MLIR - Multi-Level Intermediate Representation

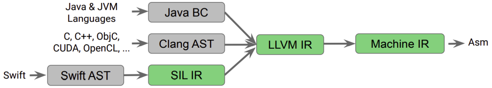
Kevin Jude Concessao

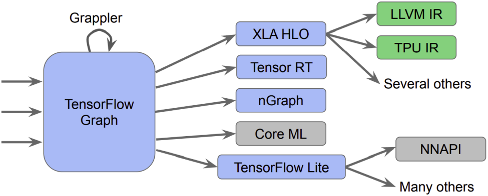
IIT Palakkad

May 7, 2021

Outline

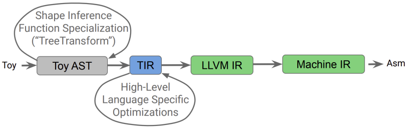
Introduction

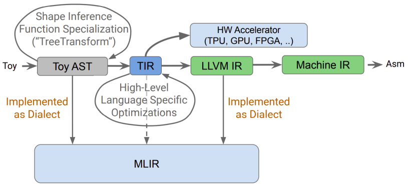




- Domain specific optimizations:
  - Devirtualization
  - Reference Count elision
  - Progressive lowering
  - Library specific optimizations
  - Better type / borrow checking
- Problems:
  - Reimplementation of pass managers, error tracking, passes, use-def chains, etc.
  - Huge expense to rebuild / repeat existing infrastructure. Wasteful repetition of effort.

Introduction - What is MLIR ?





- MLIR is a **framework** to represent **multiple levels** of `tree-based` IRs, `machine level` IRs, `graph-based` IRs
- Provides **common infrastructure** to write `optimization`, `lowering`, and `reuse` of **passes** across IRs.
- Reduces duplication of `pass infrastructure`, `location tracking / error handling`.
- Provides **minimal abstractions** for representing constructs across domains: `parallel constructs`, `polyhedral models`, ML `graph` optimizations, etc.
- Not opinionated. Extensible because of minimal abstractions.

Design Principles

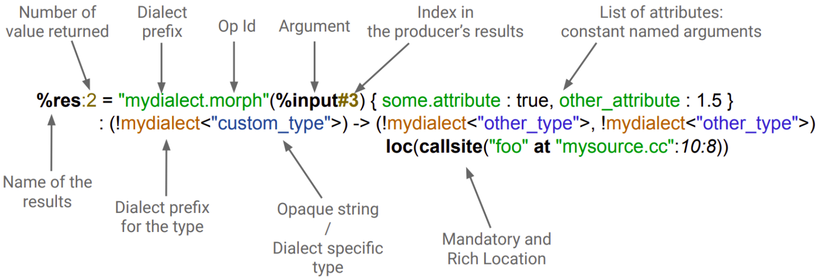
- Little builtin, everything customizable
  - Minimal number of fundamental concepts: attributes, types, operations.
  - Ability to express ML graphs, ASTs, polyhedral models, CFGs, LLVM IR, etc
  - Create reusable abstractions
- SSA and regions
  - Makes data-flow analysis simple. Well understood representation.
  - Unlike flat-linearized CFGs, there are nested regions. lift higher level abstractions (e.g., loop trees), speeding up the compilation process or extracting instruction, or SIMD parallelism.
  - To support heterogeneous compilation, the system has to support the expression of structured control flow, concurrency constructs, closures in source languages, and many other purposes.
  - Loss of normalization: lowering to a minimal subset as in LLVM.

Design Principles

- Progressive lowering
  - Benefits of combining passes was to mix constant propagation, value numbering and unreachable code elimination across dialects.
  - Passes are meant for optimizing, transforming, lowering and cleaning dialect operations.
- Maintain higher-level semantics: Only lower a construct when not need for further optimization.
- IR validation
- Declarative rewrite patterns
  - Common transformations should be implementable as rewrite rules expressed declaratively, to reason about properties of the rewrites such as complexity and completion.
  - Build machine descriptions capable of steering rewriting strategies through multiple levels of abstraction
- Source location tracking and traceability

IR Design

- Operations in MLIR are analogous to `Instruction`'s in `LLVM IR`.
- Operations take and produce zero or more values, called operands and results respectively, and these are maintained in `SSA form`.



IR Design - Nesting of Operations - Regions

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops and can have multiple blocks.        Region
  ^block(%argument: !d.type):                                   Block
    // Ops have function types (expressing mapping).
    %value = "nested.operation"() ({
      // Ops can contain nested regions.                  Region
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:                                                 Block
    "d.terminator"() [^block(%argument : !d.type)] : () -> ()
  })
// Ops can have a list of attributes.
{attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

- A **region** contains a list of blocks, and a **block** contains a list of **operations** (which may contain regions).
- **Blocks** inside a region form a **CFG**. Each block ends with a **terminator operation**.
- No ϕ nodes. MLIR uses a **functional SSA**.

What is a Dialect ?

A dialect encapsulates the following:

- A **prefix** / **namespace**.
- A collection of **types**.
- **Operations**:
  - Verifier for operation invariants (e.g. toy.print must have a single operand)
  - Semantics (has-no-side-effects, constant-folding, CSE-allowed, et cetra)
- **Passes**: analysis, transformations, and dialect conversions.
- Custom parsers and pretty-printers.

Each of the above can be implemented using a `C++ class`, or with a `TableGen DSL` program for MLIR.

`mlir-tablegen` can auto-generate these `C++ classes`. The paper calls it as `Operation Definition Specification`.

Affine Dialect Example

```
func @test() {
  affine.for %k = 0 to 10 {
    affine.for %l = 0 to 10 {
      affine.if (d0) : (8*d0 - 4 ≥ 0, -8*d0 + 7 ≥ 0)(%k) {
        // Dead code, because no multiple of 8 lies between 4 and 7
        "foo"(%k) : (index) → ()
      }
    }
  }
  return
}
```

Affine Dialect Example

```
#set0 = (d0) : (d0 * 8 - 4 ≥ 0, d0 * -8 + 7 ≥ 0)
func @test() {
  "affine.for"(){lower_bound : #map0, step : 1 : index, upper_bound : #map1} : () → () {
    ^bb1(% i0 : index)
        : "affine.for"() { lower_bound : #map0, step : 1 : index, upper_bound : #map1 } : () → () {
              ^bb2(% i1 : index) : "affine.if"(% i0){condition : #set0} : (index) → () {
                  "foo"(% i0) : (index) → ()
                  "affine.terminator"() : () → ()
                } {
                  // else block
                }
                "affine.terminator"() : () → ()
            }
    ...
```

LLVM Dialect Example

```
%13 = llvm.alloca %arg0 x !llvm.double : (!llvm.i32) -> !llvm.ptr<double>

%14 = llvm.getelementptr %13[%arg0, %arg0]
          : (!llvm.ptr<double>, !llvm.i32, !llvm.i32) -> !llvm.ptr<double>
%15 = llvm.load %14 : !llvm.ptr<double>
llvm.store %15, %13 : !llvm.ptr<double>
%16 = llvm.bitcast %13 : !llvm.ptr<double> to !llvm.ptr<i64>
%17 = llvm.call @foo(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
%18 = llvm.extractvalue %17[0] : !llvm.struct<(i32, double, i32)>
%19 = llvm.insertvalue %18, %17[2] : !llvm.struct<(i32, double, i32)>
%20 = llvm.constant(@foo : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>) :
        !llvm.ptr<func<struct<i32, double, i32> (i32)>>
%21 = llvm.call %20(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
```

Operation Definition Specification

```
def TF_AvgPoolOp : TF_Op<"AvgPool", [NoSideEffect, SameValueType]> {
  let summary = "Performs average pooling on the input.";

  let description = [{
    Each entry in 'output' is the mean of the corresponding size
    'ksize' window in `value`.
  }];

  let arguments = (ins
    TF_FpTensor:$value,

    Confined<I64ArrayAttr, [ArrayMinCount<4>]>:$ksize,
    Confined<I64ArrayAttr, [ArrayMinCount<4>]>:$strides,
    TF_AnyStrAttrOf<["SAME", "VALID"]>:$padding,
    DefaultValuedAttr<TF_ConvertDataFormatAttr, "NHWC">:$data_format
  );

  let results = (outs
    TF_FpTensor:$output
  );

  let regions = (region
    <region-constraint>:$<region-name>,
    ...
  );

  // For terminator operations
  let successors = (successor
    <successor-constraint>:$<successor-name>,
    ...
  );

  let verifier = [{ return ::verify(*this); }];
}
```

Optimization using MLIR

```
def no_op(b) {
  return transpose(transpose(b));
}
#define N 100
#define M 100

void double_transpose(int A[N][M]) {
  int B[M][N];
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
      B[j][i] = A[i][j];
    }
  }
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
      A[i][j] = B[j][i];
    }
  }
}
```

```
struct SimplifyRedundantTranspose : public RewritePattern {
  SimplifyRedundantTranspose(MLIRContext *context)
      : RewritePattern(TransposeOp::getOperationName(),
                       1, context) {}
  PatternMatchResult
    matchAndRewrite(Operation *op,
                    PatternRewriter &rewriter)
                    const override {

    TransposeOp transpose = op->cast<TransposeOp>();

    mlir::Value *transposeInput =
        transpose.getOperand();

    mlir::Operation *transposeInputInst =
        transposeInput->getDefiningOp();

    TransposeOp transposeInputOp =
        dyn_cast_or_null<TransposeOp>(transposeInputInst);
    if (!transposeInputOp)
      return matchFailure();

    rewriter.replaceOp(op, {transposeInputOp.getOperand()},
                           {transposeInputOp});
    return matchSuccess();
  }
}
```

```
def: Pat<(TransposeOp (TransposeOp $arg)), ($arg)>;
```

Optimization using MLIR

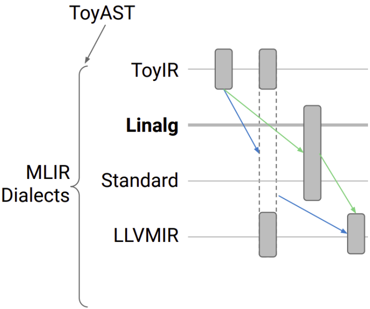
Before:

```
func @multiply_transpose(%arg0: !toy<"array">)
  attributes {toy.generic: true} {
   %0 = "toy.transpose"(%arg0) : (!toy<"array">) → !toy<"array">
   %1 = "toy.transpose"(%0)    : (!toy<"array">) → !toy<"array">
   "toy.return"(%1) : (!toy<"array">) → ()
}
```

After:

```
func @multiply_transpose(%arg0: !toy<"array">)
  attributes {toy.generic: true} {
   "toy.return"(%arg0) : (!toy<"array">) → ()
}
```

Dialect Conversion



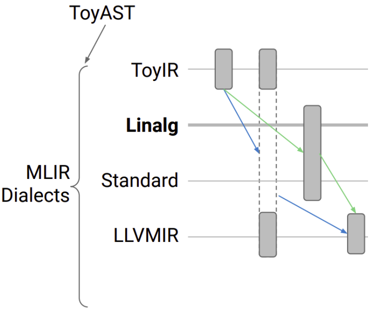
Toy Dialect:

- `toy.constant`
- `toy.reshape`
- `toy.cast`
- `toy.transpose`
- `toy.mul`
- `toy.generic_call`
- `toy.print`
- `toy.return`

Standard Dialect:

- `std.br (BranchOp)`
- `std.mulf (MulFOp)`
- `std.addi (AddIOp)`
- `std.cmpf (CmpFOp)`
- `std.xor (XOROp)`
- `std.switch (SwitchOp)`
- `std.return (ReturnOp)`

Dialect Conversion



Linalg Dialect:

- Types:
  - `linalg.range`
  - `linalg.view`
- Operations:
  - `linalg.matmul`
  - `linalg.matvec`
  - `linalg.dot`
  - `linalg.load`
  - `linalg.store`
  - `linalg.range`
  - `linalg.slice`
  - `linalg.view`

Dialect Conversion

- MLIR allows for **progressive lowering**: allows multiple dialects in the same function or module.
- Three components:
  - **Function signature** conversion

    ```
    func @foo(i64) → (f64, f64)

    func @foo(!llvm<"i64">) → !llvm<"{double, double}">
    ```

  - **Type** conversion

    ```
    i64 ⇒ !llvm<"i64">

    f32 ⇒ !llvm<"float">
    ```

  - **Operation** conversion

    ```
    addf %0, %1 : f32 ⇒ %2 = llvm.fadd %0, %1 : !llvm<"float">
    load %memref[%x] : memref<?xf32> ⇒ %3 = llvm.extractvalue %m[0] : !llvm<"{float*, i64}">
    %4 = llvm.getelementptr %3[%x] : !llvm<"float*">
    ```

Thank You

References

[1] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, *Mlir: A compiler infrastructure for the end of moore's law*, 2020. arXiv: 2002.11054 [cs.PL].

[2] M. Amini, A. Zinenko, and N. Vasilache, *Mlir tutorial: Building a compiler with mlir*, https://llvm.org/devmtg/2019-04/slides/Tutorial-AminiVasilacheZinenko-MLIR.pdf, 2020.