



# MLIR Primer:

## A Compiler Infrastructure for the End of Moore's Law

Compilers for Machine Learning Workshop, CGO 2019

Chris Lattner  
clattner@google.com

Jacques Pienaar  
jpienaar@google.com



Presenting the work of many, many, people!

# TensorFlow

Huge machine learning community

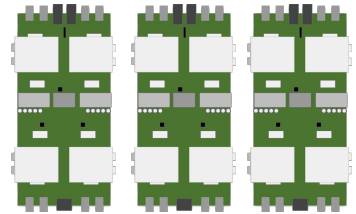
Programming APIs for many languages

Abstraction layer for accelerators:

- Heterogenous, distributed, mobile, custom ASICs...
- Urgency is driven by the “end of Moore’s law”

Open Source:

<https://tensorflow.org>



TensorFlow is a lot of things to different people, but we are here to talk about compilers.

TensorFlow is a very general system, and our work is a key part of TensorFlow future, so we cannot take simplifying assumptions - we have to be able to support the full generality of the tensor problem.

# Why a new compiler infrastructure?



We have LLVM and many other great infras, why do we need something new?  
Let's take a short detour and talk about the state of the broader LLVM compiler ecosystem.

# The LLVM Ecosystem: Clang Compiler



Green boxes are SSA IRs:

- Different levels of abstraction - operations and types are different
- Abstraction-specific optimization at both levels

Progressive lowering:

- Simpler lowering, reuse across other front/back ends

Google 

Clang follows a classic “by the book” textbook design. Oversimplifying the story here, clang has a language-specific AST, generates LLVM IR. LLVM does a lot of optimizations, then lowers to a machine level IR for code generation.

# Azul Falcon JVM



Uses LLVM IR for high level domain specific optimization:

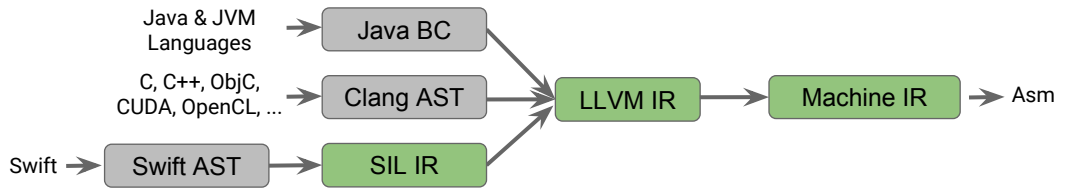
- Encodes information in lots of ways: IR Metadata, well known functions, intrinsics, ...
- Reuses LLVM infrastructure: pass manager, passes like inliner, etc.



["Falcon: An Optimizing Java JIT"](#) - LLVM Developer Meeting Oct'2017

The Azul JIT is incredibly clever in the ways it [ab]uses LLVM. This works well for them, but very complicated and really stretches the limits of what LLVM can do.

# Swift Compiler



3-address SSA IR with Swift-specific operations and types:

- Domain specific optimizations: generic specialization, devirt, ref count optzns, library-specific optzns, etc
- Dataflow driven type checking passes: e.g. definitive initialization, "static analysis" checks
- Progressive lowering makes each edge simpler!

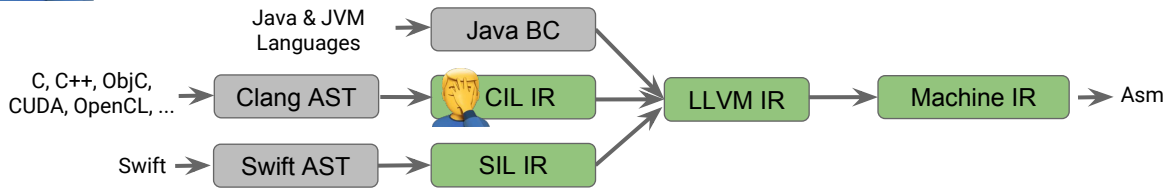


["Swift's High-Level IR"](#) - LLVM Developer Meeting Oct'2015

Swift has higher level abstractions than Java and requires data-flow specific type checking (which relies on 'perfect' location information). As such, we came up with SIL, which is similar to LLVM IR but has Swift specific operations and types. This makes it easy to do domain specific optimization, library specific optimizations and lots of other things.



## A sad aside: Clang should have a CIL!



3-address SSA IR with **Clang**-specific operations and types:

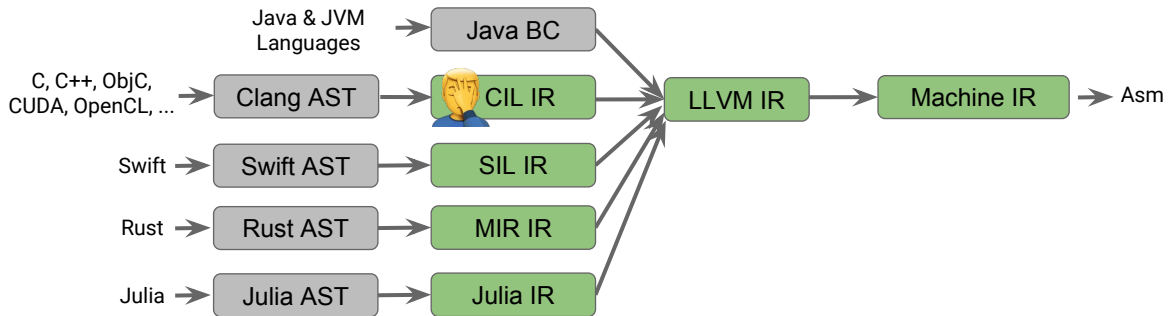
- Optimizations for `std::vector`, `std::shared_ptr`, `std::string`, ...
- Better IR for Clang Static Analyzer
- Dataflow sensitive source tooling



*Anyway, back to the talk...*

With the benefit of experience, we should have built Clang this way too, with a high level IR. Unfortunately now this is probably not going to happen as is, because a team has to build a complex mid-level SSA based optimization infra *and* know enough to reimplement Clang's IRGen. These are reasonably different skillsets and enough work that it has never happened despite the wins.

## Rust and Julia have things similar to SIL



- Dataflow driven type checking - e.g. borrow checker
- Domain specific optimizations, progressive lowering

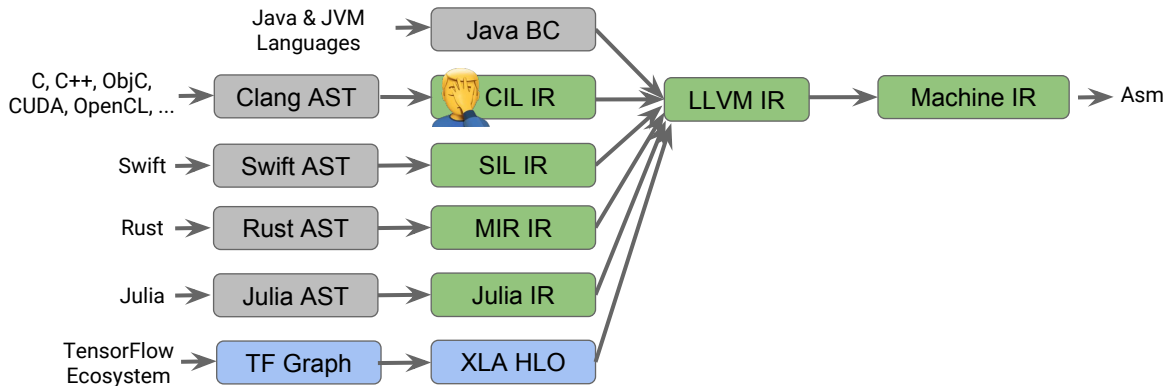


“[Introducing MIR](#)”: Rust Language Blog, “[Julia SSA-form IR](#)”: Julia docs

Swift isn't alone here, many modern high level languages are doing the same thing.



# TensorFlow XLA Compiler



- Domain specific optimizations, progressive lowering



“XLA Overview”: <https://tensorflow.org/xla/overview> (video overview)

Many frameworks in the machine learning world are targeting LLVM. They are effectively defining higher level IRs in the tensor domain, and lowering to LLVM for CPUs and GPUs. This is structurally the same thing as any other language frontend.

Blue boxes are ML “graph” IRs

## Domain Specific SSA-based IRs

Great!

- High-level domain-specific optimizations
- Progressive lowering encourages reuse between levels
- Great location tracking enables flow-sensitive “type checking”

Not great!

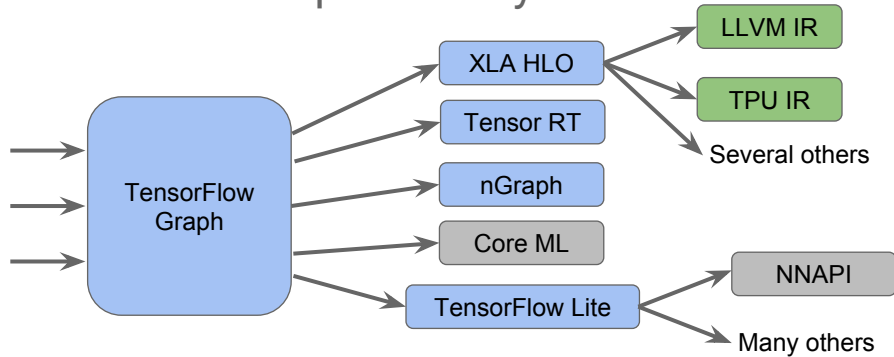
- Huge expense to build this infrastructure
- Reimplementation of all the same stuff:
  - pass managers, location tracking, use-def chains, inlining, constant folding, CSE, testing tools, ....
- Innovations in one community don't benefit the others



Let's summarize the situation here.

Type checking can be things in Swift like definitive initialization, things in Rust like affine types, or things like shape checking in an ML framework.

## The TensorFlow compiler ecosystem



Many "Graph IRs", each with challenges:

- Similar-but-different proprietary technologies: not going away anytime soon
- Fragile, poor UI when failures happen: e.g. poor/no location info, or even crashes
- Duplication of infrastructure at all levels

Google 

Coming back to the challenges we face on the TensorFlow team, I actually fibbed - the world is a lot more complicated than what was described. TensorFlow has a broad collection of graph based IRs, infrastructure for mapping back and forth between them, and very little code reuse across any of these ecosystems.

## Goal: Global improvements to TensorFlow infrastructure

SSA-based designs to generalize and improve ML “graphs”:

- Better side effect modeling and control flow representation
- Improve generality of the lowering passes
- Dramatically increase code reuse
- Fix location tracking and other pervasive issues for better QoI

No reasonable existing answers!

- ... and we refuse to copy and paste SSA-based optimizers 6 more times!



Our team is looking at making across the board improvements to this situation, but there is no good existing solution.

What is a team to do?

# Quick Tour of MLIR: Multi-Level IR

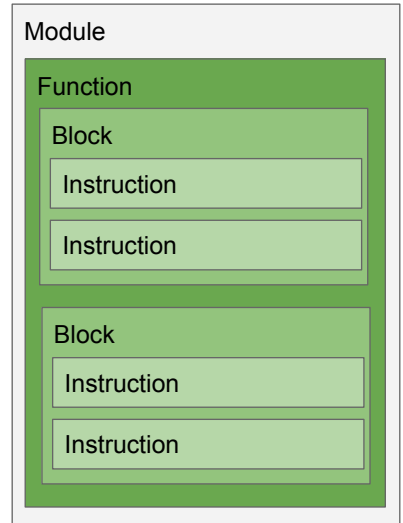


This brings us to MLIR. “ML” expands in multiple ways, principally “Multi-Level”, but also Mid Level, Machine Learning, Multidimensional Loop, and I’m sure we’ll find other clever expansions in the future.

## Many similarities to LLVM

- SSA, typed, three address
- Module/Function/Block/Instruction structure
- Round trippable textual form
- Syntactically similar:

```
func @testFunction(%arg0: i32) {  
  %x = call @thingToCall(%arg0) : (i32) -> i32  
  br ^bb1  
^bb1:  
  %y = addi %x, %x : i32  
  return %y : i32  
}
```



MLIR is highly influenced by LLVM and takes many great ideas unabashedly from it.

## MLIR Type System: some examples

### Scalars:

- f16, bf16, f32, ... i1, i8, i16, i32, ... i3, i4, i7, i57, ...

### Vectors:

- vector<4 x f32>, vector<4x4 x f16>, etc.

### Tensors, including dynamic shape and rank:

- tensor<4x4 x f32>
- tensor<4x?x?x17x? x f32>    tensor<\* x f32>

### Others:

- functions, memory buffers, quantized integers, other TensorFlow stuff, ...

Google 

MLIR has a flexible type system, but here are some examples to give you a sense of what it can do. It has rich support for modeling the tensor domain, including dynamic shapes and ranks, since that is a key part of TensorFlow.

# MLIR Instructions: an open ecosystem

No fixed / builtin list of globally known operations:

- No “instruction” vs “target-indep intrinsic” vs “target-dep intrinsic” distinction
  - Why is “add” an instruction but “add with overflow” an intrinsic in LLVM? 🐱

Passes are expected to conservatively handle unknown instructions:

- just like LLVM does with unknown intrinsics

```
func @testFunction(%arg0: i32) -> i32 {  
  %x = "any_unknown_operation_here"(%arg0, %arg0) : (i32, i32) -> i32  
  %y = "my_increment"(%x) : (i32) -> i32  
  return %y : i32  
}
```



An open ecosystem is the biggest difference from LLVM - in MLIR you can define your own operations and abstractions in the IR, suitable for the domain of problems you are trying to solve. It is *more* of a pure compiler infrastructure than LLVM is.




# MLIR Instructions Capabilities

Instructions always have: opcode and source location info

Instructions may have:

- Arbitrary number of SSA results and operands
- Attributes: guaranteed constant values
- Block arguments: e.g. for branch instructions
- Regions: discussed in later slide
- Custom printing/parsing - or use the more verbose generic syntax

```
%2 = dim %1, 1 : tensor<1024x? x f32>
```

 Dimension to extract is guaranteed integer constant, an "attribute"

```
%x = alloc() : memref<1024x64 x f32>
```

```
%y = load %x[%a, %b] : memref<1024x64 x f32>
```

Google 

So what can an instruction do? They always have an opcode and always have location info (!!).

One thing to note is that operations can customize their printing, so you'll see specialized printing for common ops like in this slide, and the default generic printer that uses double quotes.

# Complicated TensorFlow Example

```
func @foo(%arg0: tensor<8x?x?x8xf32>, %arg1: tensor<8xf32>,
         %arg2: tensor<8xf32>, %arg3: tensor<8xf32>, %arg4: tensor<8xf32>) {

  %0 = "tf.FusedBatchNorm"(%arg0, %arg1, %arg2, %arg3, %arg4)
    {data_format: "NHWC", epsilon: 0.001, is_training: false}
    : (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)
    -> (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)

  "use"(%0#2, %0#4 ...
```



To see what operations can do, let's look at a more complicated example from TensorFlow, a fused batch norm.

## Complicated TensorFlow Example: Inputs

```
func @foo(%arg0: tensor<8x?x?x8xf32>, %arg1: tensor<8xf32>,  
         %arg2: tensor<8xf32>, %arg3: tensor<8xf32>, %arg4: tensor<8xf32>) {  
  
  %0 = "tf.FusedBatchNorm"(%arg0, %arg1, %arg2, %arg3, %arg4)  
      {data_format: "NHWC", epsilon: 0.001, is_training: false}  
      : (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)  
      -> (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)  
  
  "use"(%0#2, %0#4 ...
```

→ Input SSA values and corresponding type info



As in LLVM, SSA values have types. Here there are 5 inputs and their types.

# Complicated TensorFlow Example: Results

```
func @foo(%arg0: tensor<8x?x?x8xf32>, %arg1: tensor<8xf32>,
         %arg2: tensor<8xf32>, %arg3: tensor<8xf32>, %arg4: tensor<8xf32>) {

  %0 = "tf.FusedBatchNorm"(%arg0, %arg1, %arg2, %arg3, %arg4)
    {data_format: "NHWC", epsilon: 0.001, is_training: false}
    : (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)
    -> (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)

  "use"(%0#2, %0#4) ...
```

- This op produces five results
- Each result can be used independently with # syntax
- No “tuple extracts” get in the way of transformations



This op has 5 results as well, and multiple results can be directly referenced. This makes analyses and transformations easier to write.

# Complicated TensorFlow Example: Attributes

```
func @foo(%arg0: tensor<8x?x?x8xf32>, %arg1: tensor<8xf32>,
         %arg2: tensor<8xf32>, %arg3: tensor<8xf32>, %arg4: tensor<8xf32>) {

  %0 = "tf.FusedBatchNorm"(%arg0, %arg1, %arg2, %arg3, %arg4)
    {data_format: "NHWC", epsilon: 0.001, is_training: false}
    : (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)
    -> (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)

  "use"(%0#2, %0#4 ...
```

- Named attributes
- "NHWC" is a constant, static entity, not an SSA value



Instruction can have a named dictionary of known-constant attribute values, used for things like the strides of a convolution, or the immediate value in a “load immediate” machine instruction.

## Extensible Operations Allow Multi-Level IR

TensorFlow		<pre>%x = "tf.Conv2d"(%input, %filter)       {strides: [1,1,2,1], padding: "SAME", dilations: [2,1,1,1]}       : (tensor&lt;*xf32&gt;, tensor&lt;*xf32&gt;) -&gt; tensor&lt;*xf32&gt;</pre>
XLA HLO		<pre>%m = "xla.AllToAll"(%z)       {split_dimension: 1, concat_dimension: 0, split_count: 2}       : (memref&lt;300x200x32xf32&gt;) -&gt; memref&lt;600x100x32xf32&gt;</pre>
LLVM IR		<pre>%f = "llvm.add"(%a, %b)       : (f32, f32) -&gt; f32</pre>

Also: TF-Lite, Core ML, other frontends, etc ...



Don't we end up with the JSON of compiler IRs????

Because of this flexible system, you can represent things at many different levels of abstraction, giving rise to the Multi-Level part of MLIR.

But doesn't this mean that everything is stringly typed? doesn't this mean that all transformations have to use magic numbers like `getOperand(4)`? Doesn't this mean that everything has to be written defensively to handle malformed IR?

In fact - no!

## MLIR “Dialects”: Families of defined operations

Example Dialects:

- TensorFlow, LLVM IR, XLA HLO, TF Lite, Swift SIL...

Dialects can define:

- Sets of defined operations
- Entirely custom type system

Operation can define:

- Invariants on # operands, results, attributes, etc
- Custom parser, printer, verifier, ...
- Constant folding, canonicalization patterns, ...



MLIR solves this by allowing defined operations, which have invariants placed on them - things like “this is a binary operator, the inputs and output has the same types”. This allows generation of verification and accessors, which give typed access to the operation.

Dialects can also define entirely custom types, which is how MLIR can model things like the LLVM IR type system (which has first class aggregates), the Swift type system (completely tied around Swift decl nodes), Clang in the future, and lots of other domain abstractions.

## Nested Regions

```
%2 = xla.fusion (%0 : tensor<f32>, %1 : tensor<f32>) : tensor<f32> {  
^bb0(%a0 : tensor<f32>, %a1 : tensor<f32>):  
  %x0 = xla.add %a0, %a1 : tensor<f32>  
  %x1 = xla.relu %x0 : tensor<f32>  
  return %x1  
}  
  
%7 = tf.If(%arg0 : tensor<i1>, %arg1 : tensor<2xf32>) -> tensor<2xf32> {  
  ... "then" code...  
  return ...  
} else {  
  ... "else" code...  
  return ...  
}
```

→ Functional control flow, XLA fusion node, closures/lambda, parallelism abstractions like OpenMP, etc.

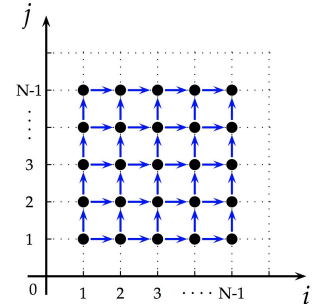


One of the other really important things of MLIR instructions is the ability to have nested regions of code in an instruction. This allows representation of “functional loops” in TensorFlow and XLO, parallelism abstractions like OpenMP, closures in source languages like Swift, etc. This makes analyses and optimizations on these much more powerful because they are suddenly intraprocedural instead of interprocedural, and code within the region can directly refer to dominating SSA values in the enclosing code.



# Bigger Example: Polyhedral IR Dialect

```
func @matmul_square(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {  
  %n = dim %A, 0 : memref<?x?xf32>  
  
  affine.for %i = 0 to %n {  
    affine.for %j = 0 to %n {  
      store 0, %C[%i, %j] : memref<?x?xf32>  
      affine.for %k = 0 to %n {  
        %a = load %A[%i, %k] : memref<?x?xf32>  
        %b = load %B[%k, %j] : memref<?x?xf32>  
        %prod = mulf %a, %b : f32  
        %c = load %C[%i, %j] : memref<?x?xf32>  
        %sum = addf %c, %prod : f32  
        store %sum, %C[%i, %j] : memref<?x?xf32>  
      }  
    }  
  }  
  return  
}
```



**affine.for** and **affine.if** represent polyhedral schedule trees:

- Polyhedral is a great match for ML kernels
- Includes systems of affine constraints, mappings, etc



One of the ways we use this today is in our polyhedral codegen framework. We don't have time to talk about this today, but we have an innovative approach combining the advantages of structured polyhedral schedule trees with SSA. We do not use ISL, ILP solvers, or exponential time polyhedral code generation like existing research systems.

We are a production system that must scale to hundreds of thousands of statements in polyhedral regions.

## MLIR vs LLVM: "Bugs" Fixed

- Constants can't trap!
- Robust source location tracking!
- Dialect-defined structured metadata!
- Block arguments instead of PHI nodes!
- SSA use-def chains allow multithreading the compiler!
- etc. :-)



Of course, given the chance to build a new infra, we learned a lot of existing systems and the mistakes we've had to live with for a long time, and fixed them. Notably, we've designed the compiler to support multithreaded compilation, because 100 hardware threads in a modern system is not unusual, and they will continue to grow.

# MLIR: Infrastructure



Next up, Jacques will take this high level of this system, dive deeper into some of the “how” it works, and give concrete examples.

## Op definition: TensorFlow LeakyRelu

- Specified using TableGen
  - LLVM Data modelling language
- Dialect can create own hierarchies
  - "tf.LeakyRelu" is a "TensorFlow unary op"
- Specify op properties (open ended)
  - Side-effect free
  - "tf.Add" has broadcasting behavior
  - "pt.add" has (scalar, tensor) input
- Name input and output operands
  - Named accessors created
- Document along with the op
- Define optimization & semantics

```
def TF_LeakyReluOp : TF_UnaryOp<"LeakyRelu",
  [NoSideEffect, SameValueType]>,
  Results<(outs TF_Tensor:$output)>
{
  let arguments = (ins
    TF_FloatTensor:$value,
    DefaultValuedAttr<F32Attr, "0.2">:$alpha
  );

  let summary = "Leaky ReLU operator";
  let description = [{
    The Leaky ReLU operation takes a tensor and returns
    a new tensor element-wise as follows:
    LeakyRelu(x) = x      if x >= 0
                  = alpha*x  else
  }];

  let constantFolding = ...;
  let canonicalizer = ...;
  let referenceImplementation = ...;
}
```



MLIR has an open op eco system so there is no need to define ops. But op definitions adds structure. The op definitions provide a central place (per dialect) to define ops. It allows us to define invariants/requirements, properties, attributes, textual format, documentation, reference implementation, ... of an operation. Serving as a single source of truth for the operation.

## Generated from op definition

- C++ class TF::LeakyReluOp
  - Accessors (value() and alpha())
  - Builder methods
    - create<TF::LeakyReluOp>(loc, ...)
  - Verify function
    - Verify number of operands, type of operands, compatibility of operands
    - Write transforms for legal ops!
- Documentation
- Serialization/translate methods
  - Ops interact with different backends
  - Finally need to generate new graphs or code

```
namespace TF {
class LeakyReluOp
  : public Op<LeakyReluOp,
             OpTrait::OneResult,
             OpTrait::HasNoSideEffect,
             OpTrait::SameOperandsAndResultType,
             OpTrait::OneOperand> {
public:
  static StringRef getOperationName() {
    return "tf.LeakyRelu";
  };
  Value* value() { ... }
  APFloat alpha() const { ... }
  static void build(...) { ... }
  bool verify() const {
    if (...) return emitOpError(
      "requires 32-bit float attribute 'alpha'");
    return false;
  }
};
} // end namespace
```



From the op definition we can generate multiple different representation. Op definition serves multiple different purposes, among which

- removes unintuitive accessor methods (one can get the stride with stride() rather than GetOperand(3)),
- builder method generation (using source location information),
- this gets rid of scattered/repeated/non-existent op verification methods and instead gathers it all together.

Beyond the C++ we also generate documentation as well serialization/translation methods to interact with external systems from the op definition.

## Specify simple patterns simply

```
def : Pat<(TF_SqueezeOp StaticShapeTensor:$arg), ( TFL_ReshapeOp $arg)>;
```

- Support M-N patterns
- Support constraints on Operations, Operands and Attributes
- Support specifying dynamic predicates
  - Similar to advocated in "Fast and Flexible Instruction Selection With Constraints", CC18
- Support native C++ code rewrites
  - Always a long tail, don't make the common case hard for the tail!

Goal: Declarative, reduces boilerplate, easy to express for all



Now we have ops, now we want to transform graphs of op. There are multiple different graph optimizations folks want to apply. Particularly common is rewrite patterns. In MLIR we want to make it simple to specify simple patterns simply. Transforms can be specified using simple DAG-to-DAG patterns. MLIR supports M-N patterns, constraints on the operations, operands and attributes, support specifying dynamic predicates on when to match a rule as well as allow native C++ code rewrites. These patterns are declarative and aims to reduces the boiler plate and make transforms easy to express.

## General function/graph transformations

- Additionally module/function passes, function passes, utility matching functions, nested loop matchers ...

```
struct Vectorize : public FunctionPass {
    Vectorize() : FunctionPass(&Vectorize::passID) {}

    PassResult runOnFunction(Function *f) override;

    static char *passID;
};
```

```
f->walk([&](Instruction *!inst) {
    foldInstruction(inst);
});
```

```
...
if (matchPattern(getOperand(1), m_Zero()))
    return getOperand(0);
...
```



Pattern rewrites are not the entire world of graph transformations. We support all the standard things you'd expect, including a pass manager, the ability to walk code ergonomically, and pattern matchers similar to LLVM's.

# Location tracking

API requires location information on each operation:

- File/line/column, op fusion, op fission
- “Unknown” is *allowed*, but discouraged and must be explicit.

Integral to our test suite!

```
// RUN: mlir-opt %s -memref-dependence-check -split-input-file -verify
...
%0 = alloc() : memref<100xf32>
for %i0 = 0 to 10 {
  %1 = load %0[%i0] : memref<100xf32>
  // ...
  // expected-note@-2 {{dependence from 0 to 1 at depth 2 = true}}
  store %1, %0[%i0] : memref<100xf32>
}
...
```

Location tracking is fundamentally baked into MLIR, and the API for creating and transforming operations requires source locations (unlike in LLVM, where they get implicitly dropped). There are multiple types of location in MLIR that is used to improve debuggability, traceability and the user experience in general. It is also integral to our test suite. For example here we show how the dependency pass reports a dependency between the load, which was assigned id 0, and store, id 1, within the same iteration of the loop, by adding a note at the location of the instruction.

Design for testability is a key part of our design, and we take it further than LLVM did.



# mlir-opt

- Similar to LLVM's opt a tool for testing compiler passes
- Every compiler transformation is unit testable:
  - Including verification logic, without dependence on earlier passes
  - Policy: every behavior changing commit includes a test case
  - E.g., loop unrolling pass test

```
func @loop_nest_simplest() {  
  // CHECK: affine.for %i0 = 0 to 100 step 2 {  
  affine.for %i = 0 to 100 step 2 {  
    // CHECK: %c1_i32 = constant 1 : i32  
    // CHECK-NEXT: %c1_i32_0 = constant 1 : i32  
    // CHECK-NEXT: %c1_i32_1 = constant 1 : i32  
    // CHECK-NEXT: %c1_i32_2 = constant 1 : i32  
    affine.for %j = 0 to 4 {  
      %x = constant 1 : i32  
    }  
  }  
  }  
  return  
}
```



mlir-opt works the same way as llvm-opt, and we use FileCheck in the same way.

## mlir-translate

- mlir-translate transforms MLIR  $\rightleftharpoons$  external format
- All the previous is transformations within MLIR
  - Progressive lowering of ops within same IR!
- Also need to
  - Transform from/back for different backends (TensorFlow, TensorFlow Lite, XLA, ...)
  - Generate code
- Decouple function/graph transformations from data transformation
  - Principle: Keep data transformations simple/direct/trivially testable & correct
  - $\rightsquigarrow$  Target dialect represents external target closely
- But what about codegen ... ?



We are interoperating with a lot of proprietary systems and building translators between many different foreign representations. We've seen many different translators which make representational lowering changes at the same time as making data structure changes. We want to be able to test our lowering, and MLIR was designed to support this testability, but many foreign systems were not designed with this in mind - we don't want to be diffing protobufs, for example.

The solution to this is to do all lowering within LLVM to a dialect that matches the foreign system as closely as possible (ideally completely isomorphic) and make the actual data-structure translation as trivial as possible. This allows us to write great tests for all the lowering logic and makes the translation more trivially correct by construction.

# LLVM dialect for codegen

- Represent LLVM IR as MLIR dialect
  - Represent LLVM types without duplicating all type definitions

```
!llvm<"{ i32, double, i32 }">
```
  - Simple translation and codegen invocation  
MLIR ops -> MLIR LLVM dialect -> LLVM IR

Function lowered to LLVM dialect



```
...
^bb2: // pred: ^bb1
  %9 = "llvm.constant"() {value: 10 : index} :
    () -> !llvm<"i64">
  %11 = "llvm.mul"(%2, %9) :
    (!llvm<"i64">, !llvm<"i64">) -> !llvm<"i64">
  %12 = "llvm.add"(%11, %6) :
    (!llvm<"i64">, !llvm<"i64">) -> !llvm<"i64">
  %13 = "llvm.extractvalue"(%arg2) {position: [0]} :
    (!llvm<"{ float* }">) -> !llvm<"float">
  %14 = "llvm.getelementptr"(%13, %12) :
    (!llvm<"float*">, !llvm<"i64">) -> !llvm<"float">
  "llvm.store"(%8, %14) :
    (!llvm<"float">, !llvm<"float*">) -> ()
...
```

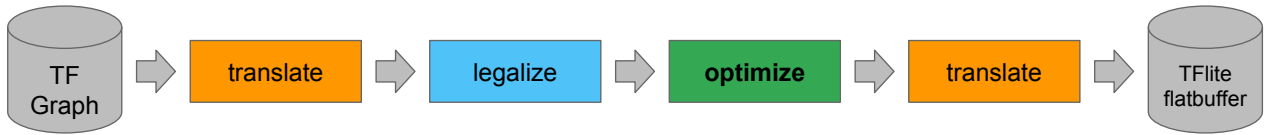
Of course, LLVM is great for C level optimization and code generation to CPUs and PTX, and as such we have an LLVM IR dialect in MLIR that we lower to, which is isomorphic to LLVM IR. This is only used as a lowering step, we don't expect people to be reimplementing existing LLVM IR optimizations on this representation (there is no point, LLVM is a good thing for what it does!!)

# Applications to TensorFlow ecosystem



TensorFlow is moving to MLIR for its core infrastructure, let's talk about a couple of the projects that are in the works.

# TensorFlow Lite Translator



- TensorFlow to TensorFlow Lite translator
  - Two different graph representations
    - Different set of ops & types
  - Different constraints/targets
- Overlapping goals with regular compilation
  - Edge devices also can have accelerators (or a multitude of them!)
  - Same lowering path, expressed as rewrite patterns
- MLIR's pluggable type system simplifying transforms & expressibility
  - Quantized types is a first class citizen in dialect

Google 

TensorFlow Lite is another graph representation with a different interpreter. The TensorFlow Lite Translator is a mini compiler that does a number of compiler passes.

Moving it to MLIR makes it easier to test and develop, and the user experience is dramatically improved due to the location tracking in MLIR.

## TF/XLA bridge

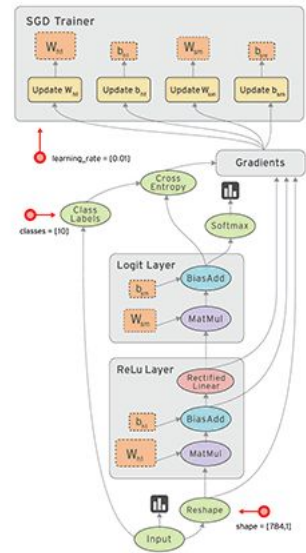
- Interop between TensorFlow and XLA
  - Consists of rewrite passes and transformation to XLA
- Large part is expanding subset of TensorFlow ops to XLA HLO
  - Many 1-M patterns
  - Simple to express as DAG-to-DAG patterns
- XLA targets from multi-node machines to edge devices
  - Not as distinct from TensorFlow Lite



Another project we are working on is to rework the integration of XLA into TensorFlow, rebuilding the lowering infrastructure that converts from TensorFlow graphs to XLA HLO.

# TensorFlow

- Unify graph optimization frameworks
  - Collaboration with Grappler team
  - Unify, extend and improve TensorFlow graph optimizations
- TensorFlow has multiple backends (XLA, TF Lite, nGraph, TensorRT, Core ML, ...) but duplicate integration paths:
  - Unifying integration paths (less code, maintenance burden, overhead for backend teams, ...)
  - Focus on core contributions



XLA isn't the only compiler in the TensorFlow ecosystem, we expect to use the same infra to support Tensor-RT, nGraph, etc.

# Future Directions



While we've built a number of things, we think that there are a lot more future directions to explore than we will be able to tackle.



## Hackability & HW/SW Research

Aiming for a super-extensible system, catalyzing next-gen accelerator research:

- domain-specific languages / annotations lower naturally to MLIR
- domain-specific HW constructs are first-class operations
- extend type system: support novel numerics, sparse types, trees (?)
- many classes of transformations have *structured* search spaces: algorithmic rewriting, graph rewriting, memory-recompute, polyhedral, and synthesis

Accelerate innovation in hardware, compiler algorithms, and applications thereof

We expect MLIR to have a long life, so we are investing heavily in getting the base infrastructure right, we expect and hope that this will catalyze the next generation of compiler research, including novel high level abstractions, parallelism constructs, etc.

# Applying machine learning to compilers

- Move past handwritten heuristics:
  - NP complete problems (full time employment theorem?)
  - Cost models that are hard or infeasible to characterize
  - Hardware explosion, model diversity, problem diversity, ... can't scale
- Autotuning, search and caching FTW
  - Separate algorithms and policy
  - Exploit structure in search space



Compilers are full of NP-complete problems and difficult/impossible to characterize problems - how can we characterize the performance of the code generated by the nvcc compiler? The natural way to handle these are to go with simple heuristics (e.g. greedy algorithms) which are not optimal, or do more exhaustive searches, which are slow for compilation. By offloading this to an offline service, you can use expensive search algorithms (e.g. RL, generic algorithms, brute force, ...) to get great results, without impacting interactive turnaround time.

## Open research topics

- Build a CIL for Clang! 🙌<sup>AST\*</sup>
- Use MLIR as an AST?
  - or build an AST-equivalent of MLIR?
- New code generation / lowering strategies
- New concurrency constructs



A lot of interesting research topics that we want to explore, from building a CIL for Clang (volunteers?!?), MLIR as an AST, AST equivalent of MLIR, new code generation/lowering strategy, new concurrency constructs.

## Open research topics

- Build a CIL for Clang! 🙌
- Use MLIR as an AST?
  - or build an AST-equivalent of MLIR?
- New code generation / lowering strategies
- New concurrency constructs

Open source soon!

We expect to open source MLIR this spring!

Thank you to the team!

Alex Cohen  
Alex Zaretskiy

Jacques Perrier  
James Murray

River Riddle  
Sergey Dav

We are hiring!  
[mlir-hiring@google.com](mailto:mlir-hiring@google.com)

Carole W. Zeng  
Dimitris Nikolakis  
Feng Li  
Hiroshi Nakajima

Ilse Isenhardt  
Pavel Laskov  
Rasmus Larsson  
Richard Wei

Mysore Sathyanarayanan  
Yanan Cao

This is work of large group of folks that I'd like the opportunity to thank.

One last note: we are hiring, so reach out!