From MHLO to Linalg on tensors

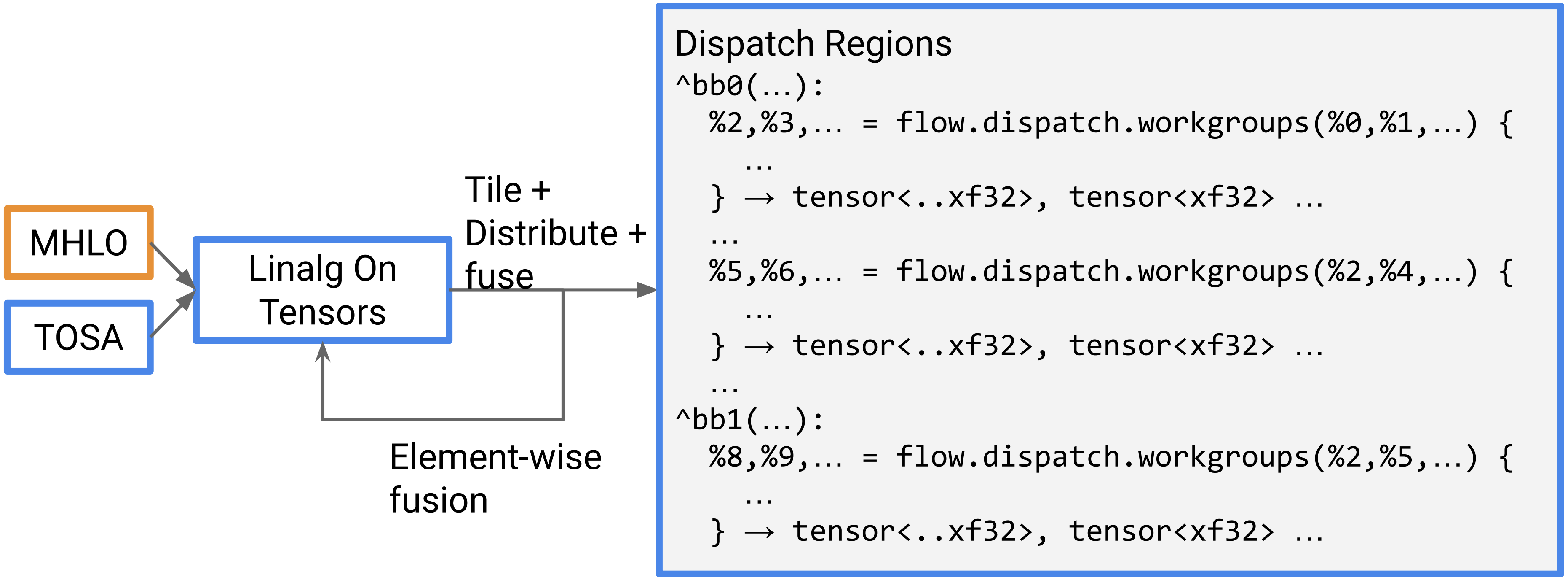
Learnings from transitioning to Linalg on tensors

Overview

- Quick overview of IREEs compilation flow

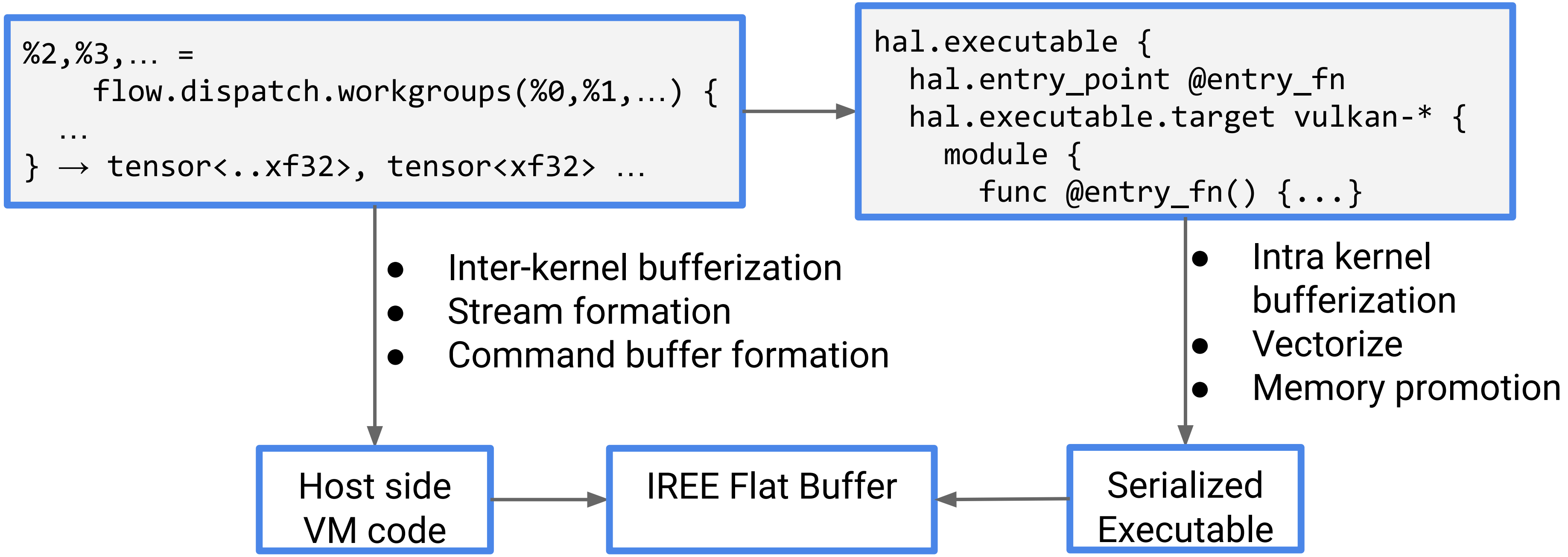
- Representational enhancements to Linalg/IREE that were needed

Current compilation flow from MHLO to Machine code



```
^bb0(...):
  %2,%3,... = flow.dispatch.workgroups(%0,%1,...) {
    ...
  } → tensor<..xf32>, tensor<xf32> ...
  ...
  %5,%6,... = flow.dispatch.workgroups(%2,%4,...) {
    ...
  } → tensor<..xf32>, tensor<xf32> ...
  ...
^bb1(...):
  %8,%9,... = flow.dispatch.workgroups(%2,%5,...) {
    ...
  } → tensor<..xf32>, tensor<xf32> ...
```

Current compilation flow from MHLO to Machine code



```
%2,%3,… =
    flow.dispatch.workgroups(%0,%1,…) {
  …
} → tensor<..xf32>, tensor<xf32> …
```

```
hal.executable {
  hal.entry_point @entry_fn
  hal.executable.target vulkan-* {
    module {
      func @entry_fn() {...}
```

- Inter-kernel bufferization
- Stream formation
- Command buffer formation

- Intra kernel bufferization
- Vectorize
- Memory promotion

Google

Where we started : MHLO as front end

- Dispatch region formation based on rules on MHLO ops
  - Loosely these rules were based on “which operations can be fused”.

```
%3 = “mhlo.dot”(%0, %1)
%4 = “mhlo.broadcast(%2)
%5 = “mhlo.add”(%4, %2)
```

```
%3 = flow.dispatch.region(%a0 = %0, %a1 = %1) {
    %4 = “mhlo.dot”(%a0, %a1) :
}
%4 = flow.dispatch.region(%a0 = %2, %a1 = %3) {
     %5 = “mhlo.broadcast(%a0) :
     %6 = “mhlo.add”(%a1, %4) :
}
```

- Elementwise fusion and tile + fuse run within a dispatch region.
- Suboptimal dispatch region creation (Github Issue)

Google

Where we started : MHLO as front end

- Preferable to have
  - one dispatch region is lowered to one kernel
  - all memory needed for the dispatch to be explicit arguments to the dispatch region
  - Better scheduling and buffer allocation opportunities

```
%res = "mhlo.pad"(%input, %v) {
  edge_padding_low = dense<[0, 1]>,
  edge_padding_high = dense<[1, 5]>
} : (tensor<2x3xi32>, tensor<i32>)
     -> tensor<3x9xi32>
```

```
%linalg.fill(%res, %v) : memref<3x9xi32>
%sv = memref.subview %res
    [0, 1] [2, 3] [1, 1] :
    memref<3x9xi32> into memref<2x4xi32>
linalg.copy(%input, %sv) : memref<2x3xi32>,
```

Google

MHLO as front end : Phase ordering issue

- Grouping at MHLO level needed to guess what the backends could fuse into a single kernel
  - MHLO ops need to know if the Linalg ops can be fused using elementwise fusion or tile + fuse.
  - As we do more fusion using Linalg, needed to map it back to the MHLO operations that could now be grouped together

- Could be different for a backend

Linalg on tensors

- Create Linalg operations at tensor level that represent the final fused kernels

```
%3 = “mhlo.dot”(%0, %1) :
    (tensor<?x?xf32>, tensor<?x?xf32>) -> tensor<?x?xf32>
%4 = “mhlo.broadcast(%2) : tensor<?xf32> -> tensor<?x?xf32>
%5 = “mhlo.add”(%4, %2) :
    (tensor<?x?xf32>, tensor<?x?xf32>) -> tensor<?x?xf32>
```

Conversion from MHLO → Linalg on tensors

```
%cst = constant 0.0 : f32
%3 = linalg.init_tensor [...]
%4 = linalg.fill(%cst, %3)
%5 = linalg.matmul ins(%0, %1 : ) outs(%4 : ) -> …
%6 = linalg.init_tensor [...]
%7 = linalg.generic {
    indexing_maps = [affine_map<(d0, d1) -> (d0), affine_map<(d0, d1) -> (d0, d1)]}
    ins(%2 : ) outs(%6 : ) {...}
%8 = linalg.init_tensor [...]
%9 = linalg.generic {
    indexing_maps = [affine_map(d0, d1) -> (d0, d1), affine_map(d0, d1) -> (d0, d1),
                     affine_map(d0, d1) -> (d0, d1)]}
    ins(%5, %7 : ) outs(%8 : ) {...}
```

Google

After element-wise fusion

```
%cst = constant 0.0 : f32
%3 = linalg.init_tensor [...]
%4 = linalg.fill(%cst, %3)
%5 = linalg.matmul ins(%0, %1 : ) outs(%4 : ) -> …
%8 = linalg.init_tensor [...]
%9 = linalg.generic {
    indexing_maps = [affine_map(d0, d1) -> (d0, d1),
                     affine_map(d0, d1) -> (d0),
                     affine_map(d0, d1) -> (d0, d1)]}
    ins(%5, %2 : ) outs(%8 : ) {...}
```

Tile + Distribute using block-cyclic distribution

```
%id_y = flow.workgroup.workgroup_id[1] // id_y
%wgs_y = flow.workgroup.workgroup_count[1] // number_y
%nwg_y = flow.workgroup.workgroup_size[1] // same as %ts_M
%start = muli %id_y, %wgx_y
%step = muli %nwg_y, %wgs_y
scf.for %iv0 = %start to %M step %step {
    ....
}
```

Tile + fuse + distribute

```
%3 = flow.dispatch.workgroups[...]-> tensor<?x?xf32> {
    %wg_x = flow.dispatch.workgroup_size[0]
    %wg_y = flow.dispatch.workgroup_size[1]
    %y0 = scf.for %iv0 = %start_y to %M step %step_y
      %y1 = scf.for %iv1 = %start_x to %N step %step_x
        %4 = linalg.init_tensor [%wg_y, %wg_x] : tensor<?x?xf32>
        %5 = linalg.fill(..., %4)
        %6 = tensor.extract_slice %0[..,..] [%wg_y, %K] [1, 1] : …
        %7 = tensor.extract_slice %1[..,..] [%K, %wg_x] [1, 1] : …
        %8 = linalg.matmul ins(%6, %7 : ) outs(%5 : )
        %9 = tensor.extract_slice %2[..] [%wg_y] [1]
        %10 = linalg.generic {...} ins(%8, %9 : ) outs(%5) {...}
  }
```

Representational enhancements to Linalg on tensors

- Adapt read-modify-write semantics

- Output Shape representation.

- Tile + distribute on Linalg on tensors.

Detour : Background on Linalg StructuredOps representation.

- Linalg on buffers for matmul

```
linalg.matmul ins(%lhs, %rhs : memref<?x?xf32>, memref<?x?xf32>)
    outs(%result : memref<?x?xf32>)
```

- In loop form

```
scf.for %iv0 = 0 to %m step 1
  scf.for %iv1 = 0 to %n step 1
    scf.for % %iv2 = 0 to %k step 1
      %0 = mulf %lhs[%iv0, %iv2], %rhs[%iv2, %iv1] : f32
      %1 = addf %0, %result[%iv0, %iv1] : f32
      store %1, %result[%iv0, %iv1] : f32, memref<?x?xf32>
```

Google

Detour : Background on Linalg StructuredOps representation.

Named ops are pre-defined `linalg.generic` operations.

```
linalg.generic {
  iterator_types = [“parallel”, “parallel”, “reduction”]} // M, N and K loops
  indexing_maps = {[affine_map<(d0, d1, d2) -> (d0, d2)>, // LHS indexing
                    affine_map<(d0, d1, d2) -> (d2, d1)>, // RHS indexing
                    affine_map<(d0, d1, d2) -> (d0, d1)>] // Output indexing
  ins(%lhs, %rhs : memref<?x?xf32>, memref<?x?xf32>)
  outs(%result : memref<?x?xf32> {
  ^bb0(%arg0: f32, %arg1 : f32, %arg2 : f32) : // Loaded scalar values
    %0 = mulf %arg0, %arg1 : f32
    %1 = addf %0, %arg2 : f32
    linalg.yield %1 : f32
  }
```

Detour : Background on Linalg StructuredOps representation.

Loop bounds are determined based on operand shapes

- `affine_map<(d0, d1, d2) -> (d0, d2)> … ins(%lhs, …)`
  - Outer loop bound : `memref.dim %lhs, 0`
  - Inner loop bound : `memref.dim %lhs, 1`
- `affine_map<(d0, d1, d2) -> (d2, d1)> … ins(.., %rhs)`
  - Inner loop bound : `memref.dim %rhs, 0`
  - Middle loop bound : `memref.dim %rhs, 1`
- `affine_map<(d0, d1, d2) -> (d0, d1)> … outs(%result)`
  - Outer loop bound : `memref.dim %result, 0`
  - Middle loop bound : `memref.dim %result, 1`

Detour : Background on Linalg StructuredOps representation.

Elementwise operations representation (with memref operands)

```
linalg.generic {
  indexing_maps = {[affine_map<(d0, d1) -> (d0, d1)>,
                    affine_map<(d0, d1) -> (d0)>, // Broadcast access
                    affine_map<(d0, d1) -> (d0, d1)>] // Output indexing
  iterator_types = [“parallel”, “parallel”]} // No reduction loops
  ins(%lhs, %rhs : memref<?x?xf32>, memref<?xf32>)
  outs(%result : memref<?x?xf32> {
  ^bb0(%arg0: f32, %arg1 : f32, %arg2 : f32) :
    %0 = addf %arg0, %arg1 : f32
    linalg.yield %0 : f32 // No output reads (only writes)
  }
```

Adapt Read-Modify-Write abstraction

- Tensor are SSA values
  - All elements “defined” simultaneously and not mutable.
- One solution

```
%result = linalg.matmul
    ins(%lhs, %rhs : tensor<?x?xf32>, tensor<?x?xf32>) -> tensor<?x?xf32>
```

  - Different computation than representation in buffer
- Make the initial value an explicit operand

```
%result = linalg.matmul
    ins(%lhs, %rhs : tensor<?x?xf32>, tensor<?x?xf32>),
    outs(%init : tensor<?x?xf32>) -> tensor<?x?xf32>
```

Google

Output shape representation

- Without reduction do not need initial value

```
%result = linalg.generic {
  indexing_maps = [affine_map<(d0, d1) -> (d0)>,
                   affine_map<(d0, d1) -> (d0, d1)>]
  iterator_types = [“parallel”, “parallel”]}
  ins(%source: tensor<?xf32>) -> tensor<?x?xf32> {
  ^bb0(%arg0 : f32):
    linalg.yield %arg0 : f32
  }
```

- Only the result value carries loop bounds of one of the loops

Google

Output shape representation

- Add a “dummy” initial value just for shape information

```
%d0 = … // Some computation generated by the producer of this IR
%d1 = … // Some computation generated by the producer of this IR
%init = linalg.init_tensor [%d0, %d1] : tensor<?x?xf32> // Output shape
%result = linalg.generic {...}
    ins(%source: tensor<?xf32>), outs(%init : tensor<?x?xf32>)
```

- Each operation carries all the information needed to compute the result (including the shape) in its operands
- Dynamic shape code-generation without any external shape information

Detour Tiling with Linalg on memref operands

- Tiling is one of the core transformations in Linalg
  - Tile + fuse is tile the consumer and compute the operands of the tile in-place.

```
linalg.matmul ins(%lhs, %rhs : memref<?x?xf32>, memref<?x?xf32>)
    outs(%result : memref<?x?xf32>
```

- Linalg tiling algorithm uses the indexing maps to generate the tiled implementation
  - Any operation that implements the structured op interface can be tiled (any named ops and linalg.generic)

Google

Detour Tiling with Linalg on memref operands

- Tiling is done using parametric tile sizes.

```
scf.for %iv0 = 0 to %M step %ts_M
  scf.for %iv1 = 0 to %N step %ts_N
    scf.for %iv2 = 0 to %K step %ts_K
      %tile_lhs = memref.subview %lhs[%iv0, %iv2] [%ts_M, %ts_K]
      %tile_rhs = memref.subview %rhs[%iv2, %iv1] [%ts_K, %ts_N]
      %tile_output = memref.subview %result[%iv0, %iv1] [%ts_M, %ts_N]
      linalg.matmul ins(%tile_lhs, %tile_rhs : …) outs(%tile_output : …)
```

Google

Detour Tiling with Linalg on memref operands with Parallel loops

- Generate tiled loops while maintaining information about parallel inter-tile loops

```
scf.parallel (%iv0, %iv1) = (0, 0) to (%M, %N) step (%ts_M, %ts_k)
  scf.for %iv2 = 0 to %K step %ts_K
    %tile_lhs = memref.subview %lhs[%iv0, %iv2] [%ts_M, %ts_K]
    %tile_rhs = memref.subview %rhs[%iv2, %iv1] [%ts_K, %ts_N]
    %tile_output = memref.subview %result[%iv0, %iv1] [%ts_M, %ts_N]
    linalg.matmul ins(%tile_lhs, %tile_rhs : …) outs(%tile_output :
…)
```

Google

Detour: Structured Control Flow on tensors

Each iteration of a loop is an iteration as a function

- Arguments are induction variable and values from previous iteration
- The body of the loop computes the result of current iteration and “yields” it which gets forwarded to the next iteration of the loop
- The value yielded by the last iteration is the result of the loop.

```
%result = scf.for %iv0 = … init (%arg1 = %init) {
  %0 = … %arg1 …
  %yield = …
  scf.yield %yield
}
```

Tiling for Linalg on tensors: Using Destructive updates

```
%result = scf.for %iv0 = 0 to %M step %ts_M init(%arg1 = %init) {
  %0 = scf.for %iv1 = 0 to %N step %ts_N init(%arg2 = %arg1) {
    %1 = scf.for %iv2 = 0 to %K step %ts_K init (%arg3 = %arg2) {
      %tile_lhs = tensor.extract_slice %lhs[%iv0, %iv2] [%ts_M, %ts_K]
      %tile_rhs = tensor.extract_slice %rhs[%iv2, %iv1] [%ts_K, %ts_N]
      %tile_init = tensor.extract_slice %arg3[%iv0, %iv1] [%ts_M, %ts_N]
      %tile_result = linalg.matmul ins(%tile_lhs, %tile_rhs : …) outs(%tile_init : …)
      %2 = tensor.insert_slice %tile_result into %arg3[%iv0, %iv1] [%ts_M, %ts_N]
      scf.yield %2
    }
    scf.yield %1
  }
  scf.yield %0
```

Representational issue with tiling for Linalg on tensors

- In tensors, each loop iteration value is forwarded to the next iteration
  - Cannot represent parallel inter-tile loops at this level.
  - Tile + Distribute that worked with memref operands, doesn’t work anymore

- Essentially this is an abstraction gap
  - You eventually have to go from a tensor representation to a memref-based representation, at that time you can get back to the same semantics as before.
  - Issue is the destructive updates.

Avoiding destructive updates during tile + distribute

- Recap : IREE does tile + distribute (+fuse) during dispatch region formation
- Create two new operations within the dispatch region
    - `flow.dispatch.tensor.load :` Load a tile of the input to the dispatch region
    - `flow.dispatch.tensor.store :` Store a tile of into the result of the dispatch region.
- Across workgroups there is no synchronization between loads/stores
    - If multiple workgroups overlap in reads/writes, then it is a race condition.

Avoiding destructive updates

```
%result = flow.dispatch.workgroups[%N, %M, 1]
  (%lhs, %rhs, %init) -> tensor<?x?xf32> =
  (%arg0 : !flow.tensor<readonly:?x?xf32>, %arg1 : !flow.tensor<readonly:?x?xf32>,
   %arg2 : !flow.tensor<readonly:?x?xf32>, %arg3 : !flow.tensor<writeonly:?x?xf32>) {
    %wg_y, %wg_x = …
    scf.for %iv0 = %start_y to %M step %step_y
      scf.for %iv1 = %start_x to %N step %step_x
        %lhs_tile = flow.tensor.load %arg0 [...][%wg_y, %K]
        %rhs_tile = flow.tensor.load %arg1 [...][%K, %wg_x]
        %init_tile = flow.tensor.load %arg2 [...][%wg_y, %wg_x]
        %result_tile = linalg.matmul ins(%lhs_tile, %rhs_tile : ) outs(%init_tile :)
        flow.tensor.store %result_tile into %arg3 [...][%wg_y, %wg_x]
  }
```

Google

Bridging the abstraction gap

- The “runtime” does not need to do anything special.
    - Conversion to memref should give back the old semantics
    - flow.tensor.load becomes a tensor.extract_slice

```
hal.executable {
  hal.interface @io {
   hal.interface.binding @arg0, access = “Read”
   hal.interface.binding @arg1, access = “Read”
   hal.interface.binding @arg2, access = “Read”
   hal.interface.binding @arg3, access = “Write”
 }
 hal.executable.target dylib-llvm-* {
   hal.executable.entry_point @entry_fn
   module { … }
```

Google

Bridging the abstraction gap

```
func @entry_fn() {
   scf.for %iv0 = %start_y to %M step %step_y
     scf.for %iv1 = %start_x to %N step %step_x
       %lhs_tile = memref.subview %arg0 [...][%wg_y, %K]
       %rhs_tile = memref.subview %arg1 [...][%K, %wg_x]
       %init_tile = memref.subview %arg2 [...][%wg_y, %wg_x]
       %result_tile = memref.subview %arg3 [...][%wg_y, %wg_x]
       linalg.copy(%init_tile, %result_tile)
       linalg.matmul ins(%lhs_tile, %rhs_tile : ) outs(%result_tile : )
```

- Intra-executable bufferization has to update results in-place
- Extra copy here is because
  - Flow → Hal lowering creates a buffer for all inputs and outputs
  - linalg.matmul on tensors has initial value and extra operand

Google

In-place updates

Marking operands of flow.dispatch.workgroups as read-write

```
%result = flow.dispatch.workgroups[%N, %M, 1](%lhs, %rhs, %init) -> %init =
  (%arg0 : !flow.tensor<readonly:?x?xf32>, %arg1 : !flow.tensor<readonly:?x?xf32>,
   %arg2 : !flow.tensor<readwrite:?x?xf32>) {
    %wg_y, %wg_x = …
    scf.for %iv0 = %start_y to %M step %step_y
      scf.for %iv1 = %start_x to %N step %step_x
        %lhs_tile = flow.tensor.load %arg0 [...][%wg_y, %K]
        %rhs_tile = flow.tensor.load %arg1 [...][%K, %wg_x]
        %init_tile = flow.tensor.load %arg2 [...][%wg_y, %wg_x]
        %result_tile = linalg.matmul ins(%lhs_tile, %rhs_tile : ) outs(%init_tile :)
        flow.tensor.store %result_tile into %arg2 [...][%wg_y, %wg_x]
  }
```

In-place updates

```
hal.executable {
  hal.interface @io {
    hal.interface.binding @arg0, access = “Read”
    hal.interface.binding @arg1, access = “Read”
    hal.interface.binding @arg2, access = “Read|Write”
  }
  ...
    func @entry_fn() {
      ...
      scf.for %iv0 = %start_y to %M step %step_y
        scf.for %iv1 = %start_x to %N step %step_x
          %lhs_tile = memref.subview %arg0 [...][%wg_y, %K]
          %rhs_tile = memref.subview %arg1 [...][%K, %wg_x]
          %result_tile = memref.subview %arg2 [...][%wg_y, %wg_x]
          linalg.matmul ins(%lhs_tile, %rhs_tile : ) outs(%result_tile : )
    }
```

Google

Intra-dispatch bufferizations

- Useful to think of bufferization in IREE as two phases
  - Inter-dispatch region bufferization
  - Intra-dispatch region bufferization
- Intra-dispatch region bufferization has different constraints
  - No additional heap allocations
  - Try to compute results in place (using readwrite annotations on dispatch region operands)