IREE CodeGen

Mahesh Ravishankar, Lei Zhang, Hanhan Wang,
Ahmed Taei, Thomas Raoux, Nicolas Vasilache

Cerebra Babelfish Device Inference (BDI) team

MLIR Open Design Meeting, 2020-08-20

Goals

- IREE has three backends (as of now)
  - VMLA
  - CPU } Codegeneration backends
  - GPU

- Vulkan/SPIR-V used to target GPUs.
  - Unlike CUDA no great library support for GEMM, Convs, etc.
  - Avoid using black-box components.

- Use/enhance Dialects/Conversions available in MLIR or TF for progressive lowering.

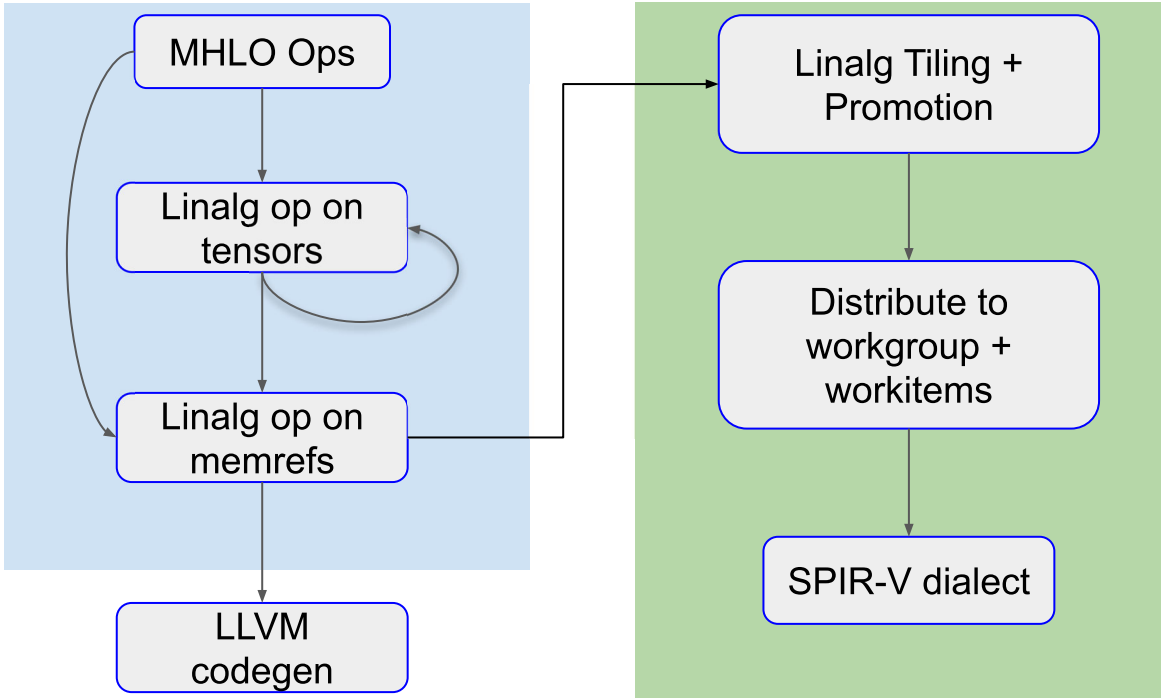
Background

- Input to IREE is an MLIR module in MHLO
  - Partitioned into atomic *execution units*, i.e. dispatch regions
  - Scheduled based on dependencies between the dispatch regions
    - Insert appropriate synchronization between the regions
    - Buffer allocation at granularity of dispatch regions
- Single dispatch regions contains operations that can be “fused” together
  - On GPU ideally a single kernel
- Progressively lowered to CPU/GPU code

Documentations Resources

- Detailed description of codegeneration pipeline
  https://google.github.io/iree/design-docs/codegen-passes
- Upto date dump of output after each pass in the pipeline
  https://google.github.io/iree/ir-examples
- MHLO op coverage
  https://google.github.io/iree/xla-op-coverage
- TF Model coverage
  https://google.github.io/iree/tf-e2e-coverage

Compilation flow



Detailed description of the IREE Code-generation pipeline is here

Running examples

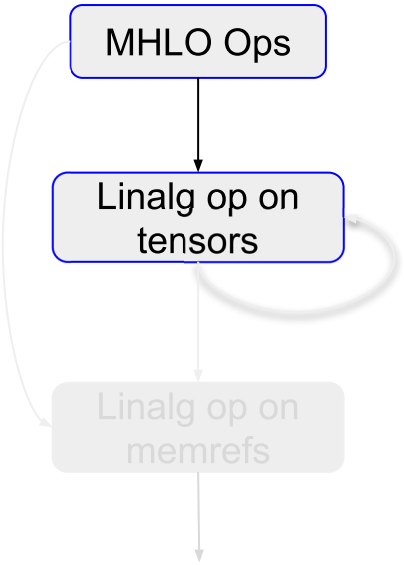
```
func @gemm() {
  %0 = ... : tensor<16x32xf32>
  %1 = ... : tensor<32x8xf32>
  %2 = “mhlo.dot”(%0, %1) : (tensor<16x32xf32>, tensor<32x8xf32>) -> tensor<16x8xf32>
  ...
}
func @elementwise {
    %0 = ... : tensor<10x15xf32>
    %1 = ... : tensor<10x15xf32>
    %2 = ... : tensor<15xf32>
    %3 = “mhlo.add”(%0, %1) : (tensor<10x15xf32>, tensor<10x15xf32) -> tensor<10x15xf32>
    %4 = “mhlo.broadcast(%2) : (tensor<15xf32) -> tensor<10x15xf32>
    %5 = “mhlo.mul”(%3, %4) : (tensor<10x15xf32>, tensor<10x15xf32>) -> tensor<10x15xf32>
    ...
}
```

MHLO To Linalg on Tensors

- Convert MHLO ops that represent element-wise operations to Linalg operations on tensors
  - `linalg.generic`
  - `linalg.indexed_generic`
  - `linalg.tensor_reshape`

- `mlho.conv, mhlo.dot, mhlo.reduce` handled later

- `hlo-legalize-to-linalg` pass in TF (here)



Elementwise operation to Linalg on tensors

```
#map0 = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
func @elementwise() {
  ...
  %3 = linalg.generic %0 %1 {..[#map0, #map1]..} {
       // add operation
  } : (tensor<10x15xf32>, tensor<10x15xf32>) -> tensor<10x15xf32>
  %4 = linalg.generic %2 {..[#map0, #map1]..} {
       ^bb0(%arg0 : f32, %arg1 : f32):
         linalg.yield %arg0 : f32
  } : (tensor<15xf32>) -> tensor<10x15xf32>
  %5 = linalg.generic %3 %4 {..[#map0, #map1]..} {
       // mul operation
  } : (tensor<10x15xf32>, tensor<10x15xf32>) -> tensor<10x15xf32>
}
```

Elementwise operation to Linalg on tensors

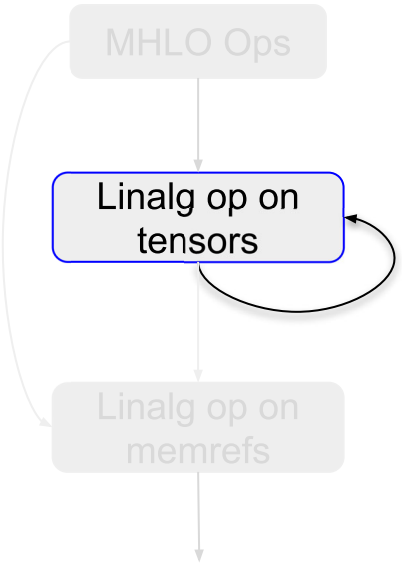
```
#map0 = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
func @elementwise() {
  ...
  %3 = linalg.generic %0 %1 {..[#map0, #map1]..} {
       // add operation
  } : (tensor<10x15xf32>, tensor<10x15xf32>) -> tensor<10x15xf32>
  %4 = linalg.generic %2 {..[#map0, #map1]..} {
       ^bb0(%arg0 : f32, %arg1 : f32):
         linalg.yield %arg0 : f32
  } : (tensor<15xf32>) -> tensor<10x15xf32>
  %5 = linalg.generic %3 %4 {..[#map0, #map1]..} {
       // mul operation
  } : (tensor<10x15xf32>, tensor<10x15xf32>) -> tensor<10x15xf32>
}
```

Linalg fusion on tensors

- Producer/Consumer fusion `RewritePattern`
  - Fixed point iteration within `GreedyPatternRewriter` to fuse all producers/consumers that are elementwise ops.

- Smaller op surface area + `LinalgOp OpInterface` makes the fusion logic very simple
  - 4 patterns provides all the functionality needed to achieve elementwise ops + broadcast fusion

- `linalg-fusion-for-tensor-ops` pass in MLIR (here)

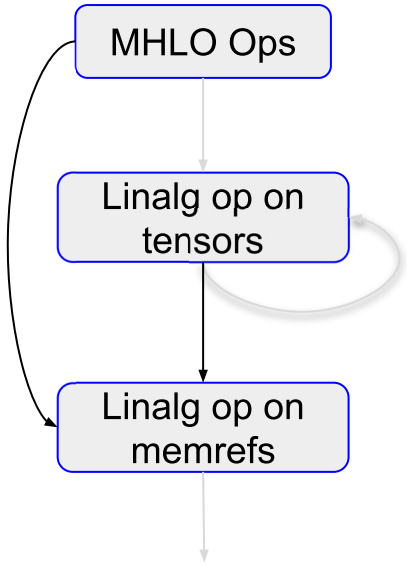


Elementwise operation after fusion

```
#map0 = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
func @elementwise() {
  ...
  %3 = linalg.generic %0, %1, %2 { .. indexing_maps = [#map0, #map0, #map1, #map0] ..} {
       ^bb0(%arg0 : f32, %arg1 : f32, %arg2 : f32, %arg3 : f32):
         %0 = addf %arg0, %arg1 : f32
         %1 = mulf %0, %arg2 : f32
         linalg.yield %1 : f32
       } : (tensor<10x15xf32>, tensor<10x15xf32>, tensor<15xf32> -> (tensor<10x15xf32>)
  ...
}
```

Tensor to Buffer Conversion

- Tensor ops to `linalg` buffer ops
  - MHLO ops with `reduction/window` iterator type.
  - Linalg op on tensors to Linalg op on buffers.
- Requires buffer allocation in general
  - In IREE happens at dispatch region boundary.
  - Avoid additional temporary buffer allocations within dispatch regions.
- `iree-codegen-hlo-to-linalg-on-buffers` pass in IREE (here)



Running examples

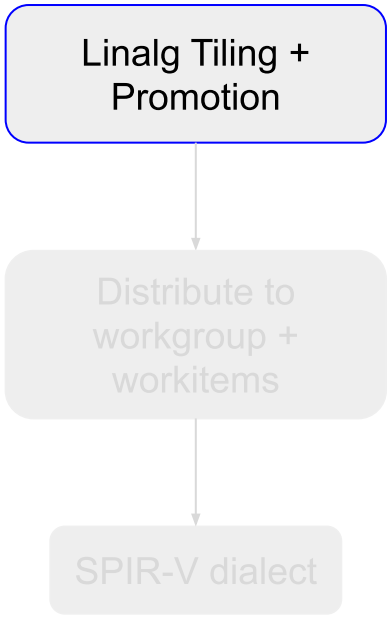
```
func @gemm() {
  %0 = ... : memref<16x32xf32>
  %1 = ... : memref<32x8xf32>
  %2 = ... : memref<16x8xf32>
  linalg.matmul(%0, %1, %2) : (memref<16x32xf32>, memref<32x8xf32>, memref<16x8xf32>
}

func @elementwise {
  %0 = ... : memref<10x15xf32>
  %1 = ... : memref<10x15xf32>
  %2 = ... : memref<15xf32>
  %3 = ... : memref<10x15xf32>
  linalg.generic %0, %1, %2, %3 {..} {
    ...
  } : memref<10x15xf32>, memref<10x15xf32>, memref<15xf32>, memref<10x15xf32>
}
```

Linalg to SPIR-V in IREE

Linalg Tiling and Fusion

- Use tiling to map to different levels of processor hierarchy
  - One level of tiling : `scf.parallel` to workgroups
  - Second level of tiling : `scf.parallel` to subgroups
  - Map tiled operation to workitems
- Subviews of the tiled operation can be promoted to Workgroup memory
- Fusion
  - At tile granularity using linalg on buffers
  - Convert to vector dialect and fusion using SSA use-def



Example tiling of `linalg.matmul` operation

```
func @gemm() {
  %0 = ... : memref<16x32xf32>
  %1 = ... : memref<32x8xf32>
  %2 = ... : memref<16x8xf32>
  scf.parallel (%iv0, %iv1) = ... {
    scf.for %iv2 = {
      ...
      %sv1 = subview %0[%iv0, iv2]
      %sv2 = subview %1[%iv2, iv0]
      %sv3 = subview %2[%iv0, iv1]
      linalg.matmul %sv1, %sv2, %sv3  : (memref<8x4xf32>, memref<4x8xf32>, memref<8x8xf32>
    }
  }
}
```

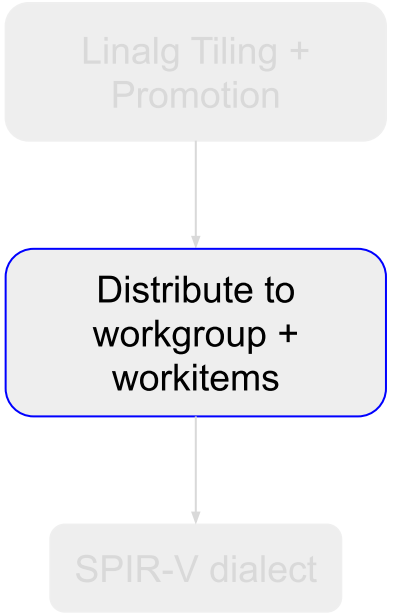
Distributed to workgroups (2D)

Promotion to workgroup memory

Workgroup-level `linalg.matmul`

Distributing to workgroup/workitems

- Inter-tile loops distributed to workgroups
  - Second level inter-tile loops distributed to subgroups
- Tiled Linalg operation lowered to loops and distributed to workitems
- Some logic to reduce control overhead when
  - Number of iterations <= Number of processors
  - Number of iterations == Number of processors
- `iree-codegen-convert-to-gpu` pass in IREE (here)



Matrix-Matrix multiply after distribution

```
func @gemm() {
  ...
  %3 = “gpu.block_id”() {“x”} : index
  %4 = “gpu.grid_dim”() {“x”} : index
  %5 = “gpu.block_id() {“y”} : index
  %6 = “gpu.grid_dim() {“y”} : index
  ...
  scf.for %iv0 = {
    …
    %9 = “gpu.thread_id()”{“x”} :
    %10 = “gpu.thread_id()”{“y”}
    scf.for %iv1 {
      ...
    }
  }
}
```

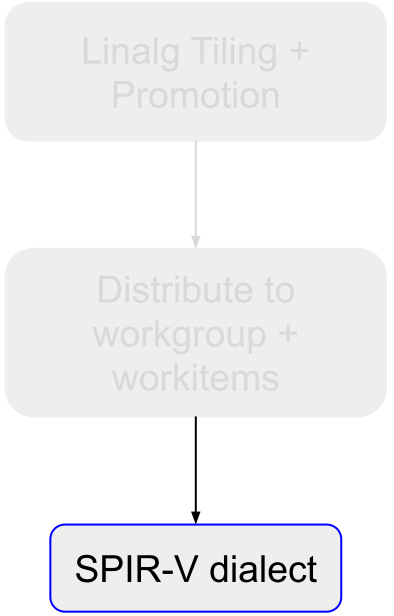
Inter-tile k-loop

Intra-tile k-loop

Conversion to SPIR-V dialect

- Aggregate all the patterns that lower to SPIR-V.
  - Standard to SPIR-V
  - SCF to SPIR-V
  - GPU To SPIR-V (for block_id, thread_id, etc.)

- Points to note
  - SPIR-V dialect has its own type system, so better to lower to SPIR-V dialect in one go.
  - Some ops (`subview`) are legalized away before lowering to SPIR-V.

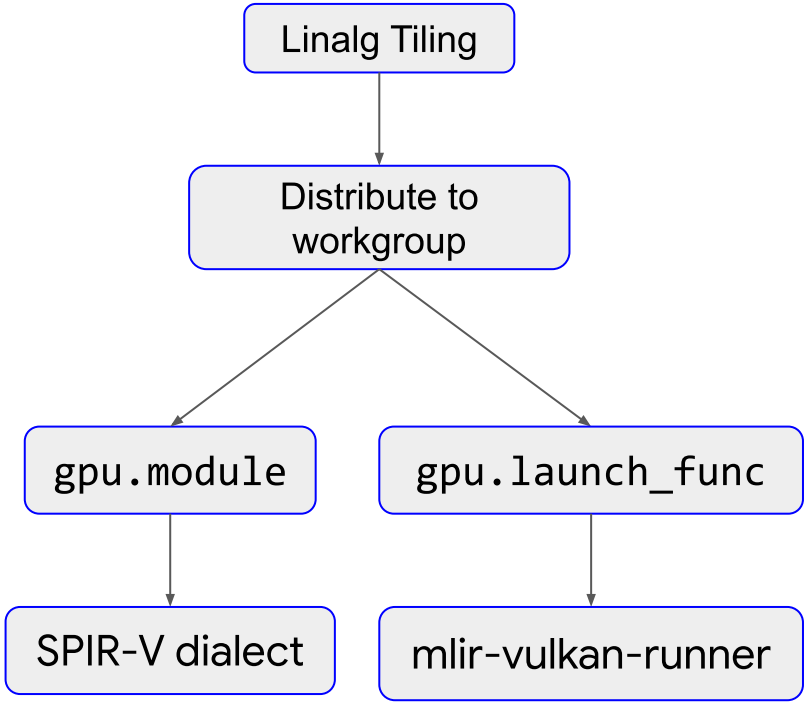


Linalg to SPIR-V in MLIR

(Alex Zinenko, Stephan Herhut, Alexander Belyaev, Denis Khalikov)

Linalg on Buffers to SPIR-V

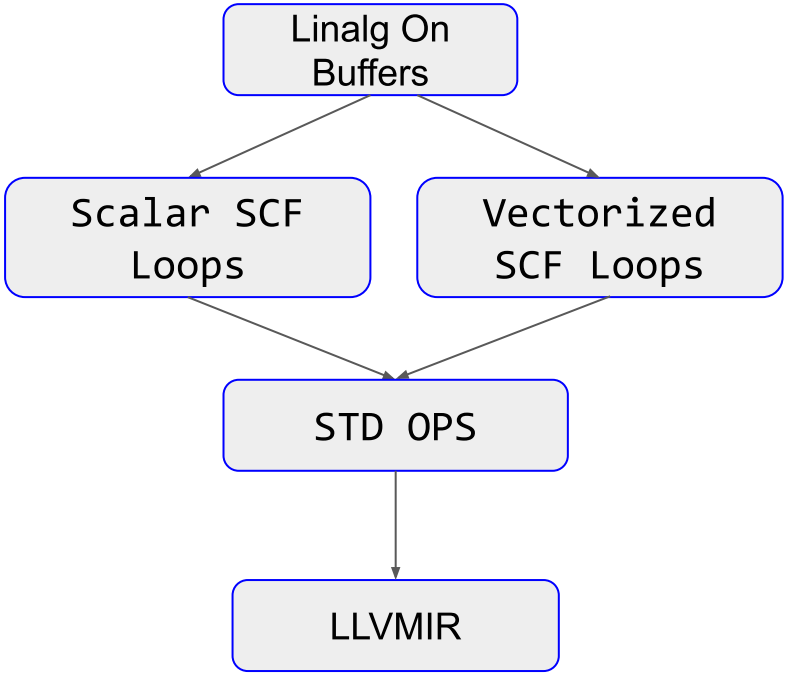
- Lowering from Linalg to SPIR-V goes through GPU dialect.
- Models both the host side and device side.
- Can allow for optimizations across host and device
  - Propagating to the device side the number of blocks/block size, etc.
- Discourse post has more details



Linalg to LLVMIR in IREE

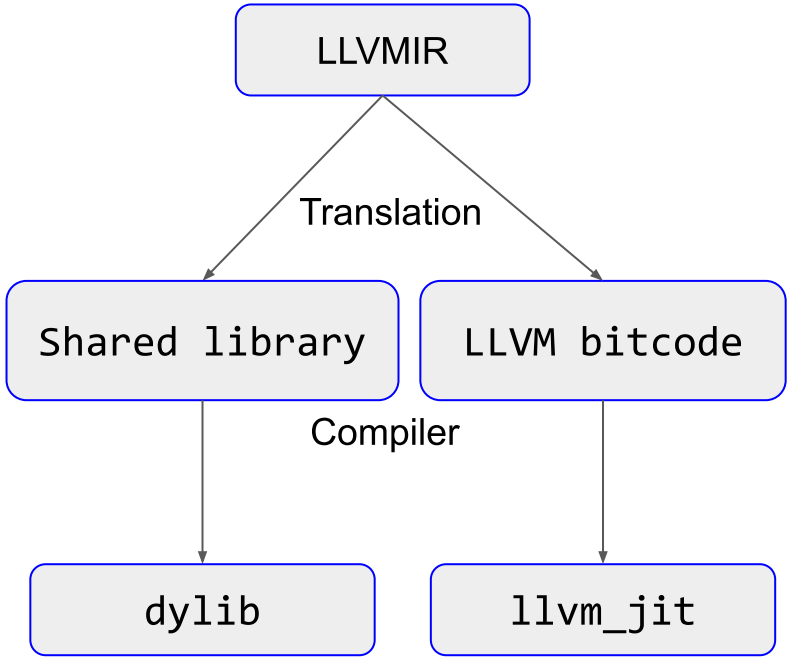
Progressive lowering Of Linalg to LLVMIR

- Goal : Efficient single core CPU code (Executables).
- LinalgOps are lowered into a loop nest over scalar arithmetic.
- We use MatMulVectorizationStrategy to control generating SIMD code for matrix-matrix multiplication, the strategy does:
  - Multi-Level hierarchical tiling of linalg.matmul
  - Efficient use of vector ops.
- For the rest of the ops as of now we relies on LLVM auto-vectorization.



IREE CPU Codegen Compilation / Runtime

- Translate LLVMIR dialect to LLVM Bitcode.
- Apply LLVM optimization passes.
- Generate executable for the specific
  - LLVM bitcode for jitting runtime
  - A Shared library for AOT dylib runtime.



Runtime HAL drivers

Example: Lowering linalg.matmul to LLVMIR

```
func @gemm() {
  %0 = ... : memref<512x512xf32>
  %1 = ... : memref<512x512xf32>
  %2 = ... : memref<512x512xf32>
  linalg.matmul(%0, %1, %2) : (memref<512x512xf32>, memref<512x512xf32>, memref<512x512xf32>
}
```

Lowering linalg.matmul to SCF

```
scf.for %arg3 = %c0 to %c512 step %c1 {
  scf.for %arg4 = %c0 to %c512 step %c1 {
    scf.for %arg5 = %c0 to %c512 step %c1 {
      %0 = load %arg0[%arg3, %arg5] : memref<512x512xf32>
      %1 = load %arg1[%arg5, %arg4] : memref<512x512xf32>
      %2 = load %arg2[%arg3, %arg4] : memref<512x512xf32>
      %3 = mulf %0, %1 : f32
      %4 = addf %2, %3 : f32
      store %4, %arg2[%arg3, %arg4] : memref<512x512xf32>
    }
  }
}
```

MatMul Tiling And Vectorization Strategy

```
scf.for %arg0 = %c0 to %c512 step %c64 {
  scf.for %arg1 = %c0 to %c512 step %c64 {
    scf.for %arg2 = %c0 to %c512 step %c64 {
      %3 = subview %1...: memref<512x512xf32> to memref<64x64xf32...
      ...
      scf.for %arg3 = %c0 to %c64 step %c32 {
        scf.for %arg4 = %c0 to %c64 step %c32 {
          scf.for %arg5 = %c0 to %c64 step %c32 {
            %6 = subview %3...: memref<64x64xf32, ... to memref<32x32xf32...
            ...
            ... = vector.transfer_read ... : memref<32x32xf32, ... vector<4xf32>
            ... = vector.transpose ... : vector<4x4xf32>, vector<4x4xf32>
            ... = vector.outerproduct ... : vector<4xf32>, vector<4xf32>
```

1 Level of memory tiling

2 Level of memory tiling

Register level tiling & vector ops

Lowering To LLVMIR Dialect

```
llvm.func @dispatch_op_name(%arg0: !llvm.ptr<ptr<i8>>, %arg1: !llvm.ptr<i32>) {
  %0 = llvm.bitcast %arg0 : !llvm.ptr<ptr<i8>> to
                            !llvm.ptr<struct<(ptr<float>, ptr<float>, ptr<float>)>>
  %1 = llvm.load %0 : !llvm.ptr<struct<(ptr<float>, ptr<float>...
  %2 = llvm.extractvalue %1[0] : !llvm.struct<(ptr<float>, ptr<float>...
  %3 = llvm.mlir.undef : !llvm.struct<(ptr<float>, ptr<float>...
  ...
  ... = llvm.mlir.constant(512 : index) : !llvm.i64
  ... = llvm.insertvalue … [3, 0] : !llvm.struct<(ptr<float>, ptr<float>...
```

Fixed ABI

Dynamic shape information passed as arguments

First argument

Static shape information recovered from IR

Packed buffer arguments

Wrapping up

Current status

- Models
  - Vision models : MobileNetV2, ResNet50
  - Language models : MT Encoder, Hotword
- Platforms
  - Android : AArch64 CPU + Mali/Adreno GPU
  - Linux/Windows : CPU + GPU (NVIDIA, and AMD)
- Latency optimization
  - Matmul codegeneration strategy on CPUs with Vector Dialect
  - Matmul codegeneration on GPUs with Vector Dialect + `spv.CooperativeMatrixMulAddNV`
- Proof of concept for dynamic shape (static rank) codegeneration

Next steps

- Integrate efficient Matmul code generation strategy into IREE pipeline
- Fusion of Matmul operations with other operations
  - Elementwise
  - Reductions
  - (any more?)
- Convolution code generation (anyone?)

MLIR infrastructure makes this all possible!