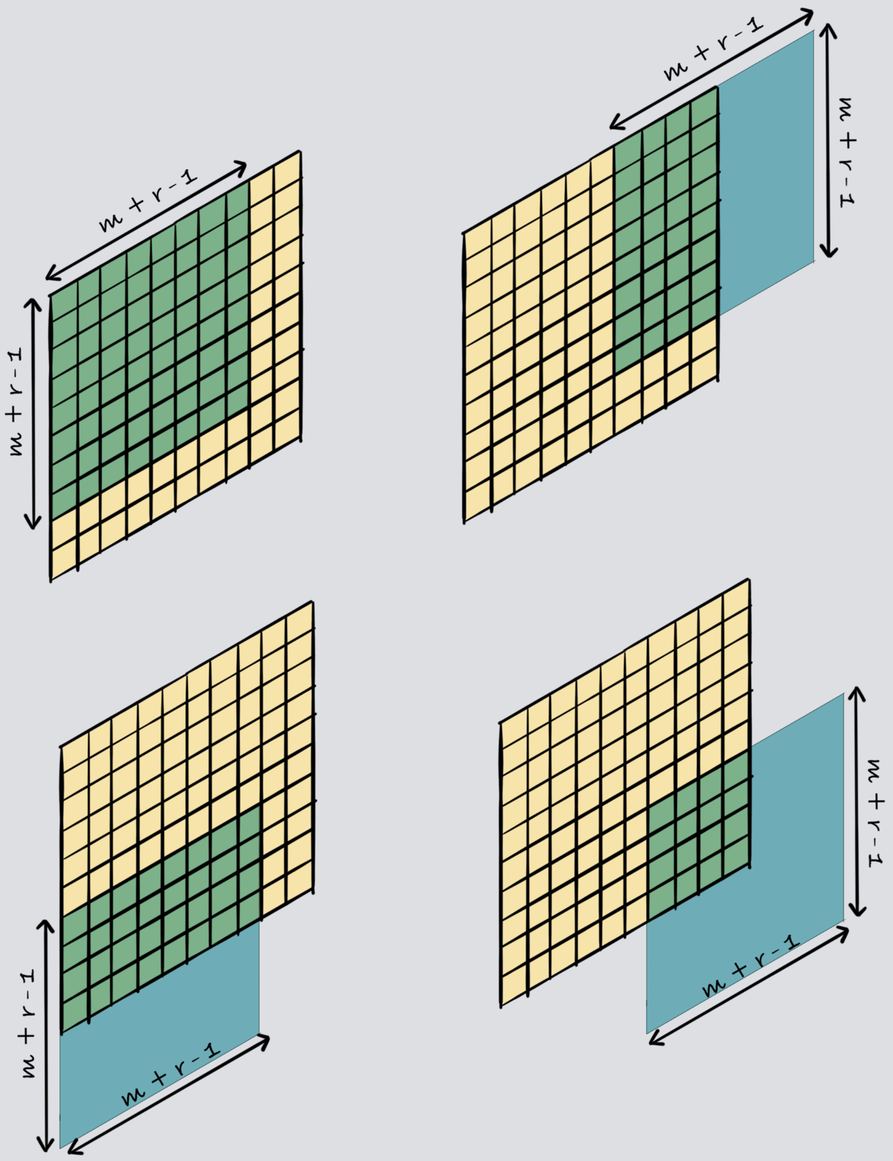
MLIR Representation

- 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: tensor<2x10x10x1280xf32>) -> tensor<2x8x8x1280xf32> {
    %cst = arith.constant dense_resource<__elided__> : tensor<64x1280x1280xf32>
    %cst_0 = arith.constant 0.000000e+00 : f32
    %0 = tensor.empty() : tensor<8x8x2x2x2x1280xf32>
    %1 = iree_linalg_ext.winograd.input_transform output_tile_size(6) kernel_size(3) image_dimensions([1, 2]) ins(%arg0 : tensor<2x10x10x1280xf32>)
outs(%0 : tensor<8x8x2x2x2x1280xf32>) -> tensor<8x8x2x2x2x1280xf32>
    %collapsed = tensor.collapse_shape %1 [[0, 1], [2, 3, 4], [5]] : tensor<8x8x2x2x2x1280xf32> into tensor<64x8x1280xf32>
    %2 = tensor.empty() : tensor<64x8x1280xf32>
    %3 = linalg.fill ins(%cst_0 : f32) outs(%2 : tensor<64x8x1280xf32>) -> tensor<64x8x1280xf32>
    %4 = linalg.batch_matmul ins(%collapsed, %cst : tensor<64x8x1280xf32>, tensor<64x1280x1280xf32>) outs(%3 : tensor<64x8x1280xf32>) ->
tensor<64x8x1280xf32>
    %expanded = tensor.expand_shape %4 [[0, 1], [2, 3, 4], [5]] : tensor<64x8x1280xf32> into tensor<8x8x2x2x2x1280xf32>
    %5 = tensor.empty() : tensor<2x12x12x1280xf32>
    %6 = iree_linalg_ext.winograd.output_transform output_tile_size(6) kernel_size(3) image_dimensions([1, 2]) ins(%expanded :
tensor<8x8x2x2x2x1280xf32>) outs(%5 : tensor<2x12x12x1280xf32>) -> 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>
    return %extracted_slice : tensor<2x8x8x1280xf32>
}
```

Tiled Representation

- The input and output transform ops have very specific tiling constraints
  - 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

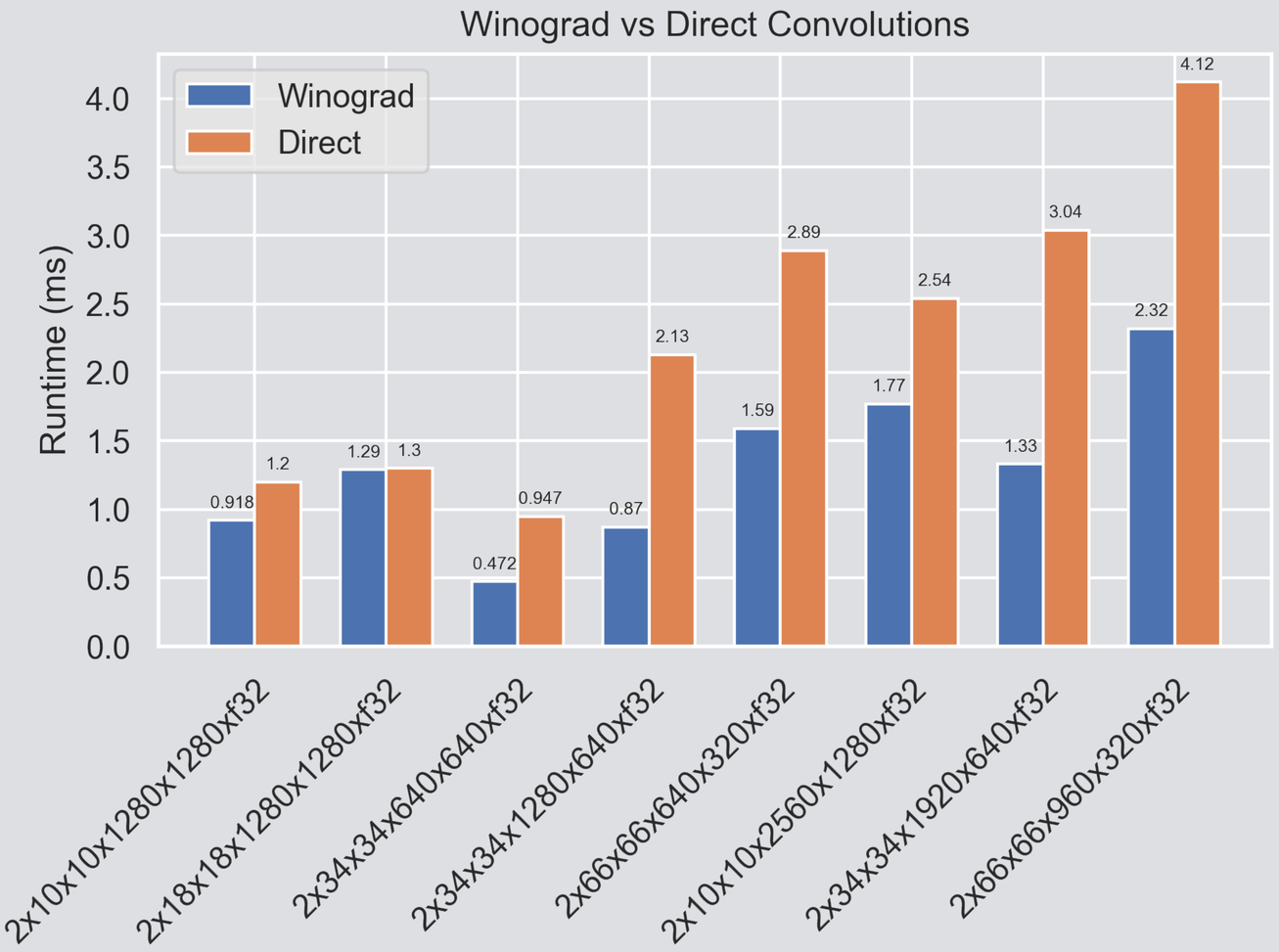
```
%workgroup_id_x = hal.interface.workgroup.id[0] : index
%subview = memref.subview %2[0, 0, 0, %4] [2, 10, 10, 32] [1, 1, 1, 1] : memref<2x10x10x1280xf32> to memref<2x10x10x32xf32, strided<[128000, 12800, 1280, 1], offset: ?>>
%subview_3 = memref.subview %3[0, 0, 0, 0, 0, %4] [8, 8, 2, 2, 2, 32] [1, 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

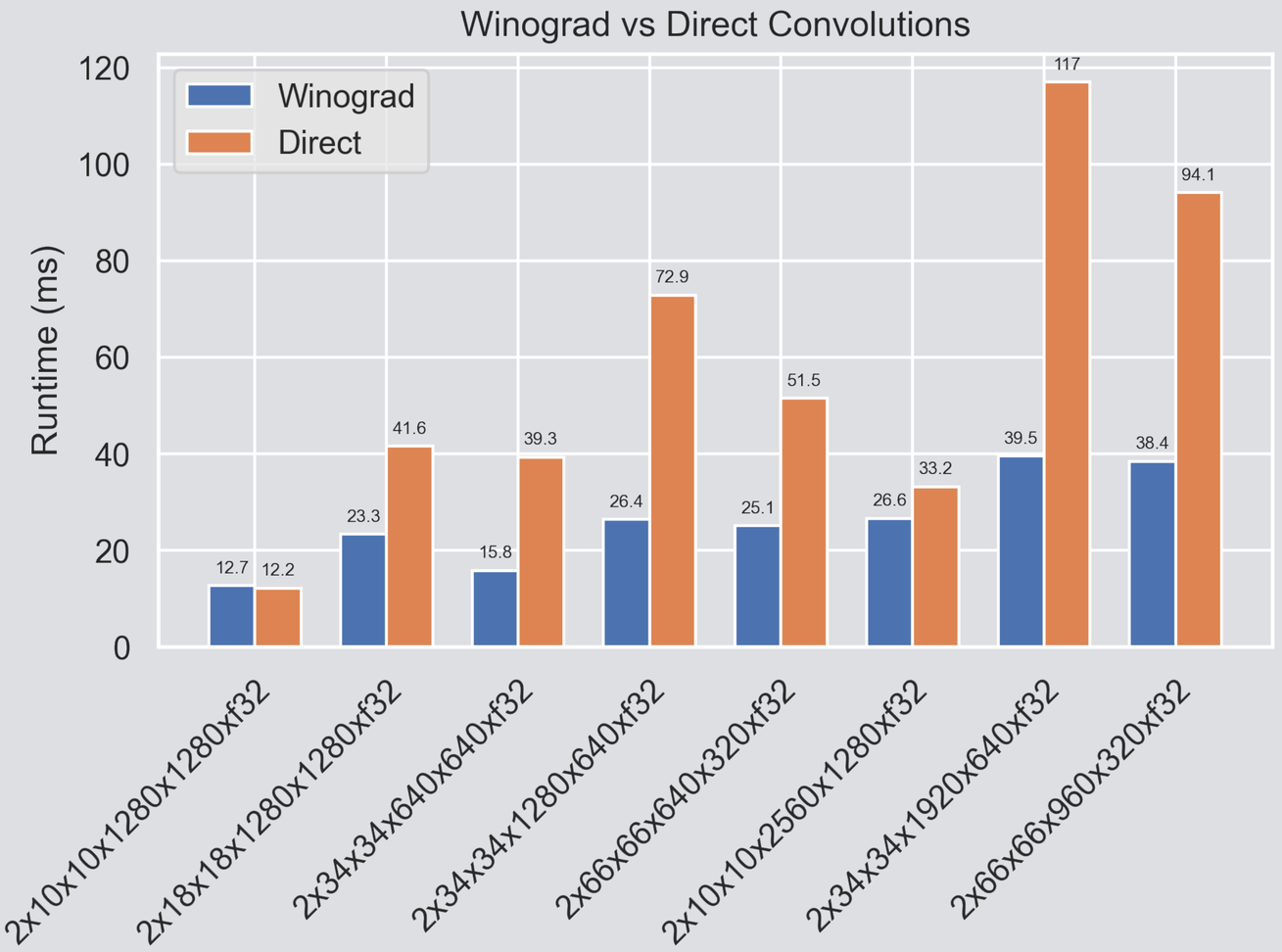
Results on GPU

- 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.

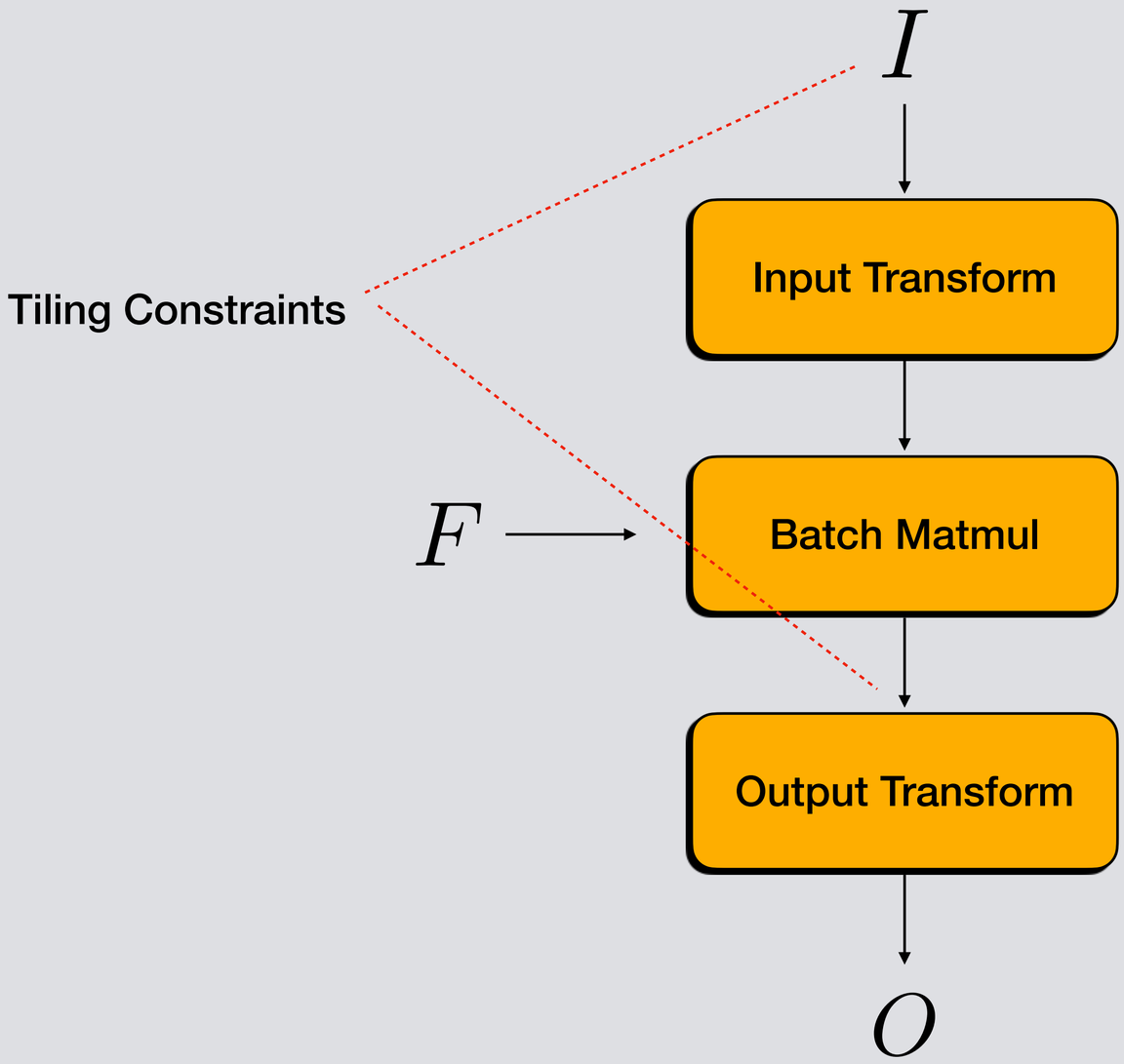


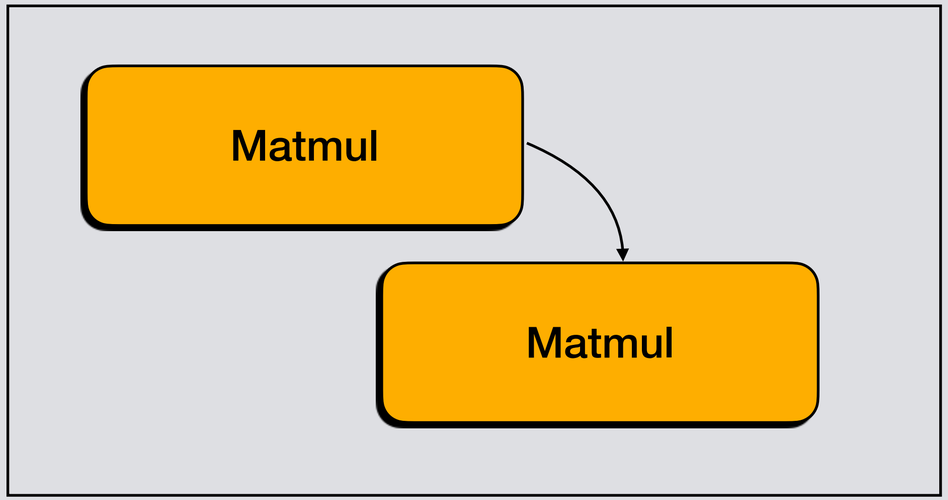
Results on CPU

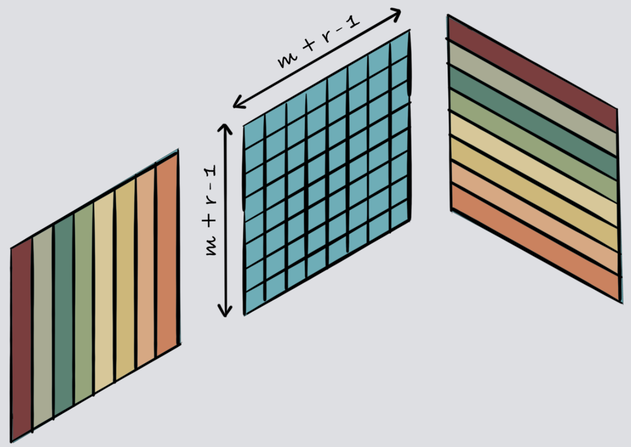
- Tested on Intel Xeon Platinum 8360Y on a variety of sizes found in ML workloads
- Speedups of up to 3.3X
- Still more improvements possible by addition levels of tiling for additional levels of cache etc.



Decomposable Operators







Conclusions & Future Work

- Flash Attention is a more efficient alternative to attention on GPUs due to tiling and fusion
- Winograd convolutions can be more efficient that direct convolution based approaches, only caveat is accuracy needs to be preserved (larger output tile sizes result in higher error)
- Flash Attention & Winograd Convolutions
  - Operators that decompose into other operators
  - Special Tiling Constraints (constraints on how to tile input/output tensors, tile size constraints, reduction tiling)
  - Could benefit from a interface that allows expressing tiling constraints
  - Deduce tile constraints from algorithm with/without annotations
  - Can be extended to add additional constraints such as where the intermediate results need to be stored (shared memory, registers) and this would have implications for layout propagation

Acknowledgements

- Special thanks to Thomas Raoux for help with flash attention
- Thanks to Mahesh Ravishankar & Lei Zhang for help with Winograd convolutions
- Thanks to nod.ai and IREE team for support

References

- Dao, T. et al (2022) FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness
- 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