Introduction to Multi Level Intermediate Representation

Prof. Uday Kumar Reddy
Ayush Raina

March 31, 2025

Multi Level Intermediate Representation (MLIR)

MLIR stands for multi level intermediate representation, it is a flexible and extensible compiler under the LLVM Project. It allows representing and optimizing code across multiple abstraction levels. It supports:

1. **Multiple Dialects:** Custom IR for different domains (e.g Tensorflow, PyTorch etc)
2. **Progressive Lowering:** Transforming high level constructs to lower-level forms.
3. **Reusability:** Shared Optimizations across different compiler pipelines.

Question. What do we mean by multiple abstraction levels ?
In MLIR, multiple abstraction levels mean representing code at different stages of computation ranging from high level domain specific applications like math operations to low level machine instructions.

- **High Level Abstraction (Tensor Dialect):** In this level of abstraction, we represent code in a domain specific way. For example, operations like matrix multiplication, convolution look like how we write them in Tensorflow or PyTorch.

```
%result = "linalg.matmul"(%A, %B) : (tensor<2x3xf32>, tensor<3x2xf32>) -> tensor<2x2xf32>
```

- **Mid Level Abstraction (Affine Dialect):** In this level of abstraction, we can see the linear algebra transformations and operations like matmul converted to a loop nest.

```
affine.for %i = 0 to 2 {
    affine.for %j = 0 to 2 {
        affine.for %k = 0 to 3 {
            %c = arith.mulf %A[%i, %k], %B[%k, %j]
            %sum = arith.addf %sum, %c
        }
    }
}
```

- **Low Level Abstraction (LLVM Dialect):** In this level of abstraction, we see machine level instruction which directly operates on pointers and raw data like load, store, getelementptr etc as shown below.

```
%ptr = llvm.getelementptr %A[%i, %k]
%val = llvm.load %ptr
```

MLIR is designed to support progressive lowering (Tensor → Affine → LLVM) and reusability of optimizations across different compiler pipelines, for example - Loop Unrolling is useful for both HPC and Tensorflow programs.

Core Structure of MLIR (Hierarchy)

1. **Operation (Op):** The smallest unit of computation in MLIR. In the example shown below **arith.addf** is the operation which adds two floating point numbers.

```
%sum = arith.addf %a, %b : f32
```

2. **Value:** Data produced or consumed by operations in SSA Form. In the above example **%sum, %a, %b** are values.
3. **Block:** A sequence of operations (similar to basic block in LLVM). Block also defines control flow by having terminators like **return** etc.

4. **Region:** A container of blocks, enabling nested control flow. For example loops or functions. An **affine.for** has region for loop contents.

5. **Attribute:** Immutable metadata attached to operations. For example constants, symbolic info etc.

```
%cst = arith.constant 10 : i32 // 10 is an attribute
```

6. **Type:** Describes the kind of data a value holds. For example **f32, tensor**$< 4 \times f32 >$ etc.

Operation (Op)

An operation represents a single computational action, similar to an instruction in other intermediate representation like LLVM IR. Each Operation includes the following components:

1. **Operands**: Values that are consumed by the operation (Inputs).
2. **Results**: Values that are produced by the operation (Outputs).
3. **Attributes**: Immutable metadata associated with the operation.
4. **Regions**: Nested containers of Operations forming control flow structures (optional).
5. **Types**: Data types for operands and results.

Basic Structure of an Operation

```
%result = <dialect>.<op_name> <operand(s)> : <type> // Structure 1
%sum = arith.addf %a, %b : f32 // Example 1

%result = <dialect>.<op_name> <operand(s)> {attributes} : <type(s)> // Structure 2
%sum = arith.addf %a, %b {fastmath = #fastmath<reassoc>} : f32 // Example 2
```

In example 2 shown above,this **fastmath** attribute is a hint for floating point arithmetic optimization. With **reassoc** compiler can reorder floating point operations.

Types of Operations

1. **Simple Operation:** are operations which perform basic computations. For example: **arith.addf, arith.subf** etc.
2. **Control Flow Operations:** are operations which manage loops and conditionals. For example: **affine.for, scf.if** etc.
3. **Memory Operations:** are operations which handle loads/stores. For example: **memref.load, memref.store** etc.
4. **Custom Operations:** are user defined operations in custom dialects.

Here is an example of loop operation with a region:

```
affine.for %i = 0 to 10 {
    %val = memref.load %A[%i] : memref<10xf32> // Load from memory
    %new_val = arith.addf %val, %c : f32 // Add a constant
    memref.store %new_val, %A[%i] : memref<10xf32> // Store back to memory
}
```

Values

The **Value** represents the data produced by or consumed by operations. We can think of it as a name for output of an operation. Values are always in SSA form means they are assigned once.

Types of Values:
1. **Operation Results:** Outputs of an operation.

```
%sum = arith.addi %a, %b : i32
```

Here **%sum** is a value produced by operation **arith.addi**. Operands **%a, %b** are input values.

2. **Block Arguments:** Inputs to a block. For example: function arguments.

```
func.func @add(%x: i32, %y: i32) -> i32 {
    %sum = arith.addi %x, %y : i32
    return %sum : i32
}
```

Here **%x, %y** are block arguments.

3. Constants: **Immutable** values (via attributes).

```
%c = arith.constant 42 : i32
```

Here **%c** holds value 42.

3. **Function Results:** Outputs of a function.
Now consider the following example:

```
func.func @square(%x: i32) -> i32 {
  %prod = arith.muli %x, %x : i32
  return %prod : i32
}
```

Here **%x** is block argument, **%prod** is result of operation **arith.muli**. Here **%prod** is a function return value.

Block

A block is a sequence of operations that execute linearly. It acts as container for:

1. **Arguments:** Input values like function parameters.
2. **Operations:** A list of instructions.
3. **Terminators:** The last operation which exits the block.

Structure of a Block

```
^block_name(%arg1: type, %arg2: type):
    %result = arith.addi %arg1, %arg2 : i32
    cf.br ^next_block(%result)
```

In the above example **^ block_name** is the block name, **%arg1, %arg2** are block arguments and **cf.br** is the terminator operation which branches to next block. Consider another example as follows in which we have blocks inside a function.

```
func.func @add(%a: i32, %b: i32) -> i32 {
  ^entry:
    %sum = arith.addi %a, %b : i32
    return %sum : i32
}
```

There can be multiple blocks inside a function also as shown in following example:

```
func.func @conditional(%x: i32) -> i32 {
  ^entry:
    %c = arith.constant 0 : i32
    %cond = arith.cmpi eq, %x, %c : i32
    cf.cond_br %cond, ^then, ^else

  ^then:
    %res1 = arith.addi %x, %c : i32
    cf.br ^exit(%res1)

  ^else:
    %res2 = arith.muli %x, %c : i32
    cf.br ^exit(%res2)

  ^exit(%result: i32):
    return %result : i32
}
```

Regions

A **Region** is a container for one or more Blocks. It defines local scope where operations and values exist. In this case we can think Blocks as instructions and region as collections of these instructions.
In MLIR, regions are always inside an operation. Consider the following example:

```
func.func @add(%a: i32, %b: i32) -> i32 {
  ^entry:                        // Block
    %sum = arith.addi %a, %b : i32
    return %sum : i32
}
```

In this **func.func** this operation defines a function. Yes functions are a special case of operations in MLIR. **Region** holds the entire function body and **Block** contains the actual computation. Here is another example with multiple regions.

```
scf.if %cond -> i32 {
  ^then:                    // Region 1 (true branch)
    %x = arith.addi %a, %b : i32
    scf.yield %x : i32
} else {
  ^else:                    // Region 2 (false branch)
    %y = arith.muli %a, %b : i32
    scf.yield %y : i32
}
```

Here if **%cond** is true then **Region 1** is executed otherwise **Region 2** is executed. Image attached below shows difference between block and region.

Regions vs. Blocks

Feature	Region	Block
Purpose	Groups multiple blocks	Holds a sequence of operations
Ownership	Owned by **operations**	Owned by **regions**
Scope	Defines local scope (SSA applies)	Local to a region
Examples	Function body, loop body	Basic block in a function

Figure 1: Difference between Block and Region

Attributes

Attributes are compile time constants that attach extra information to operations. They are immmutable and cannot be changed during execution. We can think them as annotations that give operations some special properties.

Structure of an Attribute: Attributes are generally written in $\{key = value\}$ format. For example:

```
%sum = arith.addf %a, %b {fastmath = #fastmath<reassoc>} : f32
```

We have already discussed what this means. Here is another example:

```
func.func @my_func() attributes {inline = true} {
  return
}
```

Here **inline = true** suggests that this function should be inlined.

Types

Types describe the shape and format of data used in operations. Every value in MLIR has a type. Here are some basic MLIR Type categories:

Type	Example	Description
Integer	`i32, i64`	Signed/unsigned fixed-width integers.
Floating-Point	`f32, f64`	IEEE floating-point types.
Index	`index`	Special integer for loop bounds, etc.
MemRef	`memref<4x4xf32>`	Memory reference (for tensors/arrays).
Tensor	`tensor<3x3xi32>`	Immutable multi-dimensional array.
Tuple	`tuple<i32, f32>`	Group multiple values of different types.
Function	`(i32, i32) -> i32`	Input-output type of a function.
Opaque	`!mydialect.custom`	User-defined type (from a dialect).

Table 1: Basic MLIR Type Categories

Here are some examples:

```
/* Example 1 */
%sum = arith.addi %a, %b : i32

/* Example 2 */
%t = tensor.from_elements %a, %b : tensor<2xf32>

/* Example 3 */
func.func @add(%a: i32, %b: i32) -> i32 {
    return %a : i32
}
```

In Example 1, we have **Type: i32**. In Example 2, we have **Type: tensor<2xf32>** which is a 2 element tensor of float. In example 3, we have **Function Type: (i32,i32) → i32**

Affine Dialect

The affine dialect in MLIR provides structured control flow and memory access patterns.
1. **Affine Loops and Conditionals:** These loops have a start, end and a step size, all expressed using affine expressions. Here is the syntax:

```
// Syntax
affine.for %i = <lower_bound> to <upper_bound> step <step_size> {
  // Loop body
}

// Example
affine.for %i = 0 to 10 step 1 {
  // Loop body
}

// If Conditionals
affine.if (%i < 5) {
  // Executes if %i < 5
}
```