MLIR Tutorial:

Building a Compiler with MLIR

LLVM Developers Meeting, Euro-LLVM 2019

Mehdi Amini
aminim@google.com

Alex Zinenko
zinenko@google.com

Nicolas Vasilache
ntv@google.com

Presenting the work of many, many, people!

This tutorial will walk you through the creation of a compiler using MLIR. It is intended as a companion to the keynote (https://llvm.org/devmtg/2019-04/talks.html#Keynote_1) and it is assumed that the audience saw it. The goal is to provide a more concrete view than the high-level vision presented in the keynote.

This talk is in three part:

1) We introduce a high level array-based language, with generic function and type-deduction. We then show how MLIR concepts help to design and build an IR that carry the language semantics as closely as possible to the point where you can perform every transformations that you would attempt on an AST usually: this is the foundation of what could be a C++ IR for Clang and part of the frontend like `TreeTransform` (for template instantiation for example) could be replaced by regular IR->IR passes of transformation.
2) The second part introduce the dialect-to-dialect conversion framework: after performing the high-level transformations and optimizations, we need to lower the code towards a representation more suitable for CodeGen. The dialect concept in MLIR allows to lower progressively and introduce domain-specific middle-end representations that are geared toward domain-specific optimizations. For CodeGen, LLVM is king of course but one can also implement a different lowering in order to target custom accelerators or FPGAs.
3) Finally, we showcase an example of a “middle-end” dialect that is specialized to perform transformation on “linear algebra”. It is intended to be generic and reusable. In the context of the `Toy` language we can take advantage of this dialect for a subset of the language: only the computationally intensive parts. This dialect treats Linear algebra primitives as first-class citizens from which it is easy to lower into library calls, ARM SVE, TPU, LLVM IR, coarser ISAs … It also supports key transformations (tiling, fusion, bulk memory transfers) without complex analyses.

Introduction: a Toy Language

Google

Let's Build Our Toy Language

- Mix of scalar and array computations, as well as I/O
- Array shape Inference
- Generic function
- Very limited set of operators (it's just a Toy language!):
  - element-wise addition
  - Multiplication (matmul)
  - Array Transpose

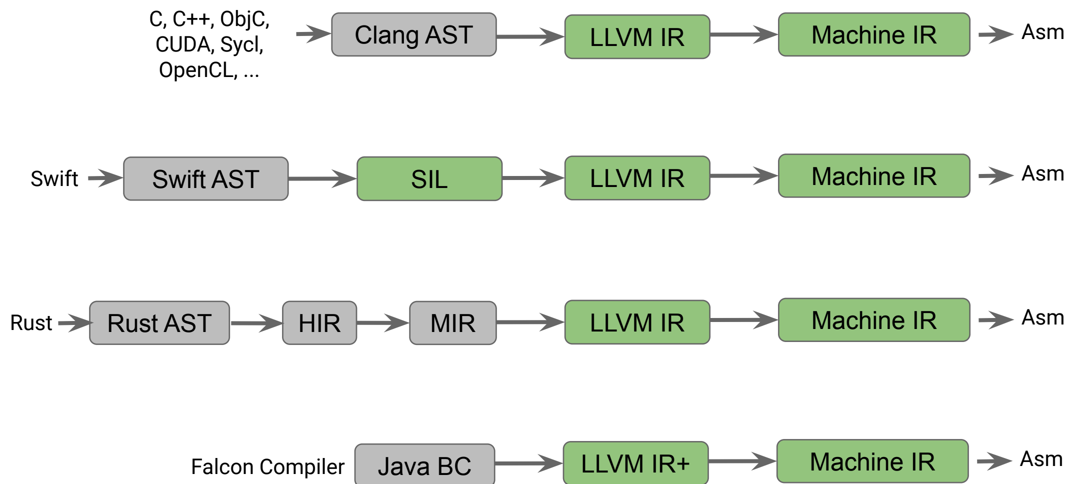
Generic function: a, b, and c aren't fully typed, think C++ template:

```
template<typename A, typename B, typename C>
auto foo(A a, B b, C c) {
  ...
```

```
def foo(a, b, c) {
  var c = a * b;
  print(transpose(c));
  var d<2, 4> = c * foo(c);
  return d;
}
```

- `var c = a * b;` — Value-based semantics / SSA
- `print(transpose(c));` — Only 2 builtin functions: print and transpose
- `var d<2, 4>` — Array reshape through explicit variable declaration

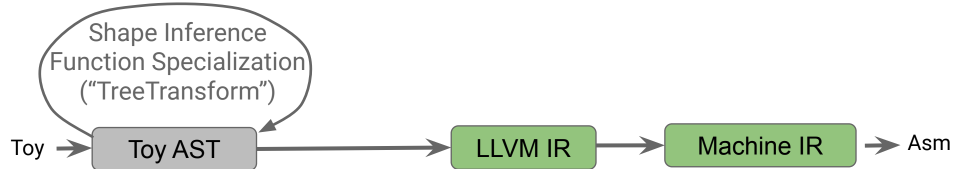
We define a high-level language to illustrate how MLIR can provide facilities for high-level representation (like an IR for Clang)



Traditional model: AST -> LLVM
Recent modern compiler added extra level of language-specific IR, refining the AST model towards LLVM, gradually lowering the representation.
Falcon took a different approach (heroic?) and embedded a higher-level representation in LLVM (through builtins and others).
What do we pick for Toy? We want something modern and future-proof as much as possible

The Toy Compiler: the "Simpler" Approach of Clang

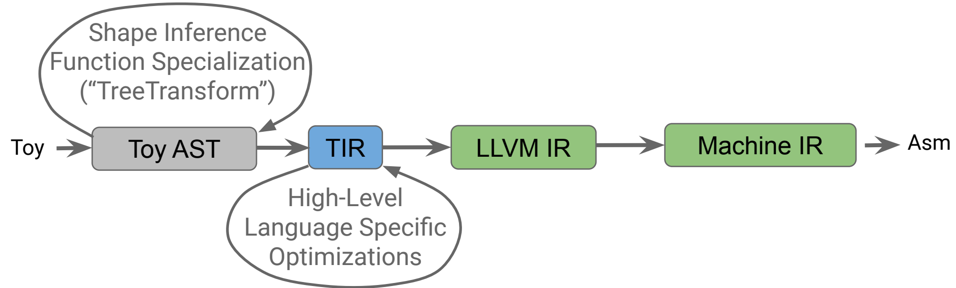
Need to analyze and transform the AST
-> heavy infrastructure!
And is the AST really the most friendly
representation we can get?



Should we follow the clang model? We have some some high-level tasks to perform before reaching LLVM.
Need a complex AST, heavy infrastructure for transformations and analysis, AST isn't great for all this.

The Toy Compiler: With Language Specific Optimizations

Need to analyze and transform the AST
-> heavy infrastructure!
And is the AST really the most friendly
representation we can get?



For more optimizations: a custom IR.
Reimplement again all the LLVM infrastructure?

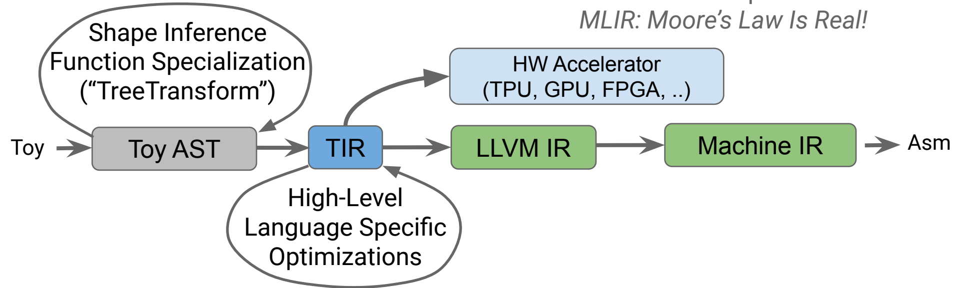
Google

For language specific optimization we can go with builtins and custom LLVM passes, but ultimately we may end up wanting our IR at the right level

Compilers in a Heterogenous World

Need to analyze and transform the AST
-> heavy infrastructure!
And is the AST really the most friendly
representation we can get?

New HW: are we extensible
and future-proof?
MLIR: Moore's Law Is Real!



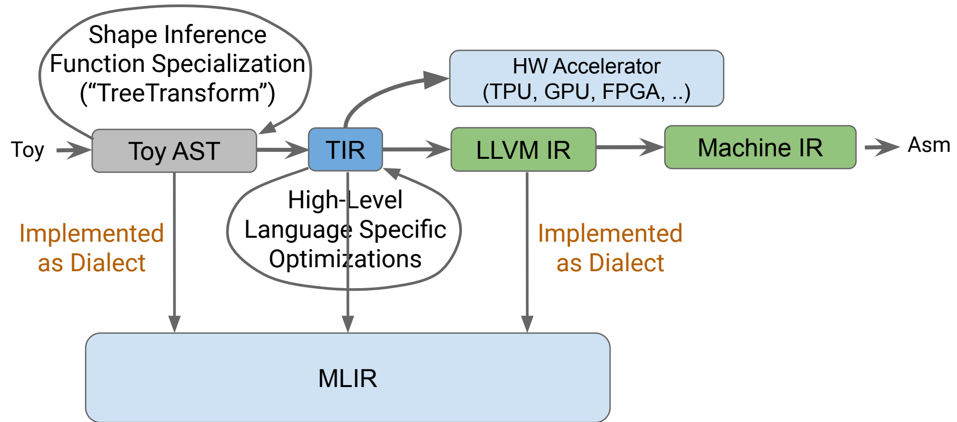
For more optimizations: a custom IR.
Reimplement again all the LLVM infrastructure?

Google

At some point we may even want to offload some part of the program to custom accelerators, requiring more concept to represent in the IR

It Is All About Dialect!

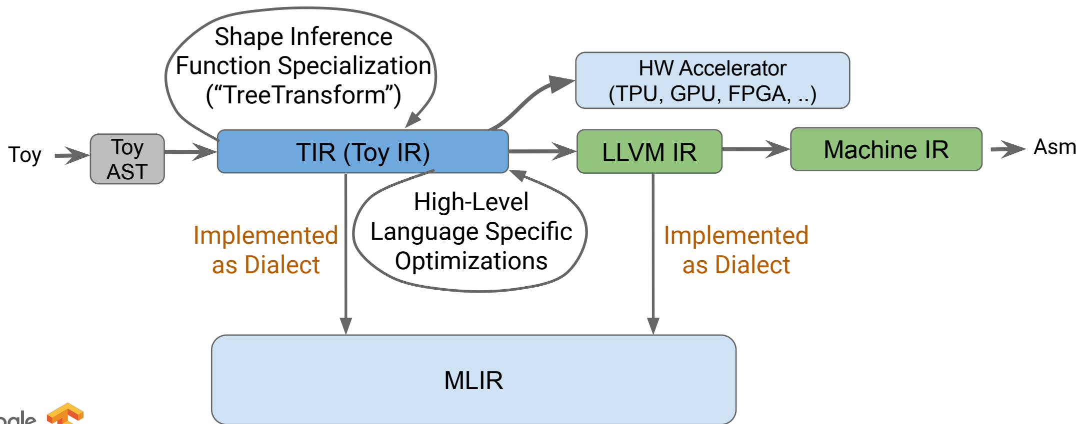
MLIR allows every level to be represented as a Dialect



Google

Adjust Ambition to Our Budget (let's fit the talk)

Limit ourselves to a single dialect for Toy IR: still flexible enough to perform shape inference, function specializations, and some high-level optimizations.



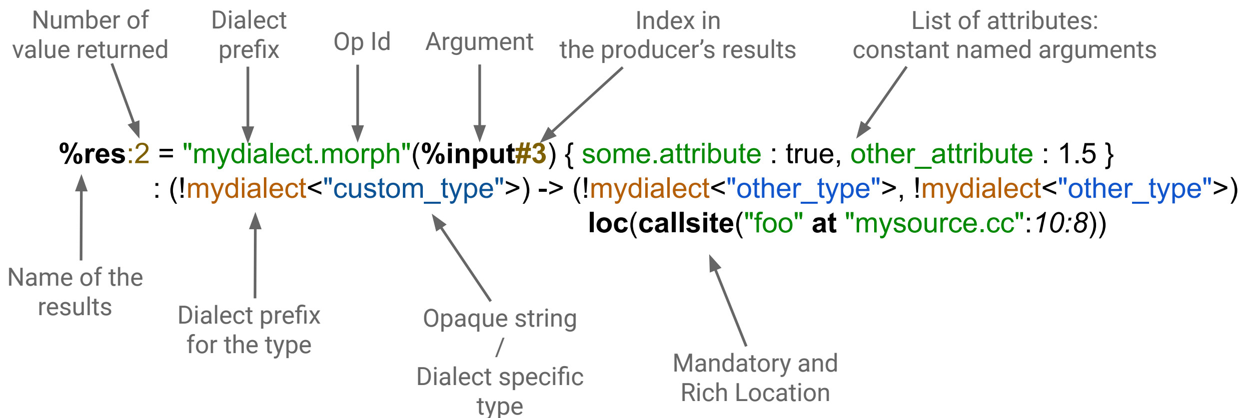
Google

MLIR Primer

Google

Operations, Not Instructions

- No predefined set of instructions
- Operations are like “opaque functions” to MLIR



In MLIR, everything is about Operations, not Instructions: we put the emphasis to distinguish from the LLVM view. Operations can be coarse grain (perform a matrix-multiplication) or can directly carry loop nest or other kind of nested “regions” (see later slides)

Example

```
func @some_func(%arg0: !random_dialect<"custom_type">)
 -> !another_dialect<"other_type"> {
  %result = "custom.operation"(%arg0)
       : (!random_dialect<"custom_type">) -> !another_dialect<"other_type">
  return %result : !another_dialect<"other_type">
}
```

Yes: this is a fully valid textual IR module: try round-tripping with *mlir-opt*!

Toy “transpose” Builtin: The Hard (and Broken) Way

```
%0 = "toy.transpose"(%arg1) : (!toy<"array<2, 3>">)
        -> !toy<"array<3, 2>"> loc("test/codegen.toy":3:14)
```

```
Operation *createTransposeOp(FuncBuilder *builder,MLIRContext *ctx,
                             Value *input_array, Location location) {
  // We bundle our custom type in a `toy` dialect.
  auto toyDialect = Identifier::get("toy", ctx);
  // Create a custom type, in the MLIR assembly it is:  !toy<"array<3, 2>">
  auto type = OpaqueType::get(toyDialect, "array<3, 2>", ctx);

  // Fill in the `OperationState` with the required fields.
  OperationState result(ctx, location, "toy.transpose");
  result.types.push_back(type); // return type
  result.operands.push_back(input_value); // argument
  Operation *newTransposeOp = builder->createOperation(result);
  return newTransposeOp;
}
```

the type and operation identifier are opaque strings

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch2/mlir/MLIRGen.cpp#L351

This is the C++ API to build an “opaque” operation to MLIR. This works out-of-the-box but is not recommended.

The “Catch”

```
func @main() {
  %0 = "toy.print"() : () -> !toy<"array<2, foo, >">
}
```

Yes: this is a fully valid textual IR module, it round-trips through *mlir-opt*!

It should not! Broken on many aspects:

- the *toy.print* builtin is not a terminator,
- it should take an operand
- it shouldn’t return any value
- this type is malformed:
    - array dims must be integer
    - the list of dimension isn’t finished!

MLIR is great: you can represent anything!
Wait, did we just creation the JSON of compiler IR?

Dialects: Defining Rules and Semantics for the IR

A MLIR dialect includes:

- A prefix (“namespace” reservation)
- A list of custom types, each its C++ class.
- A list of operations, each its name and C++ class implementation:
  - Verifier for operation invariants (e.g. *toy.print* must have a single operand)
  - Semantics (has-no-side-effects, constant-folding, CSE-allowed, ….)
- Possibly custom parser and assembly printer
- Passes: analysis, transformations, and dialect conversions.

Google

Look Ma, Something Familiar There...

Dialects are powerful enough that you can wrap LLVM IR within an MLIR Dialect

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

Google

Operations: Regions are Powerful

```
%res:2 = "mydialect.morph"(%input#3) { some.attribute : true, other_attribute : 1.5 }
          : (!mydialect<"custom_type">) -> (!mydialect<"other_type">, !mydialect<"other_type">)
                                                   loc(callsite("foo" at "mysource.cc":10:8))
              { /* One Region      */ }
              { /* Another region */ }
```

- Regions are list of basic blocks nested alongside an operation.
- Opaque to passes by default, not part of the CFG.
- Similar to a function call but can reference SSA value defined outside.
- SSA value defined inside region don't escape

In MLIR, we are referring to Operations, not Instructions:

- there is no predefined list like the LLVM Instructions, MLIR is extensible.
- they can represent any corser-grain operations, while LLVM instructions are geared towards scalar.
- An operation can hold "regions", which are arbitrary large nested section of code.

Region Example: Affine Dialect

```
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
}
```

With custom parsing/printing: affine.for operations with an attached region feels like a regular for!

Extra semantics constraints in this dialect: the if condition is an affine relationship on the enclosing loop indices.

```
#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"() : () -> ()
    }
    ...
```

Same code without custom parsing/printing: closer to the internal in-memory representation.

Example of nice syntax *and* advanced semantics using regions attached to an operation

Example: TensorFlow

```
func some_tensorflow_computation(%input : !tf.tensor<*xf32>>) -> !tf.tensor<*xf32>> {
 %fetches = "tf.graph"() ({ // This operation models the TensorFlow graph executor semantics.
  // This region attached on tf.graph operation, is using a “sea of nodes” kind of representation
  ^op_A:
   // A TensorFlow operation directly referencing a value defined outside the region (here a function
   // argument). SSA values that are live inside the region can be used inside the region directly.
   %conv = "tf.SomeOp”(%input) : (!tf.tensor<*xf32>>) -> !tf.tensor<*xf32>>>
   "tf.yield”() // The terminator for the block yields control.

  ^op_B:
   // Another TensorFlow operation which consume the SSA value from the first one.
   // This creates an implicit scheduling dependency from ^op_A to ^op_B
   %sm = "tf.SoftMax”(%conv) : (!tf<"tensor<*xf32>">) -> !tf<"tensor<*xf32>">
   "tf.yield”()
...
```

Google

The Toy IR Dialect

A Toy Dialect

```
# User defined generic function that operates on unknown shaped arguments
# It will be specialized for every call-site when the shapes are known.
def multiply_transpose(a, b) {
  return a * transpose(b);
}
```

Dialect specific types, initially unknown shapes

```
func @multiply_transpose(%arg0: !toy<"array">, %arg1: !toy<"array">)
    attributes  {toy.generic: true} {
  %0 = "toy.transpose"(%arg1) : (!toy<"array">) -> !toy<"array">
  %1 = "toy.mul"(%arg0, %0) : (!toy<"array">, !toy<"array">) -> !toy<"array">
  "toy.return"(%1) : (!toy<"array">) -> ()
}
```

Custom terminator

```
$ bin/toy-ch5 -emit=mlir example.toy
```

A Toy Dialect

```
def main() {
  var a = [[1, 2, 3], [4, 5, 6]];
  var b<2, 3> = [1, 2, 3, 4, 5, 6];
  var c = multiply_transpose(a, b);
  print(c);
}
```

```
func @main() {
  %0 = "toy.constant"() {value: dense<tensor<2x3xf64>,
            [[1., 2., 3.], [4., 5., 6.]]>} : () -> !toy<"array<2, 3>">
  %1 = "toy.constant"() {value: dense<tensor<6xf64>,
            [1., 2., 3., 4., 5., 6.]>} : () -> !toy<"array<6>">
  %2 = "toy.reshape"(%1) : (!toy<"array<6>">) -> !toy<"array<2, 3>">
  %3 = "toy.generic_call"(%0, %2) {callee: "multiply_transpose"}
            : (!toy<"array<2, 3>">, !toy<"array<2, 3>">) -> !toy<"array">
  "toy.print"(%3) : (!toy<"array">) -> ()
  "toy.return"() : () -> ()
}
```

Point of specialization, shapes are known.

Google

A Toy Dialect

```
/// This is the definition of the Toy dialect. A dialect inherits from
/// Dialect and register custom operations and types (in its constructor).
/// It can also override general behavior of dialects exposed as virtual
/// method, for example regarding verification and parsing/printing.
class ToyDialect : public Dialect {
 public:
  explicit ToyDialect(MLIRContext *ctx);

  /// Parse a type registered to this dialect. Overridding this method is
  /// required for dialects that have custom types.
  /// Technically this is only needed to be able to round-trip to textual IR.
  Type parseType(llvm::StringRef tyData, Location loc,
                 MLIRContext *context) const override;

  /// Print a type registered to this dialect. Overridding this method is
  /// only required for dialects that have custom types.
  /// Technically this is only needed to be able to round-trip to textual IR.
  void printType(Type type, llvm::raw_ostream &os) const override;
};
```

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch3/include/toy/Dialect.h#L43-L57
https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch3/mlir/ToyDialect.cpp#L95-L101

A Toy Dialect: Custom Type

```
class ToyArrayType : public Type::TypeBase<ToyArrayType, Type,
                                         detail::ToyArrayTypeStorage> {
 public:
  /// Get the unique instance of this Type from the context.
  /// A ToyArrayType is only defined by the shape of the array.
  static ToyArrayType get(MLIRContext *context,
                          llvm::ArrayRef<int64_t> shape = {});

  /// Returns the dimensions for this Toy array, or and empty range for a generic array.
  llvm::ArrayRef<int64_t> getShape();

  /// Predicate to test if this array is generic (shape haven't been inferred yet).
  bool isGeneric() { return getShape().empty(); }

  /// Return the rank of this array (0 if it is generic)
  int getRank() { return getShape().size(); }

  /// Support method to enable LLVM-style RTTI type casting.
  static bool kindof(unsigned kind) { return kind == ToyTypeKind::TOY_ARRAY; }
};
```

"Facade" for our custom type

Storage for our type data. Like in LLVM: Types are uniqued in the context

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch3/include/toy/Dialect.h#L79-L105
https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch3/mlir/ToyDialect.cpp#L45

A (Robust) Toy Dialect

Types are now properly parsed / validated

```
$ echo 'func @foo() -> !toy<"bla">' | ./bin/toyc-ch3 -emit=mlir -x mlir -
loc("<stdin>":1:21): error: Invalid Toy type 'bla', array expected

$ echo 'func @foo() -> !toy<"array<>">' | ./bin/toyc-ch3 -emit=mlir -x mlir -
loc("<stdin>":1:21): error: Invalid toy array shape '<>'

$ echo 'func @foo() -> !toy<"array<1, >">' | ./bin/toyc-ch3 -emit=mlir -x mlir -
loc("<stdin>":1:21): error: Invalid toy array shape '<1, >'

$ echo 'func @foo() -> !toy<"array<1, 2>">' | ./bin/toyc-ch3 -emit=mlir -x mlir -
func @foo() -> !toy<"array<1, 2>">
```

Google

A Toy Dialect: Custom Operation

You can write a complete C++ class like here, but you'd likely use TableGen in most cases

```
class GenericCallOp
    : public Op<GenericCallOp, OpTrait::VariadicOperands,
                      OpTrait::OneResult> {
 public:
  /// MLIR will use this to register the operation with the parser/printer.
  static llvm::StringRef getOperationName() { return "toy.generic_call"; }

  /// Operations can add custom verification beyond the traits they define.
  /// We will ensure that all the operands are Toy arrays.
  bool verify();

  /// Interface to the builder to allow:
  ///   FuncBuilder::create<GenericCallOp>(...)
  /// This method populate the `state` that MLIR use to create operations.
  /// The `toy.generic_call` operation accepts a callee name and a list of
  /// arguments for the call.
  static void build(FuncBuilder *builder, OperationState *state,
                    llvm::StringRef callee,
                    llvm::ArrayRef<Value *> arguments);

  /// Return the name of the callee by fetching it from the attribute.
  llvm::StringRef getCalleeName();
...
```

Using "traits" to constrain our operations

Specific APIs for our operation

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch3/include/toy/Dialect.h#L161-L185

A (Robust) Toy Dialect

After registration, operations are now fully checked

```
$ cat test/Examples/Toy/Ch3/invalid.mlir
func @main() {
  "toy.print"()  : () -> ()
}
$ build/bin/toyc-ch3 test/Examples/Toy/Ch3/invalid.mlir -emit=mlir
loc("test/invalid.mlir":2:8): error: 'toy.print' op requires a single operand
```

Google

Toy High-Level Transformations

Google

Generic Function Specialization: similar to template instantiation

```
# User defined generic function that operates on unknown shaped arguments
def multiply_add(a, b, c) {
  return (a * b) + c;
}
```

```
func @multiply_add(%a: !toy<"array">, %b: !toy<"array">, %c: !toy<"array">)
  attributes  {toy.generic: true} {
  %prod = "toy.mul"(%a) : (!toy<"array">, !toy<"array">) -> !toy<"array">
  %sum = "toy.add"(%prod, %c) : (!toy<"array">, !toy<"array">) -> !toy<"array">
  "toy.return"(%sum) : (!toy<"array">) -> ()
}
```

```
// Let's assume 2-dimensional array, the C++ equivalent is:
template<int Ma, int Na, int Mb, int Nb, int Mc, int Nc>
auto multiply_add(array<Ma, Na> a, array<Mb, Nb> b, array<Mc, Nc> c)
{
  auto prod = mul(a, b);
  auto sum = add(prod, c);
  return sum;
}
```

Google

Generic Function Specialization

Clang would do it on the AST (TreeTransform), let's just write a pass!
(with all the benefits about testability: lit/FileCheck)

```
// Some familiar concept...
class ShapeInferencePass : public ModulePass<ShapeInferencePass> {

  void runOnModule() override {
    auto &module = getModule();
    ...
```

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch5/mlir/ShapeInferencePass.cpp#L110

Language Specific Optimizations

What about a trivial no-op?

```
def no_op(b) {
  return transpose(transpose(b));
}
```

Clang can't optimize away these loops:

```
#define N 100
#define M 100

void sink(void *);
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];
    }
  }
  sink(A);
}
```

Language Specific Optimizations

```
struct SimplifyRedundantTranspose : public RewritePattern {
  /// We register this pattern to match every toy.transpose in the IR.
  SimplifyRedundantTranspose(MLIRContext *context)
      : RewritePattern(TransposeOp::getOperationName(), /* benefit = */ 1, context) {}

  PatternMatchResult matchAndRewrite(Operation *op,
                                     PatternRewriter &rewriter) const override {
    // Directly cast the current operation as this will only get invoked on TransposeOp.
    TransposeOp transpose = op->cast<TransposeOp>();
    // look through the input to the current transpose
    mlir::Value *transposeInput = transpose.getOperand();
    mlir::Operation *transposeInputInst = transposeInput->getDefiningOp();
    // If the input is defined by another Transpose, bingo!
    TransposeOp transposeInputOp = dyn_cast_or_null<TransposeOp>(transposeInputInst);
    if (!transposeInputOp)
      return matchFailure();

    // Use the rewriter to perform the replacement
    rewriter.replaceOp(op, {transposeInputOp.getOperand()}, {transposeInputOp});
    return matchSuccess();
```

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch4/mlir/ToyCombine.cpp#L36-L65
https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch4/mlir/ToyCombine.cpp#L155

MLIR provides a generic “canonicalization” framework, similar to InstCombine but pluggable. This is showing the full C++ class that is involved in creatign a “RewritePattern” in MLIR. However in most cases you can generate it from TableGen.

Language Specific Optimizations

```
struct SimplifyRedundantTranspose : public RewritePattern {
  /// We register this pattern to match every toy.transpose in the IR.
  SimplifyRedundantTranspose(MLIRContext *context)
      : RewritePattern(TransposeOp::getOperationName(), /* benefit = */ 1, context) {}

  PatternMatchResult matchAndRewrite(Operation *op,
                                     PatternRewriter &rewriter) const override {
    // Directly cast the current operation as this will only get invoked on TransposeOp.
    TransposeOp transpose = op->cast<TransposeOp>();
    // look throu
    mlir::Value *
    mlir::Operati
    // If the input is defined by another Transpose, bingo!
    TransposeOp transposeInputOp = dyn_cast_or_null<TransposeOp>(transposeInputInst);
    if (!transposeInputOp)
      return matchFailure();

    // Use the rewriter to perform the replacement
    rewriter.replaceOp(op, {transposeInputOp.getOperand()}, {transposeInputOp});
    return matchSuccess();
```

```
// One line of TableGen
def : Pat<(Toy_TransposeOp ( Toy_TransposeOp $arg )), ($arg)>;
```

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch4/mlir/ToyCombine.cpp#L36-L65
https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch4/mlir/ToyCombine.cpp#L155

MLIR provides a generic “canonicalization” framework, similar to InstCombine but pluggable. This is showing the full C++ class that is involved in creatign a “RewritePattern” in MLIR. However in most cases you can generate it from TableGen.

Dialect Lowering

All the way to LLVM!

Google

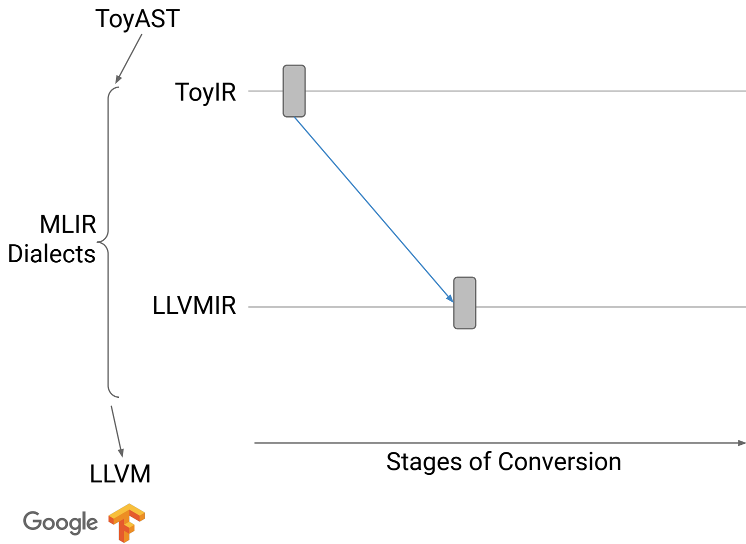
Towards CodeGen

Let’s make Toy executable!

MLIR does not have a code generator for target assembly...

Luckily, LLVM does! And we have an LLVM dialect in MLIR.

Now that we have seen how to perform high- (AST-) level transformations directly on Toy’s representation in MLIR, let’s try and make it executable. MLIR does not strive to redo all the work put into LLVM backends. Instead, it has an LLVM IR dialect, convertible to the LLVM IR proper, which we can target.



We would need to go from the Toy dialect to LLVM IR dialect within MLIR. But wait, can't we just emit LLVM IR directly from after transforming ToyIR like most compilers do?

But that is not really **M**ulti-**L**evel, is it?

MLIR has a “Standard” dialect for common operations (scalars, memory).

It also has a conversion from “Standard” to the LLVM IR dialect.

We also introduce a Linear Algebra dialect to capture commonalities between:

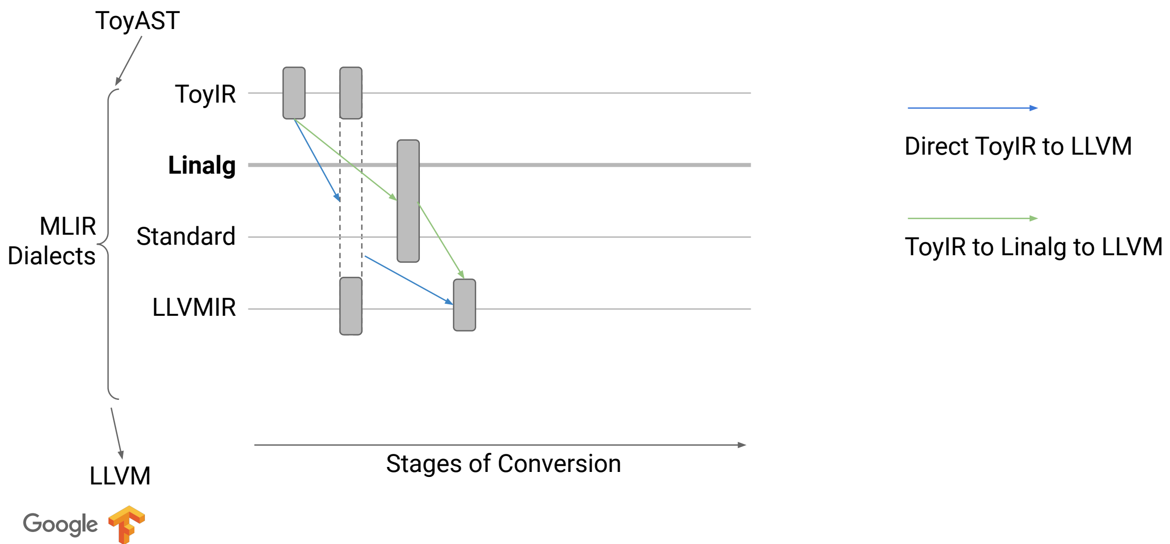
Toy, TensorFlow, BLAS…

* disclaimer: for educational purposes, we only focus on matmul / fc / gem

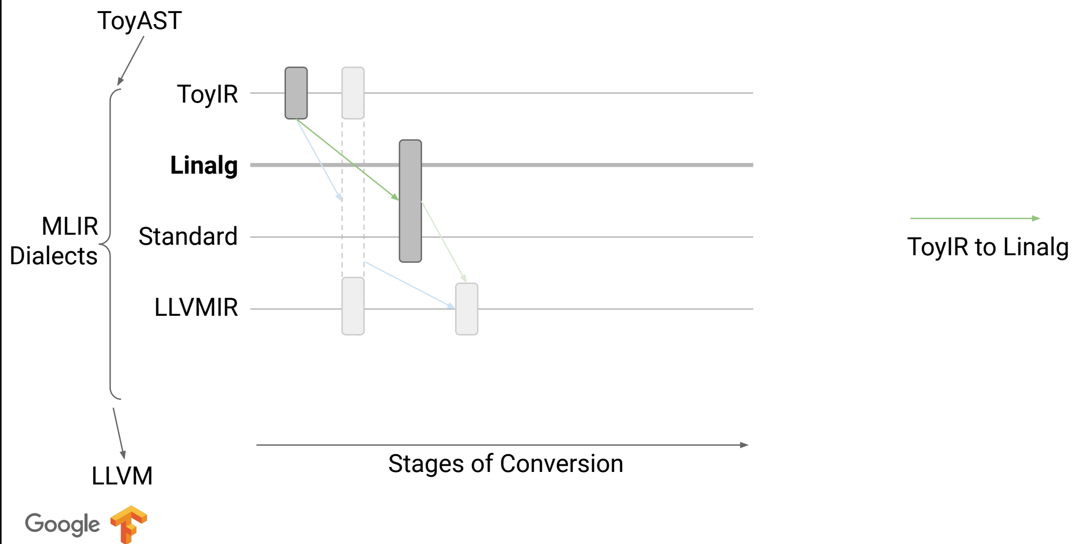
https://github.com/tensorflow/mlir/blob/master/lib/LLVMIR/Transforms/ConvertToLLVMDialect.cpp

One of the bacronyms for ML in MLIR is Multi-Level. MLIR supports multiple levels of abstractions within the same infrastructure. It provides, among others, a “standard dialect” of common scalar and vector instructions as well as common programming language concepts such as loops or first-class function values. Furthermore, it comes with lowering conversions from these concepts to the LLVM IR dialect, making it easier to implement a programming language.

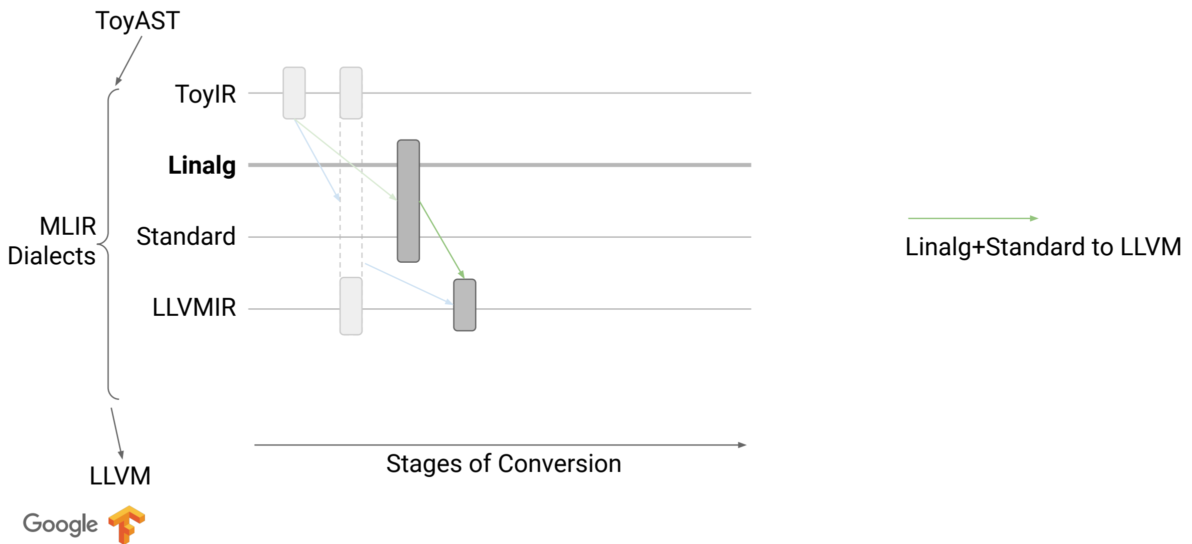
To further demonstrate the multi-level nature of MLIR, we will introduce another dialect that shares common functionality between our Toy language, TensorFlow, PyTorch, BLAS, etc. If you think they have nothing in common, they actually all support different kinds of matrix multiplication (gemm) operations. For the purposes of this tutorial, we will focus on these operations.



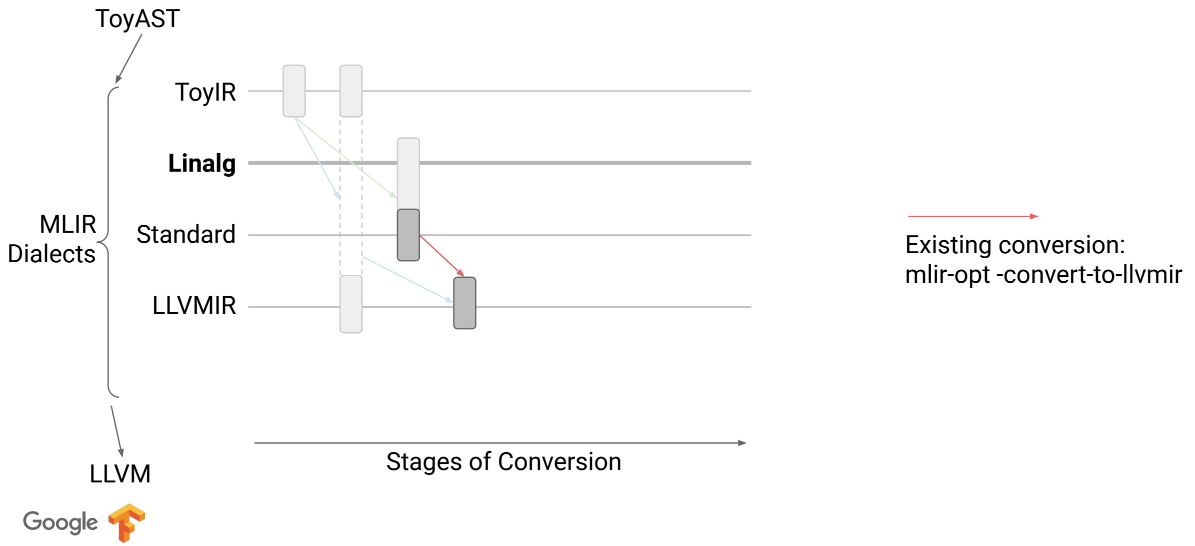
With the linear algebra dialect, we can start to define a graph where nodes are (groups of) dialects and edges are conversions between them. Thanks to the ability to mix dialects in the same module or function, MLIR naturally supports progressive partial lowering: we can lower some of the Toy operations (in particular, “print”) to the LLVM IR dialect while keeping the rest in Toy dialect for further optimization.



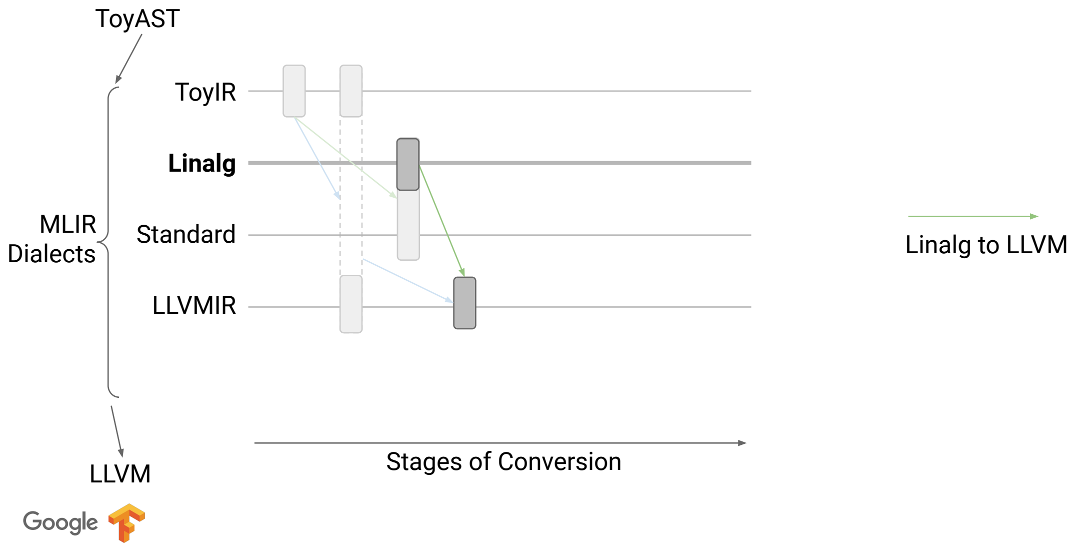
We will need to define a conversion from the Toy dialect to the combination of Linalg and Standard dialects (scalar operations and loads/stores).



As well as the conversion from the mix of Linalg and Standard dialects to the LLVM IR dialect.



In fact, MLIR already has a pass converting Standard to LLVM IR dialect.



So we only need to convert Linalg to LLVM IR and reuse the existing conversion for the rest.

Dialect Conversion

Dialect conversion requires three components:

- Function signature conversion (e.g., for result packing)

```
func @foo(i64) -> (f64, f64)

func @foo(!llvm<"i64">) -> !llvm<"{double, double}">
// typeof @foo is not !llvm<"{double,double}(i64)">
```

- Type conversion (e.g., block arguments)

```
i64         => !llvm<"i64">
f32         => !llvm<"float">
```

- Operation conversions

```
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*">
```

Dialect conversion consists of three parts as listed. Function signature conversion is optional, and is useful in cases where function-level metadata (represented as MLIR attributes) needs to be manipulated or when calling conventions must be implemented. For example, LLVM IR does not support multi-result functions while MLIR does. Therefore, the LLVM IR dialect implements a calling convention where the callee inserts multiple results in the LLVM's structure type before returning a single value and the caller extracts the values from the structure. This is already implemented in Standard to LLVM IR conversion and will be omitted from the tutorial.

Target Dialect (step 1): Linear Algebra Dialect

Let's define a linear algebra dialect we can target covering:

- memory buffer abstractions;
- common operations such as matrix multiplications.

Consider it a simplified example dialect for demonstration purposes.

* thanks to MLIR's properties, we will be able to perform advanced transformations on this dialect later on

Let's have a brief look at the target dialect -- the linear algebra dialect -- to get an understanding of what should be a result of the conversion from the Toy dialect.
(I must admit there is a hidden reason behind introducing Linalg, which will become evident in the last part of the tutorial).

Target Dialect (step 1): Linear Algebra Dialect

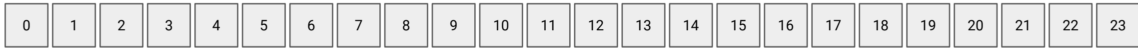
- Two types:
  - `!linalg.range` - triple of sizes
  - `!linalg.view` - sized/strided/projected view into a memory buffer (memref)
- Math operations:
  - `linalg.matmul` - matrix/matrix multiplication on 2d views
  - `linalg.matvec` - matrix/vector multiplication on 2d and 1d view
  - `linalg.dot` - dot product on 1d views
- Memory operations:
  - `linalg.view` - create a view from a memref and ranges
  - `linalg.slice` - create a view from a view and a range
  - `linalg.range` - create a range
  - `linalg.load` - load through a view
  - `linalg.store` - store through a view

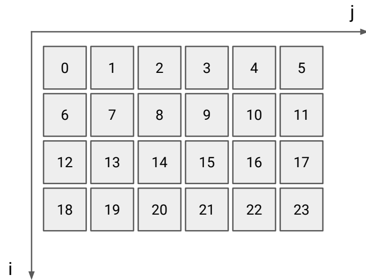
The linalg dialect introduces two new types: ranges and views, a set of mathematical and a set of memory operations.

Standard Memory Buffer - Memref

Let's define as a contiguous block of memory
indexed by multiple values
in a row-major format

```
memref<?x?x?x? x f64>
memref<4x6 x f32>
memref<42x? x i32>
```





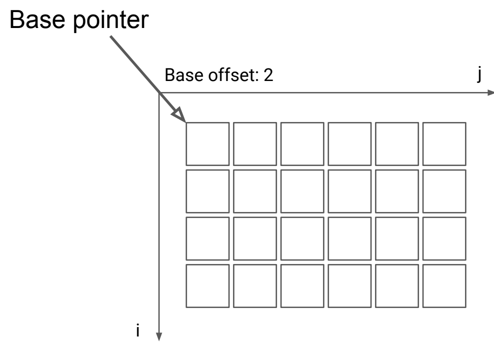

```
%memref = alloc() : memref<4x6 x f32>
%x = load %memref[%c1, %c3] : memref<4x6 x f32>
```

```
%0 = call i8* @malloc(i64 96)
%1 = bitcast i8* %0 to float*
%2 = mul i64 1, 6
%3 = add i64 %2, 3
%4 = getelementptr float, float* %1, %3
```

`memref<4x6 x f32>`

Standard Dialect in MLIR provides an abstraction for a sized memory buffer - MemRef. Particular storage details are not restricted by the spec, so each lowering can define them.

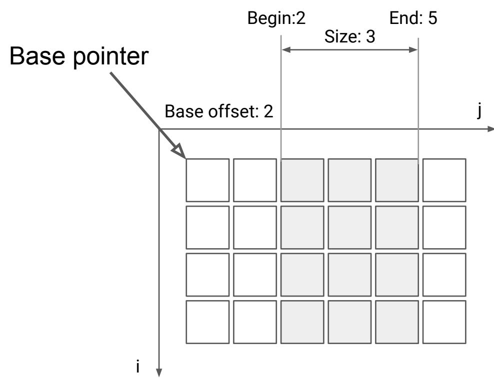
View Type Descriptor




```
%memref = alloc() : memref<4x6 x f32>
%ri = linalg.range %c2:%c5:%c2 : !linalg.range
%rj = linalg.range %c0:%c4:%c3 : !linalg.range
%v = linalg.view %memref : !linalg.view<?x?xf32>
```

Google

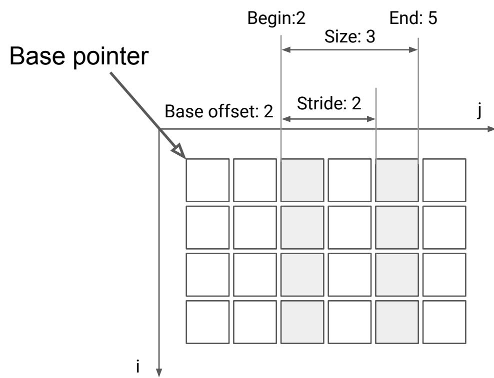
View Type Descriptor




```
%memref = alloc() : memref<4x6 x f32>
%ri = linalg.range %c0:%c4:%c3 : !linalg.range
%rj = linalg.range %c2:%c5:%c2 : !linalg.range
%v = linalg.view %memref[%ri, %rj] : !linalg.view<?x?xf32>
```

Google

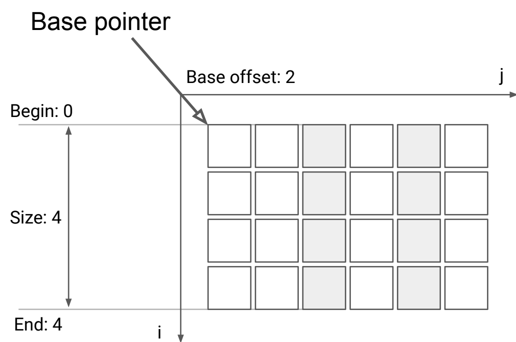
View Type Descriptor




```
%memref = alloc() : memref<4x6 x f32>
%ri = linalg.range %c0:%c4:%c3 : !linalg.range
%rj = linalg.range %c2:%c5:%c2 : !linalg.range
%v = linalg.view %memref[%ri, %rj] : !linalg.view<?x?xf32>
```

Google

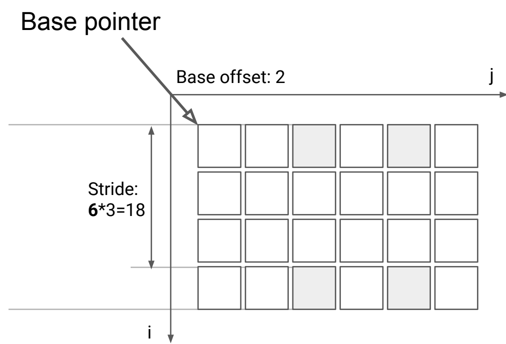
View Type Descriptor




```
%memref = alloc() : memref<4x6 x f32>
%ri = linalg.range %c0:%c4:%c3 : !linalg.range
%rj = linalg.range %c2:%c5:%c2 : !linalg.range
%v = linalg.view %memref[%ri, %rj] : !linalg.view<?x?xf32>
```

Google

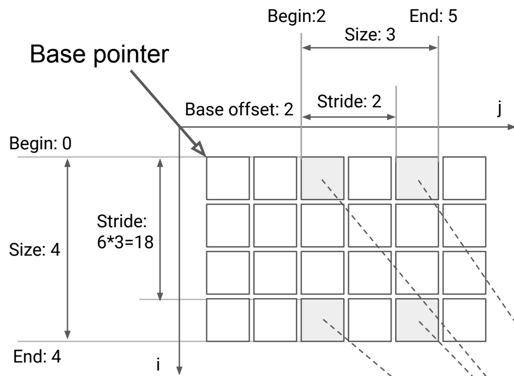
View Type Descriptor




```
%memref = alloc() : memref<4x6 x f32>
%ri = linalg.range %c0:%c4:%c3 : !linalg.range
%rj = linalg.range %c2:%c5:%c2 : !linalg.range
%v = linalg.view %memref[%ri, %rj] : !linalg.view<?x?xf32>
```

Google

View Type Descriptor

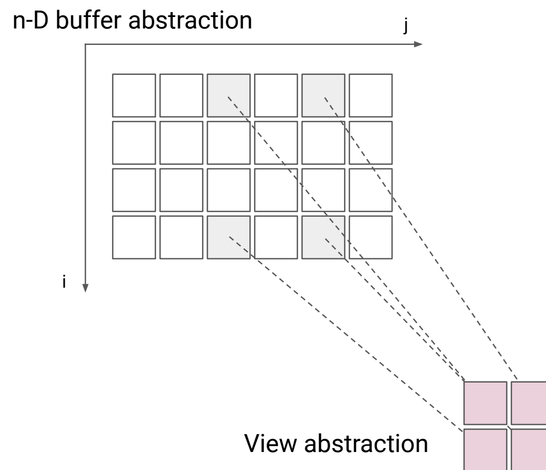



```
{ float*,  # base pointer
  i64,     # base offset
  i64[2]   # sizes
  i64[2] } # strides
```

```
%memref = alloc() : memref<4x6 x f32>
%ri = linalg.range %c2:%c5:%c2 : !linalg.range
%rj = linalg.range %c0:%c4:%c3 : !linalg.range
%v = linalg.view %memref[%ri, %rj] : !linalg.view<?x?xf32>
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/lib/ConvertToLLVMDialect.cpp

Target Dialect (step 1): Linear Algebra Dialect



The views allow one to index a multi-dimensional buffer (known in MLIR as MemRef) using compressed and projected indices. In particular, it makes possible to step over elements or dimensions.

Range Type Descriptor

Represents the range data (min, max, step) at runtime.
Let's define as a structure type.

```
linalg.range => llvm<"{i64, i64, i64}">
```

The range type is a simple triple of inclusive minimum, exclusive maximum and step, similar to Python or Fortran array indexing abstractions. It can be easily converted to an LLVM IR structure type with three integers. We assume a 64-bit architecture and use 64-bit integers for sizes.

Type Conversion

Define a Type -> Type function:

```
Type linalg::convertType(Type t) {
  /* ... */
  if (auto arrayTy = t.dyn_cast<linalg::RangeType>()) {
    llvm::Type *i64Ty = llvm::Type::getInt64Ty();
    llvm::Type *structTy = llvm::StructType::get(i64Ty, i64Ty, i64Ty);
    Type mlirType = mlir::LLVM::LLVMType(structTy, context);
    return mlirType;
  }
  /* ... */
}
```

The type conversion is defined as a function that takes a type and returns a new type. It can dispatch based on the old type, or do nothing when type conversion is not necessary. The code illustrates also illustrates how MLIR wraps LLVM IR types directly in its type system.

Type Conversion

Define a Type -> Type function:

```
Type toy::convertType(Type t) {
  /* ... */
  if (auto arrayTy = t.dyn_cast<toy::ToyArrayType>()) {
    Type converted = MemRefType::get(
        arrayTy.getShape(),         // same shape
        arrayTy.getElementType(),  // same element type
        /*memorySpace=*/0, /*mapComposition=*/{});
    return converted;
  }
  /* ... */
}
```

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch5/mlir/EarlyLowering.cpp

Conversion from Toy arrays to LLVM IR is equally simple.

Operation Conversion

```
class OpConversion : public DialectOpConversion {
  SmallVector<Value *, 4> rewrite(Operation *op, ArrayRef<Value *> operands,
                                  FuncBuilder &rewriter) const override {
```

Transformed results

Original operation
(with original operands)

IRBuilder

Transformed operarands

https://github.com/tensorflow/mlir/blob/master/include/mlir/Transforms/DialectConversion.h

Operation conversions are defined by deriving DialectOpConversion. This is a class that belongs to the MLIR pattern-matching infrastructure and is similar to those produce by Tablegen-generated high-level rewrites. It takes into account the necessity to change types. Similarly to the graph rewriter, it first needs to match the operation, which is a trivial "isa" check that we omit here for simplicity. The actual rewriting happens in the "rewrite" function.

Target Dialect (step N): LLVM IR

LLVM IR is represented as an MLIR *dialect*, containing

- `mlir::LLVM::LLVMType` opaquely wrapping any `llvm::Type *` into MLIR;
- LLVM instructions replicated as MLIR operations.

Instructions are defined in TableGen, easy to extend.

#results | name (op code) | operand type and name

```
def LLVM_LoadOp : LLVM_OneResultOp<"load">, Arguments<(ins LLVM_Type:$addr)>,
                  LLVM_Builder<"$res = builder.CreateLoad($addr);">
```

call to llvm::IRBuilder

https://github.com/tensorflow/mlir/blob/master/include/mlir/LLVMIR/LLVMOps.td

Before going into details about the rewriting, let's examine the structure of the LLVM IR dialect which we will target. It defines a single Type subclass that wraps LLVM types as they are, reusing LLVM's printing and parsing hooks. LLVM IR instructions are replicated as MLIR operations, which are defined using Tablegen. While some LLVM IR intrinsics may still be missing, they are very easy to add using the concise operation description syntax we designed around Tablegen.

Target Dialect (step N): LLVM IR

LLVM IR is represented as an MLIR *dialect*, containing

- `mlir::LLVM::LLVMType` opaquely wrapping any `llvm::Type *` into MLIR;
- LLVM instructions replicated as MLIR operations.

Instructions are defined in TableGen, easy to extend.

```
%1 = llvm.load %0 : !llvm<"float*">

%42 = llvm.getelementptr %41[%30, %32, %31] : !llvm<"{i64[3]}">
```

https://github.com/tensorflow/mlir/blob/master/include/mlir/LLVMIR/LLVMOps.td

LLVM IR dialect operations look similar to the actual LLVM IR, prefixed with “llvm” and with MLIR flavor of avoiding trivially inferrable types and placing the types in trailing positions.

Operation Conversion

```
class OpConversion : public DialectOpConversion {
  SmallVector<Value *, 4> rewrite(Operation *op, ArrayRef<Value *> operands,
                                  FuncBuilder &rewriter) const override {
```

https://github.com/tensorflow/mlir/blob/master/include/mlir/Transforms/DialectConversion.h

Back to the structure of the conversion function, there are the following essential interfacing points.

Operation Conversion (`linalg.range`)

```
SmallVector<Value *, 4> rewrite(Operation *op, ArrayRef<Value *> operands,
                                FuncBuilder &rewriter) const override {
  auto rangeOp = op->cast<linalg::RangeOp>();
  auto rangeDescriptorType =
      linalg::convertLinalgType(rangeOp.getResult()->getType());

  using namespace intrinsics;
  auto context = edsc::ScopedContext(rewriter, op->getLoc());


  Value *rangeDescriptor = undef(rangeDescriptorType);
  rangeDescriptor = insertvalue(rangeDescriptorType, rangeDescriptor,
                                operands[0], makePositionAttr(rewriter, 0));
  rangeDescriptor = insertvalue(rangeDescriptorType, rangeDescriptor,
                                operands[1], makePositionAttr(rewriter, 1));
  rangeDescriptor = insertvalue(rangeDescriptorType, rangeDescriptor,
                                operands[2], makePositionAttr(rewriter, 2));
  return {rangeDescriptor};
}
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/lib/ConvertToLLVMDialect.cpp

This is the entire conversion of the linalg.range op to LLVM IR. I don't expect you to read this from the slide.

Operation Conversion (`linalg.range`)

```
SmallVector<Value *, 4> rewrite(Operation *op, ArrayRef<Value *> operands,
                                FuncBuilder &rewriter) const override {
```



```
Value *rangeDescriptor = undef(rangeDescriptorType);
rangeDescriptor = insertvalue(rangeDescriptorType, rangeDescriptor,
                              operands[0], makePositionAttr(rewriter, 0));
```

Instead, let's focus on how new operations can be constructed. We use MLIR's declarative builders API here, where the function name corresponds to the operation to create, and where one can compose function calls to pass the results of one operation as operands to another one. Depending on the operation constructors, one may need to specify return types, operands and attributes.

Operation Conversion (`linalg.range`)

```
SmallVector<Value *, 4> rewrite(Operation *op, ArrayRef<Value *> operands,
                                FuncBuilder &rewriter) const override {




  Value *rangeDescriptor = rewriter.create<LLVM::UndefOp>(
      op->getLoc(), rangeDescriptorType);
  rangeDescriptor = rewriter.create<LLVM::InsertValueOp>(
      op->getLoc(), rangeDescriptorType, rangeDescriptor,
      operands[0], makePositionAttr(rewriter, 0));
```

Classical IRBuilder syntax is also available.

LLVM-flavored IRBuilder syntax is also available, with additional templating to support MLIR's extendable instruction set.

Putting It All Together

```
class Lowering : public DialectConversion {
public:
```

1. Type conversion

2. Function signature conversion

3. Operation conversion

```
private:
  llvm::BumpPtrAllocator allocator;
};
```

https://github.com/tensorflow/mlir/blob/master/include/mlir/Transforms/DialectConversion.h

Finally, after having defined the type and operation conversions, we can put everything together in a form accepted by the dialect conversion infrastructure. To do so, we derive from the DialectConvesion class and override the three functions that correspond to the three components of the conversion.

Putting It All Together

```
class Lowering : public DialectConversion {
public:
  // This gets called for block and region arguments, and attributes.
  Type convertType(Type t) override { return linalg::convertLinalgType(t); }
```

2. Function signature conversion

3. Operation conversion

```
private:
  llvm::BumpPtrAllocator allocator;
};
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/lib/ConvertToLLVMDialect.cpp

Type conversion calls the function we defined.

Putting It All Together

```
class Lowering : public DialectConversion {
public:
  // This gets called for block and region arguments, and attributes.
  Type convertType(Type t) override { return linalg::convertLinalgType(t); }

  // This gets called for functions.
  FunctionType convertFunctionSignatureType(FunctionType type,
      ArrayRef<NamedAttributeList> argAttrs,
      SmallVectorImpl<NamedAttributeList> &convertedArgAttrs) { /*...*/ }
```

3. Operation conversion

```
private:
  llvm::BumpPtrAllocator allocator;
};
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/lib/ConvertToLLVMDialect.cpp

Function conversion can just call to the default (parent) implementation to perform one-to-one conversion of function types and results using “convertType” we defined above. The existing part of MLIR’s conversion infrastructure will take care of the calling conventions.

Putting It All Together

```
class Lowering : public DialectConversion {
public:
  // This gets called for block and region arguments, and attributes.
  Type convertType(Type t) override { return linalg::convertLinalgType(t); }

  // This gets called for functions.
  FunctionType convertFunctionSignatureType(FunctionType type,
      ArrayRef<NamedAttributeList> argAttrs,
      SmallVectorImpl<NamedAttributeList> &convertedArgAttrs) { /*...*/ }

  // This gets called once to set up operation converters.
  llvm::DenseSet<DialectOpConversion *>
  initConverters(MLIRContext *context) override {
    return ConversionListBuilder<RangeOpConversion, SliceOpConversion,
                                 ViewOpConversion>::build(allocator, context);
  }

private:
  llvm::BumpPtrAllocator allocator;
};
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/lib/ConvertToLLVMDialect.cpp

Finally, we override the initConverters function that is called once before every module conversion to populate the list of supported conversions, giving the caller the opportunity to use different conversions depending on the module. Inside, we use a helper function from MLIR to allocate instances of the list of conversion classes in LLVM's BumpPtrAllocator.

Putting It All Together

```
class Lowering : public DialectConversion {
public:
  // This gets called for block and region arguments, and attributes.
  Type convertType(Type t) override { return linalg::convertLinalgType(t); }

  // This gets called for functions.
  FunctionType convertFunctionSignatureType(FunctionType type,
      ArrayRef<NamedAttributeList> argAttrs,
      SmallVectorImpl<NamedAttributeList> &convertedArgAttrs) { /*...*/ }

  // This gets called once to set up operation converters.
  llvm::DenseSet<DialectOpConversion *>
  initConverters(MLIRContext *context) override {
    return ConversionListBuilder<RangeOpConversion, SliceOpConversion,
                                 ViewOpConversion>::build(allocator, context);
  }

private:
  llvm::BumpPtrAllocator allocator;
};
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/lib/ConvertToLLVMDialect.cpp

This is the glue code putting together a conversion from Linalg to LLVM IR, the code for Toy To LLVM IR is quite similar and you may find both in the repository. Conversions are orthogonal to the pass management infrastructure, and each pass may choose to run one or multiple conversions. IR verifier does not kick in in the conversions, letting them temporarily break the validity of operations, in particular use incorrect types, as long as the validity is restored at the end of the pass.
This completes the illustration of the lowering infrastructure. We can now look at the more interesting transformations that MLIR allows one to perform using the Linalg dialect we defined above.

A Dialect for Linear Algebra Optimizations

Let’s see how to use the infrastructure we’ve seen so far to build a minimal dialect which supports more advanced transformations.

Building a Linalg Dialect : Rationale

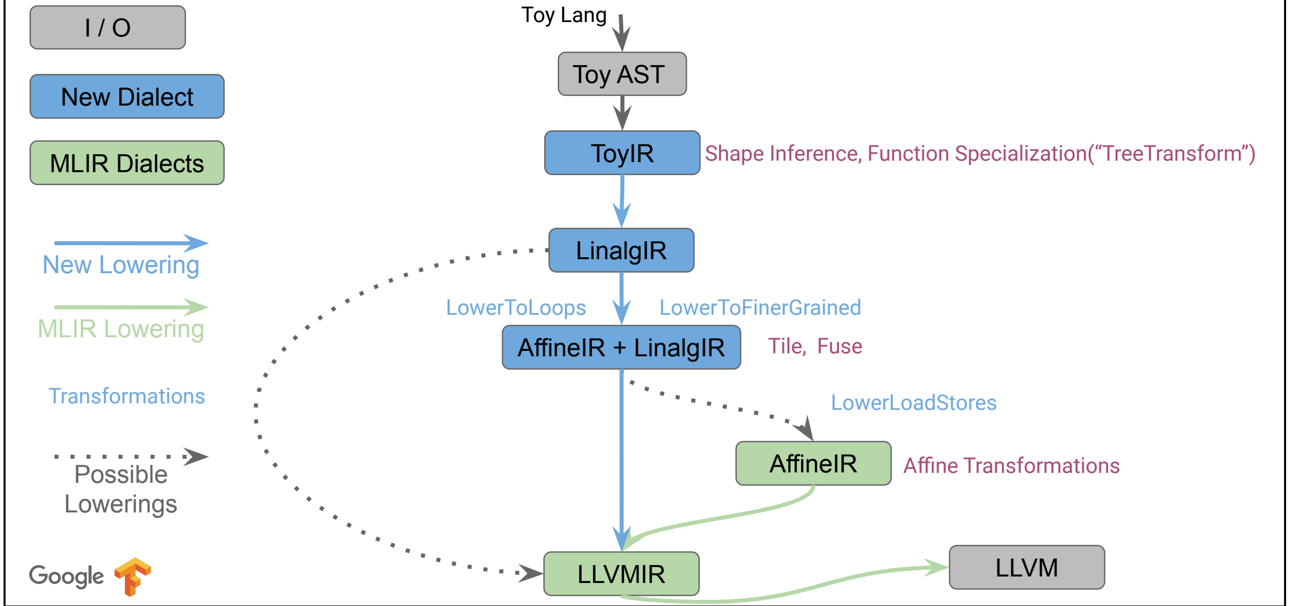
Explore building a `linalg` dialect and compiler with the following properties:

- Linear algebra primitives as first-class citizens
- From which it is easy to lower into:
  - library calls,
  - ARM SVE, TPU, coarser ISAs ...
  - LLVM IR
- Supports key transformations (tiling, fusion, bulk memory transfers)
  - Without complex analyses

Moore's Law
Dennard Scaling
Dark Silicon

Locking-in performance gains from good library implementations is a must.
Optimize across loops and library calls for locality and custom implementations.

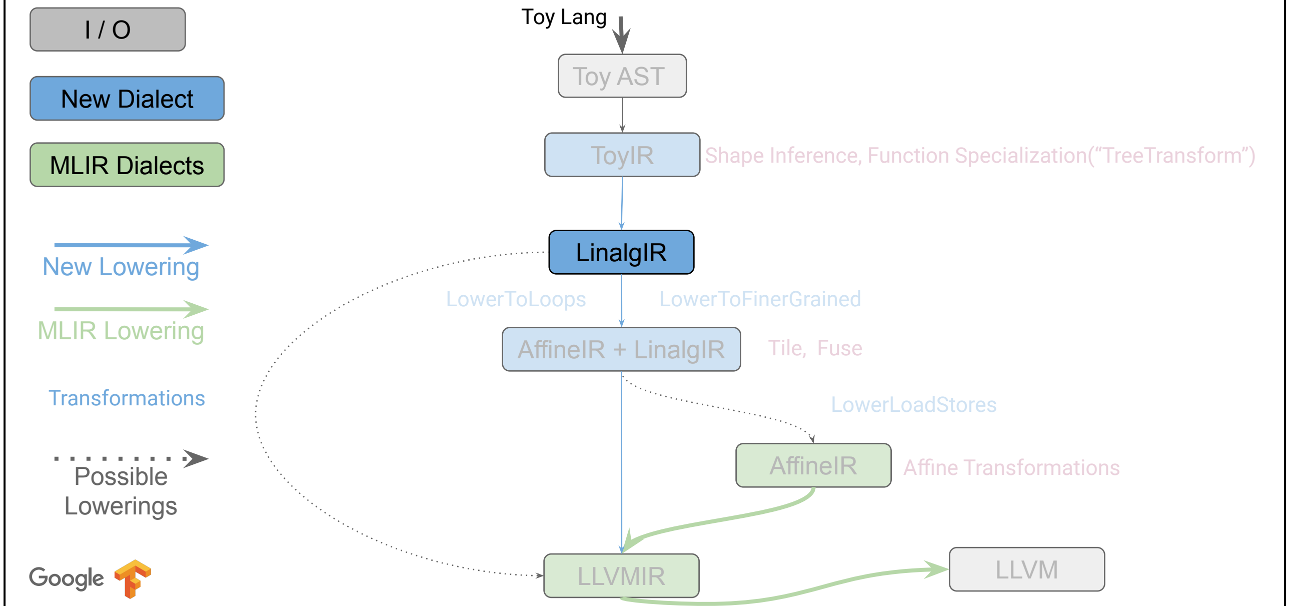
MLIR makes it easy to spin a new IR and experiment.
Let's see how this translates to a problem that is difficult to even represent with a low-level IR such as LLVM.



Here is the whole end to end picture of the system we are building in this tutorial.
The blue boxes and arrows are the pieces we have concretely built.
The green boxes and arrows already existed in MLIR and we just connected to them.

Linalg Type System

Google



Let's look at the type system, it is fully contained within the Linalg box.

Linalg Type System And Type Building Ops

- RangeType: RangeOp create a (min, max, step)-triple of `index` (intptr_t)

```
%0 = linalg.range %c0:%arg1:%c1 : !linalg.range
```

intptr_t intptr_t intptr_t

- Used for stepping over
  - loop iterations (loop bounds)
  - data structures

Google

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/include/linalg1/RangeOp.h
https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/include/linalg1/RangeType.h

Linalg Type System And Type Building Ops

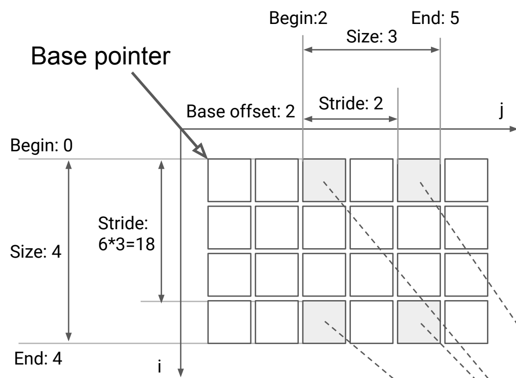
- ViewType: ViewOp creates an n-d *"indexing"* over a `MemRefType`


```
%8 = linalg.view %7[%r0, %r1] : !linalg.view<?x?xf32>


%9 = linalg.view %7[%r0, %row] : !linalg.view<?xf32>
```

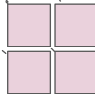
https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/include/linalg1/ViewOp.h
https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/include/linalg1/ViewType.h

View Type Descriptor in LLVM IR




```
{ float*,  # base pointer
  i64,     # base offset
  i64[2]   # sizes
  i64[2] } # strides
```

```
%memref = alloc() : memref<4x6 x f32>
%ri = linalg.range %c2:%c5:%c2 : !linalg.range
%rj = linalg.range %c0:%c4:%c3 : !linalg.range
%v = linalg.view %memref[%ri, %rj] : !linalg.view<?x?xf32>
```



https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/lib/ConvertToLLVMDialect.cpp

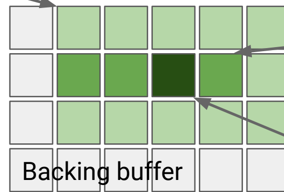
From the point of view of Linalg, this is a separate concern hidden behind implementation details: linalg types and operations can operate and compose at a higher level of abstraction and avoid analyses on more complex details.

Linalg Type System And Type Building Ops

- SliceOp creates a strict “sub-view” of a ViewType along a dimension

```
linalg.slice %8[*, %c0] : !linalg.view<?xf32>
```

intptr_t

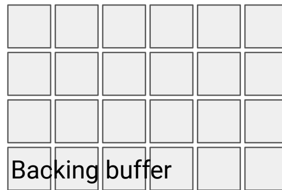


https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg1/include/linalg1/SliceOp.h

Linalg View: Digging Deeper

- Views over contiguous memory regions
  - Fortran, APL, boost::multi_array
  - Machine Learning Community: XLA, Torch, TVM

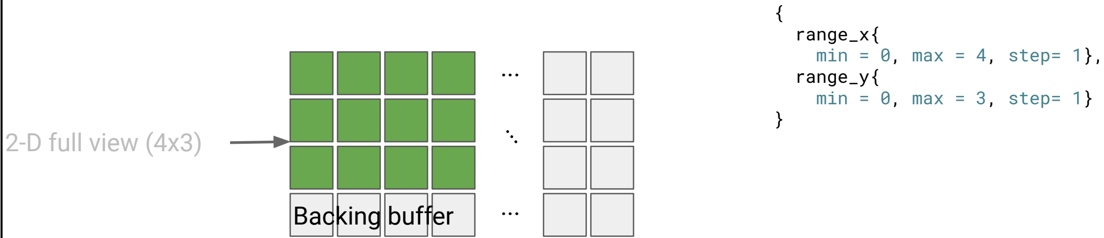
- Simplifying assumptions for analyses and IR construction
  - E.g. non-overlapping rectangular memory regions (symbolic shapes)



Google

Linalg View: Digging Deeper

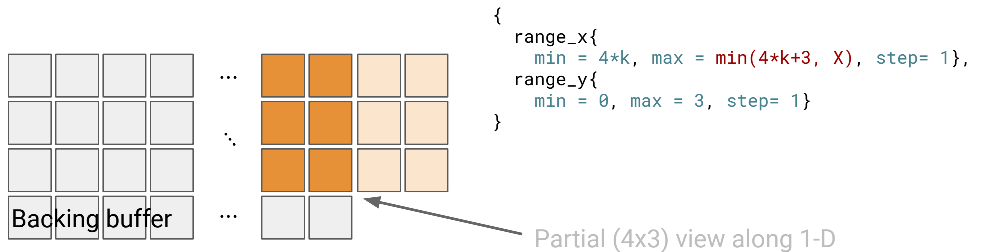
- Simplifying assumptions for analyses and IR construction
  - E.g. non-overlapping rectangular memory regions (symbolic shapes)
  - Data abstraction encodes boundary conditions



In linalg, a given 4x3 view can adapt to the shape of the backing buffer. If the view is mapped to a region fully contained within the buffer, it is a “full view”.

Linalg View: Digging Deeper

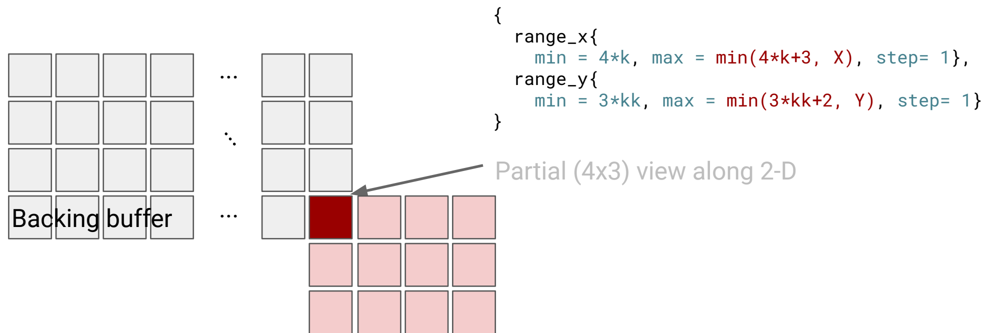
- Simplifying assumptions for analyses and IR construction
  - E.g. non-overlapping rectangular memory regions (symbolic shapes)
  - Data abstraction encodes boundary conditions



In linalg, a given 4x3 view can adapt to the shape of the backing buffer. If the view is mapped to a region not fully contained within the buffer, it is a “partial view”.
A partial view can be intersected with a full view of the whole backing buffer to handle boundary conditions without requiring min/max loop bound conditions.

Linalg View: Digging Deeper

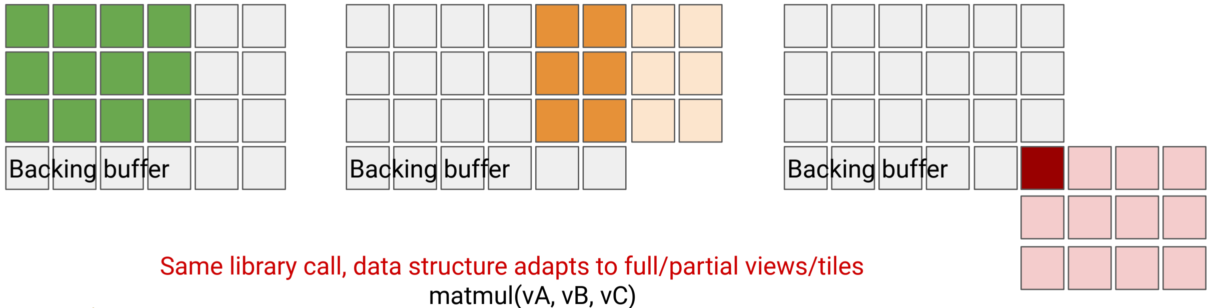
- Simplifying assumptions for analyses and IR construction
  - E.g. non-overlapping rectangular memory regions (symbolic shapes)
  - Data abstraction encodes boundary conditions



Google

Linalg View

- Simplifying assumptions for analyses and IR construction
  - E.g. non-overlapping rectangular memory regions (symbolic shapes)
  - Data abstraction encodes boundary conditions



Same library call, data structure adapts to full/partial views/tiles

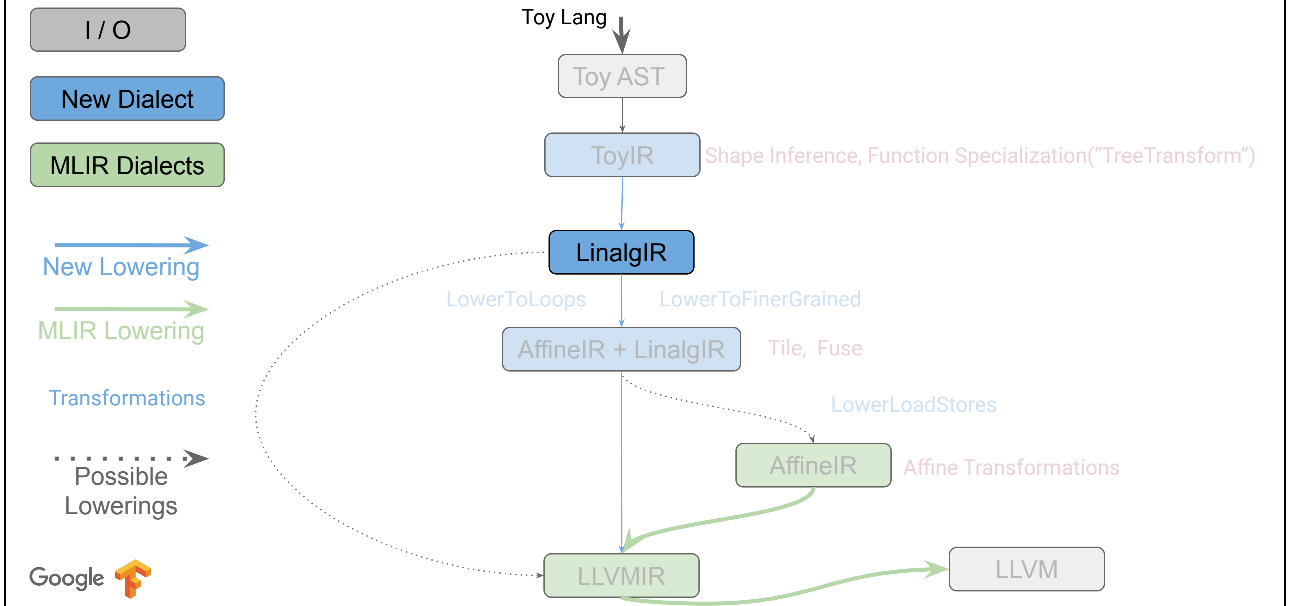
matmul(vA, vB, vC)

Google

Since the view encodes the boundary conditions dynamically, we can “just call” library operations on views (e.g. BLAS3 gemm)

Linalg Operations

Google



Let's now look at the operations we define in Linalg, still in the LinalgIR box.

Defining Linalg Operations

- `linalg.dot`, `linalg.matvec`, `linalg.matmul` operate on `ViewType`
  - parse, build, verify, print

```
LogicalResult linalg::MatmulOp::verify() {
  // Generic verification
  if (failed(LinalgBaseType::verify()))
    return failure();
  // Op-specific verification knows about expected ViewType ranks
  auto *A = getOperand(0), *B = getOperand(1), *C = getOperand(2);
  unsigned index = 0;
  for (auto *v : {A, B, C}) {
    if (getViewRank(v) != 2)
      return emitOpError(
        "operand " + Twine(index++) + " must be of rank 2");
  }
  return success();
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg2/lib/TensorOps.cpp

As we saw in the previous part of the tutorial, creating a new mlir op always consist in subclassing mlir::Op and defining the parse, build, verify and print methods.
Here is an example of MatmulOp::verify, it just checks that all operands are of rank 2.

Defining Matmul

- `linalg.matmul` operates on `view<?x?xf32>, view<?x?xf32>, view<?x?xf32>`

```
func @call_linalg_matmul(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>){
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  %rM = linalg.range %c0:%M:%c1 : !linalg.range
  %rN = linalg.range %c0:%N:%c1 : !linalg.range
  %rK = linalg.range %c0:%K:%c1 : !linalg.range
  %4 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  %6 = linalg.view %B[%rK, %rN] : !linalg.view<?x?xf32>
  %8 = linalg.view %C[%rM, %rN] : !linalg.view<?x?xf32>
  linalg.matmul(%4, %6, %8) : !linalg.view<?x?xf32>
  return
}
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg2/lib/TensorOps.cpp

This is an example of usage of linalg.matmul with views constructed from the full range of memref.
The constant and dim operations are standard MLIR operations.
Memref is a standard MLIR type and we build linalg on top of these existing constructs.

Defining Matvec

- `linalg.matvec` operates on `view<?x?xf32>, view<?xf32>, view<?xf32>`

```
func @call_linalg_matvec(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>,
%row: index, %col: index){
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  %rM = linalg.range %c0:%M:%c1 : !linalg.range
  %rN = linalg.range %c0:%N:%c1 : !linalg.range
  %rK = linalg.range %c0:%K:%c1 : !linalg.range
  %4 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  %6 = linalg.view %B[%rK, %rN] : !linalg.view<?x?xf32>
  %8 = linalg.view %C[%rM, %rN] : !linalg.view<?x?xf32>
  %9 = linalg.slice %6[*, %col] : !linalg.view<?xf32>
  %10 = linalg.slice %8[*, %col] : !linalg.view<?xf32>
  linalg.matvec(%4, %9, %10) : !linalg.view<?xf32>
  return
}
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg2/lib/TensorOps.cpp

Similarly, a matvec takes a single column slice of the backing view (for B and C) and operates on 1-D views for B and C.
This is because we chose to define matvec this way (other definitions would have been possible too).

Defining Dot

- `linalg.dot` operates on `view<?xf32>, view<?xf32>, view<f32>`

```
func @call_linalg_dot(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>,
%row: index, %col: index){
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  %rM = linalg.range %c0:%M:%c1 : !linalg.range
  %rN = linalg.range %c0:%N:%c1 : !linalg.range
  %rK = linalg.range %c0:%K:%c1 : !linalg.range
  %4 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  %6 = linalg.view %B[%rK, %rN] : !linalg.view<?x?xf32>
  %8 = linalg.view %C[%rM, %rN] : !linalg.view<?x?xf32>
  %9 = linalg.slice %6[*, %col] : !linalg.view<?xf32>
  %10 = linalg.slice %8[*, %col] : !linalg.view<?xf32>
  %11 = linalg.slice %4[%row, *] : !linalg.view<?xf32>
  %12 = linalg.slice %10[%row] : !linalg.view<f32>
  linalg.dot(%11, %9, %12) : !linalg.view<f32>
```

A dot product operates on further row slices.

Generalizing to LinalgBaseOp

- LinalgBaseOp<NumParallel, NumReduction, NumInputs, NumOutputs>
  - Reads and writes `linalg.view` input/output parameters
  - `linalg.dot`, `linalg.matvec`, `linalg.matmul`
  - Pointwise operations, broadcast, reduce, arbitrary transposes
  - inner, outer, Kronecker, Hadamard products

- `Linalg` keeps high-level operators as long as possible and lowers gradually

- A few properties, specified declaratively, enable analysis and transformations

Analysis on loops has similarities to raising.
Instead use a declarative lowering strategy.

More generally, it is possible to define a generic linalg operation that exposes a few properties and encompasses many linear algebra operations.
This tutorial does not consider more operations than the ones already introduced but operates on the properties to create generic lowerings and transformations that could apply to all such operations.

A Simple Transformation

Google

SliceOp Folding Strawman Transformation

- `linalg.slice` used to create sub-views, but they create chains
  - In a real system would get defined away.
  - This is a strawman pass to showcase SSA and MLIR APIs.

SliceOp Folding: Goal

- `linalg.slice` are used throughout for sub-views, but they create chains

```
func @linalg_dot(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>,
%row: index, %col: index) {
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  %rM = linalg.range %c0:%M:%c1 : !linalg.range
  %rN = linalg.range %c0:%N:%c1 : !linalg.range
  %rK = linalg.range %c0:%K:%c1 : !linalg.range
  %4 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  %6 = linalg.view %B[%rK, %rN] : !linalg.view<?x?xf32>
  %8 = linalg.view %C[%rM, %rN] : !linalg.view<?x?xf32>
  %9 = linalg.slice %6[*, %col] : !linalg.view<?xf32>
  %10 = linalg.slice %8[*, %col] : !linalg.view<?xf32>
  %11 = linalg.slice %4[%row, *] : !linalg.view<?xf32>
  %12 = linalg.slice %10[%row] : !linalg.view<f32>
  linalg.dot(%11, %9, %12) : !linalg.view<f32>
}
```

```
...
%9 = linalg.view %B[%rK, %ccol]
%11 = linalg.view %A[%row, %rK]
%12 = linalg.view %C[%row, %col]
linalg.dot {%11, %9} -> {%12}
```

SliceOp Folding: Implementation

- `linalg.slice` are used throughout for sub-views, but they create chains

MLIR provides the SSA graph traversal, rewrite, propagation, cleanups, pretty-printing

```
void linalg::foldSlices(Function *f) {
  f->walk<SliceOp>([](SliceOp sliceOp) {
    auto *sliceResult = sliceOp.getResult();
    auto viewOp = createFullyComposedView(sliceResult);
    sliceResult->replaceAllUsesWith(viewOp.getResult());
    sliceOp.erase();
  });
}
```

Some details in here related to the type system

f->walk traverses the IR in postorder and allows in-place rewrites and erasure without invalidating iterators.
This is a lower level implementation detail, such a transformation would typically be exposed via an mlir::Pass or am mlir::RewritePattern.

Lowering

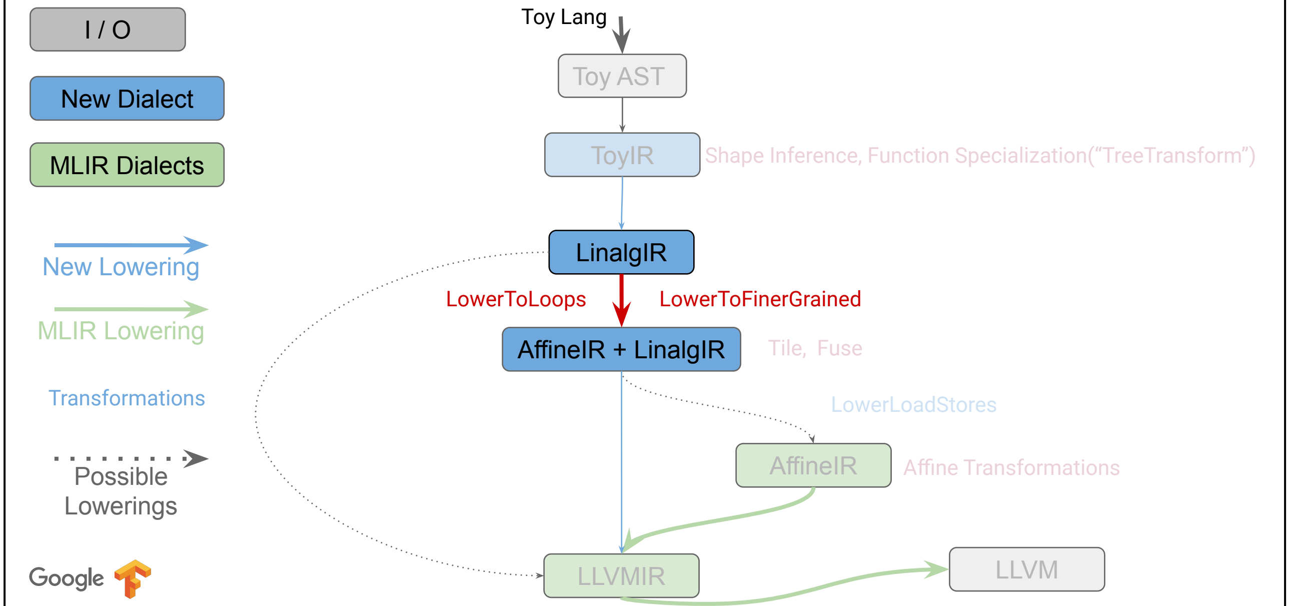
Google

General Partial Lowering Strategy

Ops declare properties (i.e. contracts they respect)

External transformations use these properties to gradually lower parts of the IR

Analyses are minimal (only SSA use-def chains)



We now look at how to reduce coarse grained Linalg ops into finer grained Linalg ops and loops.

LinalgBaseOp Property 1: emitScalarImplementation

Every LinalgBaseOp *"declares"* its scalar form, *given enclosing loops*

- dot : `C() = select(r_i == 0, 0, C()) + A(r_i) * B(r_i)` *given par: () red (r_i)*
- matvec : `C(i) = select(r_j == 0, 0, C(i)) + A(i, r_j) * B(r_j)` *given par: (i) red (r_j)*
- matmul : `C(i, j) = select(r_k == 0, 0, C(i, j)) + A(i, r_k) * B(r_k, j)`
  given par: (i, j) red (r_k)

Given enclosing loops

- Explicit handles allow composition (e.g. emit loop nest, emit tiled version, ...)

We use an index notation close to Einstein notation or einsum.
A linalg operation has enclosing parallel and reduction loops (prefixed by r_i).
Loop order is in the order passed to emitScalarImplementation.

LinalgBaseOp Property 1: emitScalarImplementation

```
void linalg::DotOp::emitScalarImplementation(
    llvm::ArrayRef<Value *> parallelIvs, llvm::ArrayRef<Value *> reductionIvs) {
  using IndexedValue = TemplatedIndexedValue<linalg::intrinsics::load,
                                             linalg::intrinsics::store>;
  assert(reductionIvs.size() == 1);
  auto innermostLoop = getForInductionVarOwner(reductionIvs.back());
  auto *body = innermostLoop.getBody();
  ScopedContext scope( // account for affine.terminator in loop.
      FuncBuilder(body, std::prev(body->end(), 1)), innermostLoop.getLoc());
  FloatType fTy = ...;
  IndexHandle zero(constant_index(0));
  ValueHandle zerof =
      constant_float(llvm::APFloat::getZero(fTy.getFloatSemantics()), fTy);
  IndexHandle r_i(reductionIvs[0]);
  IndexedValue A(getOperand(0)), B(getOperand(1)), C(getOperand(2));
  C() = select(r_i == zero, zerof, *C()) + A(r_i) * B(r_i);
}
```

C++ sugaring with mlir::edsc allows expressing `emitScalarImplementation` directly in indexing notation, given the ordered enclosing loops passed to emitScalarImplementation.
All this can also be written in a more traditional llvm fashion using mlir::FuncBuilder and get/setInsertionPoint.

LinalgBaseOp Property 1: emitScalarImplementation

With this simple property, write a 20 line generic pass that expands any LinalgBaseOp

emitScalarImplementation

```
%c0 = constant 0 : index
%c1 = constant 1 : index
%M = dim %A, 0 : memref<?x?xf32>
%N = dim %C, 1 : memref<?x?xf32>
%K = dim %A, 1 : memref<?x?xf32>
%rM = linalg.range %c0:%M:%c1 :
%rN = linalg.range %c0:%N:%c1 :
%rK = linalg.range %c0:%K:%c1 :
%4 = linalg.view %A[%rM, %rK] :
%6 = linalg.view %B[%rK, %rN] :
%8 = linalg.view %C[%rM, %rN] :
linalg.matmul(%4, %6, %8) :
```

```
func @matmul_as_loops(%arg0: memref<?x?xf32>,
      %arg1: memref<?x?xf32>, %arg2: memref<?x?xf32>) {
  %cst = constant 0.000000e+00 : f32
  %M = dim %arg0, 0 : memref<?x?xf32>
  %N = dim %arg2, 1 : memref<?x?xf32>
  %K = dim %arg0, 1 : memref<?x?xf32>
  affine.for %i0 = 0 to %M {
    affine.for %i1 = 0 to %N {
      affine.for %i2 = 0 to %K {
        %3 = cmpi "eq", %i2, %c0 : index
        %6 = load %arg2[%i3, %i4] : memref<?x?xf32>
        %7 = select %3, %cst, %6 : f32
        %9 = load %arg1[%i2, %i4] : memref<?x?xf32>
        %10 = load %arg0[%i3, %i2] : memref<?x?xf32>
        %11 = mulf %10, %9 : f32
        %12 = addf %7, %11 : f32
        store %12, %arg2[%i3, %i4] : memref<?x?xf32>
```

A generic pass can be written that creates parallel and reduction affine.for operations and call emitScalarImplementation in the scope of the innermost loop.
This emits the IR for matmul_as_loops, nested within the %i2 loop.

LinalgBaseOp Property 2: writeAsFinerGrainTensorContraction

- Ops "*declare*" how to lower themselves
  - As a mix of `affine.for` and `linalg` ("*matching APIs*")
  - Can be interpreted as a "*decreasing potential function*" for lowering
  - Dialect boundaries are not rigid
  - MLIR SSA, verification, etc.. just work on mix of ops from different dialects

Google

Similarly to **`emitScalarImplementation`**, ops also expose a property that can be used by an external transformation to rewrite the op using finer grained op.

LinalgBaseOp Property 2: writeAsFinerGrainTensorContraction

Lowering Between Linalg Ops: Matmul starting point

```
func @matmul(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  %rM = linalg.range %c0:%M:%c1 : !linalg.range
  %rN = linalg.range %c0:%N:%c1 : !linalg.range
  %rK = linalg.range %c0:%K:%c1 : !linalg.range
  %4 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  %6 = linalg.view %B[%rK, %rN] : !linalg.view<?x?xf32>
  %8 = linalg.view %C[%rM, %rN] : !linalg.view<?x?xf32>
  linalg.matmul(%4, %6, %8) : !linalg.view<?x?xf32>
  return
}
```

```
Matmul: C(i, j) = scalarC + A(i, r_k) * B(r_k, j)
Matvec:    C(i) = scalarC + A(i, r_j) * B(r_j)
```

Looking at the index form of linalg::matmul and linalg::matvec makes it easier to determine what this rewriting should do.

LinalgBaseOp Property 2: writeAsFinerGrainTensorContraction

Lowering Between Linalg Ops: Matmul as Matvec

```
func @matmul_as_matvec(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  %rM = linalg.range %c0:%M:%c1 : !linalg.range
  %rK = linalg.range %c0:%N:%c1 : !linalg.range
  %5 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  affine.for %col = 0 to %N {
    %7 = linalg.view %B[%rK, %col] : !linalg.view<?xf32>
    %8 = linalg.view %C[%rM, %col] : !linalg.view<?xf32>
    linalg.matvec(%5, %7, %8) : !linalg.view<?xf32>
  }
  return
}
```

```
Drop "j" from:
Matmul: C(i, j) = scalarC + A(i, r_k) * B(r_k, j)
Matvec:    C(i) = scalarC + A(i, r_j) * B(r_j)
```

"Interchange" due to library impedance mismatch

Looking at the index form of linalg::matmul and linalg::matvec makes it easier to determine what this rewriting should do: it should take slices along loop "j" for B and C and call matvec.
This is specific to the convention we took for implementing the scalar form of matmul and matvec.
Depending on how matmul and matvec are implemented in scalar form, details may change (this is referred to as a "library impedance mismatch").

LinalgBaseOp Property 2: writeAsFinerGrainTensorContraction

```
// In some notional index notation, we have defined:
//   Matmul as: C(i, j) = scalarC + A(i, r_k) * B(r_k, j)
//   Matvec as:    C(i) = scalarC + A(i, r_j) * B(r_j)
// So we must drop the `j` loop from the Matmul.
// Parallel dimensions permute: do it declaratively.
void linalg::MatmulOp:: writeAsFinerGrainTensorContraction()
  auto *op = getOperation();
  ScopedContext scope(FuncBuilder(op), op->getLoc());
  IndexHandle j;
  auto *vA(getInputView(0)), *vB(...), *vC(...);
  Value *range = getViewRootIndexing(vB, 1).first;
  linalg::common::LoopNestRangeBuilder(&j, range)({
      matvec(vA, slice(vB, j, 1), slice(vC, j, 1)),
  });
}
```

Extracting this information with an analysis from transformed and tiled loops would take a lot of effort.
With high-level dialects the problem can be defined away.

This **writeAsFinerGrainTensorContraction** property is thus operation-specific and is written explicitly as part of designing the linalg operations. Using the C++ sugaring with mlir::edsc we can just take the proper slices as determined in the previous slice and the IR is emitted in the proper scope.
All this can also be written in a more traditional llvm fashion using mlir::FuncBuilder and get/setInsertionPoint.

LinalgBaseOp Property 2: writeAsFinerGrainTensorContraction

```
void linalg::lowerToFinerGrainedLinalg(Function *f) {
  f->walk([](Operation *op) {
    if (auto matmulOp = op->dyn_cast<linalg::MatmulOp>())
      matmulOp. writeAsFinerGrainTensorContraction();
    else if (auto matvecOp = op->dyn_cast<linalg::MatvecOp>())
      matvecOp. writeAsFinerGrainTensorContraction();
    else
      return;
    op->erase();
  });
}
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg3/lib/Transforms.cpp

Just like before, f->walk traverses the IR in postorder and allows in-place rewrites and erasure without invalidating iterators.
This is a lower level implementation detail, such a transformation would typically be exposed via an mlir::Pass or am mlir::RewritePattern.

LinalgBaseOp Property 2: writeAsFinerGrainTensorContraction

Lowering Between Linalg Ops: Matmul as Matvec as Dot

```
func @matmul_as_dot(%A:memref<?x?xf32>, %B:memref<?x?xf32>, %C:memref<?x?xf32>) {
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  affine.for %i0 = 0 to %N {
    %3 = linalg.range %c0:%2:%c1 : !linalg.range
    %5 = linalg.view %B[%3, %i0] : !linalg.view<?xf32>
    affine.for %i1 = 0 to %M {
      %7 = linalg.view %A[%i1, %3]:!linalg.view<?xf32>
      %8 = linalg.view %C[%i1, %i0]:!linalg.view<f32>>
      linalg.dot(%7, %5, %8) : !linalg.view<f32>
    }
  }
  return
}
```

"Interchange" due to library impedance mismatch

```
Matmul: C(i, j) = scalarC + A(i, r_k) * B(r_k, j)
Drop "i" from:
Matvec:    C(i) = scalarC + A(i, r_j) * B(r_j)
Dot   :     C() = scalarC +    A(r_i) * B(r_i)
```

Going from matvec to dot proceeds similarly at the next level.
Depending on the ISA supported by a particular target hardware, one can lower all the way to loops or just stop at the right level.

LinalgBaseOp Property 2: writeAsFinerGrainTensorContraction

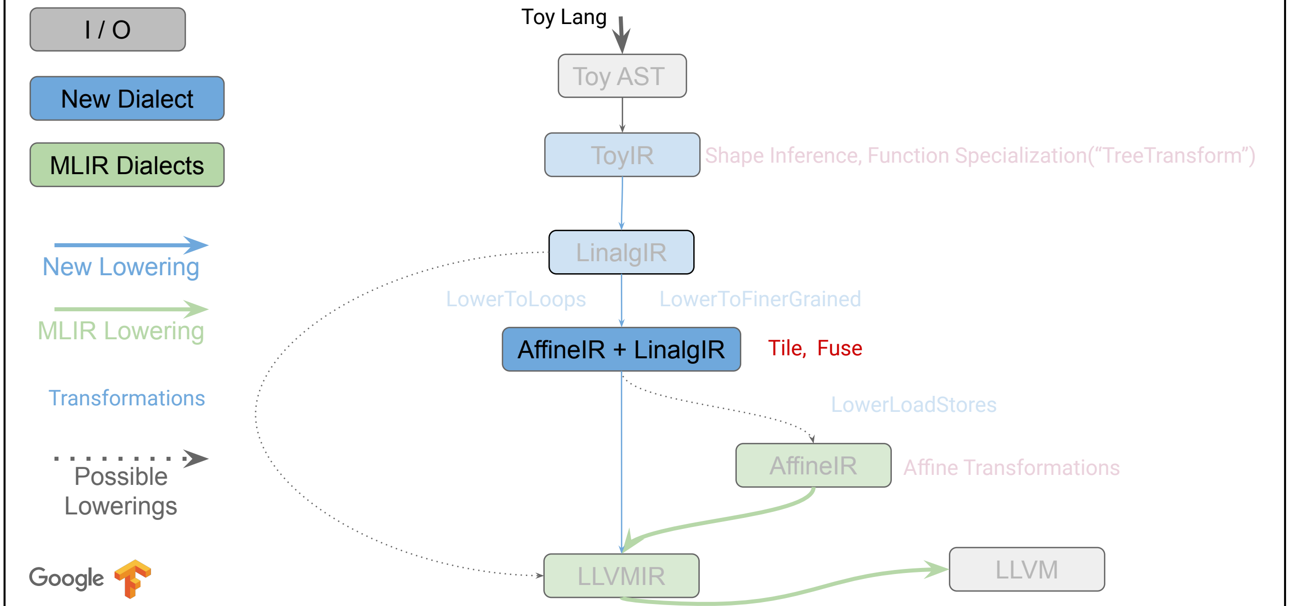
```
// In some notional index notation, we have defined:
//   Matvec as: C(i) = scalarC + A(i, r_j) * B(r_j)
//      Dot as:  C() = scalarC +    A(r_i) * B(r_i)
// So we must drop the `i` loop from the Matvec.
void linalg::Matvec:: writeAsFinerGrainTensorContraction()
  auto *op = getOperation();
  ScopedContext scope(FuncBuilder(op), op->getLoc());
  IndexHandle i;
  auto *vA(getInputView(0)), *vB(...), *vC(...);
  Value *range = getViewRootIndexing(vB, 1).first;
  linalg::common::LoopNestRangeBuilder(&i, range)({
      dot(slice(vA, i, 0), vB, slice(vC, i, 0)),
  });
}
```

Extracting this information with an analysis from transformed and tiled loops would take a lot of effort.
With high-level dialects the problem can be defined away.

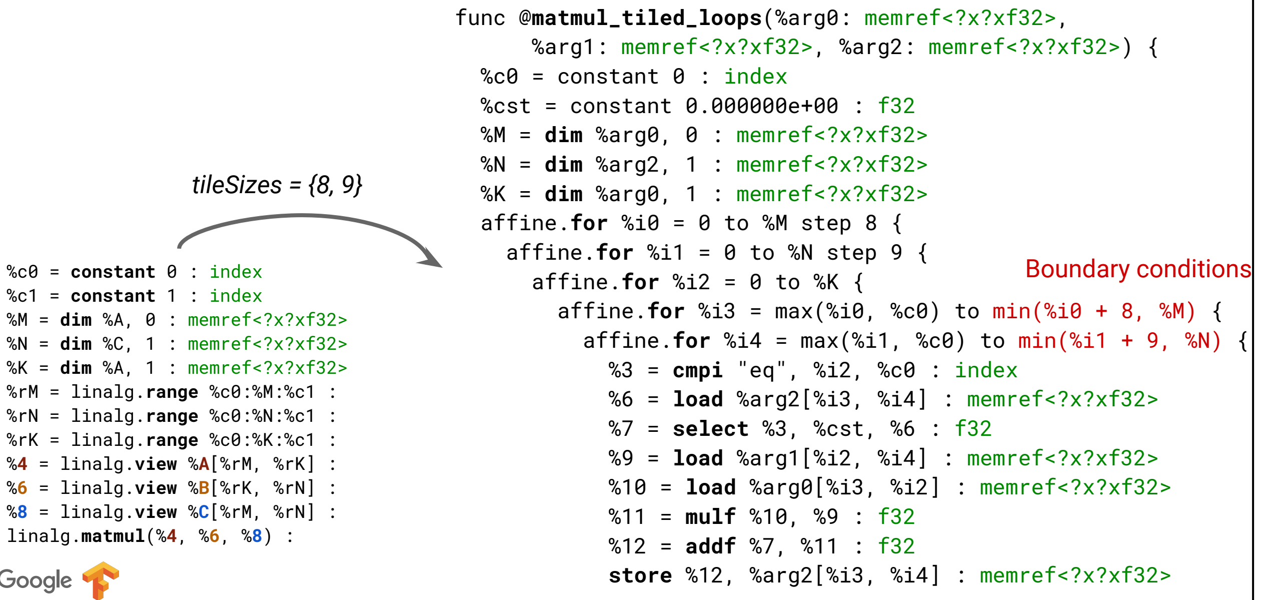
Google

Transformations

Google



Let's now look at how we can implement tiling and op-level fusion in a generic fashion using declarative properties.



Here is what we would lie to achieve.
We have already discussed how linalg.matmul knows to lower itself to a scalar form.
We want to generalize this to “linalg.matmul knows how to lower itself to a tiled scalar form”.
Making this a property of the operation allows the design of specialize transformations that are correct by construction and don’t require complex analyses for legality or complex traversals for application of the transformation.

Loop Tiling Declaration

- An op *“declares”* how to tile itself maximally on loops
  - For LinalgBase this is easy: perfect loop nests
  - Can be tiled declaratively with **mlir::tile**

```
void linalg::lowerToTiledLoops(mlir::Function *f,
                               ArrayRef<uint64_t> tileSizes) {
  f->walk([tileSizes](Operation *op) {
    if (emitTiledLoops(op, tileSizes).hasValue())
      op->erase();
  });
}
```

Works with imperfectly nested + interchanges

```
llvm::Optional<SmallVector<mlir::AffineForOp, 8>>
linalg::emitTiledLoops(Operation *op, ArrayRef<uint64_t> tileSizes) {
  auto loops = emitLoops(op);
  if (loops.hasValue())
    return mlir::tile(*loops, tileSizes, loops->back());
  return llvm::None;
}
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg4/lib/Transforms.cpp

We have conveniently declared operations that, in practice, consist of perfectly nested loop nests.
In the code above, `emitLoops` calls the `emitScalarImplementation` method that we introduced previously.
It also return the emitted loops which allows composition with tiling in a declarative fashion.
`mlir::tile` is a core mlir transformation which lets one tile a perfectly nested loop nest by apply stripmine-and-sink.
The transformation itself is more generally applicable, see the documentation for more details.

View Tiling Declaration

- A LinalgOp *"declares"* how to tile itself with views
  - Step 1: *"declare"* mapping from loop to views `loopsToOperandRangesMap`
  - Step 2: tile loops by *tileSizes*
  - Step 3: apply *mapping* on tiled loops to get tiled views (i.e. sub-views)
  - Step 4: rewrite as tiled loops over sub-views

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg4/lib/Transforms.cpp

We have seen how an op can specify how to write itself in tiled loop form.
Similarly, an op can specify how to rewrite itself in tiled view form.
This is all achieved by having an op declare how its loops map to views:
`loopsToOperandRangesMap`.
Steps 2-4 are then mechanical and generic, only using this property.

LinalgBaseOp Property 3: loopsToOperandRangesMap

- A Linalg *"declares"* how to lower itself to `affine.for`
  - Declare AffineMap attributes of loops to views

```
// Attributes: declare mapping of loop ranges to view ranges.
SmallVector<AffineMap, 4> linalg::MatmulOp::loopsToOperandRangesMap() {
  … // define d0, d1, d2 boilerplate
  // A(M, K), B(K, N), C(M, N):
  //   (d0, d1, d2) -> ((d0, d2), (d2, d1), (d0, d1))
  return SmallVector<AffineMap, 4>{
    AffineMap::get(3, 0, {d0, d2}, {}),  // A(M, K): (d0, d1, d2) -> (d0, d2)
    AffineMap::get(3, 0, {d2, d1}, {}),  // B(K, N): (d0, d1, d2) -> (d2, d1)
    AffineMap::get(3, 0, {d0, d1}, {})}; // C(M, N): (d0, d1, d2) -> (d0, d1)
}
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg4/lib/Transforms.cpp

The property that enable the correspondence between loop tiling and view tiling is an affine map (see affine-map in https://github.com/tensorflow/mlir/blob/master/g3doc/LangRef.md)
Without reproducing deeper details of the doc, we note that affine maps have a few interesting properties:

1. It is a core mlir *type* and is thus static. It allows expressing a static mapping from source to target dimensions.
2. An affine map can be applied to a list of input *values* and returns a list of result *values*. It allows applying the static mapping to dynamic values and express dynamic correspondences and transformations between SSA values.
3. The static type supports simple mathematical "affine" operations, simplifications and canonicalizations.

In other words, an affine map can be thought of as a static std::map with algebraic properties that makes it suitable for tiling compositions and transformations.

View Tiling Declaration

```
func @matmul_tiled_views(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  affine.for %i0 = 0 to %M step 8 {
    affine.for %i1 = 0 to %N step 9 {
      %4 = affine.apply (d0) -> (d0 + 8)(%i0)
      %5 = linalg.range %i0:%4:%c1 : !linalg.range   needs range intersection
      %7 = linalg.range %c0:%K:%c1 : !linalg.range
      %8 = linalg.view %A[%5, %7] : !linalg.view<?x?xf32>
      %10 = linalg.range %c0:%M:%c1 : !linalg.range
      %12 = affine.apply (d0) -> (d0 + 9)(%i1)
      %13 = linalg.range %i1:%12:%c1 : !linalg.range   needs range intersection
      %14 = linalg.view %B[%10, %13] : !linalg.view<?x?xf32>
      %15 = linalg.view %C[%5, %13] : !linalg.view<?x?xf32>
      linalg.matmul(%8, %14, %15) : !linalg.view<?x?xf32>
```

Recursive linalg.**matmul** call

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg4/lib/Transforms.cpp

Simply defining an affine map per operation allows writing a generic transformation that tiles an operation on views.
This gives a recursive form for all operations that conform to the 3 declarative properties we have defined so far.
A recursive form is a powerful rewrite, it allows memory footprint reduction when combined with other transformations (e.g. fusion as we will see next).

TileAndFuseProducerOf

- A LinalgOp *“declares”* how to tile itself with views
  - Step 1: *“declare”* mapping from loop to views
  - Step 2: tile loops by *tileSizes*
  - Step 3: apply *mapping* on tiled loops to get tiled views (i.e. sub-views)
  - Step 4: rewrite as tiled loops over sub-views
  - Step 5: follow SSA use-def chain to find producers of inputs
  - Step 6: clone producer of sub-view in local scope
  - Step 7: cleanup

Loop and View Tiling (Steps 1–4)

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg5/lib/Transforms.cpp

Using the same affine-map property and the duality between loop and view tiling, we can also write a generic “tile and fuse” operation that simultaneously reduces the memory footprint and improves loacality.
The reason we don’t separate fusion from tiling in this tutorial is that we use SSA value equality to determine whether a view produced by an operation is used by another operation.
Had we lost the information we would need to to write a more complex analysis to determine that fusion is possible.

TileAndFuseProducerOf

- 3-D tiling + fusion
  - 2 `linalg.matmul`

```
%c0 = constant 0 : index
%c1 = constant 1 : index
%M = dim %A, 0 : memref<?x?xf32>
%N = dim %C, 1 : memref<?x?xf32>
%K = dim %A, 1 : memref<?x?xf32>
%O = dim %E, 1 : memref<?x?xf32>
%rM = linalg.range %c0:%M:%c1 :
%rN = linalg.range %c0:%N:%c1 :
%rK = linalg.range %c0:%K:%c1 :
%4 = linalg.view %A[%rM, %rK] :
%6 = linalg.view %B[%rK, %rN] :
%8 = linalg.view %C[%rM, %rN] :
%9 = linalg.view %D[%rN, %rO] :
%10 = linalg.view %E[%rM, %rO] :
linalg.matmul(%4, %6, %8) :
linalg.matmul(%8, %9, %10) :
```

tileSizes = {7, 8, 9}

```
%M = dim %A, 0 : memref<?x?xf32>
%N = dim %C, 1 : memref<?x?xf32>
%K = dim %A, 1 : memref<?x?xf32>
%O = dim %E, 1 : memref<?x?xf32>
affine.for %i0 = 0 to %M step 7 {
  affine.for %i1 = 0 to %O step 8 {
    affine.for %i2 = 0 to %N step 9 {
      %5 = affine.apply (d0) -> (d0 + 7)(%i0)
      %6 = linalg.range %i0:%5:%c1 : !linalg.range
      %8 = affine.apply (d0) -> (d0 + 9)(%i2)
      %9 = linalg.range %i2:%8:%c1 : !linalg.range
      %10 = linalg.view %arg2[%6, %9] : !linalg.view<?x?xf32>
      %12 = affine.apply (d0) -> (d0 + 8)(%i1)
      %13 = linalg.range %i1:%12:%c1 : !linalg.range
      %14 = linalg.view %arg3[%9, %13] : !linalg.view<?x?xf32>
      %15 = linalg.view %arg4[%6, %13] : !linalg.view<?x?xf32>
      %17 = linalg.range %c0:%K:%c1 : !linalg.range
      %18 = linalg.view %arg0[%6, %17] : !linalg.view<?x?xf32>
      %20 = linalg.range %c0:%N:%c1 : !linalg.range
      %21 = linalg.view %arg1[%17, %20] : !linalg.view<?x?xf32>
      %22 = linalg.view %arg2[%6, %20] : !linalg.view<?x?xf32>
      linalg.matmul(%18, %21, %22) : !linalg.view<?x?xf32>
      linalg.matmul(%10, %14, %15) : !linalg.view<?x?xf32>
```

https://github.com/tensorflow/mlir/blob/master/examples/Linalg/Linalg5/lib/Transforms.cpp

In this example we only demonstrate that we can perform 3-D tiling and fusion using views and operations.
We decouple the problem of legality and transformation application from the problem of profitability.
This does not suggest such a fusion is a good idea: the tradeoff for fusion is a strict increase in arithmetic complexity.
A transformation that looks at relative costs and benefits of locality, parallelism and recomputation should drive the decisions but is outside of the scope of this tutorial.

Conclusion

Google

MLIR = Low Impedance Mismatch

IR design involves multiple tradeoffs

- Iterative process, constant learning experience

MLIR allows mixing levels of abstraction with non-obvious compounding benefits

- Dialect-to-dialect lowering is easy
- Ops from different dialects can mix in same IR
  - Lowering from “A” to “D” may skip “B” and “C”
- Avoid lowering too early and losing information
  - Help define hard analyses away

No forced IR impedance mismatch

Fresh look at problems

With the benefit of hindsight here are some takeaways.
Impedance mismatch between LLVMIR and programmers gave rise to *many* systems and countless rewrites of similar infrastructure, with varying quality.
MLIR makes this impedance mismatch go away.

Recap

MLIR is a great infrastructure for higher-level compilation

- Gradual and partial lowerings to mixed dialects
  - All the way to LLVMIR and execution
- Reduce impedance mismatch at each level

MLIR provides all the infrastructure to build dialects and transformations

- At each level it is the same LLVM-style infrastructure

Demonstrated this on a Toy language with a linear algebra dialect

- Tutorial available on github

Google

Getting Involved

Google

MLIR is Open Source!

Visit us at github.com/tensorflow/mlir:

- Code, documentation, examples
- Developer mailing list at: mlir@tensorflow.org

Still early days:

- Contributions not accepted yet - still setting up CI, etc.

Google

Thank you to the team!

Questions?

We are hiring!
mlir-hiring@google.com

Google