IREE CPU/GPU Code Generation Pipeline

This document is intended to provide an overview of the codegen pipeline within IREE used to generate CPU/GPU code. It intends to give an overview of the main passes used, the objective of the pass, the current implementation, and what it is expected to achieve in the long term.

Note that while the code generation pipeline supports dynamic shapes, this work is very preliminary. The description of this is not covered here.

Input to the codegen pipeline

The input to the code generation pipeline is the module within the `hal.executable.variant` operation. Functions within this module that do **not** have `Visibility::Private` are the *entry point* functions of the dispatch region. These are the functions that are *invoked* by the IREE runtime. In addition, each dispatch region also contains a `hal.interface` operation that describes the ABI to use for the dispatch region. Two examples of the input to the code generation pipeline are shown below. In both of these, a single dispatch function contains a sequence of MHLO operations that the dispatch region creation has grouped into a single region. Ideally the grouped operations are fused into a single kernel.

```
hal.executable.variant "vulkan*" {
  module attributes {spv.target_env = ...} {
    func @main_ex_dispatch() {
      %c0 = constant 0 : index
      %0 = hal.interface.load.tensor @legacy_io::@arg0,
             offset = %c0 : tensor<32x24xf32>
      %1 = hal.interface.load.tensor @legacy_io::@arg1,
             offset = %c0 : tensor<24x16xf32>
      %2 = "mhlo.dot"(%0, %1) {precision_config = ["DEFAULT", "DEFAULT"]} :
             (tensor<32x24xf32>, tensor<24x16xf32>) -> tensor<32x16xf32>
      hal.interface.store.tensor %2, @legacy_io::@ret0,
        offset = %c0 : tensor<32x16xf32>
      return
    }
    hal.interface private @legacy_io  {
      hal.interface.binding @arg0, set=0, binding=0,
        type="StorageBuffer", access="Read"
      hal.interface.binding @arg1, set=0, binding=1,
        type="StorageBuffer", access="Read"
      hal.interface.binding @ret0, set=0, binding=2,
        type="StorageBuffer", access="Write|Discard"
    }
  }
}
```

Snippet 1 : Dispatch region with matrix-matrix multiply operation.

```
hal.executable.variant "vulkan*" {
  module attributes {spv.target_env = ...} {
    func @main_ex_dispatch() {
      %c0 = constant 0 : index
      %0 = hal.interface.load.tensor @legacy_io::@arg0,
             offset = %c0 : tensor<10x15xf32>
      %1 = hal.interface.load.tensor @legacy_io::@arg1,
             offset = %c0 : tensor<10x15xf32>
      %2 = hal.interface.load.tensor @legacy_io::@arg2,
             offset = %c0 : tensor<15xf32>
      %3 = "mhlo.add"(%0, %1) :
         (tensor<10x15xf32>, tensor<10x15xf32>) -> tensor<10x15xf32>
      %4 = "mhlo.broadcast"(%2) : (tensor<15xf32>) -> tensor<10x15xf32>
      %5 = "mhlo.multiply"(%3, %4) :
         (tensor<10x15xf32>, tensor<10x15xf32>) -> tensor<10x15xf32>
      hal.interface.store.tensor %5, @legacy_io::@ret0,
        offset = %c0 : tensor<10x15xf32>
      return
    }
    hal.interface private @legacy_io  {
      hal.interface.binding @arg0, set=0, binding=0,
        type="StorageBuffer", access="Read"
      hal.interface.binding @arg1, set=0, binding=1,
        type="StorageBuffer", access="Read"
      hal.interface.binding @arg2, set=0, binding=2,
        type="StorageBuffer", access="Read"
      hal.interface.binding @ret0, set=0, binding=3,
        type="StorageBuffer", access="Write|Discard"
    }
  }
}
```

Snippet 2 : Dispatch region with element-wise operations.

Roadmap Note: The current implementation might not actually fuse the operations grouped into a dispatch region into a single kernel. It is possible to end up with multiple kernels per dispatch region. Over time we plan to address this by using fusion at different levels (see below).

The inputs to the dispatch region are materialized within the entry point function using the `hal.interface.load.tensor` operation, This operation returns a `tensor` view of the buffer used to store the inputs. Similarly the result of the dispatch region are *written* out using the `hal.interface.store.tensor` operation.

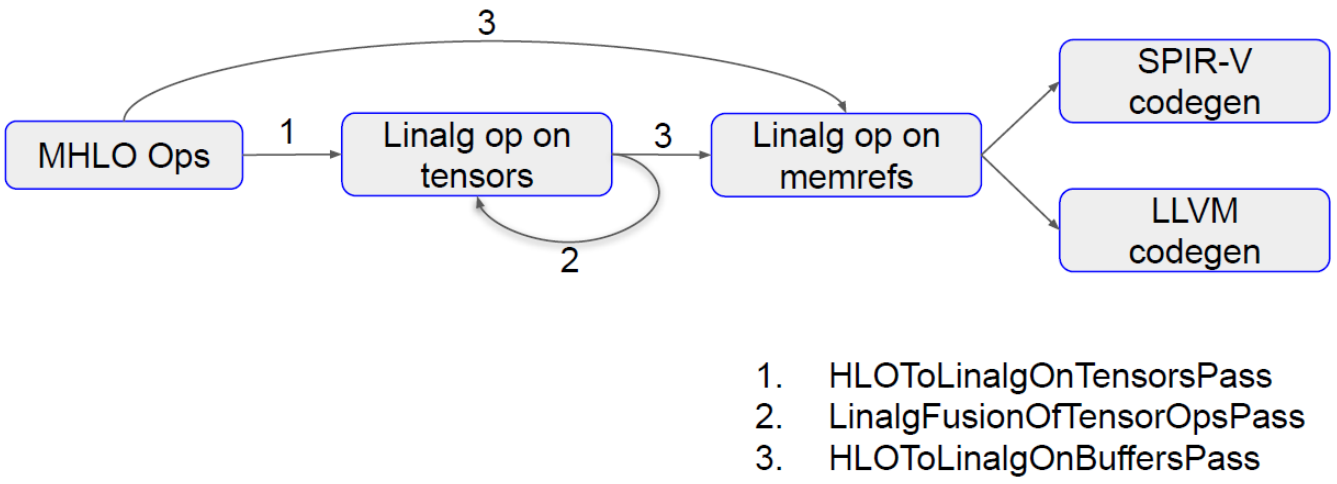
The main constraint that the code generation operates under is that it should not require additional (temporary) buffers to execute the operations grouped together within a dispatch region. The rationale behind this constraint is that buffer allocation/synchronization in IREE happens at the granularity of dispatch regions, allowing the scheduler to make better decision about where to insert appropriate synchronizations.

The IR after all the passes used in the lowering from MHLO to SPIR-V for the above two examples can be found here (matrix-matrix multiply op, elementwise ops). Below is a description of the major passes used.

Conversion from MHLO dialect to Linalg on buffers

The code generation pipeline heavily relies on use of Structured Operations, specifically the Linalg Dialect. Both, the Linalg operations on `tensor`s and on `memref`s are central to the progressive lowering approach followed here. The first part of the code generation pipeline is to convert the MHLO operations on `tensor`s to Linalg operation on `memref`s. This part of the pipeline is common to both CPU and GPU code generation.

The steps involved in this conversion is shown below. Each of the arrows represents a pass in the pipeline:


MHLO Ops
Linalg op on tensors
Linalg op on memrefs
SPIR-V codegen
LLVM codegen
1
2
3
3
1. HLOToLinalgOnTensorsPass
2. LinalgFusionOfTensorOpsPass
3. HLOToLinalgOnBuffersPass

The next sections describe each of these passes in more detail.

MHLO to Linalg on tensors

The first step is to convert MHLO operations to Linalg on tensors. This is done using the HLOToLinalgPass from Tensorflow. An example of the conversion is shown below, where the `mhlo.add`, `mhlo.broadcast` and `mhlo.multiply` operations are converted to `linalg.generic` operations on tensors.

```
#map0 = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
%3 = linalg.generic
       {args_in = 2 : i64, args_out = 1 : i64,
        indexing_maps = [#map0, #map0, #map0],
        iterator_types = ["parallel", "parallel"]} %0, %1 {
     ^bb0(%arg0: f32, %arg1: f32):  // no predecessors
       %5 = addf %arg0, %arg1 : f32
       linalg.yield %5 : f32
     } : tensor<10x15xf32>, tensor<10x15xf32> -> tensor<10x15xf32>
%4 = linalg.generic
       {args_in = 1 : i64, args_out = 1 : i64,
        indexing_maps = [#map1, #map0],
        iterator_types = ["parallel", "parallel"]} %2 {
     ^bb0(%arg0: f32):  // no predecessors
       linalg.yield %arg0 : f32
     }: tensor<15xf32> -> tensor<10x15xf32>
%5 = linalg.generic
       {args_in = 2 : i64, args_out = 1 : i64,
        indexing_maps = [#map0, #map0, #map0],
        iterator_types = ["parallel", "parallel"]} %3, %4 {
     ^bb0(%arg0: f32, %arg1: f32):  // no predecessors
       %5 = mulf %arg0, %arg1 : f32
       linalg.yield %5 : f32
     }: tensor<10x15xf32>, tensor<10x15xf32> -> tensor<10x15xf32>
```

Snippet 3 : MHLO to Linalg conversion for element-wise operations

At the time of writing the representation of Linalg on `tensor`s does not model reduction iterator types completely. Specifically, the reduction in Linalg is modeled using read-modify-write approach, i.e. each iteration of the reduction loop reads the value stored in the output, adds its contribution, and writes back to the same location. This means the output has to be *initialized* to the null element of the reduction operator (i.e. 0 if the reduction is done using addition). This works for operations on buffers. Since tensors are SSA values they cannot be updated in-place. As a result, the reduction semantics does not map as well to `tensor`s. For now it is treated as a convention that when the Linalg operation is converted to use `memref`s it has to be initialized appropriately before performing the reduction. Due to this, the conversion from MHLO op to Linalg op is only done for operations which do not need a *reduction* iterator type in the converted Linalg op. Consequently, only element-wise operations, broadcast operations and data movement operations (like copy and transpose) are converted to Linalg operations at this stage.

Roadmap note: One long term solution for the above is to have operations on tensors that have *reduction* iterator type to take an additional argument that contains the initial value of the result tensor. When the operation is converted to use `memref`s, the buffer for the initial value operand can be reused for the result. The details involved have not been fully worked out yet.

Fusion of Linalg on tensor operations

The Linalg on `tensor` operations generated at the previous step are fused using the LinalgFusionOfTensorOps from MLIR. Since `tensor`s are SSA values, fusion at this stage can be done without using alias analysis or dependence analysis based on reads and writes. Instead the use-def chains for the `tensor` values can be used to implement producer-consumer fusion. This stage fuses most elementwise operations, broadcast operations and data movement operations. An example of the fused op is shown below.

```
#map0 = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
%3 = linalg.generic
       {args_in = 3 : i64, args_out = 1 : i64,
        indexing_maps = [#map0, #map0, #map1, #map0],
        iterator_types = ["parallel", "parallel"]} %0, %1, %2 {
      ^bb0(%arg0: f32, %arg1: f32, %arg2: f32):  // no predecessors
        %4 = addf %arg0, %arg1 : f32
        %5 = mulf %4, %arg2 : f32
        linalg.yield %5 : f32
      }: tensor<10x15xf32>, tensor<10x15xf32>, tensor<15xf32> ->
tensor<10x15xf32>
```

Snippet 4: Fusion of Linalg operation on tensors for element-wise operations shown in Snippet 3

Conversion of Linalg on tensors to Linalg on buffers

Post fusion all the operation on `tensor`s are converted to analogous operations on `memref`s. In general, this requires a buffer allocation pass. In IREE, buffer allocation happens at the granularity of dispatch region, and as mentioned earlier, the dispatch region is not expected to use any additional temporary buffers. So instead of having another buffer allocation pass within the code generation pipeline, a simpler approach is used within IREE:

- For each `hal.interface.store.tensor` an `iree.placeholder` operation is created. The latter uses the same `hal.interface.binding` as the former, but returns a `memref` view of the output of the dispatch region instead of a `tensor` view. This `iree.placeholder` operation is added to start of the entry point function.

- A map is constructed that for a given `tensor` records the `memref` value to use during the conversion. In this map the `tensor` value used in the `hal.interface.store.tensor` is mapped to the `memref` value returned by the created `iree.placeholder` operation.

- The Dialect Conversion framework is used to implement a set of patterns that convert from operations on `tensor`s to operation on `memref`s,

    - A `hal.interface.load.tensor`, is replaced with an `iree.placeholder` to get the `memref` view of the input to the dispatch region.

    - All Linalg operation on `tensor`s (expected to be just `linalg.generic` or `linalg.indexed_generic` operations) are converted to the corresponding operation on `memref`s. Instead of returning a `tensor` value the converted operation takes an additional `memref` operand as argument. This `memref` is where the result of the operation is populated. Current implementation looks for the `memref` to use from the map constructed previously. If there is no `memref` associated with the result `tensor` the conversion fails.

    - At this stage, any `mhlo` operation not converted to a Linalg operation are directly converted to a Linalg operation on buffers. This is done for operations that when converted to Linalg have a *reduction* iterator type. Some examples of ops converted this way are

        - `mhlo.dot`
        - `mhlo.reduce`
        - `mhlo.conv`
        - `mhlo.reduce_window`.

      Since the specification of the Linalg operations require the output `memref` to be initialized appropriately, a `linalg.fill` operation is used to achieve this.

Roadmap Note : Right now the code-generation pipeline relies on fusion of operations on tensor level. In the near future, we want to be able to fuse operations like `linalg.matmul` and `linalg.conv` with consumers/producers that are element-wise operations using the fusion of Linalg operation on `memref`s.

At this stage of the compilation all operations must have been converted to Linalg operations on buffers. Shown below are the IR at the end of this stage for the two examples in Snippets 1 and 2.

```
func @main_ex_dispatch() {
  %0 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@ret0} : memref<32x16xf32>
  %c0 = constant 0 : index
  %1 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg0} : memref<32x24xf32>
  %2 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg1} : memref<24x16xf32>
  %cst = constant 0.000000e+00 : f32
  linalg.matmul(%1, %2, %0) :
    memref<32x24xf32>, memref<24x16xf32>, memref<32x16xf32>
  return
}
```

Snippet 5 : Matrix-matrix multiply after conversion to Linalg operation on `memref`s.

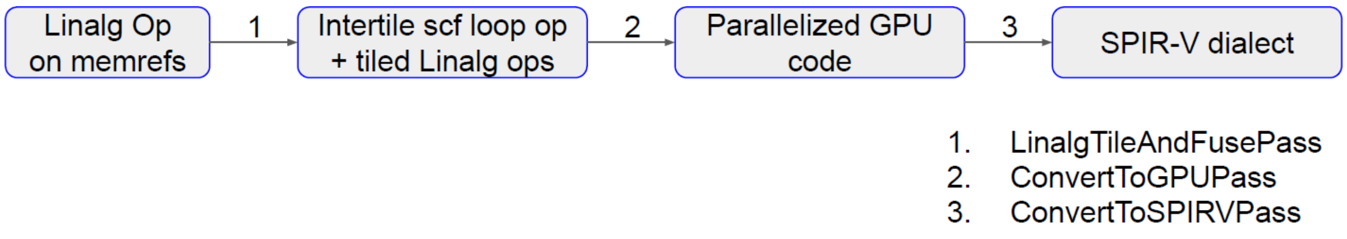
```
#map0 = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
func @main_ex_dispatch() {
  %0 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@ret0} : memref<10x15xf32>
  %c0 = constant 0 : index
  %1 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg0} : memref<10x15xf32>
  %2 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg1} : memref<10x15xf32>
  %3 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg2} : memref<15xf32>
  linalg.generic
    {args_in = 3 : i64, args_out = 1 : i64,
     indexing_maps = [#map0, #map0, #map1, #map0],
     iterator_types = ["parallel", "parallel"]} %1, %2, %3, %0 {
  ^bb0(%arg0: f32, %arg1: f32, %arg2: f32, %arg3: f32):  // no predecessors
    %4 = addf %arg0, %arg1 : f32
    %5 = mulf %4, %arg2 : f32
    linalg.yield %5 : f32
  }: memref<10x15xf32>, memref<10x15xf32>, memref<15xf32>,
memref<10x15xf32>
  return
}
```

Snippet 6 : Elementwise operations after conversion to Linalg operation on `memref`s

The rest of the code-generation differs on whether the compilation is for CPU (using LLVM) or for GPU (using SPIR-V).

Conversion from Linalg on buffers to SPIR-V dialect

The following sections describe the progressive lowering of Linalg operation on buffers to SPIR-V dialect. Once lowered to the SPIR-V dialect, it can be serialized into a SPIR-V binary using the serialization mechanism provided by the SPIR-V dialect. The steps involved in the lowering are described below, with each of the arrows representing a pass.



These passes are described below in more detail.

Tiling and fusion on buffer operations

The GPU hardware typically provides multiple-levels of compute hierarchy, namely *workgroup* level, *subgroup* level and *workitem* level. These map to blocks, warps and threads, respectively, in CUDA terminology. Tiling is a way to map the computations to each level of the compute hierarchy. For example 3-D tiling a `linalg.matmul` operation decomposes the computation into several tiled matrix-matrix multiplies. Tiling transformation in Linalg dialect generates the outer-loops that iterate over tiled `linalg.matmul` operations. These outer loops can be mapped to different workgroups, if they are parallel. The tiled `linalg.matmul` operation can be further tiled to map to subgroups. Finally, the tiled operation can be lowered to loops with individual iterations mapped to workitems. The LinalgTileAndFusePass uses the Linalg Tiling patterns (defined here) to tile operations like `linalg.matmul`, `linalg.conv` and `linalg.*_pooling`. The result of tiling the code in Snippet 5 is shown below. As expected there are 2-parallel loops that iterate over tiles of the original iteration space (i.e. inter-tile loops) and can be distributed to workgroups.

```
func @main_ex_dispatch_0()
  attributes {
    spv.entry_point_abi = {local_size = dense<[8, 8, 1]> : vector<3xi32>}}
{
  %cst = constant 0.000000e+00 : f32
  %c32 = constant 32 : index
  %c24 = constant 24 : index
  %c16 = constant 16 : index
  %c0 = constant 0 : index
  %c4 = constant 4 : index
  %0 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@ret0} : memref<32x16xf32>
  %1 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg0} : memref<32x24xf32>
  %2 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg1} : memref<24x16xf32>
  linalg.fill(%cst, %0) : f32, memref<32x16xf32>
  scf.parallel (%arg0, %arg1) = (%c0, %c0) to (%c32, %c16) step (%c8, %c8)
{
    scf.for %arg2 = %c0 to %24 step %c4 {
      ...
```

```
        %5 = subview %1[%arg0, %arg2]...
        ...
        %8 = subview %2[%arg2, %arg1]...
        ...
        %11 = subview %0[%arg0, %arg1]..
        linalg.matmul {__internal_linalg_transform__ = "workgroup"} %5, %8,
%11...
      }
      scf.yield
    }
    return
  }
```

Snippet 7 : `linalg.matmul` after tiling.

Tile Size and Workgroup Size

When operations that are to be tiled exist within the dispatch function (like `linalg.matmul` or `linalg.conv`), this pass also decides the 1. Tile size to be used for the tiling. 1. The workgroup size to be used.

The tile size and workgroup size are closely linked since the code within the tiled loops are to be collectively executed by the entire workgroup. In other words, all workitems in the workgroup collaborate to execute the tiled `linalg.matmul`.

Roadmap Note : Currently the tile sizes used in this pass are hard-wired. Not much effort has been put into finding ideal tile size for each operation on different hardware. The value used is meant to be a baseline to test functionality, with performance considerations addressed over time.

Markers

Downstream passes have to handle tiled Linalg operations and untiled Linalg operation that might exist in the same function in different ways. For example, while the former are to be executed collectively by workitems within a workgroup, the latter have to be executed by all workitems across workgroups. One way to distinguish these two operations is to use the marker mechanism in Linalg (LinalgTransformationFilter). This is a `StrAttr` whose value can be used to encode the scope of the operation. For example, in Snippet 7 above, the tiled `linalg.matmul` operation has a marker `workgroup` to indicate that this operation needs to be executed by a workgroup in a collective manner. At this time, the code-generation pipeline uses only the `workgroup` marker.

Roadmap Note : Markers are meant to be short-lived, ideally set and consumed within the same pass. In the current pipeline the lifetime spans passes to allow lowering to different hierarchies. The separate passes that implement the lowering from Linalg to SPIR-V can be combined into a single pass, relying A -> B -> C translation mechanism of the Dialect Conversion framework to implement the progressive lowering. In interest of separation of concerns and for better debuggability these passes are kept separate at the cost of having lifetimes of markers span passes.

Promoting subviews to use workgroup local memory and use of synchronizations

`Workgroup` memory (or `shared memory` in CUDA terminology) can be used to prefetch the inputs to the tiled operation. For example in the matrix-matrix multiply case, the same data row (column) of the LHS (RHS) matrix is read by multiple workitems. Prefetching the data into `Workgroup` memory can reduce the number of loads to `StorageClass` memory by an order of magnitude. This transformation can be achieved by using the `Linalg Promotion` which modifies the `subview`s that are the operands to the tiled Linalg operation to use a new `memref` object. The size of this `memref` is computed from the size of the `subview`. This `memref` object is later lowered to use `Workgroup` memory Storage Class. The snippet below shows this transformation when applied to `linalg.matmul` (along with tiling). The newly created `memref` objects are annotated with the memory space `3` to indicate that they are to be lowered to use `Workgroup` memory. The copy of data from the original `memref` into the new `memref`, as well as the necessary synchronization constructs are generated as well. Note the memory space annotation used here is consistent with what address space annotations used in NVVM.

```
func @matmul_tile()
  attributes {
    spv.entry_point_abi = {local_size = dense<[8, 8, 1]> : vector<3xi32>}}
{
  %c32 = constant 32 : index
  %c24 = constant 24 : index
  %c16 = constant 16 : index
  %c4 = constant 4 : index
  %c8 = constant 8 : index
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %0 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg0} : memref<32x24xf32>
  %1 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg1} : memref<24x16xf32>
  %2 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@ret0} : memref<32x16xf32>
  scf.parallel (%arg0, %arg1) = (%c0, %c0) to (%c32, %c16) step (%c8, %c8)
{
    scf.for %arg2 = %c0 to %c24 step %c4 {
      ...
      %5 = subview %0[%arg0, %arg2]...
      ...
      %8 = subview %1[%arg2, %arg1]...
      ...
      %11 = subview %2[%arg0, %arg1]...
      %12 = alloc(%c8, %c4) : memref<?x?xf32, 3>
      %13 = subview %12[%c0, %c0]...
      %14 = alloc(%c4, %c8) : memref<?x?xf32, 3>
      %15 = subview %14[%c0, %c0]...
      linalg.copy(%5, %13) {__internal_linalg_transform__ = "workgroup"}
        : memref<?x?xf32, #map2>, memref<?x?xf32, #map2, 3>
      spv.ControlBarrier "Workgroup", "Workgroup", "AcquireRelease"
      linalg.copy(%8, %15) {__internal_linalg_transform__ = "workgroup"}
        : memref<?x?xf32, #map2>, memref<?x?xf32, #map2, 3>
      spv.ControlBarrier "Workgroup", "Workgroup", "AcquireRelease"
      linalg.matmul {__internal_linalg_transform__ = "workgroup"} %13, %15,
%11...
```

```
        spv.ControlBarrier "Workgroup", "Workgroup", "AcquireRelease"
        dealloc %12 : memref<?x?xf32, 3>
        dealloc %14 : memref<?x?xf32, 3>
      }
      scf.yield
    }
    return
  }
```

Snippet 8: `linalg.matmul` after tiling and promotion of operand subviews to use `Workgroup` memory.

Distributing to workgroups and workitems

After tiling the operations within the dispatch functions are either `scf.parallel` operations or Linalg operations.

- The outer `scf.parallel` operations represent parallel loops that are to be distributed across workgroups. The distribution here assumes that the number of workgroups along each dimension is equal to the number of iterations of the `scf.parallel` operation.

- Linalg operations that are not tiled, and are therefore **not within** `scf` operations, are lowered to loops. The resulting outer `scf.parallel` operations are collapsed to have a single induction variable. This loop is then distributed across workitems using their `GlobalInvocationId`, (which is same as `blockIdx * blockDim + threadIdx` in CUDA terminology).

- Linalg operations that are tiled, and are therefore **within** `scf` operations, are lowered to loops and the iterations of the `scf.parallel` operations are mapped to workitems using their `LocalInvocationId` (which is same as `threadIdx` in CUDA terminology). Note that these operations are tagged with the `workgroup` marker which makes it easy to disambiguate from the case where Linalg operations are outside of `scf` operations. Here too, the distribution assumes that the workgroup size is greater than or equal to the number of iterations of the partitioned loop.

These transformations are applied by the `ConvertToGPUPass`. Below is the result of applying this pass to Snippet 7. The outer `scf.parallel` loop is distributed across workgroups. The tiled `linalg.matmul` operation is lowered to loops, and the outer `scf.parallel` operation generated during this lowering are distributed across workitems within the workgroup.

```
func @main_ex_dispatch_0_dispatch_1()
  attributes {
    spv.entry_point_abi = {local_size = dense<[8, 8, 1]> : vector<3xi32>}}
{
  %c24 = constant 24 : index
  %c8 = constant 8 : index
  %c4 = constant 4 : index
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %0 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@ret0} : memref<32x16xf32>
  %1 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg0} : memref<32x24xf32>
  %2 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg1} : memref<24x16xf32>
  %3 = "gpu.block_id"() {dimension = "x"} : () -> index
  %4 = "gpu.block_id"() {dimension = "y"} : () -> index
  %5 = muli %4, %c8 : index
  %6 = muli %3, %c8 : index
  scf.for %arg0 = %c0 to %c24 step %c4 {
    ...
    %15 = subview %1[%5, %arg0]
    ...
    %20 = subview %2[%arg0, %6]
    %21 = subview %0[%5, %6]
    %22 = "gpu.thread_id"() {dimension = "x"} : () -> index
    %23 = "gpu.thread_id"() {dimension = "y"} : () -> index
    %24 = cmpi "slt", %23, %10 : index
    %25 = cmpi "slt", %22, %19 : index
    %26 = and %24, %25 : i1
    scf.if %26 {
      scf.for %arg1 = %c0 to %14 step %c1 {
        %27 = load %15[%23, %arg1] : memref<?x?xf32, #map0>
        %28 = load %20[%arg1, %22] : memref<?x?xf32, #map1>
        %29 = load %21[%23, %22] : memref<?x?xf32, #map1>
        %30 = mulf %21, %22 : f32
        %31 = addf %23, %24 : f32
        store %25, %15[%23, %22] : memref<4x?xf32, #map1>
      }
    }
  }
  return
}
```

Snippet 9: `linalg.matmul` after distributing parallel inter-tile loops to workgroups and intra-tile loops to workitems.

Snippet 6 shows the fused element-wise operations represented using a `linalg.generic` operation. This operation is not tiled in the `LinalgTileAndFusePass`. So the `ConvertToGPUPass` lowers this operation to `scf.parallel` loops, which are collapsed into a `scf.parallel` operation with a single induction variable. This loop is then distributed across workitems using the `GlobalInvocationId`. The resulting IR is shown below.

```
func @main_ex_dispatch_0()
  attributes {
    spv.entry_point_abi = {local_size = dense<[32, 1, 1]> : vector<3xi32>}}
{
  %c50 = constant 50 : index
  %c5 = constant 5 : index
  %0 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@ret0} : memref<10x15xf32>
  %1 = iree.placeholder for "interface buffer"
         {binding = @legacy_io::@arg0} : memref<10x15xf32>
  %2 = iree.placeholder for "interface buffer"
```

```
          {binding = @legacy_io::@arg1} : memref<10x15xf32>
    %3 = iree.placeholder for "interface buffer"
          {binding = @legacy_io::@arg2} : memref<15xf32>
    %4 = "gpu.block_id"() {dimension = "x"} : () -> index
    %5 = "gpu.block_dim"() {dimension = "x"} : () -> index
    %6 = "gpu.thread_id"() {dimension = "x"} : () -> index
    %7 = muli %4, %5 : index
    %8 = addi %7, %6 : index
    %9 = cmpi "slt", %8, %c50 : index
    scf.if %9 {
      %10 = divi_signed %8, %c5 : index
      %11 = remi_signed %8, %c5 : index
      %12 = load %1[%10, %11] : memref<10x15xf32>
      %13 = load %2[%10, %11] : memref<10x15xf32>
      %14 = load %3[%11] : memref<15xf32>
      %15 = addf %12, %13 : f32
      %16 = mulf %15, %14 : f32
      store %16, %0[%10, %11] : memref<10x15xf32>
    }
    return
  }
```

Snippet 10: Distributing the iterations for pointwise operations for GPU execution.

Lowering to SPIR-V dialect

The last step is to take the result of the previous pass and lowering it to SPIR-V dialect. Since SPIR-V dialect is *closed*, i.e. it has a separate type system, its best to lower all the operations to SPIR-V in one step. This is done by applying all the patterns that lower all the different IR constructs into SPIR-V within the `ConvertToSPIRVPass`. These are

- GPU dialect to SPIR-V conversion.
- SCF dialect to SPIR-V conversion.
- Standard dialect to SPIR-V conversion.
- Patterns that lower the `iree.placeholder` instruction into a SPIR-V.

Once applied the resulting IR is in SPIR-V dialect that can be serialized to a SPIR-V binary.