MLIR

Uday Kumar Reddy B

udayb@iisc.ac.in

Dept of CSA
Indian Institute of Science

MLIR

- Open-sourced by Google in Apr 2019
- ML in MLIR: Multi-level
- Ability to represent code at multiple levels in a unified way
- First class abstractions for multi-dimensional arrays (tensors), loop nests, affine maps/sets, and more

Outline

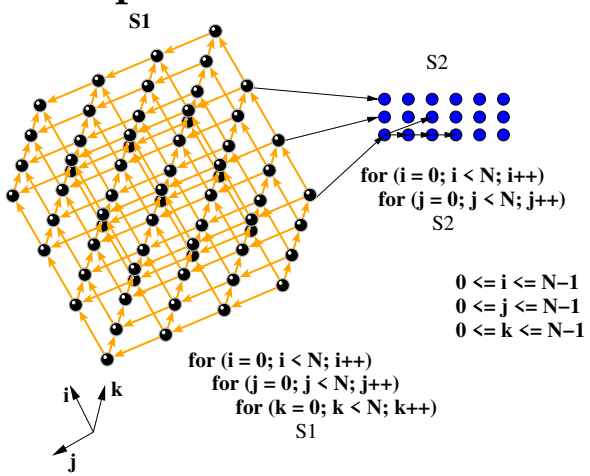
- MLIR Representation
- Polyhedral Notions in MLIR
- Analyses and Transformations

MLIR: Multi-level Intermediate Representation

1. Ops on tensor types form

```
%patches = "tf.reshape"(%patches, %minus_one, %minor_dim_size)
                : (tensor<? x ? x ? x ? x ? x f32>, index, index) -> tensor<? x ? x f32>
%mat_out = "tf.matmul"(%patches_flat, %patches_flat){transpose_a : true}
                : (tensor<? x ? x f32>, tensor<? x ? x f32>) -> tensor<? x ? x f32>
%vec_out = "tf.reduce_sum"(%patches_flat) {axis: 0} : (tensor<? x ? x f32>) -> tensor<? x f32>
```

2. Loop-level/mid-level form




```
#map0 = affine_map<(d0) -> (d0)>
#map1 = affine_map<(d0) -> (d0 + 4)>
affine.for %i = 0 to 8 step 4 {
  affine.for %j = 0 to 8 step 4 {
    affine.for %k = 0 to 8 step 4 {
      affine.for %ii = #map0(%i) to #map1(%i) {
        affine.for %jj = #map0(%j) to #map1(%j) {
          affine.for %kk = #map0(%k) to #map1(%k) {
            %5 = affine.load %lhs[%ii, %kk] : memref<8 x 8 x f32>
            %6 = affine.load %rhs[%kk, %jj] : memref<8 x 8 x f32>
            %7 = affine.load %out[%ii, %jj] : memref<8 x 8 x f32>
            %8 = arith.mulf %5, %6 : f32
            %9 = arith.addf %7, %8 : f32
            affine.store %9, %out[%ii, %jj] : memref<8 x 8 x f32>
          }
        }
      }
    }
  }
}
```

3. Low-level form: closer to hardware

```
%v1 = memref.load %a[%i2, %i3] : memref<256 x 64 x vector<16 x f32>>
%v2 = memref.load %b[%i2, %i3] : memref<256 x 64 x vector<16 x f32>>
%v3 = arith.addf %v1, %v2 : vector<16 x f32>
memref.store %v3, %d[%i2, %i3] : memref<256 x 64 x vector<16 x f32>>
```

MLIR Design Principles

1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels

MLIR Design Principles

1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels

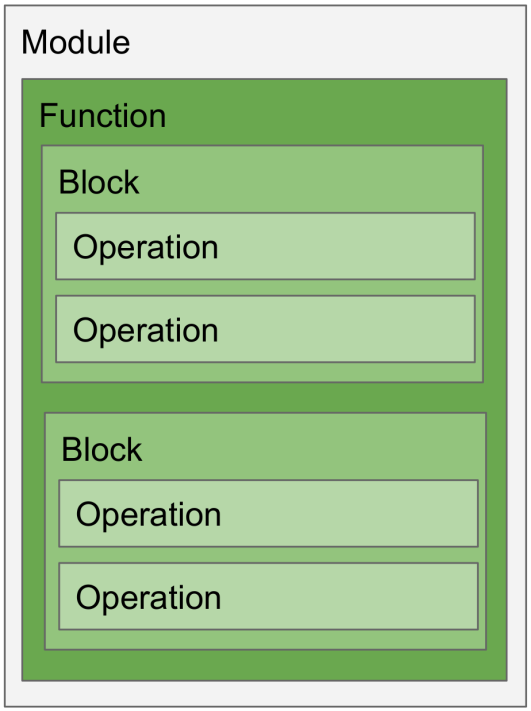
MLIR Design Principles

1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels

MLIR - Basic Concepts

- SSA, typed, three address
- Module/Function/Block/Operation structure
- Operations can hold a "region", which is a list of blocks

```
func.func @test(%arg: i32) {
  %x = call  @thing_to_call(%arg) :  (i32)  -> i32
  cf.br ^bb1
^bb1:
  %y = arith.addi %x, %x : i32
  return %y : i32
}
```



SSA REPRESENTATION

- Functional SSA representation
- No ϕ nodes
- Instead, blocks take arguments

```
func.func @condbr_simple() -> (i32) {
  %cond = "foo"() : () -> i1
  %a = "bar"() : () -> i32
  %b = "bar"() : () -> i64
  cf.cond_br %cond, ^bb1(%a : i32), ^bb2(%b : i64)

^bb1(%x : i32):
  %w = "foo_bar"(%x) : (i32) -> i64
  cf.br ^bb2(%w: i64)

^bb2(%y : i64):
  %z = "abc"(%y) : (i64) -> i32
  return %z : i32
}
```

MLIR Operations

- Operations always have a name and source location info
- Operations may have:
  - Arbitrary number of SSA results and operands
  - Attributes: guaranteed constant values
  - Block operands: e.g. for branch operations
  - Regions: discussed later
  - Custom printing/parsing - or use the more verbose generic syntax

```
%size = tensor.dim %T, 1 : tensor<1024x? x f32>
// Dimension to extract is a guaranteed integer constant, an attribute .
%x = memref.alloc() : memref<1024x64xf32>
%y = affine.load %x[%a, %b] : memref<1024x64xf32>
```

Operations with Regions

- An MLIR Region is a list of blocks
- Operations in MLIR can have nested regions

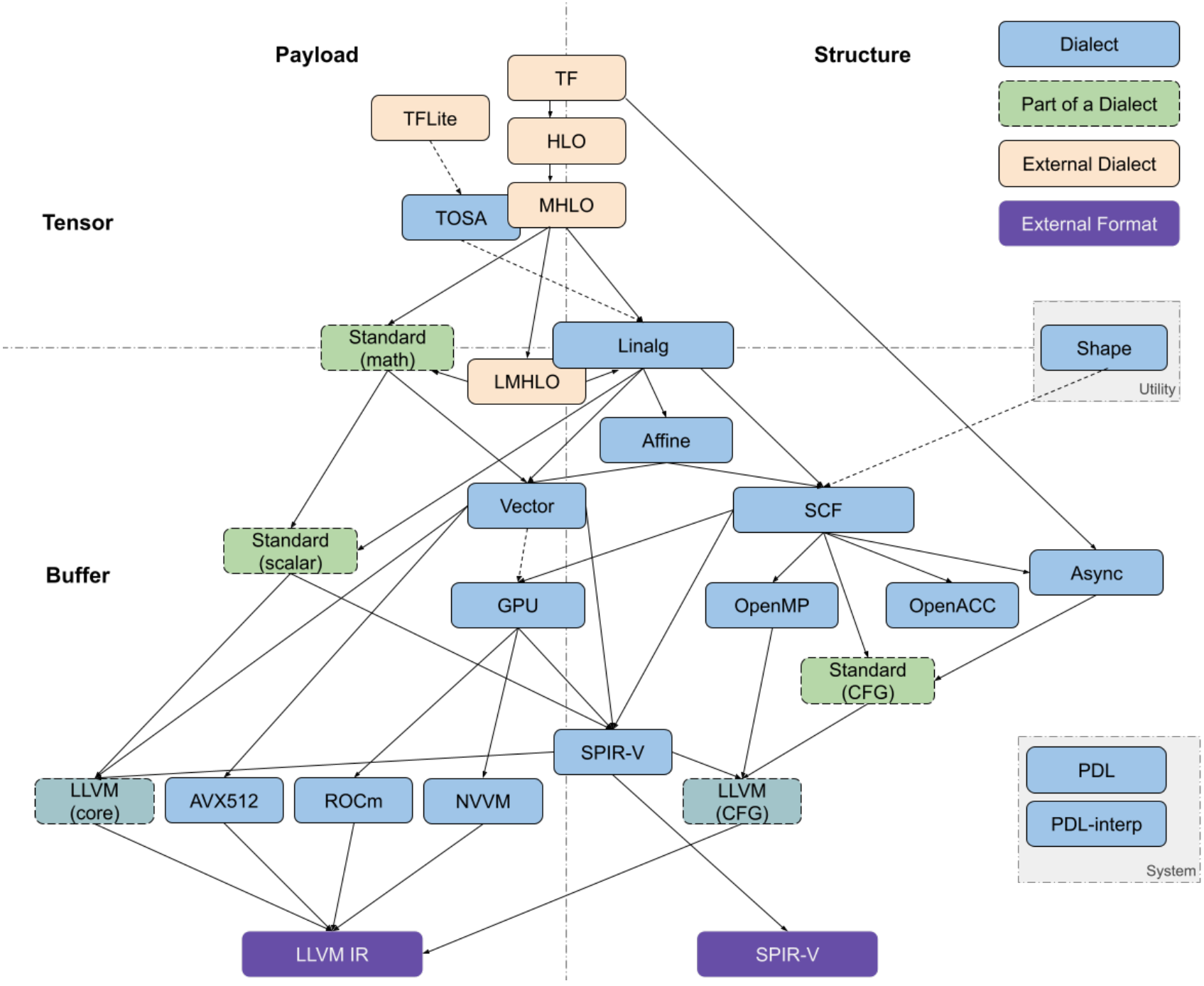
```
%2 = xla.fusion (%0 : tensor<f32>,
                 %1 : tensor<f32>) : tensor<f32> {
^bb0(%a0 : tensor<f32>, %a1 : tensor<f32>):
  %x0 = xla.add %a0, %a1 : tensor<f32>
  %x1 = xla.relu  %x0 : tensor<f32>
  return %x1
}
```

```
func.func @loop_nest_unroll(%arg0: index) {
  affine.for %arg1 = 0 to 100 step 2 {
    affine.for %arg2 = 0 to #map1(%arg0) {
      %0 = "foo"()  :  ()  -> i32
    }
  }
  return
}
```

- Can be used to represent:
  - functional control flow
  - fusion nodes
  - closures/lambdas
  - structured looping/conditional constructs (*for*, *if*, *while*)
  - Parallelism abstractions like OpenMP
  - Launch/dispatch kernel abstractions `gpu.launch`

Dialects in MLIR

- A collection of operations and types suitable for a specific task
- Typically correspond to a programming model, frontend, or a backend
- Example dialects: TensorFlow dialect, LLVM dialect, Affine dialect, NVIDIA GPU dialect
- You can have a mix of dialects


Payload
Structure
Tensor
Buffer
Dialect
Part of a Dialect
External Dialect
External Format
TF
TFLite
HLO
MHLO
TOSA
Standard (math)
Linalg
LMHLO
Shape
Utility
Affine
Vector
SCF
Standard (scalar)
Async
GPU
OpenMP
OpenACC
Standard (CFG)
SPIR-V
LLVM (core)
AVX512
ROCm
NVVM
LLVM (CFG)
PDL
PDL-interp
System
LLVM IR
SPIR-V

Outline

- MLIR Representation
- Polyhedral Notions in MLIR
- Analyses and Transformations

Affine expressions in MLIR

- Affine for functions is linear + constant
  - Addition of identifiers, multiplication with a constant, floordiv, mod, ceildiv with respect to a positive constant
- Examples of affine functions of i, j:
  $i + j$, $2i - j$, $i + 1$, $2i + 5$,
  $i/128 + 1$, $i\%8$, $(i + j)/8$,
  $((d0 * 9216 + d1 * 128) \bmod 294912)$ floordiv 147456
- Not affine: ij, i/j, j/N, $i^2 + j^2$, $a[j]$

Affine Maps

- An affine map maps zero or more identifiers to one or more result affine expressions

$$
\begin{aligned}
\#map1 &= (d0) \rightarrow ((d0 \text{ floordiv } 4) \bmod 2) \\
\#map2 &= (d0) \rightarrow (d0 - 4) \\
\#map3 &= (d0) \rightarrow (d0 + 4) \\
\#map4 &= (d0, d1) \rightarrow (d0 * 16 - d1 + 15) \\
\#map5 &= (d0, d1, d2, d3) \rightarrow (d2 - d0 * 16, d3 - d1 * 16)
\end{aligned}
$$

- Why affine maps? What can they express?
  - Loop IV mappings for nearly every useful loop transformation, data layout transformations, placement functions / processor mappings / distributions: block, cyclic, block-cyclic, multi-dimensional array subscripts, loop bound expressions, conditionals

Where are affine maps Used in MLIR?

1. IV remappings: to map old IVs to new IVs

(i, j)	Identity
(j, i)	Interchange
$(i, i + j)$	Skew j
$(2i, j)$	Scale i by two
$(i, j + 1)$	Shift j
$(\lfloor \frac{i}{32} \rfloor, \lfloor \frac{j}{32} \rfloor, i, j)$	Tile (rectangular)
...	

```
#map = (d0) -> (2*d0 - 1)

affine.for %i = 0 to #map(%N) {
  affine.for %j = 0 to 3  {
    %v = affine.load %0[%i + %j] : memref<100xf32>
    "op1"(%v) : (f32)  -> ()
  }
}
%w = "op"(%s, %t) {map: affine_map<(d0, d1) -> (d1, d0)>}
```

2. Loop bounds
3. Memref access subscripts
4. As an attribute for any operation

Polyhedral Structures in the IR

1. Affine expressions
   - Eg: $(d0 + 1) \bmod 2$
2. Affine maps
   - Eg: $(d0, d1) \rightarrow (d1, d0/128, d0 \bmod 128)$
3. Integer sets
   - Eg: $\{(d0, d1)[s1] : d0 \geq 0, d0 \leq s1, d1 == 512\}$
4. Affine apply operation (`affine.apply`)

```
%a = affine.apply (d0, d1) -> (d0 + d1) (%i, %j)
```

5. Affine 'for' operation (`affine.for`)
6. Affine 'if' operation (`affine.if`)

Polyhedral Structures in the IR

1. Affine expressions
   - Eg: $(d0 + 1) \bmod 2$
2. Affine maps
   - Eg: $(d0, d1) \rightarrow (d1, d0/128, d0 \bmod 128)$
3. Integer sets
   - Eg: $\{(d0, d1)[s1] : d0 \geq 0, d0 \leq s1, d1 == 512\}$
4. Affine apply operation (`affine.apply`)

```
%a = affine.apply (d0, d1) -> (d0 + d1) (%i, %j)
```

5. Affine 'for' operation (`affine.for`)
6. Affine 'if' operation (`affine.if`)

Types Relevant for Dense Matrices/Tensors

1. *tensor* A value that is a multi-dimensional array of elemental values

```
%d = "tf.Add"(%e, %f) : (tensor<?x42x?xf32>, tensor<?x42x?xf32>) -> tensor<?x42x?xf32>
```

2. *memref* A buffer in memory or a view on a buffer, has a layout map, memory space qualifier, symbols bound to its dynamic dimensions

```
%N = affine.apply (d0) -> (8 * (d0 ceildiv 8))  (%S)
%M = affine.apply (d0) -> (2 * d0) (%N)
#tmap = affine_map<(d0, d1) -> (d1 floordiv 32, d0 floordiv  128,  d1 mod 32, d0 mod 128)>
#shift  = affine_map<(d0, d1)[s0, s1]  -> (d0 + s0,  d1 + s1)>
%A = memref.alloc() : memref<1024x64xf32, #tmap, 0>
%B = memref.alloc(%M, %N)[%x, %y] : memref<?x?xf32, #tmap, 1>
%C = memref.alloc(%M, %M)[%x, %y] : memref<?x?xf32, #shift, 1>
```

Integer Sets

- An integer set is primarily used for conditionals
- It is also powerful as an attribute to specify constraints on symbols (esp. shape symbols)

```
// An example two-dimensional integer set with two symbols.
#set = affine_set<(d0, d1)[s0, s1]
   : d0 >= 0, -d0 + s0 - 1 >= 0, d1 >= 0, -d1 + s1 - 1 >= 0>

affine.if #set(%i, %j)[%M, %N] {
  %v = affine.load %A[%i] : memref<256xf32>
}
```

Outline

- MLIR Representation
- Polyhedral Notions in MLIR
- Analyses and Transformations

Analyses and Transformations: what's curently present

- Several techniques are available:
  - GCD test
    $2i + 4j - 8k - 1 = 0 \quad \rightarrow$ No solution
  - GCD tightening:
    $16i \geq 16j - 15, \;\; 16i \leq 16j \quad \rightarrow$ i = j
  - Gaussian elimination
    $i = j - 1, \;\; 0 \leq j \leq i$
  - Fourier-Motzkin elimination: eliminate a variable from a system of linear inequalities
    $i \leq j + 1, \;\; j = k, \;\; k \leq 16, \;\; i \geq 32$
- FlatAffineConstraints
- Fast Presburger library (FPL)

GENERALIZED SLICING-BASED LOOP FUSION

- A generalized slicing-based loop fusion approach
- Can trade off redundant computation for locality / memory minimization
- Post fusion, forwarding of 'affine.store' to affine.load, elimination of intermediate arrays can be performed
- Fixed size local buffers are created when possible to pass intermediate data

```
affine.for %i = 0 to 64 {
  %v = affine.load %in[%i] : memref<64xf32>
  affine.store %v, %out[%i floordiv 4, %i mod 4]
       : memref<16x4xf32>
}

affine.for %i = 0 to 16 {
  affine.for %j = 0 to 4 {
    %w = affine.load %out[%i, %j] : memref<16x4xf32>
    "foo"(%w) : (f32) -> ()
  }
}
```

```
affine.for %i = 0 to 16 {
  affine.for %j = 0 to 4 {
    %v = affine.load %in[4 * %i + %j] : memref<64xf32>
    "foo"(%v) : (f32) -> ()
  }
}
```