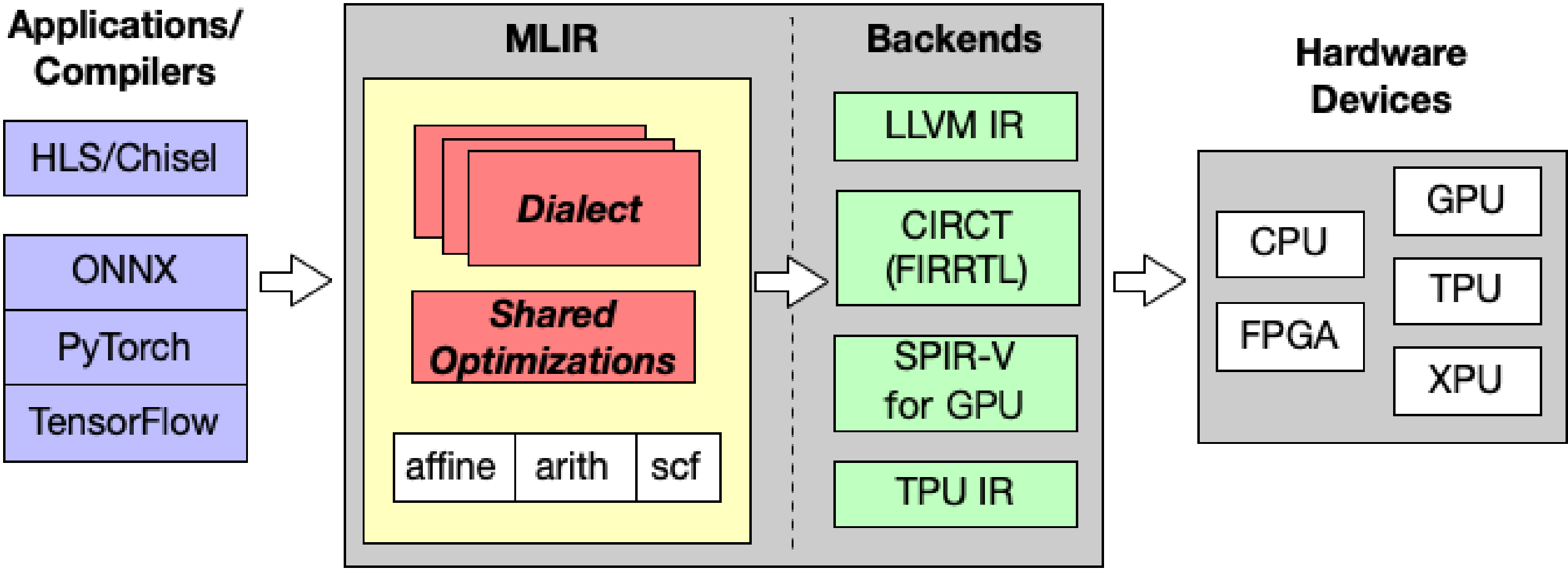
Compilers 101

MLIR - Part 2

Previously

- MLIR overview
- MLIR practical applications

MLIR



http://lastweek.io/notes/MLIR/

Today

- Available transformations and optimizations overview
- Custom dialects
- Writing a pass
- MLIR connections with other languages
- Debugging

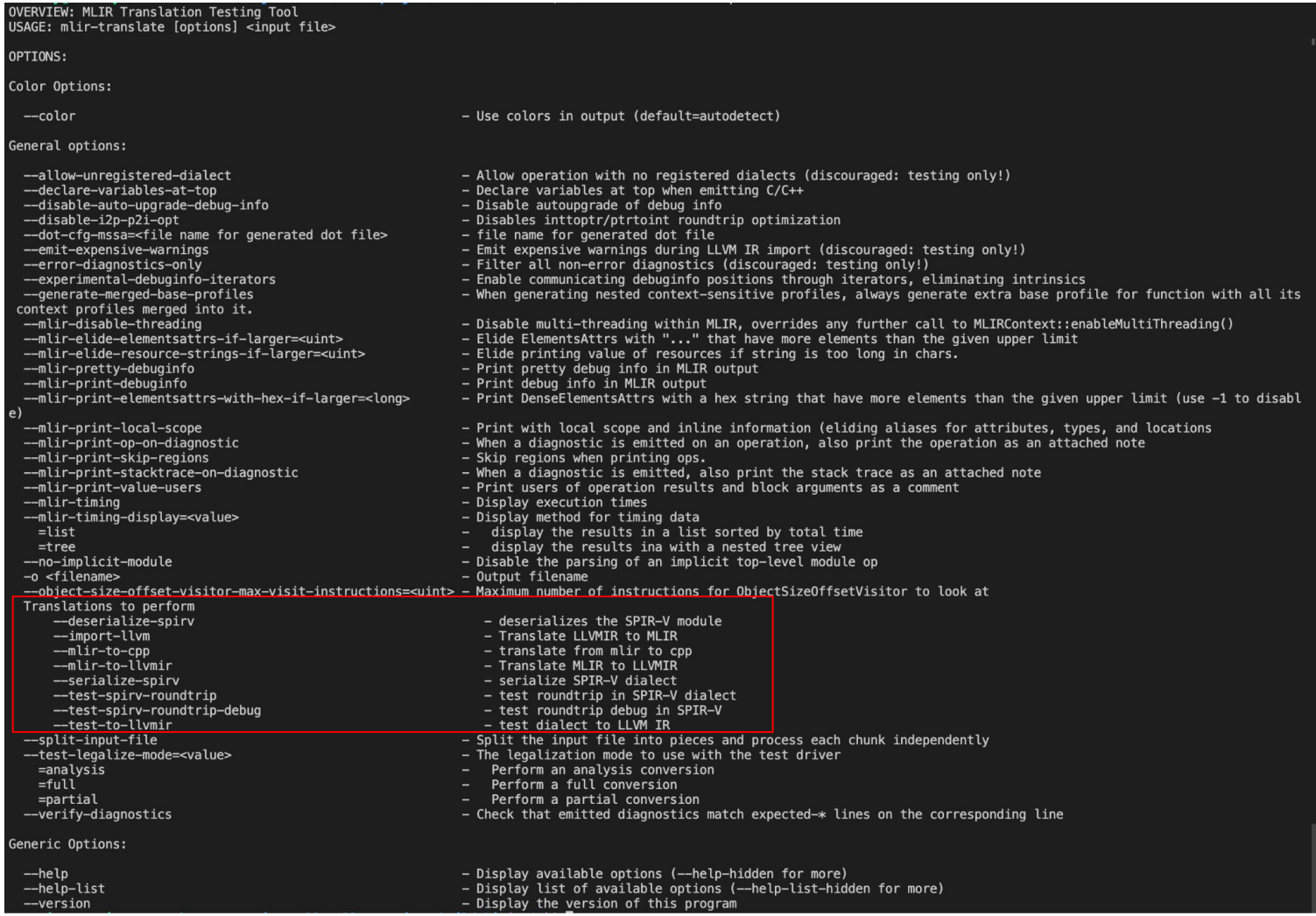
mlir-translate

Translate MLIR to…

- LLVM
- SPIR-V
- C++

Input: MLIR

Output: target representation



List of targets:
https://github.com/llvm/llvm-project/tree/main/mlir/lib/Target

mlir-opt

Responsible for optimizing MLIR code

Allows to run passes on MLIR

Input: MLIR

Output: MLIR

Help:  https://godbolt.org/z/x5zjxbq4T

mlir-opt + mlir-translate

opt

https://godbolt.org/z/75a8xTrGx

translate

https://godbolt.org/z/z9s9q1x1P

Example:

Translate the following code to LLVM IR

```
func.func @vector_add(%A: memref<100xf32>, %B: memref<100xf32>, %C: memref<100xf32>) {
  %c0 = arith.constant 0 : index
  %c100 = arith.constant 100 : index
  %c1 = arith.constant 1 : index
  scf.for %i = %c0 to %c100 step %c1 {
    %a = memref.load %A[%i] : memref<100xf32>
    %b = memref.load %B[%i] : memref<100xf32>
    %sum = arith.addf %a, %b : f32
    memref.store %sum, %C[%i] : memref<100xf32>
  }
  return
}
```

1. convert to LLVM dialect
2. translate MLIR with LLVM dialect to LLVM IR

Using MLIR

From test-tensor-e2e.mlir
git@f2f61a99f7f754f3e4

No optimization

.mlir **mlir-opt** -arith-bufferize
-linalg-bufferize -tensor-bufferize
-func-bufferize -finalizing-bufferize
-buffer-deallocation-pipeline
-convert-bufferization-to-memref
-convert-linalg-to-loops
-convert-arith-to-llvm
-convert-scf-to-cf -convert-cf-to-llvm
-finalize-memref-to-llvm
-convert-func-to-llvm
-reconcile-unrealized-casts

.mlir

Still MLIR

.mlir **mlir-translate** -mlir-to-llvmir .ll

.ll **clang** -x ir -O3 .exe (not that fast)

* assuming environment is configured correctly, eventual (c)make, etc

Source: https://llvm.org/devmtg/2023-10/slides/techtalks/Zinenko-MLIRisNotAnMLCompiler.pdf

Content

- **Available transformations and optimizations overview**
- Custom dialects
- Writing a pass
- MLIR connections with other languages
- Debugging

Rewind back to LLVM IR

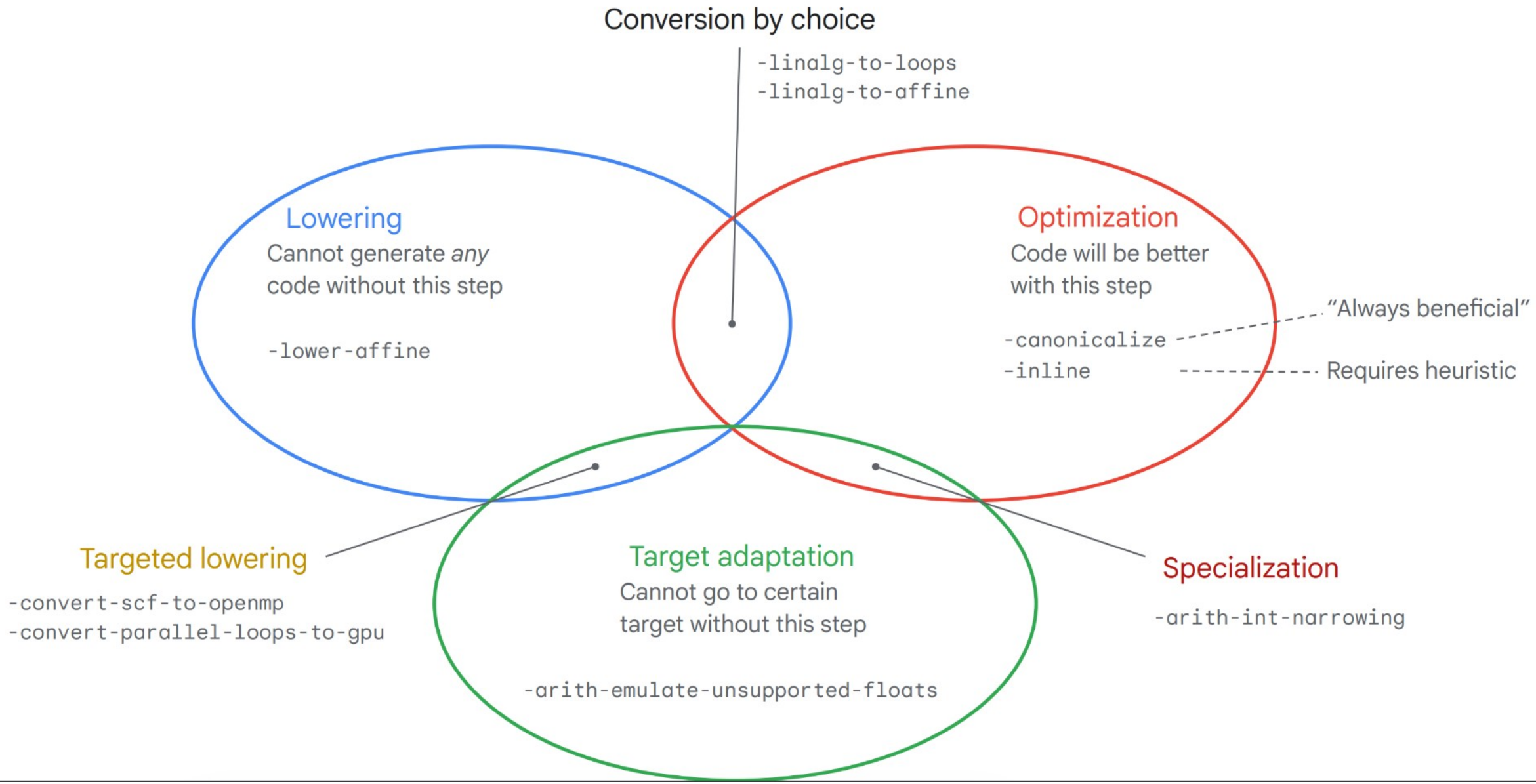
LLVM IR passes classifications

- Types
  - Analysis pass
  - Transformation pass
  - Utility pass
- Run scope
  - Module pass
  - Function pass
  - Loop pass
  - …

What about MLIR classification?

- It could be the same
- but in addition to that usually people outline different classification rules

Categorizing `mlir-opt` Passes: Function



https://llvm.org/devmtg/2023-10/slides/techtalks/Zinenko-MLIRisNotAnMLCompiler.pdf

Available transformations

Reference page:

- https://mlir.llvm.org/docs/Passes/

Lowering

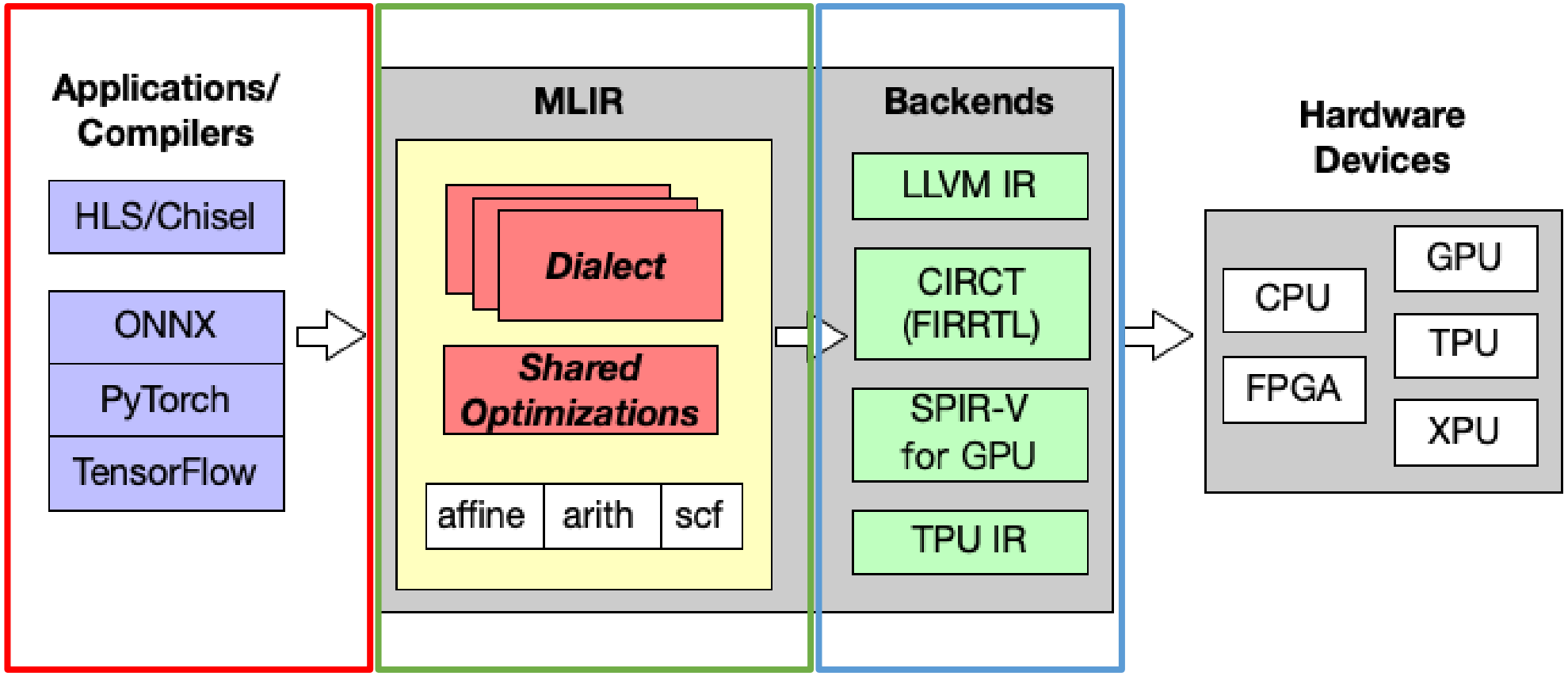
Lowering is the process of transforming a high-level representation of a program into a lower-level representation

Lowering involves transforming this high-level representation into a more concrete, lower-level representation that is closer to the target hardware's capabilities and constraints.

One of common lowering pipelines:

ML framework/DSL -> (TOSA) -> Linalg -> Affine -> SCF -> LLVM -> LLVM IR

Lowering



http://lastweek.io/notes/MLIR/

One of common lowering pipelines:

ML framework/DSL -> (TOSA) -> Linalg -> Affine -> SCF -> LLVM -> LLVM IR

TOSA to Linalg

https://godbolt.org/z/T9G4zPb99

https://github.com/llvm/llvm-project/blob/76aa042dde6ba9ba57c680950f5818259ee02690/mlir/test/Dialect/Tosa/ops.mlir#L226C12-L226C20

```
func.func @test_add(%arg0: tensor<13x21x1xf32>, %arg1: tensor<13x21x3xf32>) -> tensor<13x21x3xf32> {
  %0 = tosa.add %arg0, %arg1 : (tensor<13x21x1xf32>, tensor<13x21x3xf32>) -> tensor<13x21x3xf32>
  return %0 : tensor<13x21x3xf32>
}
```

mlir-opt -pass-pipeline="builtin.module(func.func(tosa-to-linalg))"

```
#map = affine_map<(d0, d1, d2) -> (d0, d1, 0)>
#map1 = affine_map<(d0, d1, d2) -> (d0, d1, d2)>
module {
  func.func @test_add(%arg0: tensor<13x21x1xf32>, %arg1: tensor<13x21x3xf32>) -> tensor<13x21x3xf32> {
    %0 = tensor.empty() : tensor<13x21x3xf32>
    %1 = linalg.generic {indexing_maps = [#map, #map1, #map1], iterator_types = ["parallel", "parallel", "parallel"]} ins
    (%arg0, %arg1 : tensor<13x21x1xf32>, tensor<13x21x3xf32>) outs(%0 : tensor<13x21x3xf32>) {
    ^bb0(%in: f32, %in_0: f32, %out: f32):
      %2 = arith.addf %in, %in_0 : f32
      linalg.yield %2 : f32
    } -> tensor<13x21x3xf32>
    return %1 : tensor<13x21x3xf32>
  }
}
```

https://github.com/llvm/llvm-project/blob/76aa042dde6ba9ba57c680950f5818259ee02690/mlir/test/Dialect/Linalg/affine.mlir#L6

Linalg conversion to parallel loops

```
1  func.func @matmul(%arg0: memref<?xi8>, %M: index, %N: index, %K: index) {
2    %c0 = arith.constant 0 : index
3    %c1 = arith.constant 1 : index
4    %A = memref.view %arg0[%c0][%M, %K] : memref<?xi8> to memref<?x?xf32>
5    %B = memref.view %arg0[%c0][%K, %N] : memref<?xi8> to memref<?x?xf32>
6    %C = memref.view %arg0[%c0][%M, %N] : memref<?xi8> to memref<?x?xf32>
7    linalg.matmul ins(%A, %B: memref<?x?xf32>, memref<?x?xf32>)
8                  outs(%C: memref<?x?xf32>)
9    return
10 }
```

mlir-opt -pass-pipeline="builtin.module(func.func(convert-linalg-to-parallel-loops))"

https://godbolt.org/z/b3jjfhx6j

```
1  module {
2    func.func @matmul(%arg0: memref<?xi8>, %arg1: index, %arg2: index, %arg3: index) {
3      %c1 = arith.constant 1 : index
4      %c0 = arith.constant 0 : index
5      %view = memref.view %arg0[%c0][%arg1, %arg3] : memref<?xi8> to memref<?x?xf32>
6      %view_0 = memref.view %arg0[%c0][%arg3, %arg2] : memref<?xi8> to memref<?x?xf32>
7      %view_1 = memref.view %arg0[%c0][%arg1, %arg2] : memref<?xi8> to memref<?x?xf32>
8      scf.parallel (%arg4, %arg5) = (%c0, %c0) to (%arg1, %arg2) step (%c1, %c1) {
9        scf.for %arg6 = %c0 to %arg3 step %c1 {
10         %0 = memref.load %view[%arg4, %arg6] : memref<?x?xf32>
11         %1 = memref.load %view_0[%arg6, %arg5] : memref<?x?xf32>
12         %2 = memref.load %view_1[%arg4, %arg5] : memref<?x?xf32>
13         %3 = arith.mulf %0, %1 : f32
14         %4 = arith.addf %2, %3 : f32
15         memref.store %4, %view_1[%arg4, %arg5] : memref<?x?xf32>
16       }
17       scf.reduce
18     }
19     return
20   }
21 }
```

Affine loops vectorization

https://godbolt.org/z/T3q1qzxqc

```
1  func.func @vector_add(%A: memref<128xf32>, %B: memref<128xf32>, %C: memref<128xf32>) {
2    affine.for %i = 0 to 128 {
3      %a = affine.load %A[%i] : memref<128xf32>
4      %b = affine.load %B[%i] : memref<128xf32>
5      %sum = arith.addf %a, %b : f32
6      affine.store %sum, %C[%i] : memref<128xf32>
7    }
8    return
9  }
```

mlir-opt -affine-super-vectorize=virtual-vector-size=128

```
1  module {
2    func.func @vector_add(%arg0: memref<128xf32>, %arg1: memref<128xf32>, %arg2: memref<128xf32>) {
3      affine.for %arg3 = 0 to 128 step 128 {
4        %cst = arith.constant 0.000000e+00 : f32
5        %0 = vector.transfer_read %arg0[%arg3], %cst : memref<128xf32>, vector<128xf32>
6        %cst_0 = arith.constant 0.000000e+00 : f32
7        %1 = vector.transfer_read %arg1[%arg3], %cst_0 : memref<128xf32>, vector<128xf32>
8        %2 = arith.addf %0, %1 : vector<128xf32>
9        vector.transfer_write %2, %arg2[%arg3] : vector<128xf32>, memref<128xf32>
10     }
11     return
12   }
13 }
```

Canonicalizer

https://godbolt.org/z/Yd47xj6df

The purpose of the canonicalizer - transform MLIR code into a canonical form (to a standardized, simplified representation of code that allows for easier analysis and optimization)

```
func.func @vector_add(%A: memref<100xf32>, %B: memref<100xf32>, %C: memref<100xf32>) {
  %c0 = arith.constant 0 : index
  %c50 = arith.constant 50 : index
  %c1 = arith.constant 1 : index
  %constant100 = arith.addi %c50, %c50 : index
  scf.for %i = %c0 to %constant100 step %c1 {
    %a = memref.load %A[%i] : memref<100xf32>
    %b = memref.load %B[%i] : memref<100xf32>
    %sum = arith.addf %a, %b : f32
    memref.store %sum, %C[%i] : memref<100xf32>
  }
  return
}
```

```
module {
  func.func @vector_add(%arg0: memref<100xf32>, %arg1: memref<100xf32>, %arg2: memref<100xf32>) {
    %c0 = arith.constant 0 : index
    %c1 = arith.constant 1 : index
    %c100 = arith.constant 100 : index
    scf.for %arg3 = %c0 to %c100 step %c1 {
      %0 = memref.load %arg0[%arg3] : memref<100xf32>
      %1 = memref.load %arg1[%arg3] : memref<100xf32>
      %2 = arith.addf %0, %1 : f32
      memref.store %2, %arg2[%arg3] : memref<100xf32>
    }
    return
  }
}
```

Canonicalization is an important part of compiler IR design: it makes it easier to implement reliable compiler transformations and to reason about what is better or worse in the code, and it forces interesting discussions about the goals of a particular level of IR. Dan Gohman wrote an article exploring these issues; it is worth reading if you're not familiar with these concepts.

From https://mlir.llvm.org/docs/Canonicalization/

Canonicalization patterns

How canonicalizer understands what to optimize?

```
void arith::AddIOp::getCanonicalizationPatterns(RewritePatternSet &patterns,
                                                MLIRContext *context) {
  patterns.add<AddIAddConstant, AddISubConstantRHS, AddISubConstantLHS,
               AddIMulNegativeOneRhs, AddIMulNegativeOneLhs>(context);
}
```

https://github.com/llvm/llvm-project/blob/02dfbbff1937b3e2c1ee1cd4a5ad0a9f03ee23ea/mlir/lib/Dialect/Arith/IR/ArithOps.cpp#L319-L323

```
// addi(addi(x, c0), c1) -> addi(x, c0 + c1)
def AddIAddConstant :
    Pat<(Arith_AddIOp:$res
          (Arith_AddIOp $x, (ConstantLikeMatcher APIntAttr:$c0), $ovf1),
          (ConstantLikeMatcher APIntAttr:$c1), $ovf2),
        (Arith_AddIOp $x, (Arith_ConstantOp (AddIntAttrs $res, $c0, $c1)),
                DefOverflow)>;
```

https://github.com/llvm/llvm-project/blob/7a484d3a1f630ba9ce7b22e744818be974971470/mlir/include/mlir/Dialect/Arith/IR/ArithOps.td#L244

```
209  def Arith_AddIOp : Arith_IntBinaryOpWithOverflowFlags<"addi", [Commutative]> {
210    let summary = "integer addition operation";
211    let description = [{
212      Performs N-bit addition on the operands. The operands are interpreted as
213      unsigned bitvectors. The result is represented by a bitvector containing the
214      mathematical value of the addition modulo 2^n, where `n` is the bitwidth.
215      Because `arith` integers use a two's complement representation, this operation
216      is applicable on both signed and unsigned integer operands.
217
218      The `addi` operation takes two operands and returns one result, each of
219      these is required to be the same type. This type may be an integer scalar type,
220      a vector whose element type is integer, or a tensor of integers.
221
222      This op supports `nuw`/`nsw` overflow flags which stands stand for
223      "No Unsigned Wrap" and "No Signed Wrap", respectively. If the `nuw` and/or
224      `nsw` flags are present, and an unsigned/signed overflow occurs
225      (respectively), the result is poison.
226
227      Example:
228
229      ```mlir
230      // Scalar addition.
231      %a = arith.addi %b, %c : i64
232
233      // Scalar addition with overflow flags.
234      %a = arith.addi %b, %c overflow<nsw, nuw> : i64
235
236      // SIMD vector element-wise addition.
237      %f = arith.addi %g, %h : vector<4xi32>
238
239      // Tensor element-wise addition.
240      %x = arith.addi %y, %z : tensor<4x?xi8>
241      ```
242    }];
243    let hasFolder = 1;
244    let hasCanonicalizer = 1;
245  }
```

Ways to define canonicalization patterns

1. TableGen

```
// addi(addi(x, c0), c1) -> addi(x, c0 + c1)
def AddIAddConstant :
    Pat<(Arith_AddIOp:$res
          (Arith_AddIOp $x, (ConstantLikeMatcher APIntAttr:$c0), $ovf1),
          (ConstantLikeMatcher APIntAttr:$c1), $ovf2),
        (Arith_AddIOp $x, (Arith_ConstantOp (AddIntAttrs $res, $c0, $c1)),
              DefOverflow)>;
```

https://github.com/llvm/llvm-project/blob/02dfbbff1937b3e2c1ee1cd4a5ad0a9f03ee23ea/mlir/lib/Dialect/Arith/IR/ArithCanonicalization.td#L35-L93

2. Implement <OpClass>::canonicalize in C++ code (for more sophisticated cases)

```
LogicalResult SelectOp::canonicalize(SelectOp op, PatternRewriter &rewriter) {
  auto notOp = op.getPred().getDefiningOp<tosa::LogicalNotOp>();
  if (!notOp)
    return failure();
  rewriter.modifyOpInPlace(op, [&]() {
    op.getOperation()->setOperands(
        {notOp.getInput1(), op.getOnFalse(), op.getOnTrue()});
  });
  return success();
}
```

https://github.com/llvm/llvm-project/blob/02dfbbff1937b3e2c1ee1cd4a5ad0a9f03ee23ea/mlir/lib/Dialect/Tosa/IR/TosaCanonicalizations.cpp#L67C15-L76

Patterns

Common structure in MLIR.

There are the following patterns present in pass infrastructure:

- fold and canonicalization (for canonicalizer pass)
- rewrite (operation rewriting/replacement)
- conversion (for conversion passes)

They are applied in random order and multiple times to make sure that IR is canonicalized/rewritten/converted.

More here:

Pattern Rewriting : Generic DAG-to-DAG Rewriting

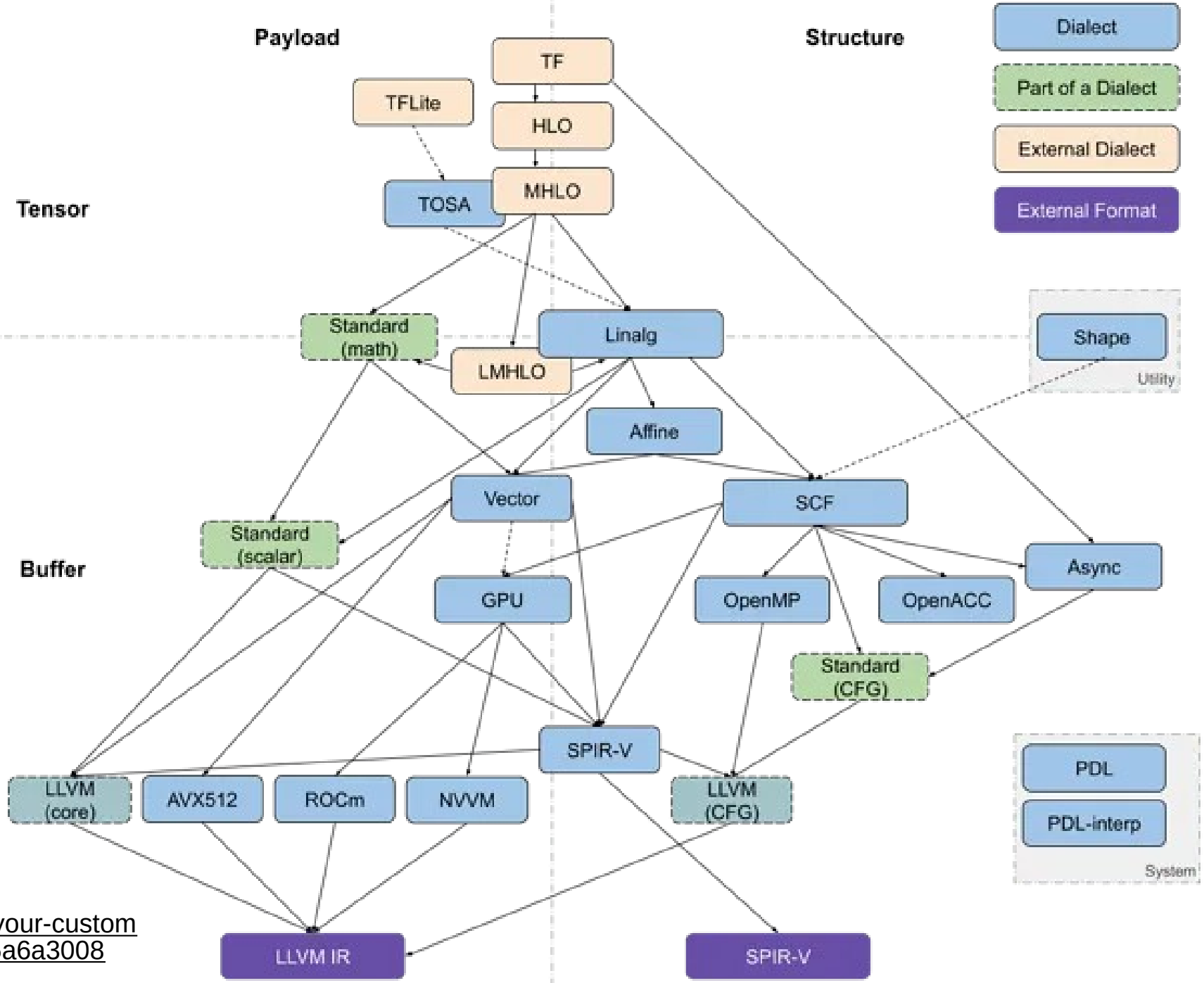
https://mlir.llvm.org/docs/PatternRewriter/

Content

- Available transformations and optimizations overview
- **Custom dialects**
- Writing a pass
- MLIR connections with other languages
- Debugging

Dialects

One of possible ways to illustrate dialects in structured way



https://medium.com/sniper-ai/mlir-tutorial-create-your-custom-dialect-lowering-to-llvm-ir-dialect-system-1-1f125a6a3008

Creating a custom dialect

Dialect definition files are separated:

- `mlir/include/mlir/Dialect/<dialect_name>` (include files)
- `mlir/include/mlir/Dialect/<dialect_name>/IR` (operations include files)
- `mlir/include/mlir/Dialect/<dialect_name>/Transforms` (transforms include files)
- `mlir/lib/Dialect/<dialect_name>` (sources)
- `mlir/lib/Dialect/<dialect_name>/IR` (operations sources)
- `mlir/lib/Dialect/<dialect_name>/Transforms` (transforms sources)

Learn by example: Math dialect

https://github.com/llvm/llvm-project/tree/main/mlir/include/mlir/Dialect/Math

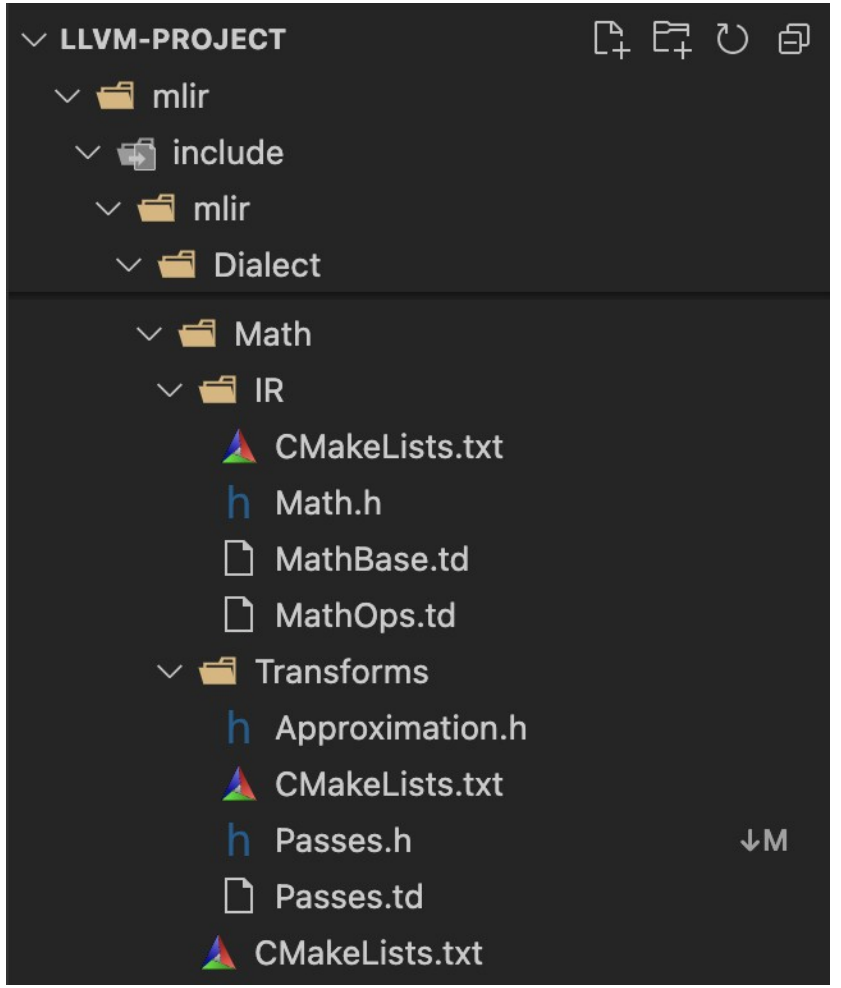
IR:

- Dialect description
- Dialect attributes and types
- Operations description

Transforms:

- Dialect specific passes

other directories depend on dialect specifics



Dialect description

Docs:
https://mlir.llvm.org/docs/DefiningDialects/AttributesAndTypes/

Example:
https://github.com/llvm/llvm-project/blob/main/mlir/include/mlir/Dialect/Arith/IR/ArithBase.td

```
154   def Arith_IntegerOverflowAttr :
155       EnumAttr<Arith_Dialect, Arith_IntegerOverflowFlags, "overflow"> {
156     let assemblyFormat = "`<` $value `>`";
157   }
```

```
12  include "mlir/IR/EnumAttr.td"
13  include "mlir/IR/OpBase.td"
14
15  def Arith_Dialect : Dialect {
16    let name = "arith";
17    let cppNamespace = "::mlir::arith";
18    let description = [{
19      The arith dialect is intended to hold basic integer and floating point
20      mathematical operations. This includes unary, binary, and ternary arithmetic
21      ops, bitwise and shift ops, cast ops, and compare ops. Operations in this
22      dialect also accept vectors and tensors of integers or floats. The dialect
23      assumes integers are represented by bitvectors with a two's complement
24      representation. Unless otherwise stated, the operations within this dialect
25      propagate poison values, i.e., if any of its inputs are poison, then the
26      output is poison. Unless otherwise stated, operations applied to `vector`
27      and `tensor` values propagates poison elementwise.
28    }];
29
30    let hasConstantMaterializer = 1;
31    let useDefaultAttributePrinterParser = 1;
32  }
33
34  // The predicate indicates the type of the comparison to perform:
35  // (un)orderedness, (in)equality and less/greater than (or equal to) as
36  // well as predicates that are always true or false.
37  def Arith_CmpFPredicateAttr : I64EnumAttr<
38      "CmpFPredicate", "",
39      [
40        I64EnumAttrCase<"AlwaysFalse", 0, "false">,
41        I64EnumAttrCase<"OEQ", 1, "oeq">,
42        I64EnumAttrCase<"OGT", 2, "ogt">,
43        I64EnumAttrCase<"OGE", 3, "oge">,
44        I64EnumAttrCase<"OLT", 4, "olt">,
45        I64EnumAttrCase<"OLE", 5, "ole">,
46        I64EnumAttrCase<"ONE", 6, "one">,
47        I64EnumAttrCase<"ORD", 7, "ord">,
48        I64EnumAttrCase<"UEQ", 8, "ueq">,
49        I64EnumAttrCase<"UGT", 9, "ugt">,
50        I64EnumAttrCase<"UGE", 10, "uge">,
51        I64EnumAttrCase<"ULT", 11, "ult">,
52        I64EnumAttrCase<"ULE", 12, "ule">,
53        I64EnumAttrCase<"UNE", 13, "une">,
54        I64EnumAttrCase<"UNO", 14, "uno">,
55        I64EnumAttrCase<"AlwaysTrue", 15, "true">,
56      ]> {
57    let cppNamespace = "::mlir::arith";
58  }
```

Operations definition

Operation Definition Specification (ODS)

Docs: https://mlir.llvm.org/docs/DefiningDialects/Operations/

```
def MyOp : Op<"my_op", []> {
  let arguments = (ins F32Attr:$attr);

  let builders = [
    OpBuilder<(ins "float":$val)>
  ];
}
```

```
class MyOp : /*...*/ {
  /*...*/
  static void build(::mlir::OpBuilder &builder, ::mlir::OperationState &state,
                    float val);
};
```

TableGen definition
*Ops.td

Auto-generated code
*Ops.h.inc and *Ops.cpp.inc

https://github.com/llvm/llvm-project/blob/76aa042dde6ba9ba57c680950f5818259ee02690/mlir/include/mlir/Dialect/Arith/IR/ArithOps.td#L209

Arith AddI operation example

```
23  // Base class for Arith dialect ops. Ops in this dialect have no memory
24  // effects and can be applied element-wise to vectors and tensors.
25  class Arith_Op<string mnemonic, list<Trait> traits = []> :
26      Op<Arith_Dialect, mnemonic,
27          traits #
28          [DeclareOpInterfaceMethods<VectorUnrollOpInterface>, NoMemoryEffect] #
29          ElementwiseMappable.traits>;
30
31  // Base class for integer and floating point arithmetic ops. All ops have one
32  // result, require operands and results to be of the same type, and can accept
33  // tensors or vectors of integers or floats.
34  class Arith_ArithOp<string mnemonic, list<Trait> traits = []> :
35      Arith_Op<mnemonic, traits # [SameOperandsAndResultType]>;
```

```
43  // Base class for binary arithmetic operations.
44  class Arith_BinaryOp<string mnemonic, list<Trait> traits = []> :
45      Arith_ArithOp<mnemonic, traits> {
46    let assemblyFormat = "$lhs `,` $rhs attr-dict `:` type($result)";
47  }
```

```
140 class Arith_IntBinaryOpWithOverflowFlags<string mnemonic, list<Trait> traits = []> :
141     Arith_BinaryOp<mnemonic, traits #
142         [Pure, DeclareOpInterfaceMethods<InferIntRangeInterface>,
143          DeclareOpInterfaceMethods<ArithIntegerOverflowFlagsInterface>]>,
144     Arguments<(ins SignlessIntegerLike:$lhs, SignlessIntegerLike:$rhs,
145         DefaultValuedAttr<
146             Arith_IntegerOverflowAttr,
147             "::mlir::arith::IntegerOverflowFlags::none">:$overflowFlags)>,
148     Results<(outs SignlessIntegerLike:$result)> {
149
150   let assemblyFormat = [{ $lhs `,` $rhs (`overflow` `` $overflowFlags^)?
151                           attr-dict `:` type($result) }];
152 }
```

```
209 def Arith_AddIOp : Arith_IntBinaryOpWithOverflowFlags<"addi", [Commutative]> {
210   let summary = "integer addition operation";
211   let description = [{
212     Performs N-bit addition on the operands. The operands are interpreted as
213     unsigned bitvectors. The result is represented by a bitvector containing the
214     mathematical value of the addition modulo 2^n, where `n` is the bitwidth.
215     Because `arith` integers use a two's complement representation, this operation
216     is applicable on both signed and unsigned integer operands.
217
218     The `addi` operation takes two operands and returns one result, each of
219     these is required to be the same type. This type may be an integer scalar type,
220     a vector whose element type is integer, or a tensor of integers.
221
222     This op supports `nuw`/`nsw` overflow flags which stands stand for
223     "No Unsigned Wrap" and "No Signed Wrap", respectively. If the `nuw` and/or
224     `nsw` flags are present, and an unsigned/signed overflow occurs
225     (respectively), the result is poison.
226
227     Example:
228
229     ```mlir
230     // Scalar addition.
231     %a = arith.addi %b, %c : i64
232
233     // Scalar addition with overflow flags.
234     %a = arith.addi %b, %c overflow<nsw, nuw> : i64
235
236     // SIMD vector element-wise addition.
237     %f = arith.addi %g, %h : vector<4xi32>
238
239     // Tensor element-wise addition.
240     %x = arith.addi %y, %z : tensor<4x?xi8>
241     ```
242   }];
243   let hasFolder = 1;
244   let hasCanonicalizer = 1;
245 }
```

Content

- Available transformations and optimizations overview
- Custom dialects
- **Writing a pass**
- MLIR connections with other languages
- Debugging

Writing a pass

Pass manager

Docs: https://mlir.llvm.org/docs/PassManagement/

Pass writing by example: LICM

Loop Invariant Code Motion

`-loop-invariant-code-motion` ¶

Hoist loop invariant instructions outside of the loop

https://mlir.llvm.org/docs/Passes/#-loop-invariant-code-motion

Pass writing by example: LICM definition in headers

mlir > include > mlir > Transforms > Passes.td

```
327   def LoopInvariantCodeMotion : Pass<"loop-invariant-code-motion"> {
328     let summary = "Hoist loop invariant instructions outside of the loop";
329     let constructor = "mlir::createLoopInvariantCodeMotionPass()";
330   }
```

TableGen Definition

https://github.com/llvm/llvm-project/blob/76aa042dde6ba9ba57c680950f5818259ee02690/mlir/include/mlir/Transforms/Passes.td#L336-L339

```
77   /// Creates a loop invariant code motion pass that hoists loop invariant
78   /// instructions out of the loop.
79   std::unique_ptr<Pass> createLoopInvariantCodeMotionPass();
```

Header Definition

https://github.com/llvm/llvm-project/blob/76aa042dde6ba9ba57c680950f5818259ee02690/mlir/include/mlir/Transforms/Passes.h#L78-L80

Pass writing by example: LICM usage

```
83    void runOnOperation() override {
84      MLIRContext *context = &getContext();
85      RewritePatternSet patterns =
86          linalg::getLinalgTilingCanonicalizationPatterns(context);
87      patterns.add<ExtractSliceOfPadTensorSwapPattern>(context);
88      scf::populateSCFForLoopCanonicalizationPatterns(patterns);
89      FrozenRewritePatternSet frozenPatterns(std::move(patterns));
90      OpPassManager pm(func::FuncOp::getOperationName());
91      pm.addPass(createLoopInvariantCodeMotionPass());
92      pm.addPass(createCanonicalizerPass());
93      pm.addPass(createCSEPass());
94      do {
95        (void)applyPatternsAndFoldGreedily(getOperation(), frozenPatterns);
96        if (failed(runPipeline(pm, getOperation())))
97          this->signalPassFailure();
98      } while (succeeded(fuseLinalgOpsGreedily(getOperation())));
99    }
```

https://github.com/llvm/llvm-project/blob/76aa042dde6ba9ba57c680950f5818259ee02690/mlir/test/lib/Dialect/Linalg/TestLinalgFusionTransforms.cpp#L91C22-L91C28

Pass writing by example: LICM implementation

mlir > lib > Transforms > C++ LoopInvariantCodeMotion.cpp > ...

```cpp
//===- LoopInvariantCodeMotion.cpp - Code to perform loop fusion-----------===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//
//
// This file implements loop invariant code motion.
//
//===----------------------------------------------------------------------===//

#include "mlir/Transforms/Passes.h"

#include "mlir/IR/PatternMatch.h"
#include "mlir/Interfaces/LoopLikeInterface.h"
#include "mlir/Interfaces/SideEffectInterfaces.h"
#include "mlir/Transforms/LoopInvariantCodeMotionUtils.h"

Matthias Springer, 6 months ago | 2 authors (Michele Scuttari and others)
namespace mlir {
#define GEN_PASS_DEF_LOOPINVARIANTCODEMOTION
#define GEN_PASS_DEF_LOOPINVARIANTSUBSETHOISTING
#include "mlir/Transforms/Passes.h.inc"
} // namespace mlir

using namespace mlir;

Matthias Springer, 6 months ago | 5 authors (Matthias Springer and others)
namespace {
/// Loop invariant code motion (LICM) pass.
Michele Scuttari, 20 months ago | 3 authors (Michele Scuttari and others)
struct LoopInvariantCodeMotion
    : public impl::LoopInvariantCodeMotionBase<LoopInvariantCodeMotion> {
  void runOnOperation() override;
};
```

```cpp
void LoopInvariantCodeMotion::runOnOperation() {
  // Walk through all loops in a function in innermost-loop-first order. This
  // way, we first LICM from the inner loop, and place the ops in
  // the outer loop, which in turn can be further LICM'ed.
  getOperation()->walk(
      [&](LoopLikeOpInterface loopLike) { moveLoopInvariantCode(loopLike); });
}
```

```cpp
std::unique_ptr<Pass> mlir::createLoopInvariantCodeMotionPass() {
  return std::make_unique<LoopInvariantCodeMotion>();
}
```

`runOnOperation` is the main function to be implemented by pass developer

Operation walker

Walk through all operation within this operation

```
block->walk([&](Operation *op) {
  for (Value result : op->getResults())
    defValues.insert(result);
  for (Value operand : op->getOperands())
    useValues.insert(operand);
});
```

https://github.com/llvm/llvm-project/blob/ffc9a30938ae5c42c03f9c563db1465876b4def6/mlir/lib/Analysis/Liveness.cpp#L70

Walk through all operations implementing specific interface

```
void runOnOperation() override {
  // Topologically sort the regions of the operation without SSA dominance.
  getOperation()->walk([](RegionKindInterface op) {
    for (auto it : llvm::enumerate(op->getRegions())) {
      if (op.hasSSADominance(it.index()))
        continue;
      for (Block &block : it.value())
        sortTopologically(&block);
    }
  });
}
```

https://github.com/llvm/llvm-project/blob/ffc9a30938ae5c42c03f9c563db1465876b4def6/mlir/lib/Transforms/TopologicalSort.cpp#L26

Walk through all operations of specific type (e.g. `scf.for`)

```
void runOnOperation() override {
  SmallVector<scf::ForOp, 4> loops;
  getOperation()->walk([&](scf::ForOp forOp) {
    if (getNestingDepth(forOp) == loopDepth)
      loops.push_back(forOp);
  });
```

https://github.com/llvm/llvm-project/blob/ffc9a30938ae5c42c03f9c563db1465876b4def6/mlir/test/lib/Dialect/SCF/TestLoopUnrolling.cpp#L57

Rewind back to LLVM IR

LLVM IR passes classifications

- Types
  - Analysis pass
  - Transformation pass
  - Utility pass
- Run scope
  - Module pass
  - Function pass
  - Loop pass
  - …

Operation walker gives an opportunity to run the pass on any operations

What about MLIR classification?

- It could be the same
- but in addition to that usually people outline different classification rules

Content

- Available transformations and optimizations overview
- Custom dialects
- Writing a pass
- **MLIR connections with other languages**
- Debugging

MLIR C API

Adds interoperability with many other programming languages

- C (first of all), C++
  - C API: https://mlir.llvm.org/docs/CAPI/
- Python
  - Built-in project: https://mlir.llvm.org/docs/Bindings/Python/
- Rust
  - https://github.com/raviqqe/melior
- others…

LLVM is written in C++. Why not C++? Why libraries usually choose C for interoperability?

C for interoperability

Why C over C++?

- Mangled names in C++
  - OOP
  - Templates
- Memory model differences
  - Pointers and references
  - Different allocation techniques
- C is more common

MLIR Python bindings

```python
from mlir.ir import Context, InsertionPoint, Location, Module, Operation

with Context() as ctx:
  module = Module.create()

  # Prepare for inserting operations into the body of the module and indicate
  # that these operations originate in the "f.mlir" file at the given line and
  # column.
  with InsertionPoint(module.body), Location.file("f.mlir", line=42, col=1):
    # This operation will be inserted at the end of the module body and will
    # have the location set up by the context manager.
    Operation(<...>)

    # This operation will be inserted at the end of the module (and after the
    # previously constructed operation) and will have the location provided as
    # the keyword argument.
    Operation(<...>, loc=Location.file("g.mlir", line=1, col=10))

    # This operation will be inserted at the *beginning* of the block rather
    # than at its end.
    Operation(<...>, ip=InsertionPoint.at_block_begin(module.body))
```

```python
from mlir.ir import Context, Module
from mlir.dialects import builtin

with Context():
  module = Module.create()
  with InsertionPoint(module.body), Location.unknown():
    # Operations can be created in a generic way.
    func = Operation.create(
        "func.func", results=[], operands=[],
        attributes={"function_type":TypeAttr.get(FunctionType.get([], []))},
        successors=None, regions=1)
    # The result will be downcasted to the concrete `OpView` subclass if
    # available.
    assert isinstance(func, func.FuncOp)
```

https://mlir.llvm.org/docs/Bindings/Python/

MLIR Python bindings

Projects who use:

- mlir-aie
  - https://github.com/Xilinx/mlir-aie/blob/b8d512fa4368f5f9c77b76835323990e73cc099c/python/compiler/util.py#L414
- xDSL
  - https://github.com/ingomueller-net/xdsl/blob/e67c2e00351cb7073dfa91a7e0e8e6e65bc60632/src/xdsl/mlir_converter.py#L129
- Similar interface is exposed by languages that use MLIR (e.g. Triton)
  - https://github.com/microsoft/triton-shared/blob/e771c1e604c9c8883e50ed8f0e13c647156aa7c3/backend/compiler.py#L172

and others…

Content

- Available transformations and optimizations overview
- Custom dialects
- Writing a pass
- MLIR connections with other languages
- **Debugging**

MLIR debugging

- Debug sequence of passes
  - -mlir-print-ir-after-failure
  - -mlir-print-ir-before-all / -mlir-print-ir-after-all
- Disable multithreading
  - -mlir-disable-threading
- Print generic ops to pinpoint error in operation arguments
  - -mlir-print-op-generic
- Use tools to find culprit operations in IR (e.g. mlir-reduce)
  - Reduce buggy IR to minimal one where the issue is still reproducible
    - https://mlir.llvm.org/docs/Tools/mlir-reduce/
- other useful tips can be found here: https://mlir.llvm.org/getting_started/Debugging/

Core materials on MLIR

Official MLIR documentation: https://mlir.llvm.org/

Lab assignment #4

- Write a pass in MLIR infrastructure
- Deadline: May, 21
- Where to seek help
  - https://mlir.llvm.org/docs/PassManagement/
  - Contact teachers

Next time

- Python internals
  - AST
  - bytecode (IR)
  - Interpreter
  - JIT
  - …

Test

https://forms.gle/HbN5qJU1sViVkSSP6

Submission time: **10 minutes**

В чем отличие классификации пассов в MLIR и в LLVM IR?

Your answer

Зачем нужен и что делает canonicalizer pass?

Your answer

Backup: me@gooddoog.ru

Extra materials

- How to build a compiler with LLVM and MLIR: https://www.youtube.com/playlist?list=PLlONLmJCfHTo9WYfsoQvwjsa5ZB6hjOG5
- 2019 EuroLLVM Developers' Meeting: T. Shpeisman & C. Lattner "MLIR: Multi-Level Intermediate Representation for Compiler Infrastructure" https://www.youtube.com/watch?v=qzljG6DKgic
- 2023 LLVM Dev Mtg - MLIR Is Not an ML Compiler, and Other Common Misconceptions: https://www.youtube.com/watch?v=lXAp6ZAWyBY
- MLIR CodeGen Dialects for Machine Learning Compilers: https://www.lei.chat/posts/mlir-codegen-dialects-for-machine-learning-compilers/
- MLIR: Scaling Compiler Infrastructure for Domain Specific Computation: https://rcs.uwaterloo.ca/~ali/cs842-s23/papers/mlir.pdf
- MLIR: A Compiler Infrastructure for the End of Moore's Law: https://arxiv.org/pdf/2002.11054
- Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations: https://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf