MLIR: Multi-Level Intermediate Representation Compiler Infrastructure

2019 European LLVM Developers Meeting

Tatiana Shpeisman
shpeisman@google.com

Chris Lattner
clattner@google.com

Presenting the work of many, many, people!

TensorFlow

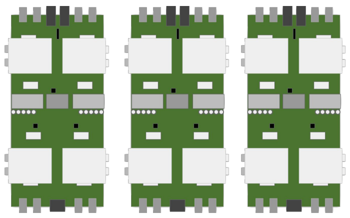
Huge machine learning community

Programming APIs for many languages

Abstraction layer for accelerators:
- Heterogenous, distributed, mobile, custom ASICs…
- Urgency driven by the “end of Moore’s law”

Open Source:

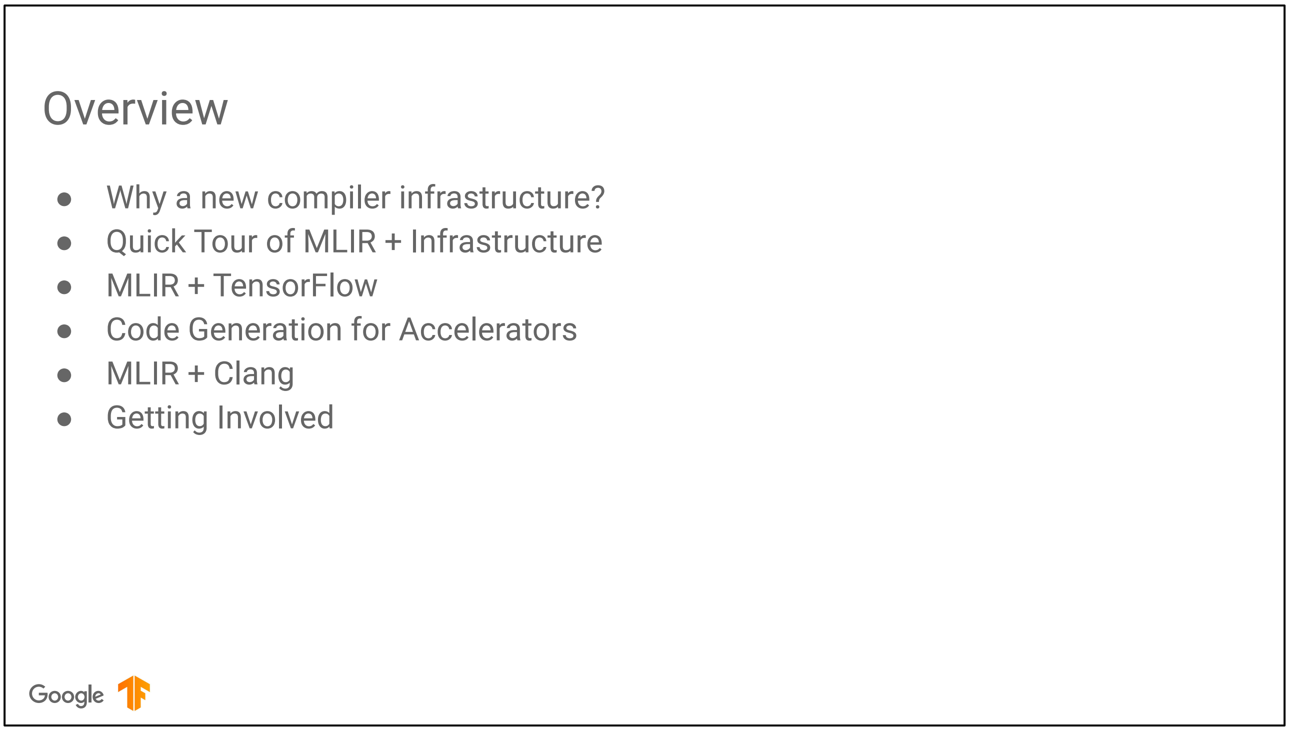
https://tensorflow.org



Google

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

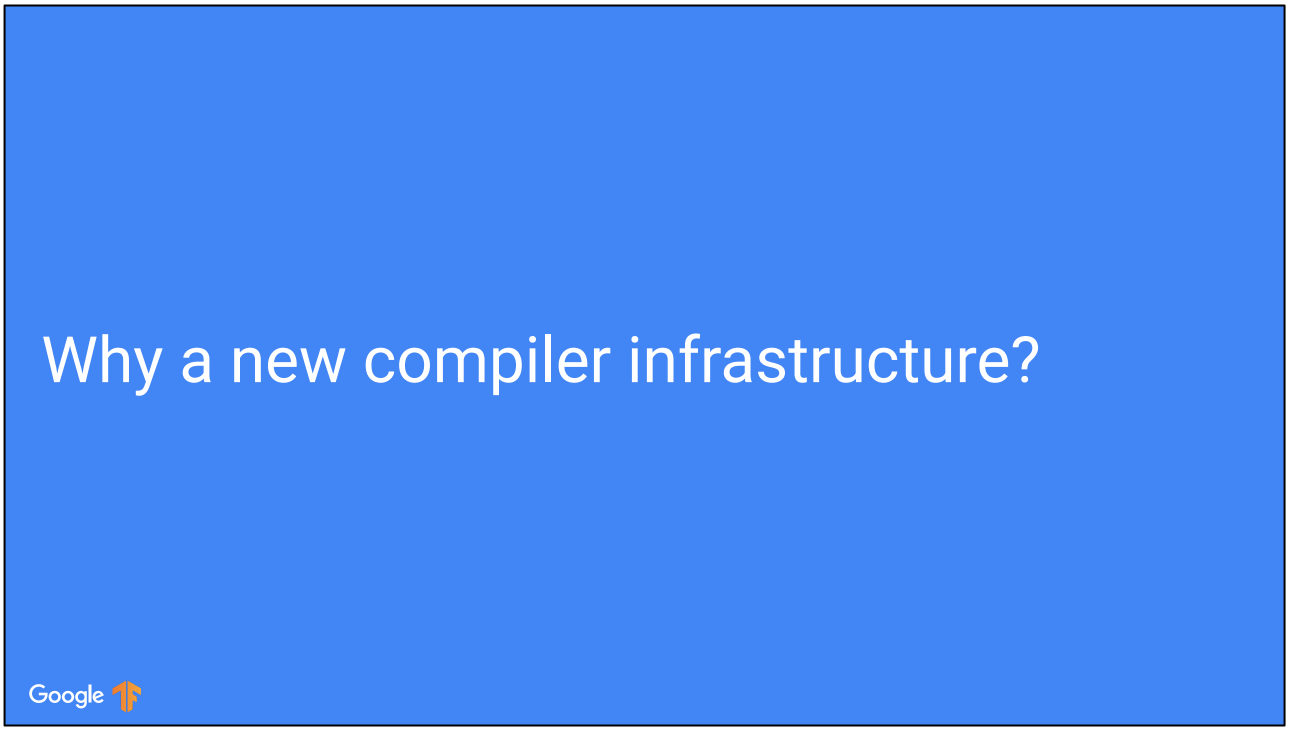
TensorFlow is a very general system, and our work is a key part of TensorFlow future, so we cannot take simplifying assumptions - we have to be able to support the full generality of the tensor problem.



Overview

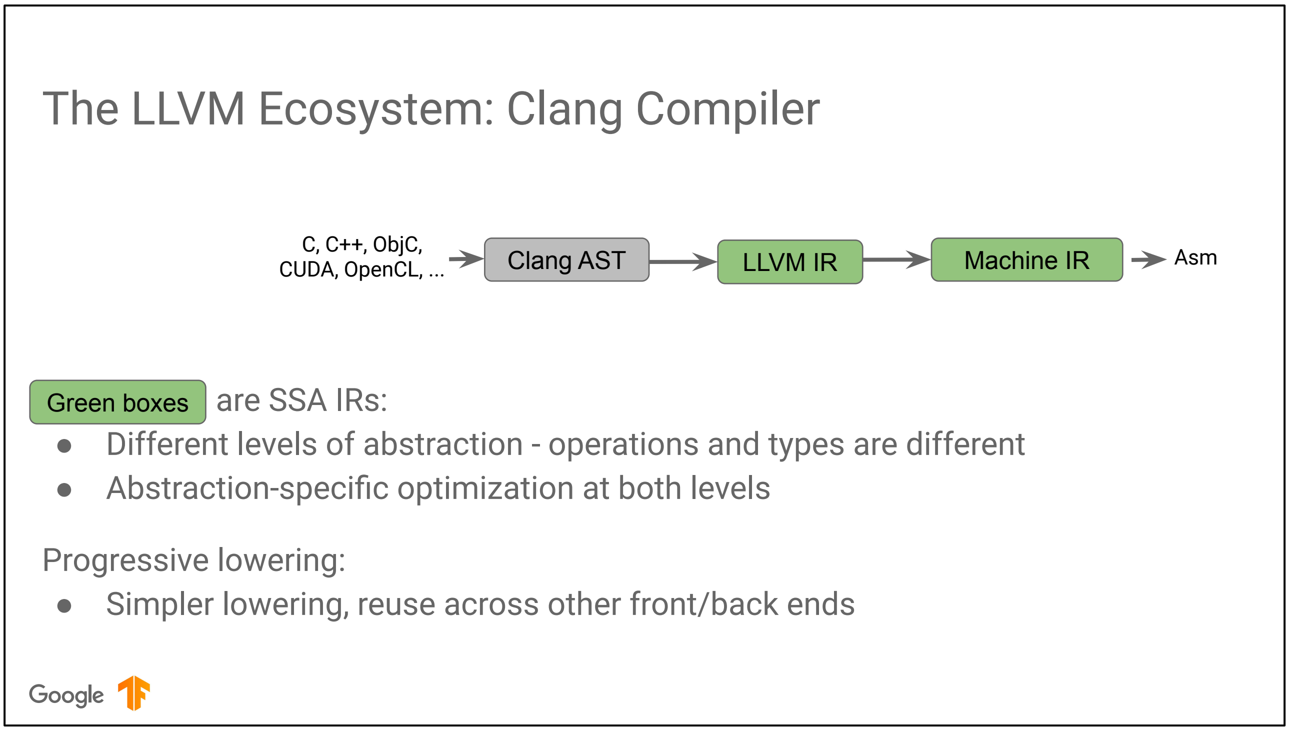
- Why a new compiler infrastructure?
- Quick Tour of MLIR + Infrastructure
- MLIR + TensorFlow
- Code Generation for Accelerators
- MLIR + Clang
- Getting Involved

If you've seen the C4ML talk, this talk has some common material, but is expanded and improved.



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

$$\begin{tikzcd}
\text{C, C++, ObjC, CUDA, OpenCL, ...} \arrow[r] & \boxed{\text{Clang AST}} \arrow[r] & \boxed{\text{LLVM IR}} \arrow[r] & \boxed{\text{Machine IR}} \arrow[r] & \text{Asm}
\end{tikzcd}$$

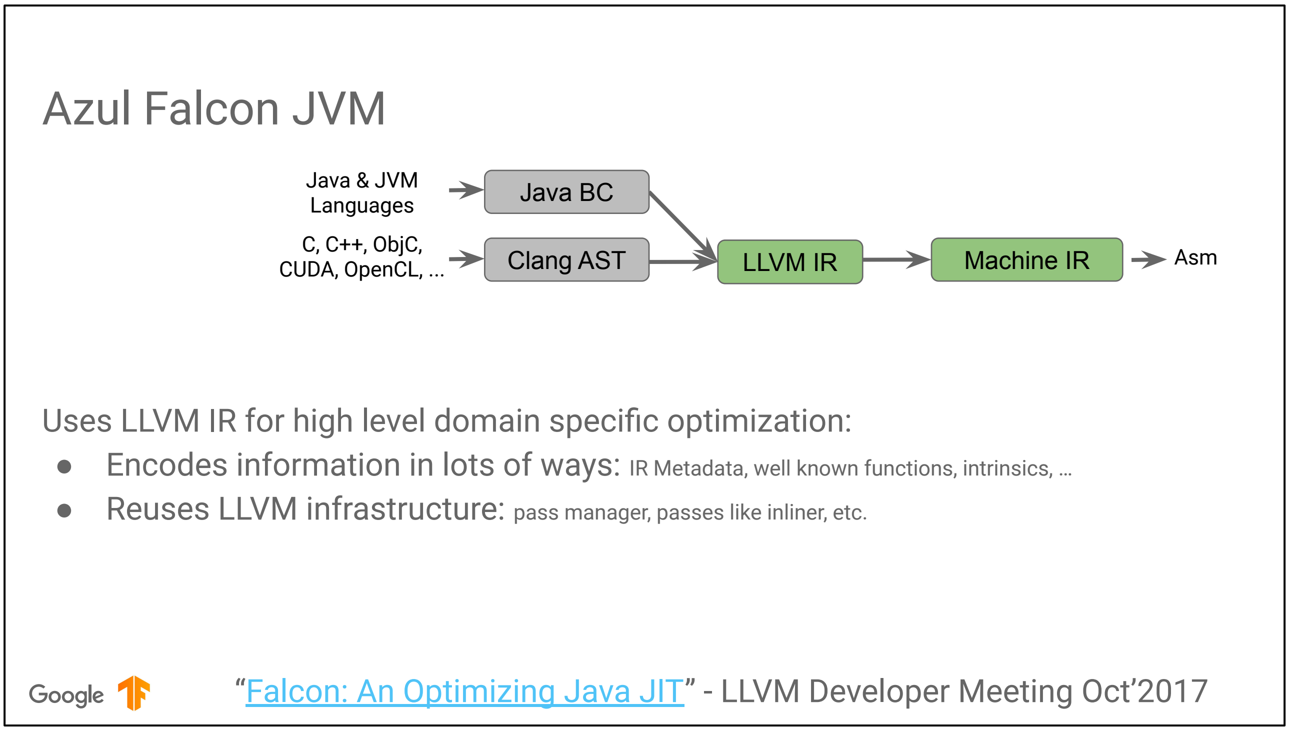
Green boxes are SSA IRs:

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

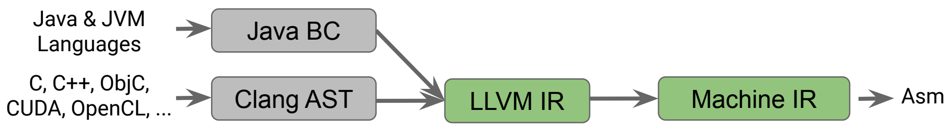
Progressive lowering:

- Simpler lowering, reuse across other front/back ends

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:

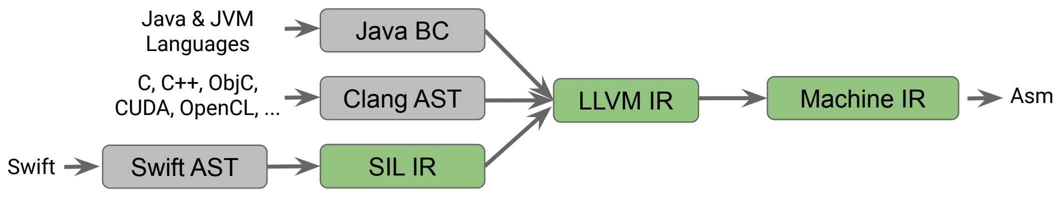
- Encodes information in lots of ways: IR Metadata, well known functions, intrinsics, ...
- Reuses LLVM infrastructure: pass manager, passes like inliner, etc.

Google

"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



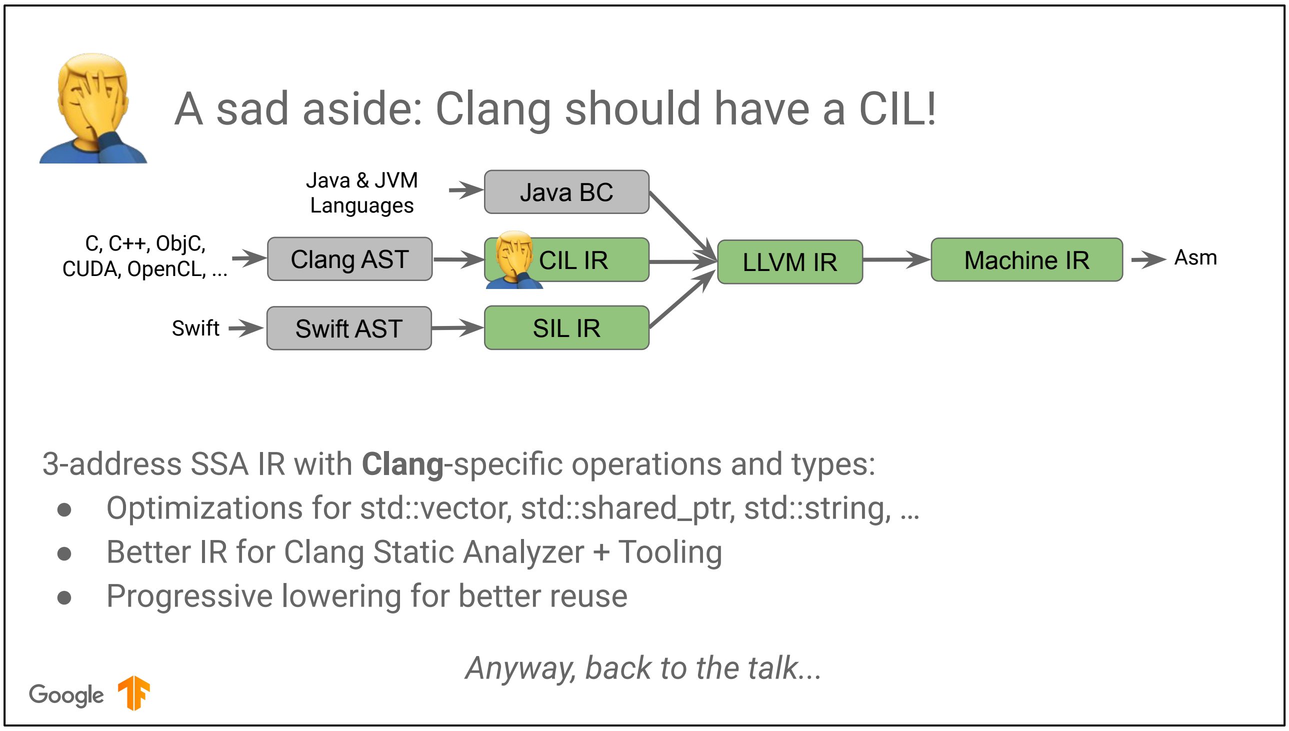
3-address SSA IR with Swift-specific operations and types:

- 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!

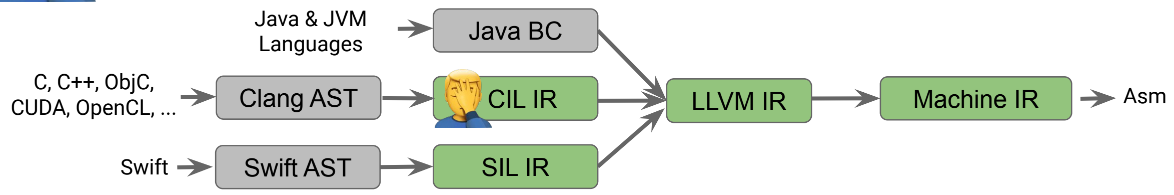
Google

“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!

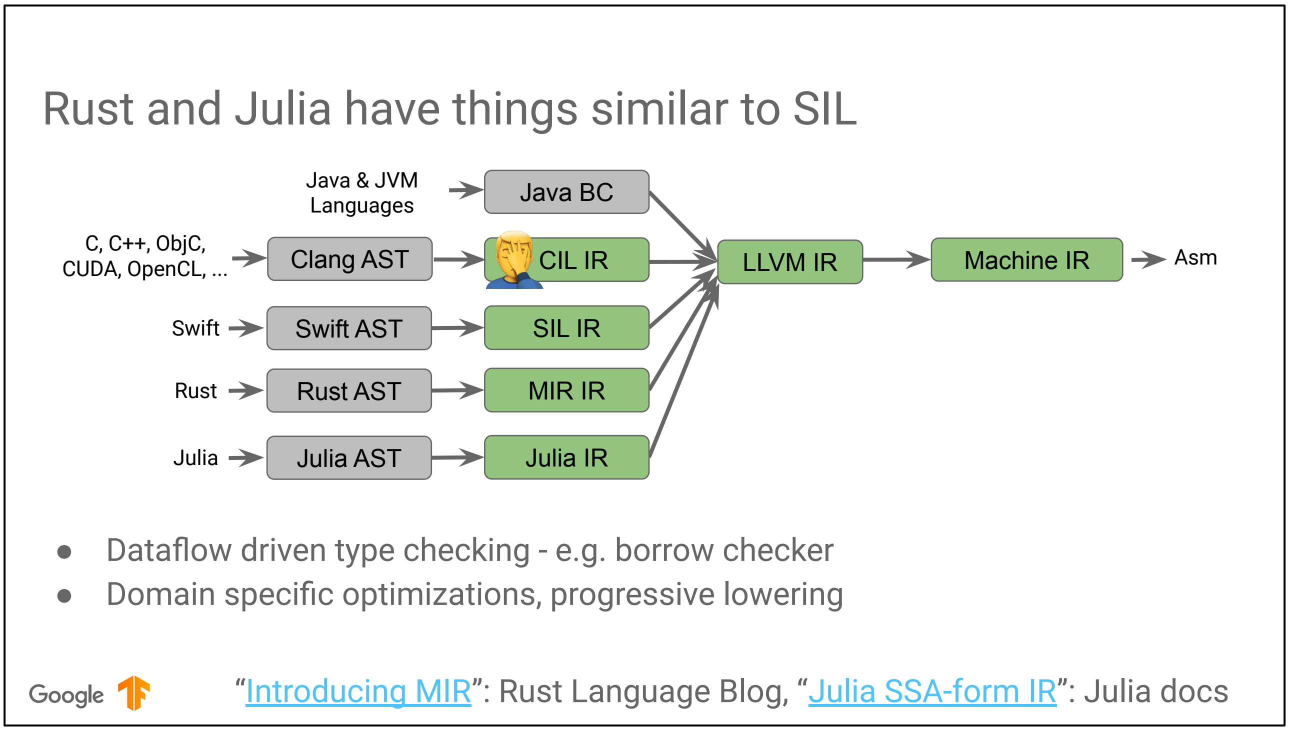


3-address SSA IR with **Clang**-specific operations and types:

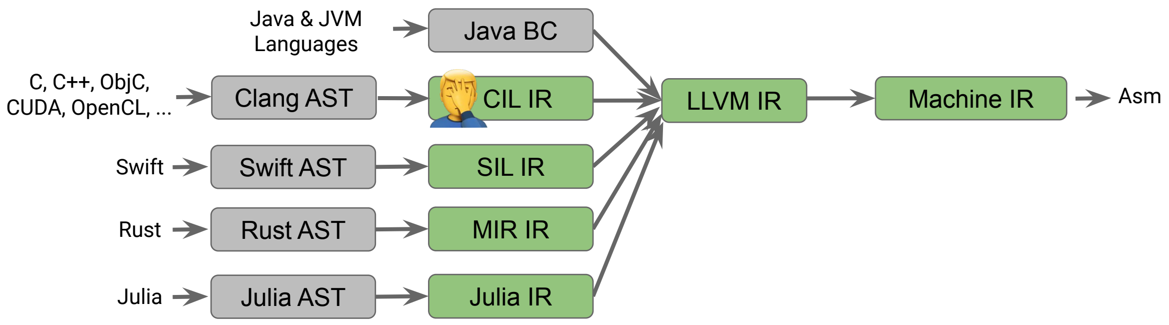
- Optimizations for std::vector, std::shared_ptr, std::string, …
- Better IR for Clang Static Analyzer + Tooling
- Progressive lowering for better reuse

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



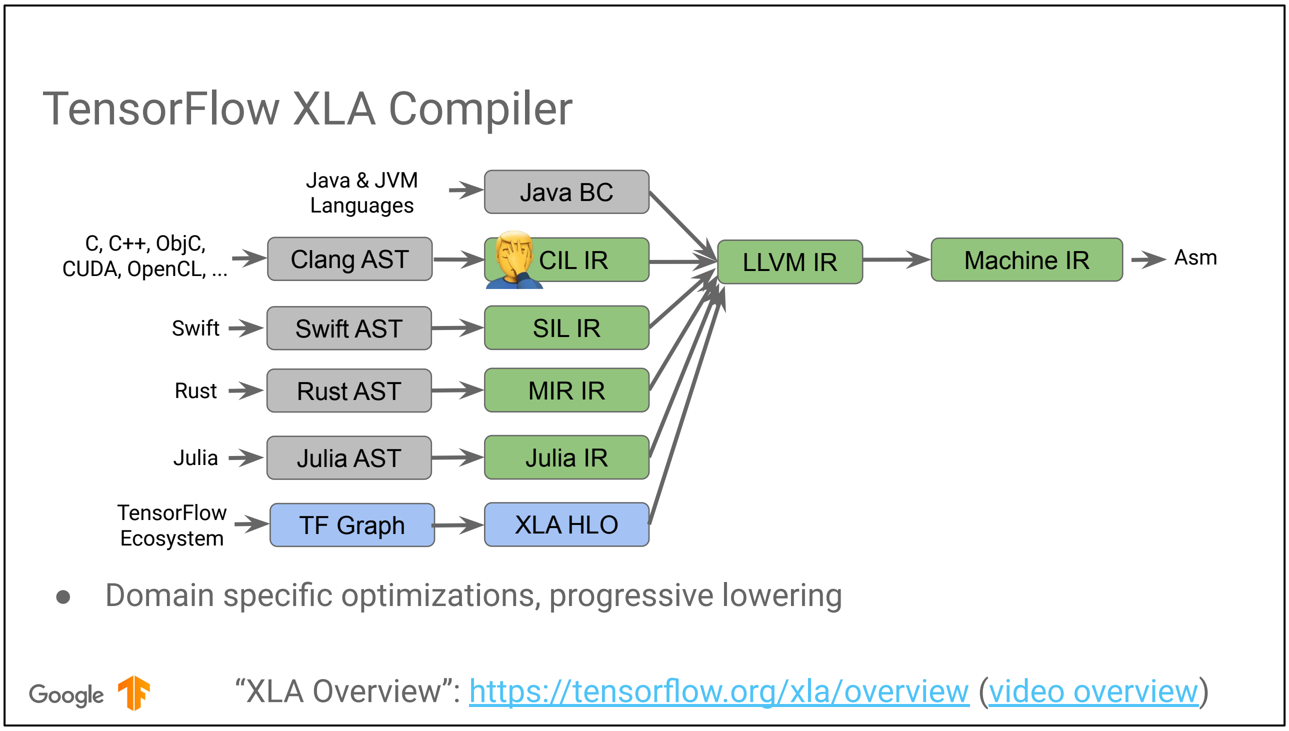
- Dataflow driven type checking - e.g. borrow checker
- Domain specific optimizations, progressive lowering

Google

"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.

Technically these aren't all SSA, but close enough for the purposes of this talk.



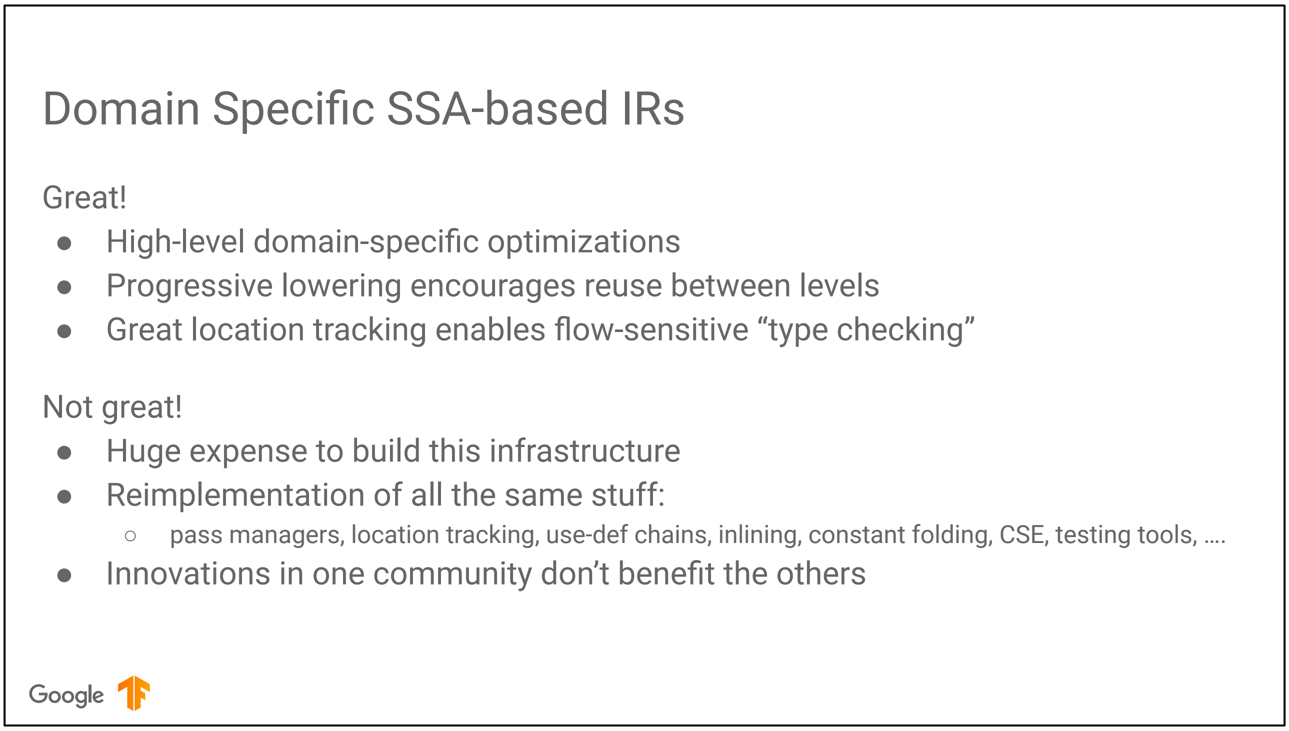
TensorFlow XLA Compiler

- 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!

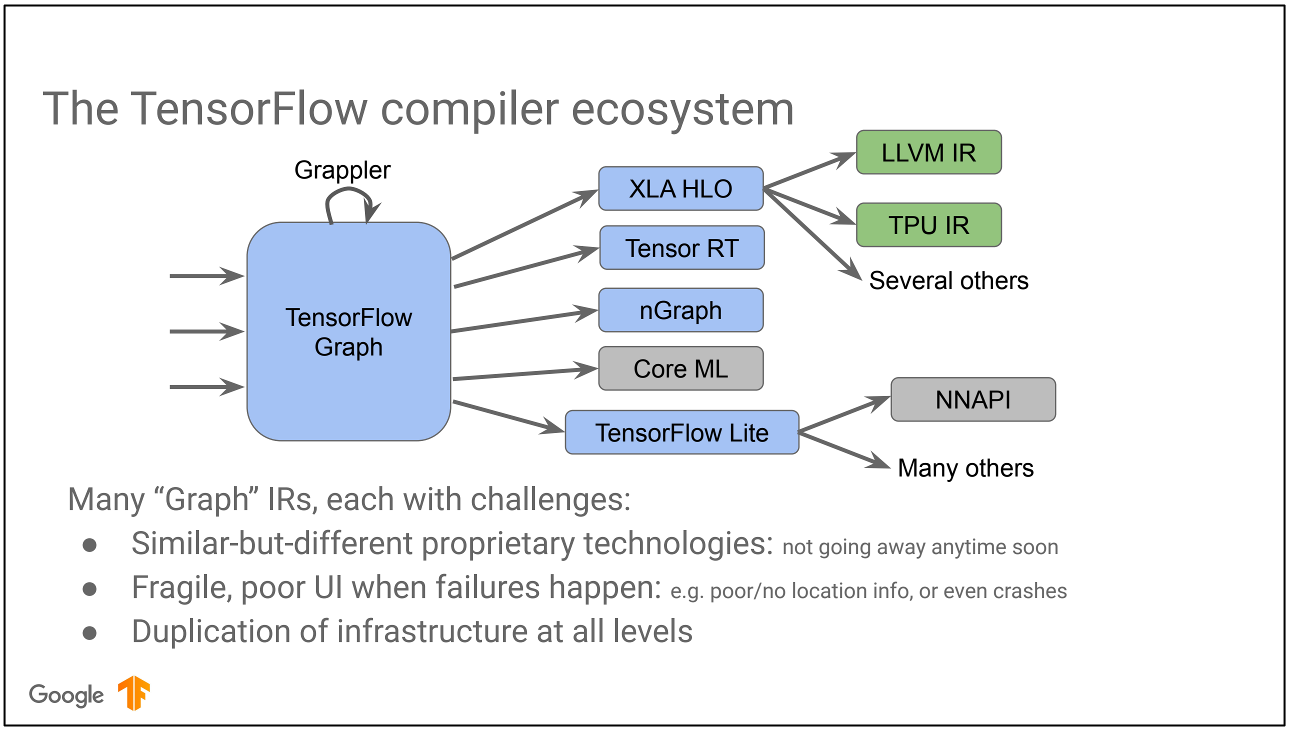
- High-level domain-specific optimizations
- Progressive lowering encourages reuse between levels
- Great location tracking enables flow-sensitive “type checking”

Not great!

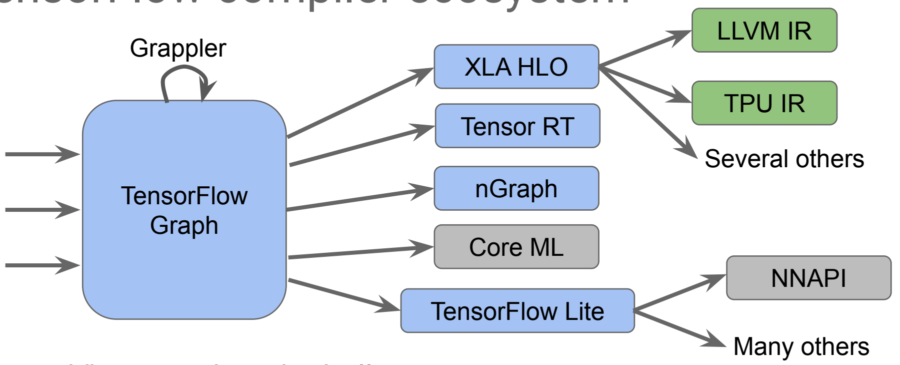
- Huge expense to build this infrastructure
- Reimplementation of all the same stuff:
    - pass managers, location tracking, use-def chains, inlining, constant folding, CSE, testing tools, ….
- Innovations in one community don’t benefit the others

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.



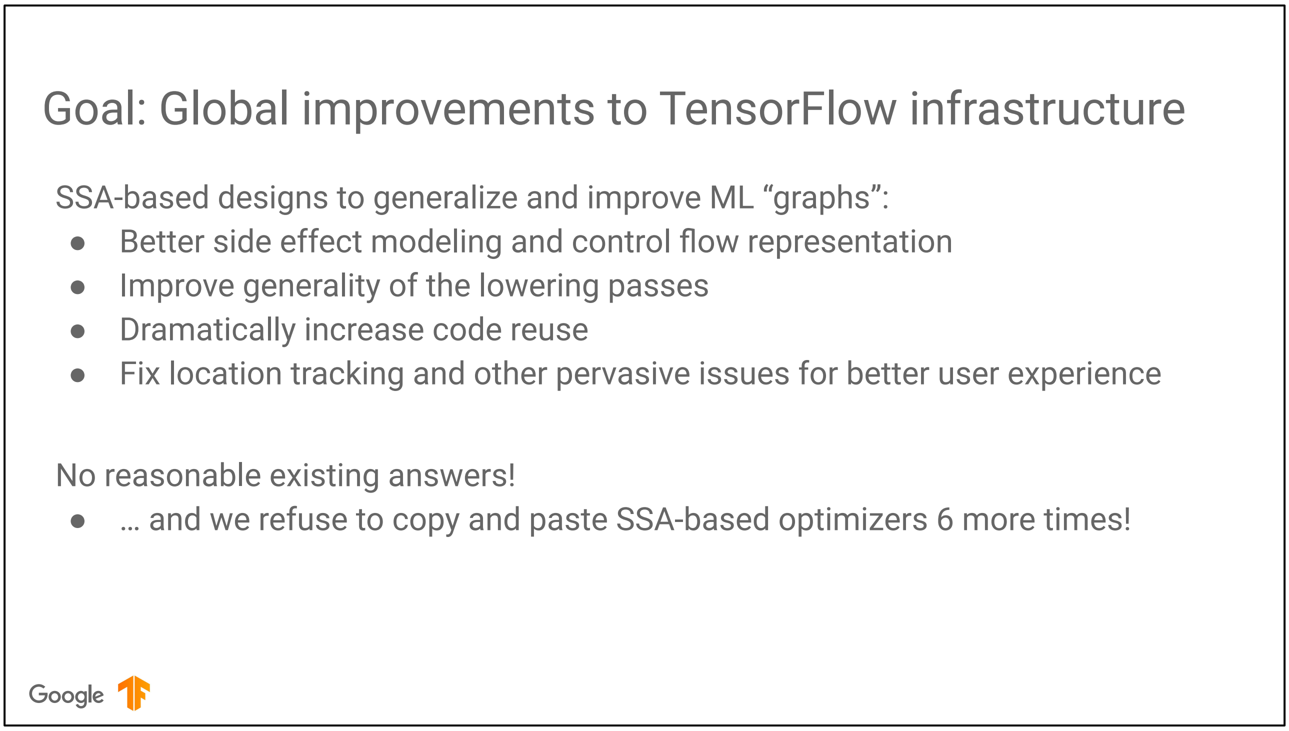
The TensorFlow compiler ecosystem



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

Coming back to the challenges we face on the TensorFlow team, we 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”:

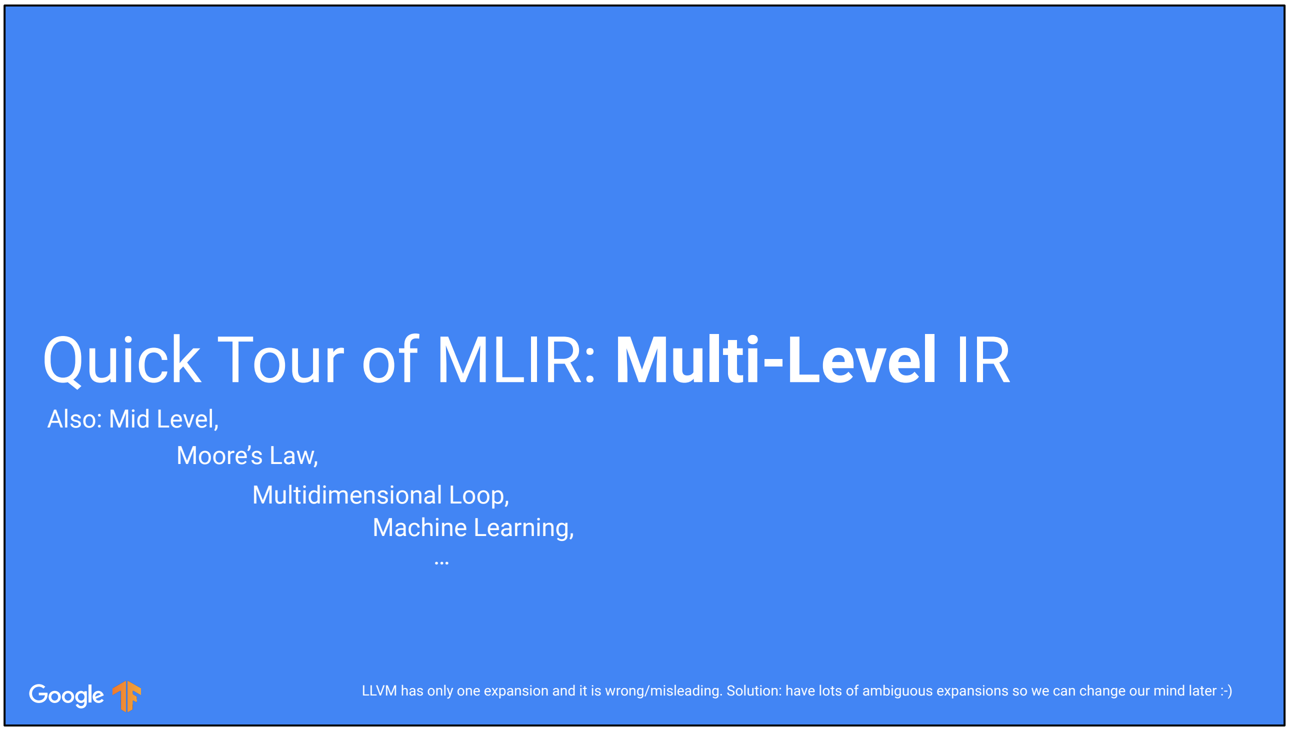
- Better side effect modeling and control flow representation
- Improve generality of the lowering passes
- Dramatically increase code reuse
- Fix location tracking and other pervasive issues for better user experience

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?

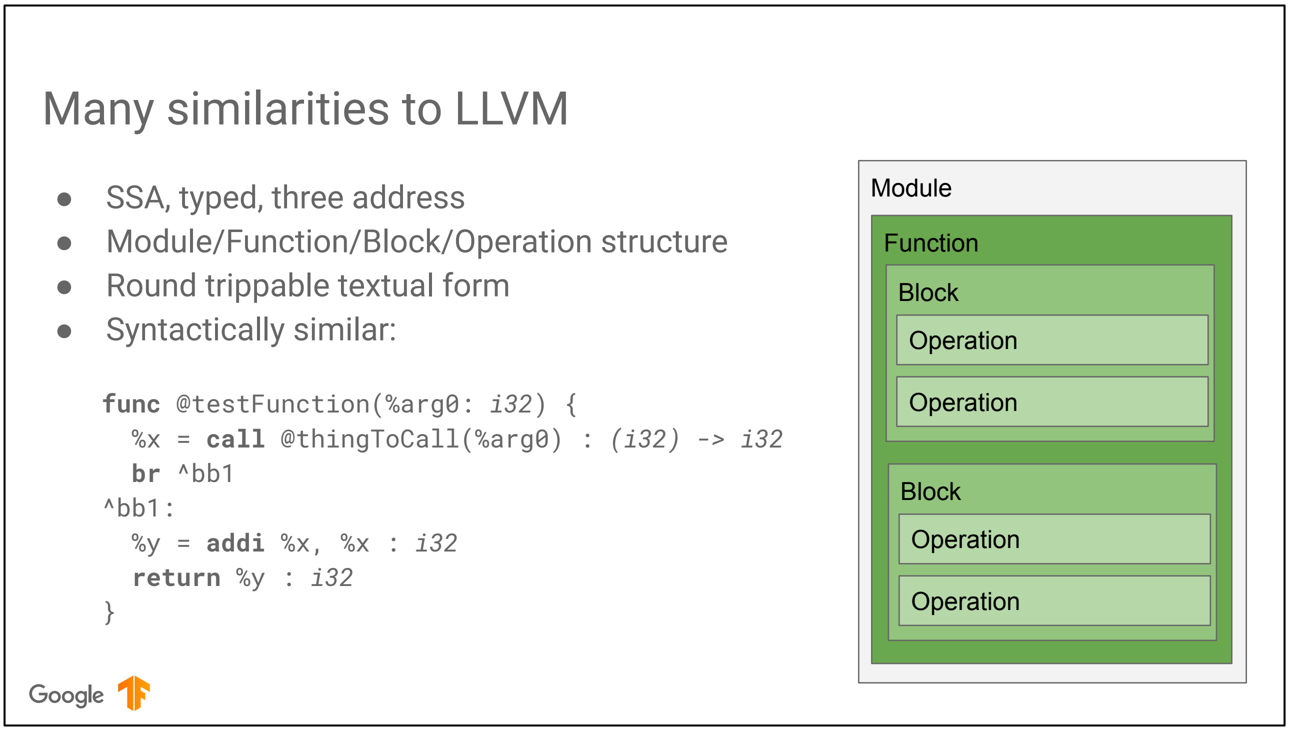


Quick Tour of MLIR: **Multi-Level** IR

Also: Mid Level,
Moore's Law,
Multidimensional Loop,
Machine Learning,
...

LLVM has only one expansion and it is wrong/misleading. Solution: have lots of ambiguous expansions so we can change our mind later :-)

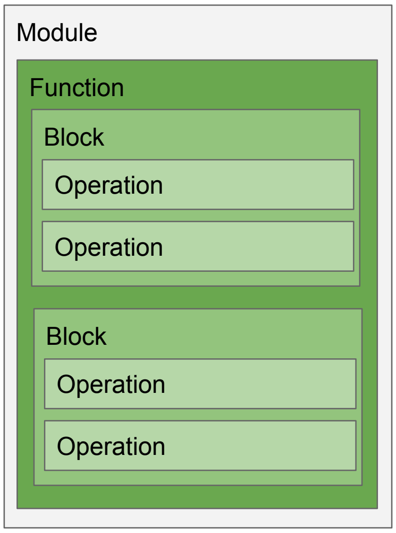
This brings us to MLIR. "ML" expands in multiple ways, principally "Multi-Level", but also Mid Level, Machine Learning, Multidimensional Loop, Moore's Law, and I'm sure we'll find other clever expansions in the future.



Many similarities to LLVM

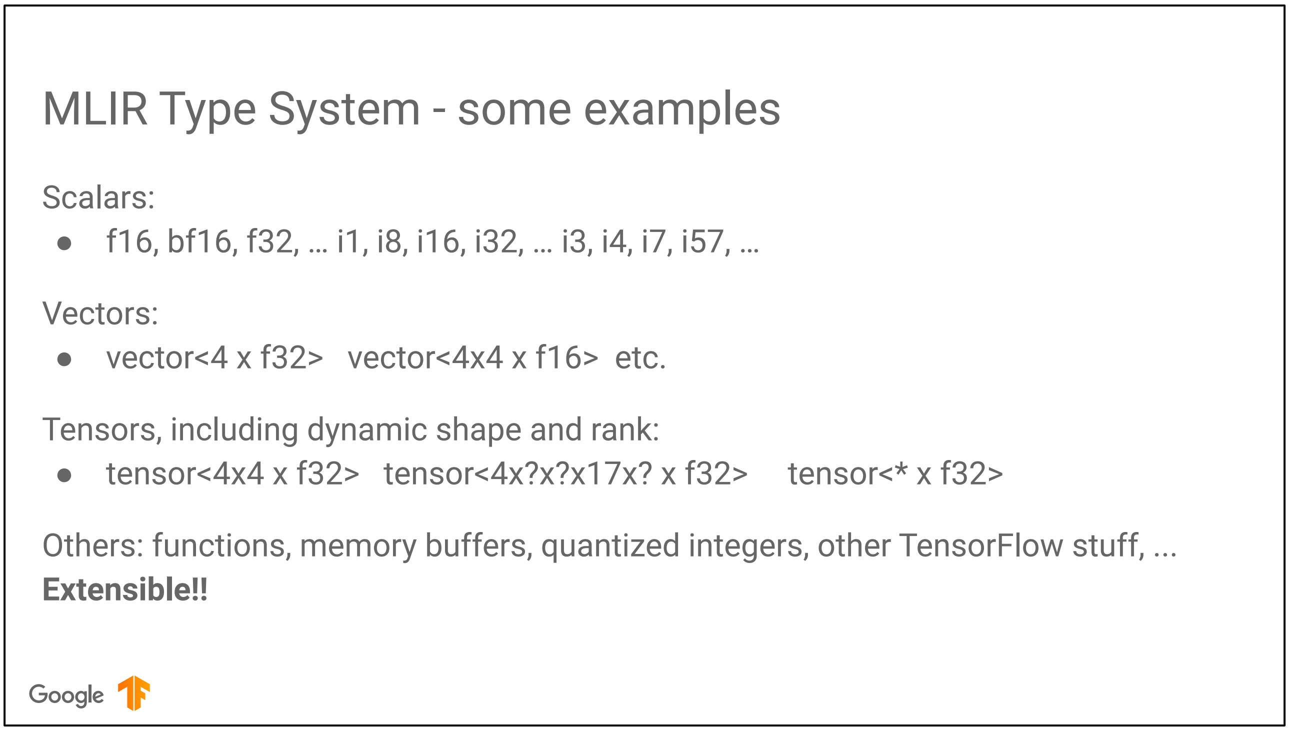
- SSA, typed, three address
- Module/Function/Block/Operation structure
- Round trippable textual form
- Syntactically similar:

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



Google

MLIR is highly influenced by LLVM and unabashedly reuses many great ideas 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.

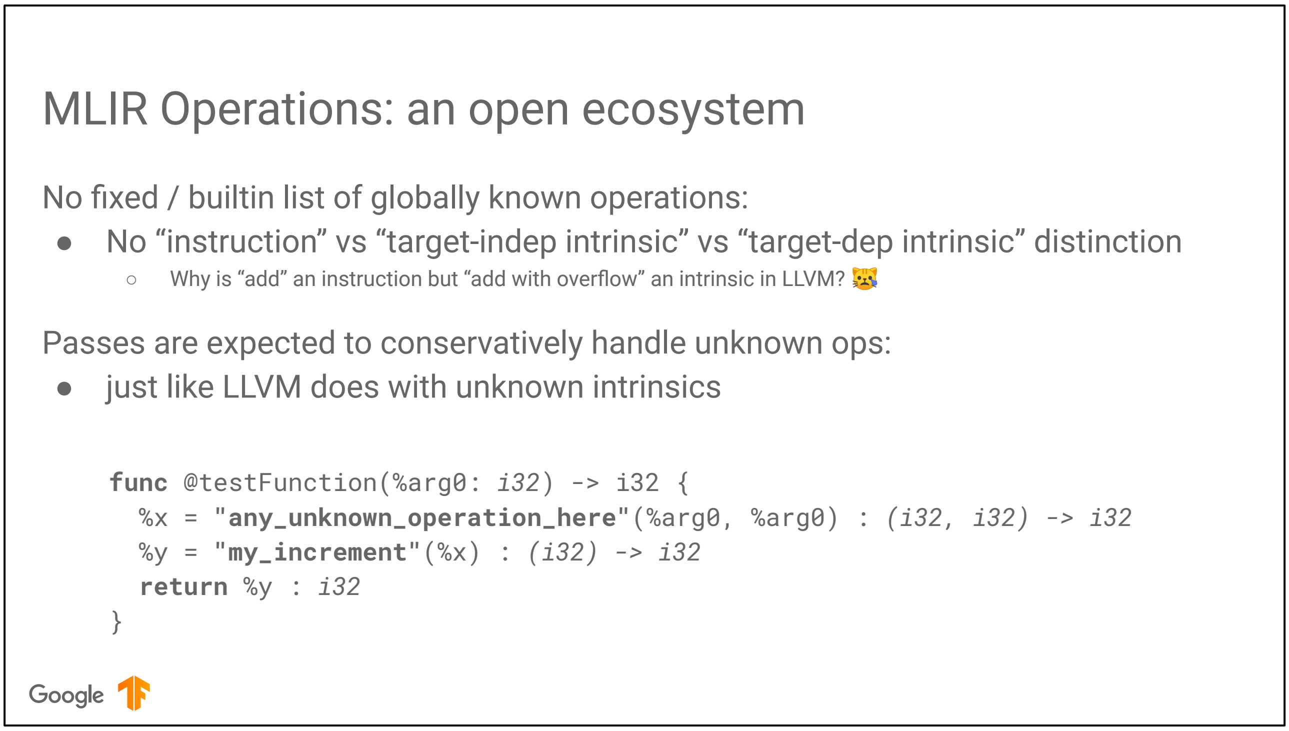
Tensors, including dynamic shape and rank:

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

Others: functions, memory buffers, quantized integers, other TensorFlow stuff, ...
Extensible!!

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 Operations: 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? 😿

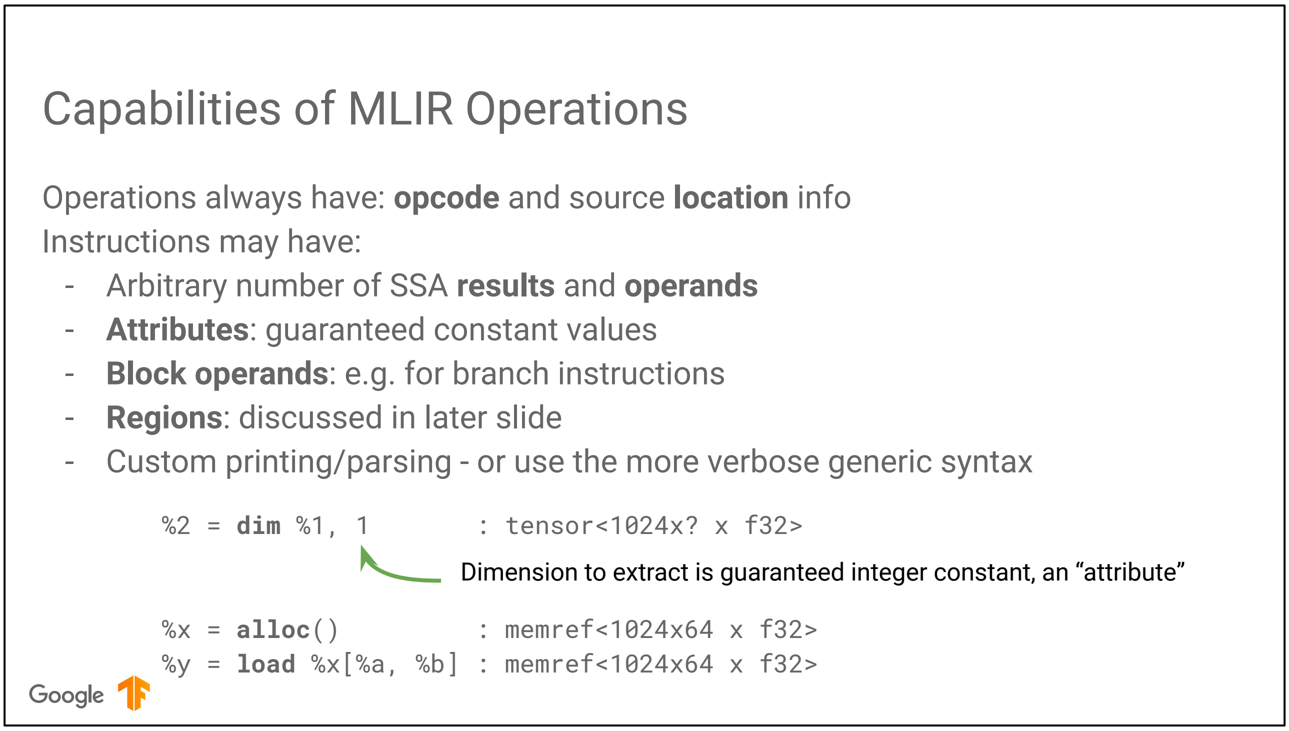
Passes are expected to conservatively handle unknown ops:

- 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
}
```

Google

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.



Capabilities of MLIR Operations

Operations always have: **opcode** and source **location** info
Instructions may have:

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

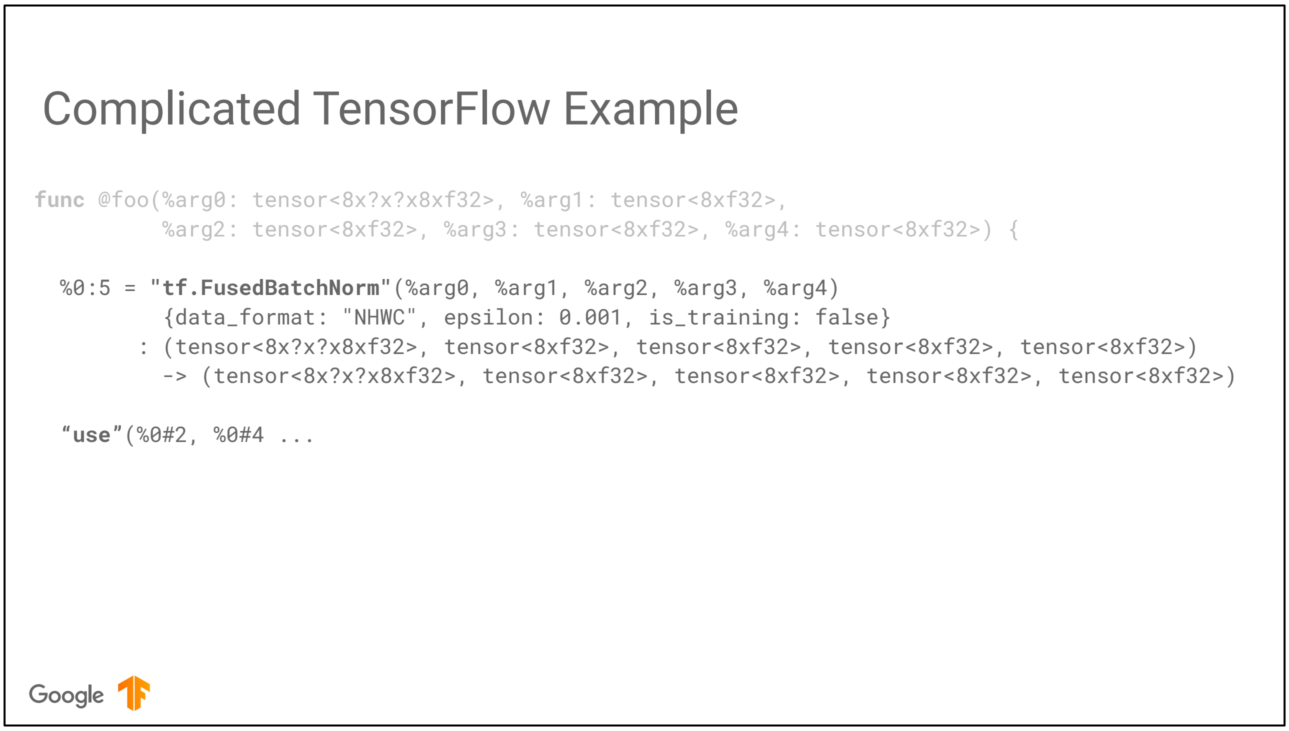
```
%2 = dim %1, 1        : tensor<1024x? x f32>
```

Dimension to extract is guaranteed integer constant, an “attribute”

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

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

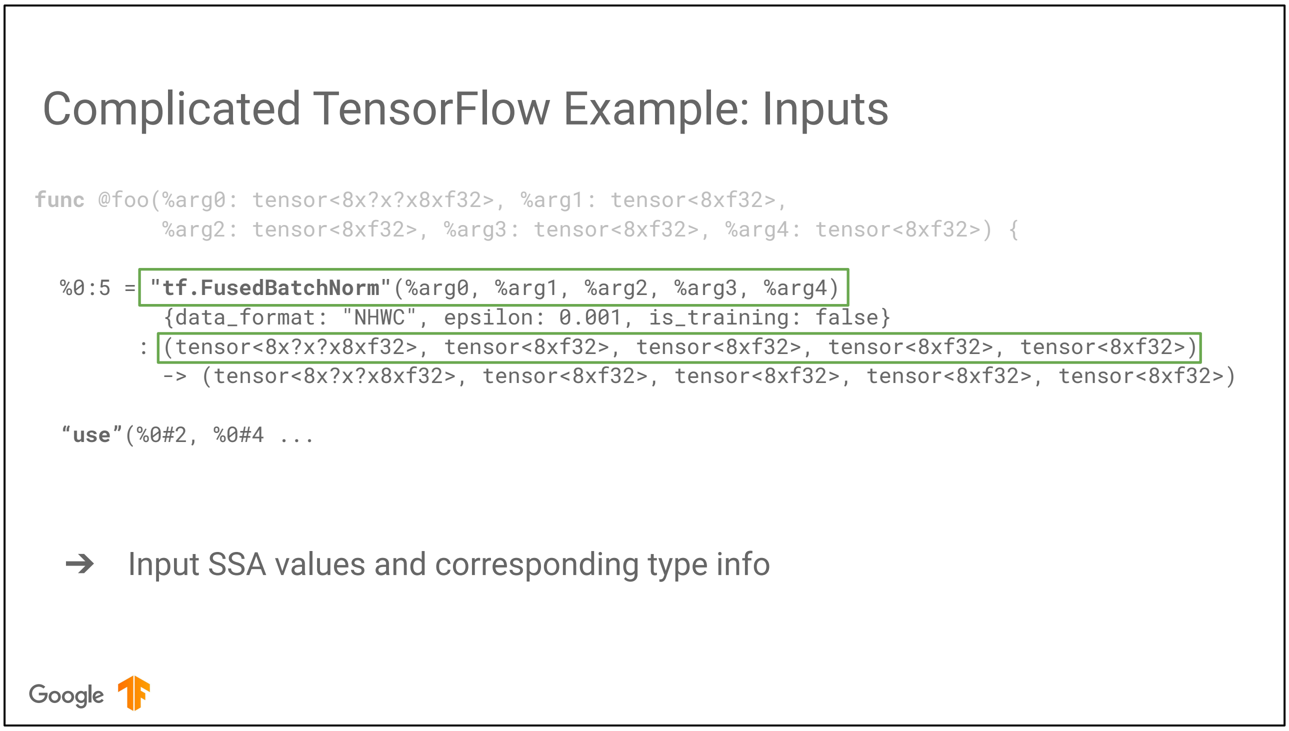
One thing to note is that operations can customize their printing, so you’ll see specialized printing for common ops like in this slide, and the default generic printer that uses double quotes.



Complicated TensorFlow Example

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

  %0:5 = "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.



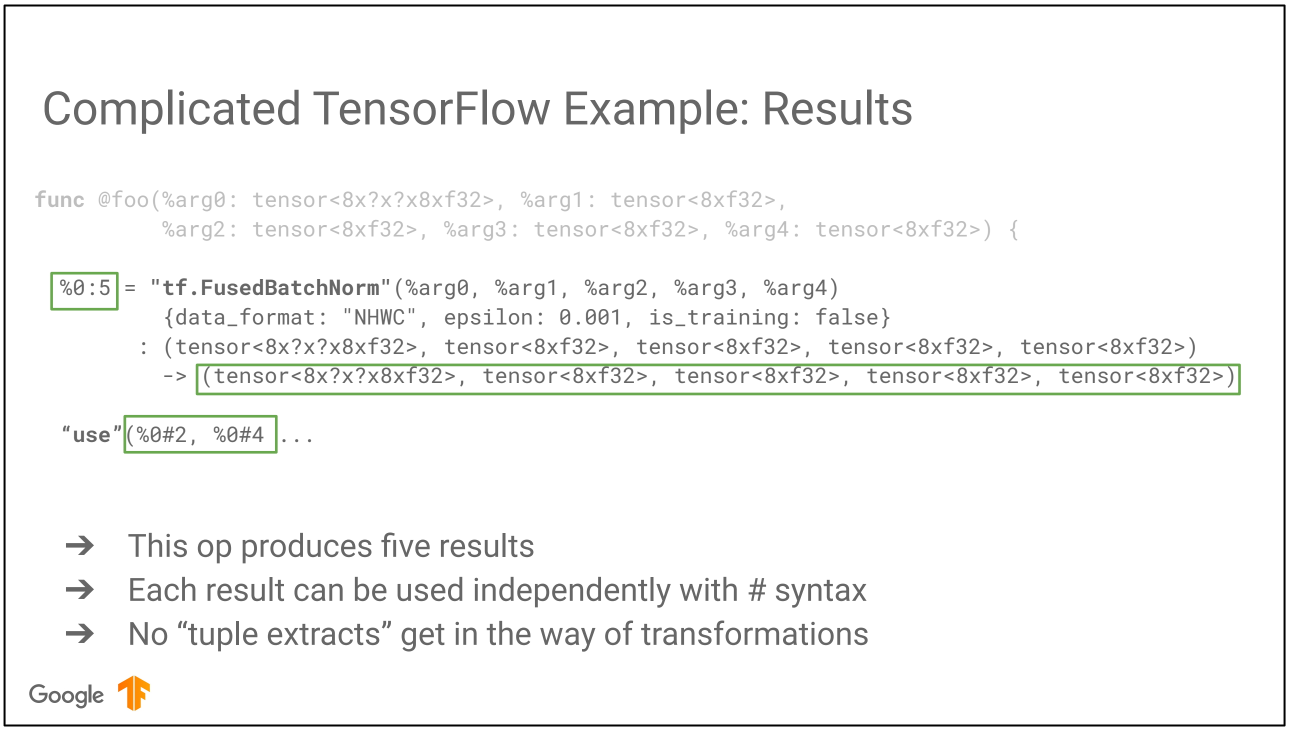
Complicated TensorFlow Example: Inputs

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

  %0:5 = "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

Google

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



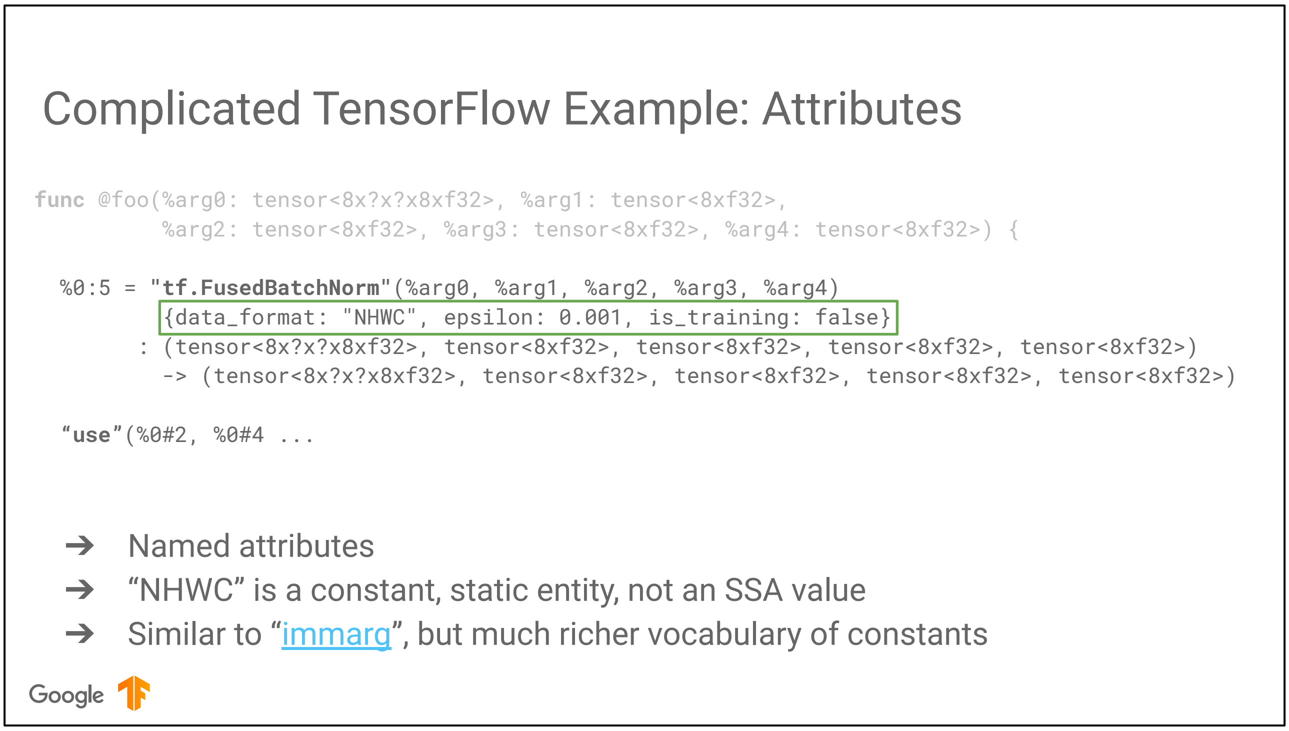
Complicated TensorFlow Example: Results

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

  %0:5 = "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

Google

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



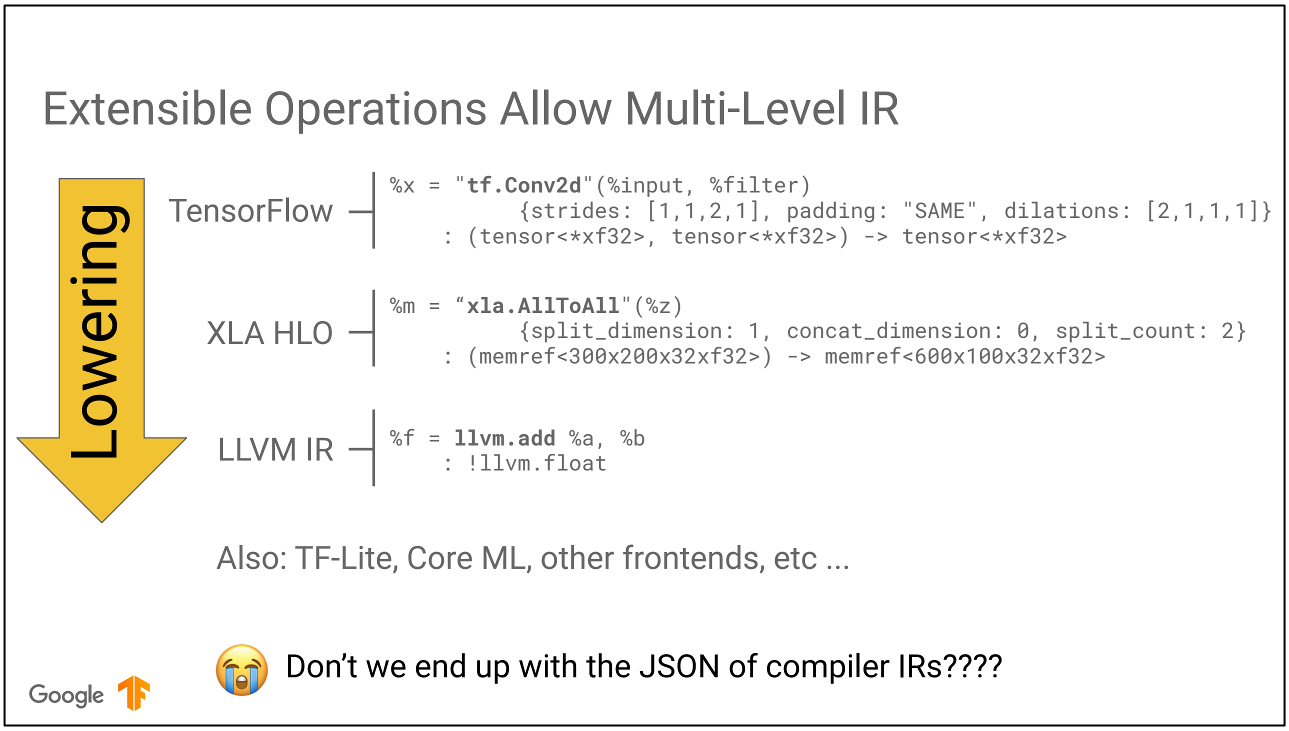
Complicated TensorFlow Example: Attributes

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

  %0:5 = "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
- ➔ Similar to "immarg", but much richer vocabulary of constants

Google

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

Lowering
TensorFlow
XLA HLO
LLVM IR

```
%x = "tf.Conv2d"(%input, %filter)
          {strides: [1,1,2,1], padding: "SAME", dilations: [2,1,1,1]}
    : (tensor<*xf32>, tensor<*xf32>) -> tensor<*xf32>
```

```
%m = "xla.AllToAll"(%z)
          {split_dimension: 1, concat_dimension: 0, split_count: 2}
    : (memref<300x200x32xf32>) -> memref<600x100x32xf32>
```

```
%f = llvm.add %a, %b
    : !llvm.float
```

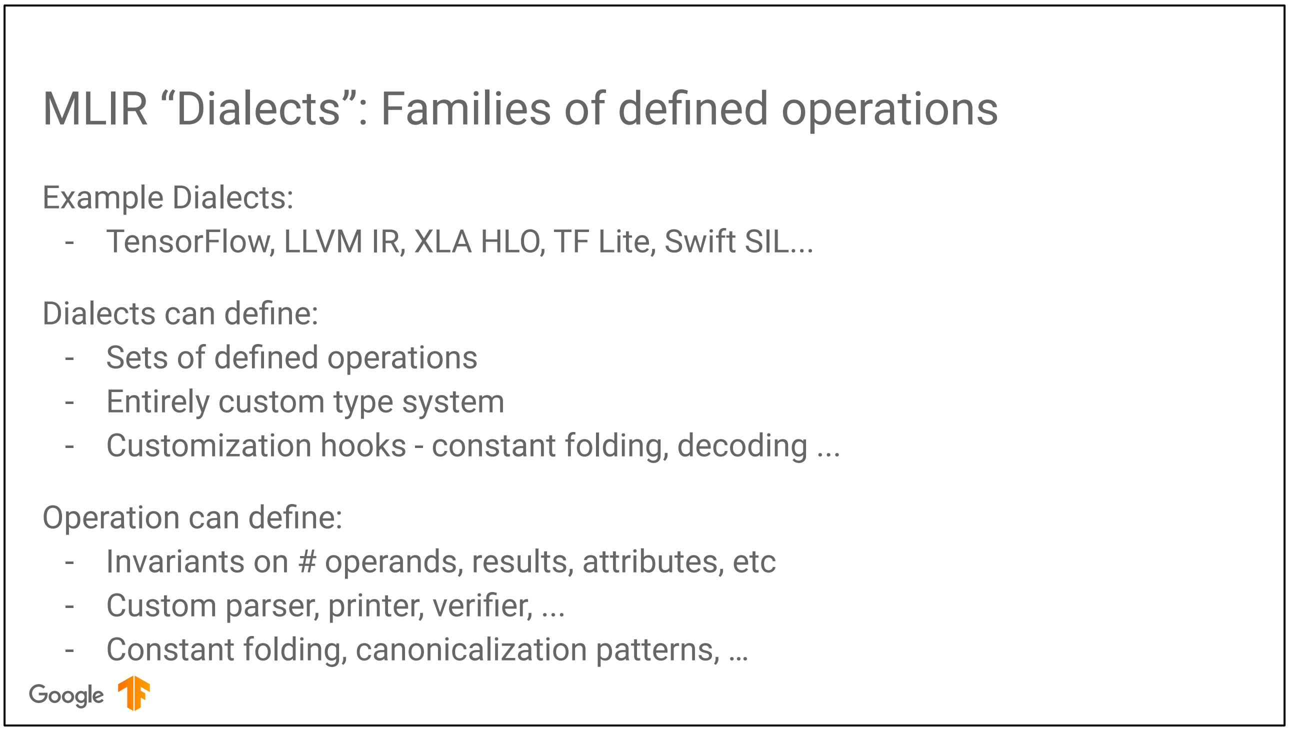
Also: TF-Lite, Core ML, other frontends, etc ...

😭 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:

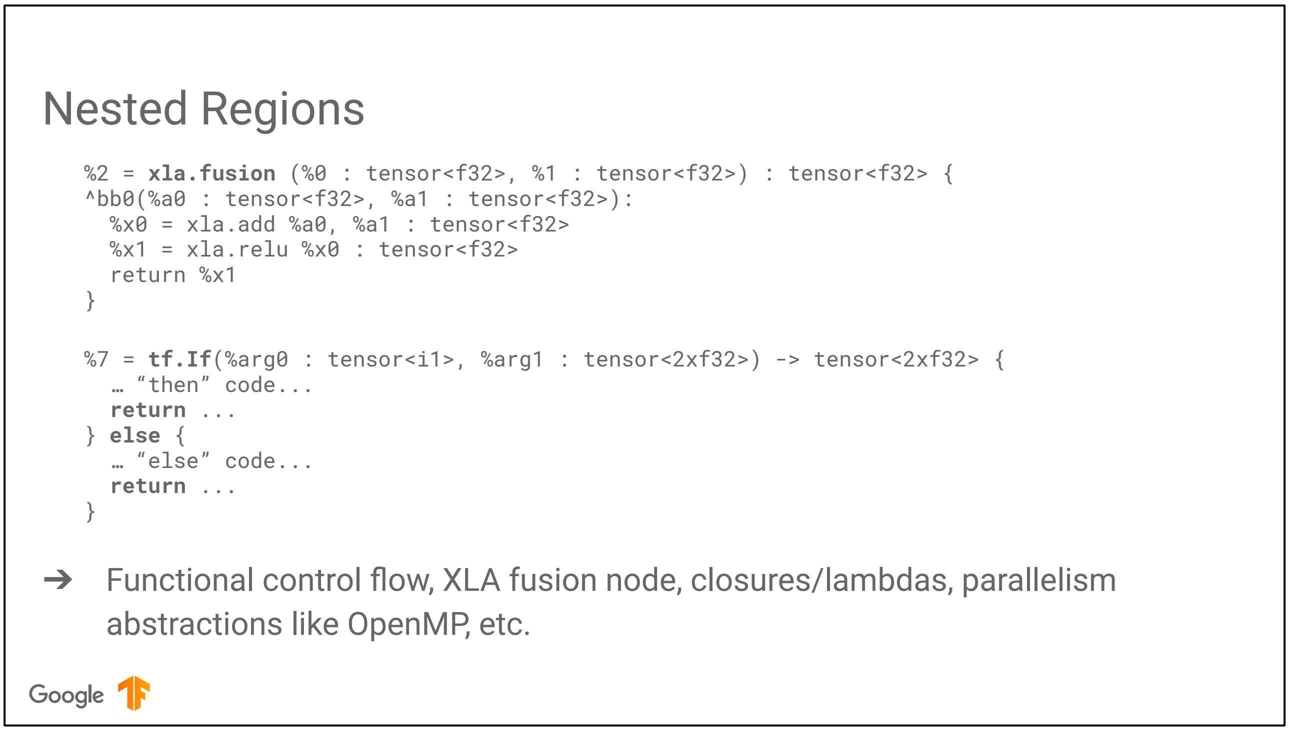
- Sets of defined operations
- Entirely custom type system
- Customization hooks - constant folding, decoding ...

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.

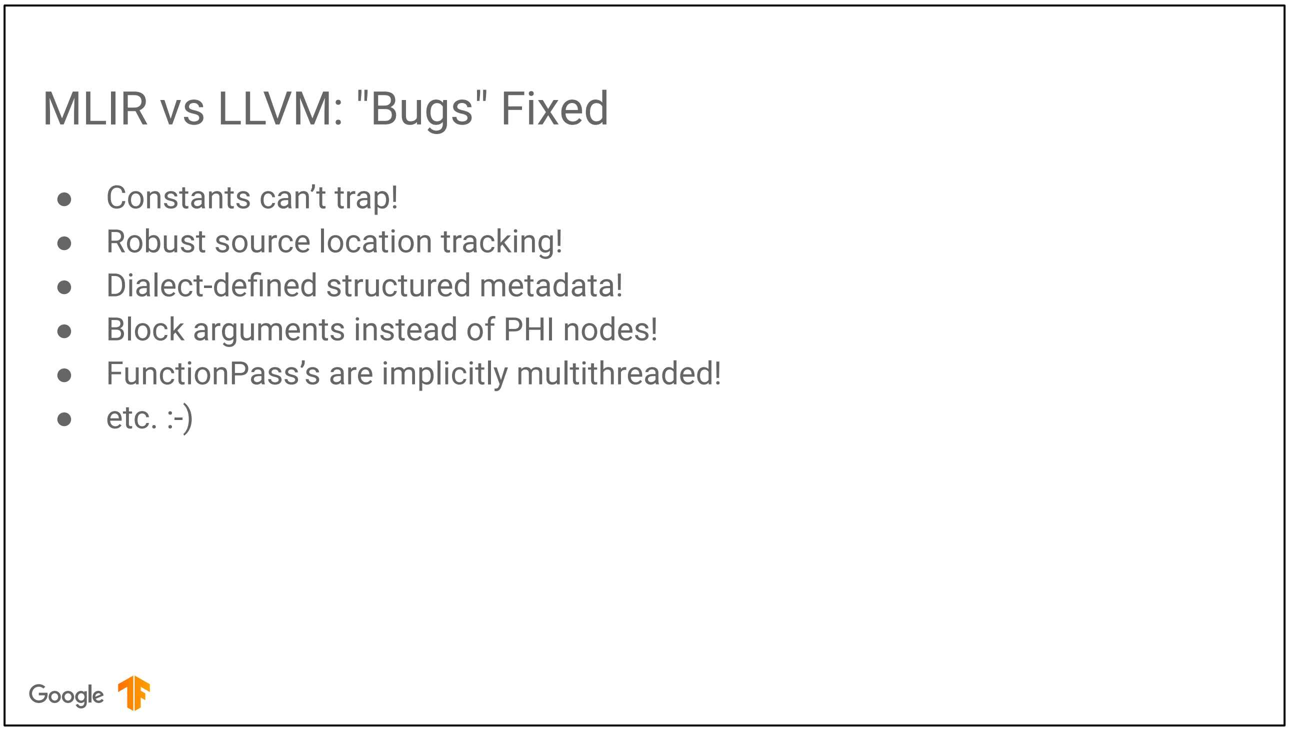


Nested Regions

```
%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/lambdas, 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.



MLIR vs LLVM: "Bugs" Fixed

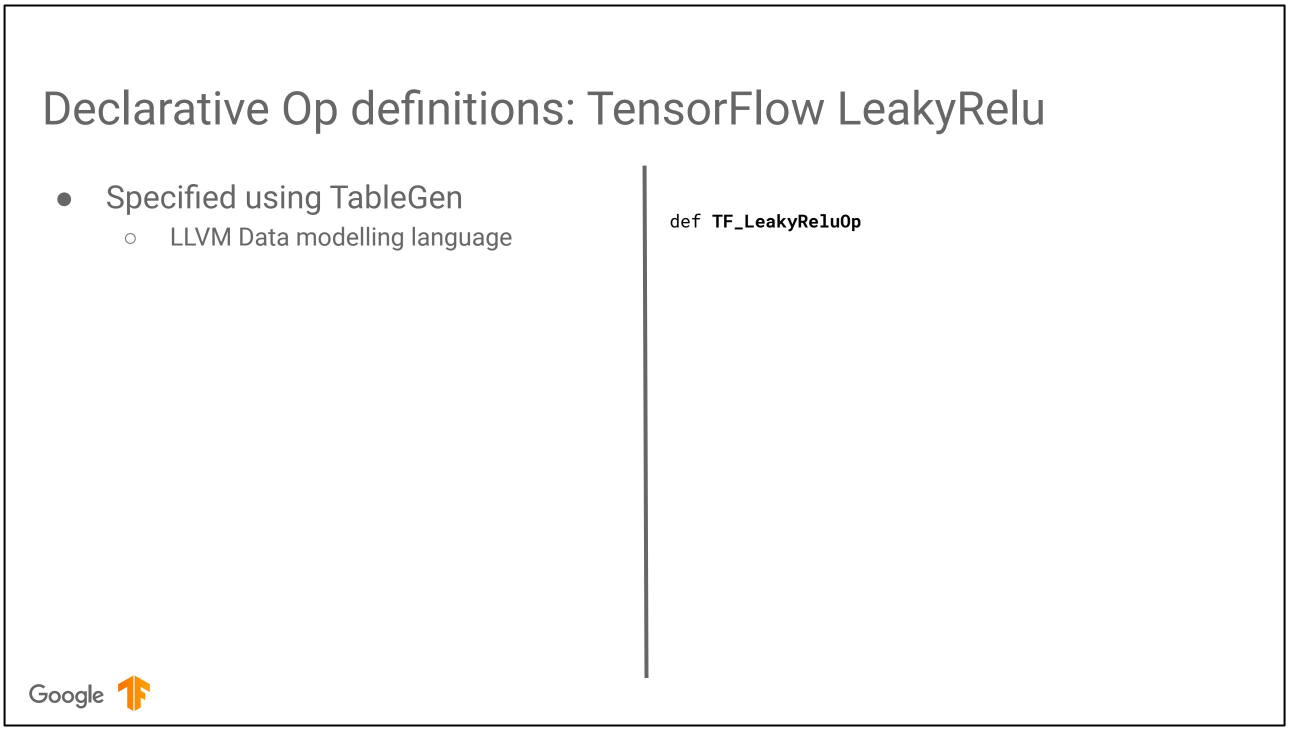
- Constants can't trap!
- Robust source location tracking!
- Dialect-defined structured metadata!
- Block arguments instead of PHI nodes!
- FunctionPass's are implicitly multithreaded!
- etc. :-)

Google

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



Declarative Op definitions: TensorFlow LeakyRelu

- Specified using TableGen
    - LLVM Data modelling language

```
def TF_LeakyReluOp
```

Google

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.

Declarative Op definitions: TensorFlow LeakyRelu

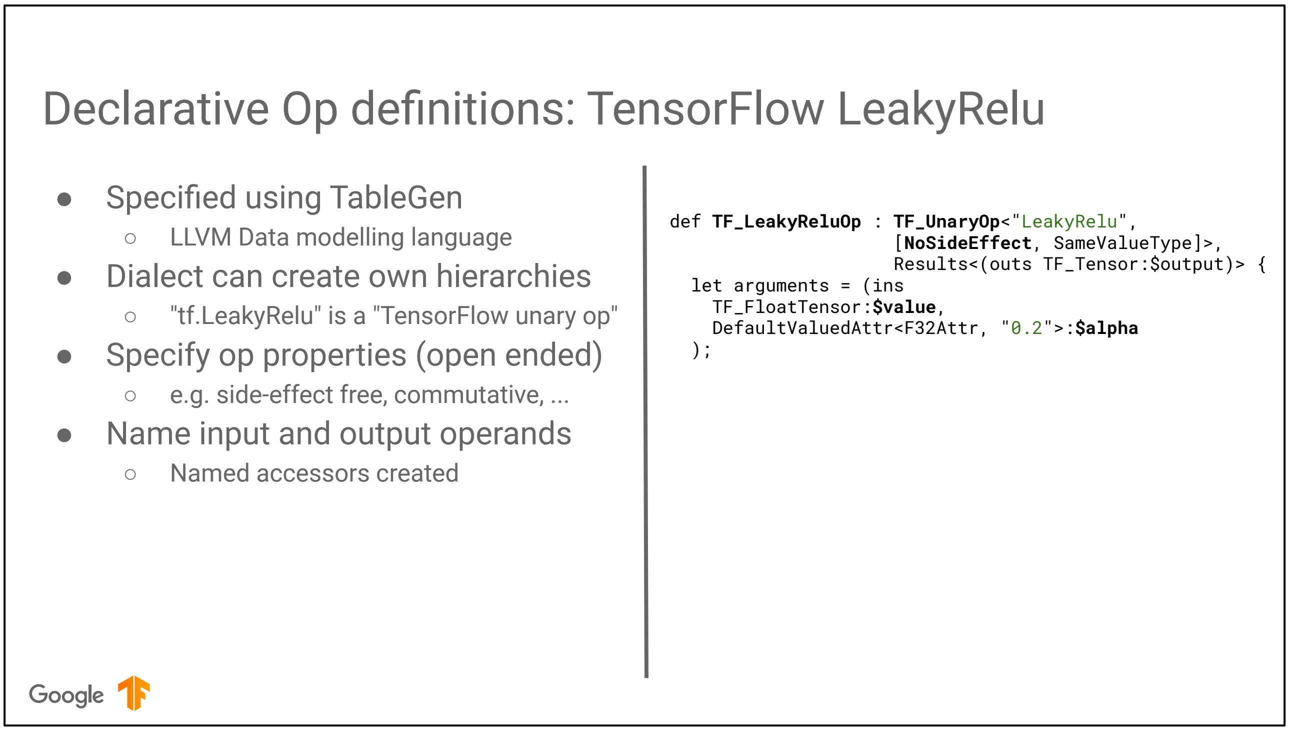
- Specified using TableGen
  - LLVM Data modelling language
- Dialect can create own hierarchies
  - "tf.LeakyRelu" is a "TensorFlow unary op"

```
def TF_LeakyReluOp : TF_UnaryOp<"LeakyRelu",
```

Declarative Op definitions: 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)
  - e.g. side-effect free, commutative, ...

```
def TF_LeakyReluOp : TF_UnaryOp<"LeakyRelu",
                     [NoSideEffect, SameValueType]>,
```



Declarative Op definitions: 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)
  - e.g. side-effect free, commutative, ...
- Name input and output operands
  - Named accessors created

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

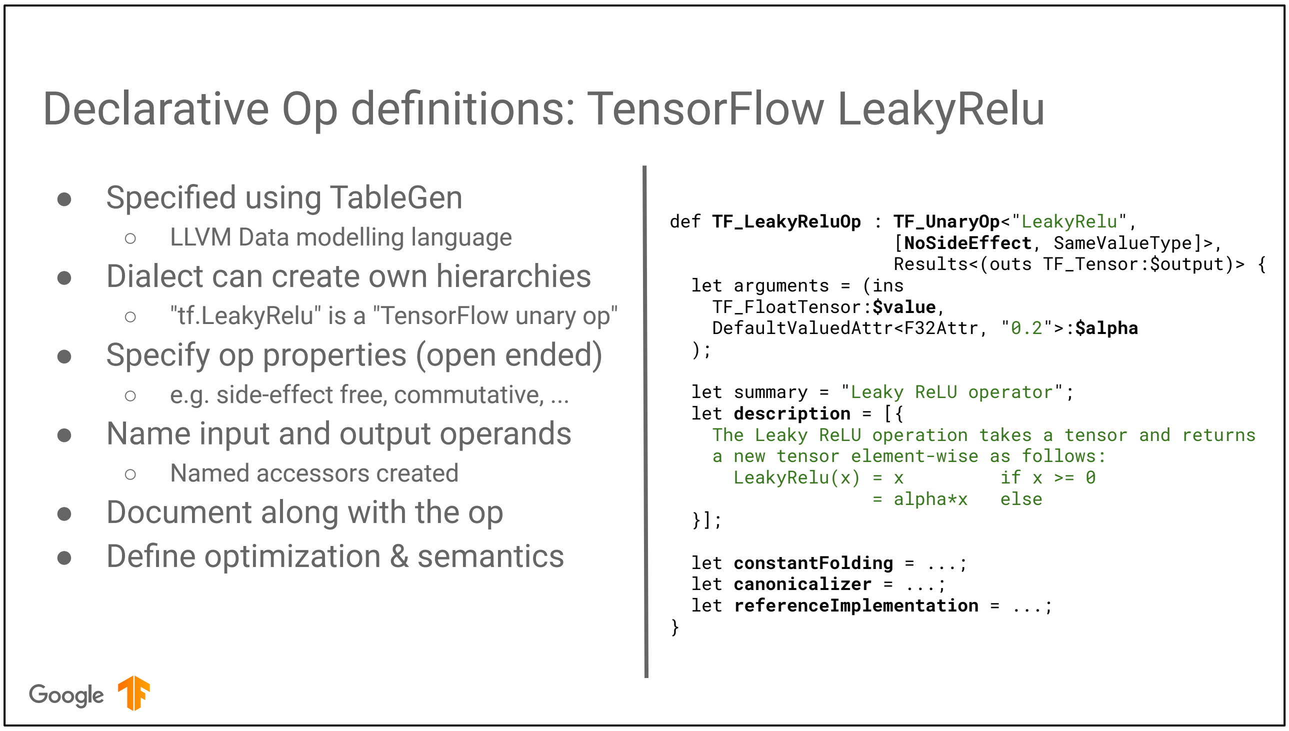
Google

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.

Declarative Op definitions: 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)
  - e.g. side-effect free, commutative, ...
- Name input and output operands
  - Named accessors created
- Document along with the op

```
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
  }];
```



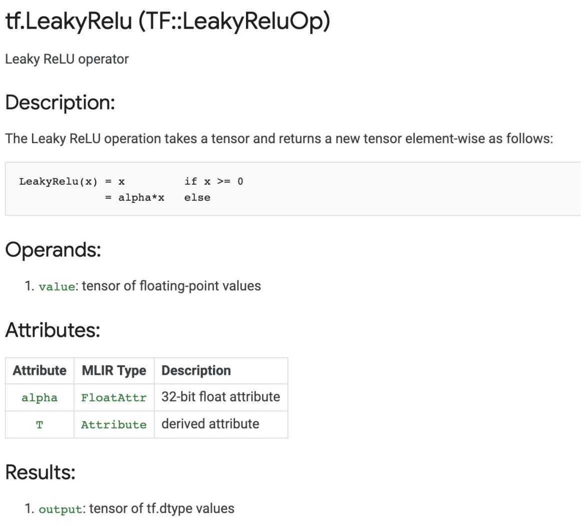
Declarative Op definitions: TensorFlow LeakyRelu

- Specified using TableGen
  - LLVM Data modelling language
- Dialect can create own hierarchies
  - "tf.LeakyRelu" is a "TensorFlow unary op"
- Specify op properties (open ended)
  - e.g. side-effect free, commutative, ...
- 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 documentation



tf.LeakyRelu (TF::LeakyReluOp)

Leaky ReLU operator

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
```

Operands:

1. `value`: tensor of floating-point values

Attributes:

Attribute	MLIR Type	Description
`alpha`	`FloatAttr`	32-bit float attribute
`T`	`Attribute`	derived attribute

Results:

1. `output`: tensor of tf.dtype values

One of the things you get from declarative op descriptions is the documentation that goes with the dialect.

Generated C++ Code

- C++ class TF::LeakyReluOp

```
namespace TF {
class **LeakyReluOp**
    : public Op<LeakyReluOp,
                OpTrait::OneResult,
                OpTrait::HasNoSideEffect,
                OpTrait::SameOperandsAndResultType,
                OpTrait::OneOperand> {
 public:
  static StringRef getOperationName() {
    return "tf.LeakyRelu";
  };








  ...
};
} // end namespace
```

Generated C++ Code: Typed accessors

- C++ class TF::LeakyReluOp
- Typed accessors:
  - value() and alpha()

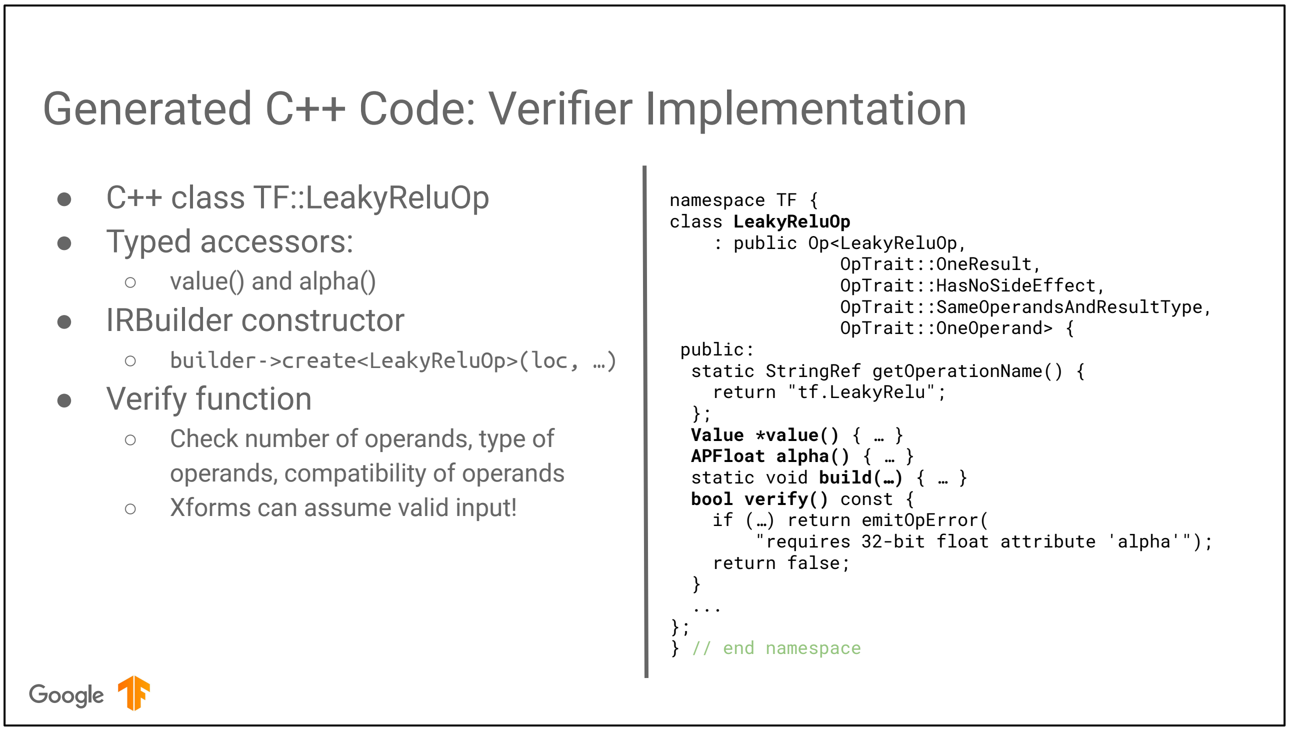
```
namespace TF {
class LeakyReluOp
    : public Op<LeakyReluOp,
                OpTrait::OneResult,
                OpTrait::HasNoSideEffect,
                OpTrait::SameOperandsAndResultType,
                OpTrait::OneOperand> {
 public:
  static StringRef getOperationName() {
    return "tf.LeakyRelu";
  };
  Value *value() { … }
  APFloat alpha() { … }







  ...
};
} // end namespace
```

Generated C++ Code: Typed IRBuilder constructor

- C++ class TF::LeakyReluOp
- Typed accessors:
    - value() and alpha()
- IRBuilder constructor
    - `builder->create<LeakyReluOp>(loc, …)`

```
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() { … }
  static void build(…) { … }





  ...
};
} // end namespace
```

Google



Generated C++ Code: Verifier Implementation

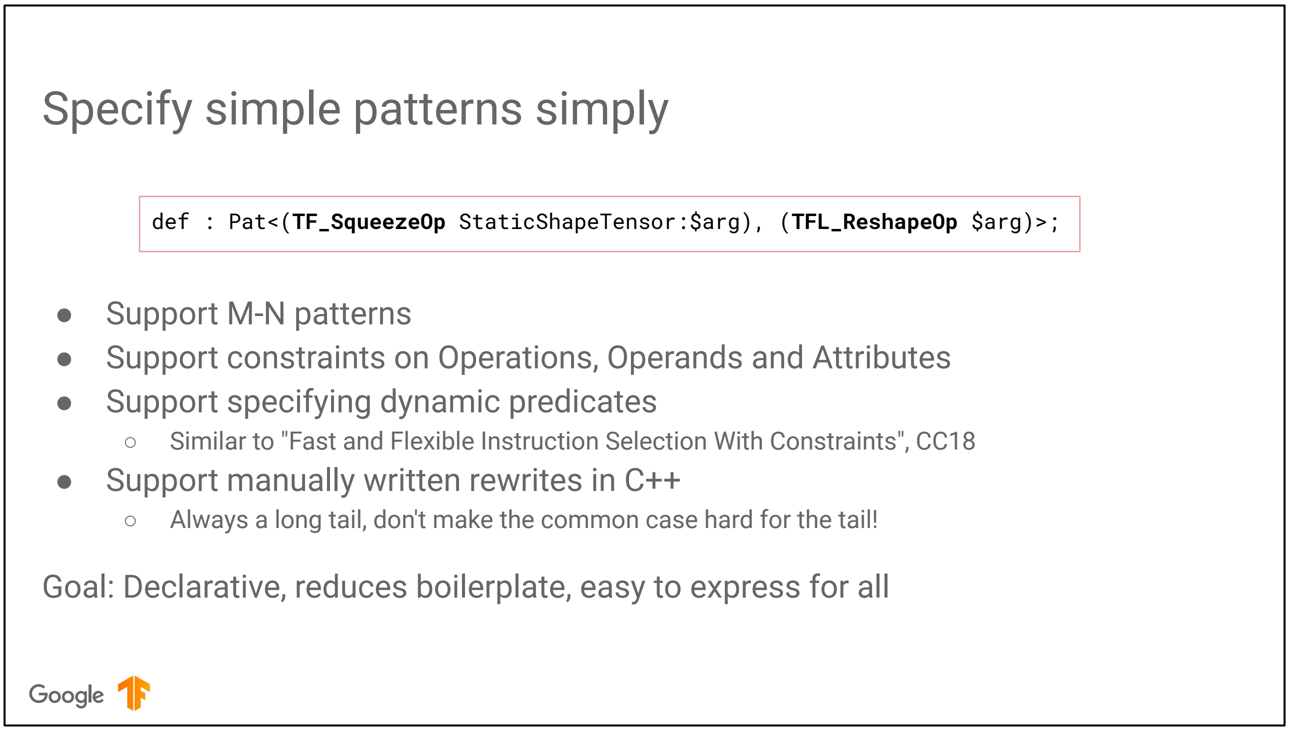
- C++ class TF::LeakyReluOp
- Typed accessors:
  - value() and alpha()
- IRBuilder constructor
  - `builder->create<LeakyReluOp>(loc, …)`
- Verify function
  - Check number of operands, type of operands, compatibility of operands
  - Xforms can assume valid input!

```
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() { … }
  static void build(…) { … }
  bool verify() const {
    if (…) return emitOpError(
        "requires 32-bit float attribute 'alpha'");
    return false;
  }
  ...
};
} // end namespace
```

Google

Another thing you get is a C++ class that corresponds to the op. This allows type safe dynamic casting to the class (instead of stringly defined matches), and the class has all the named accessors and other stuff that powers the operation.

You can still poke at the low level “mlir::Operation*” class (which corresponds to “llvm::Instruction*”) and use things like getOperand(4) if you have reason to.



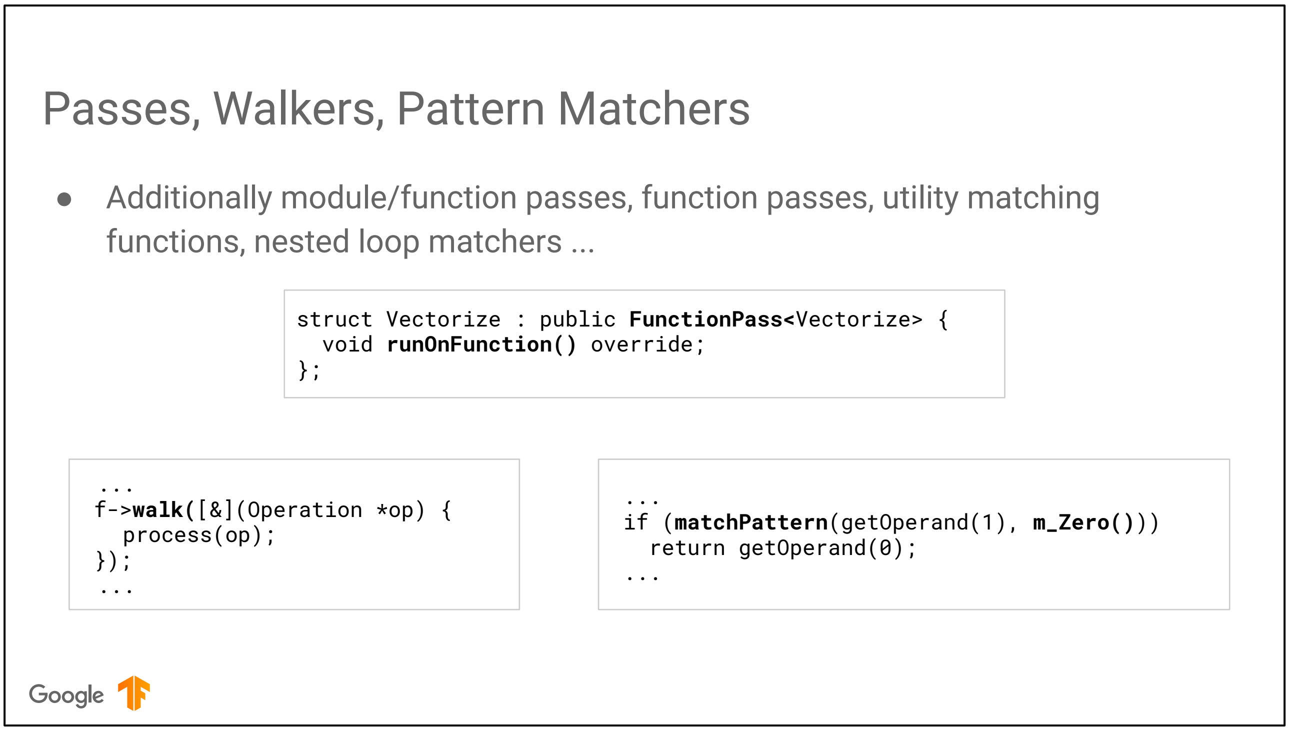
Specify simple patterns simply

```
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 "Fast and Flexible Instruction Selection With Constraints", CC18
- Support manually written rewrites in C++
  - 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 boilerplate and make transforms easy to express.



Passes, Walkers, Pattern Matchers

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

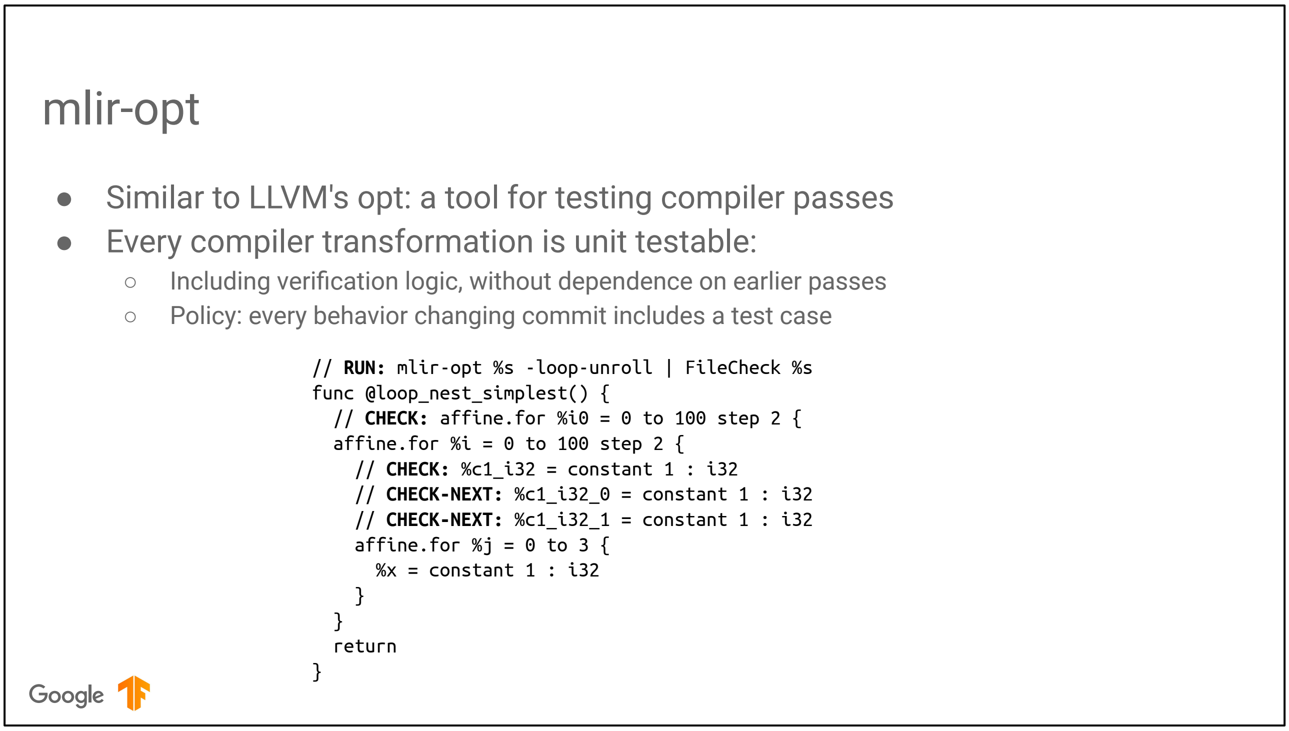
```
struct Vectorize : public FunctionPass<Vectorize> {
  void runOnFunction() override;
};
```

```
...
f->walk([&](Operation *op) {
   process(op);
});
...
```

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

Google

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.

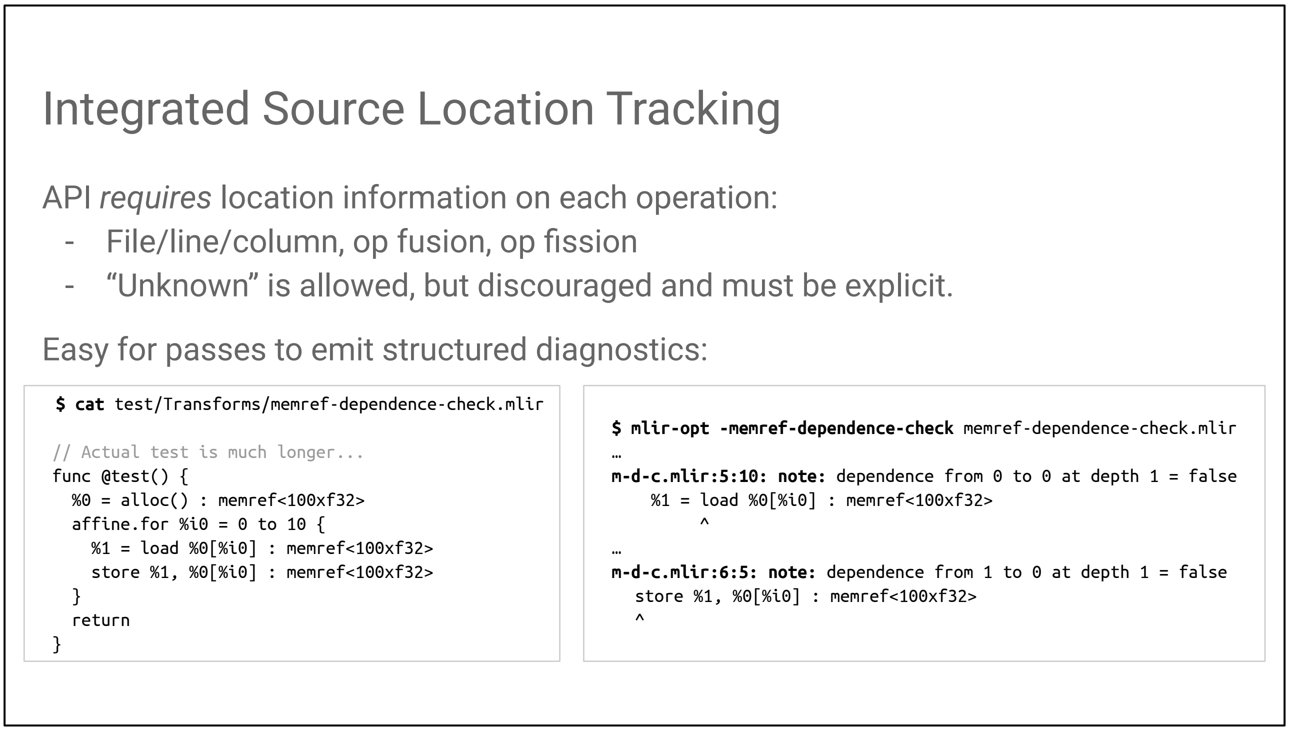


mlir-opt

- 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

```
// RUN: mlir-opt %s -loop-unroll | FileCheck %s
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
    affine.for %j = 0 to 3 {
      %x = constant 1 : i32
    }
  }
  return
}
```

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



Integrated Source Location Tracking

API *requires* location information on each operation:

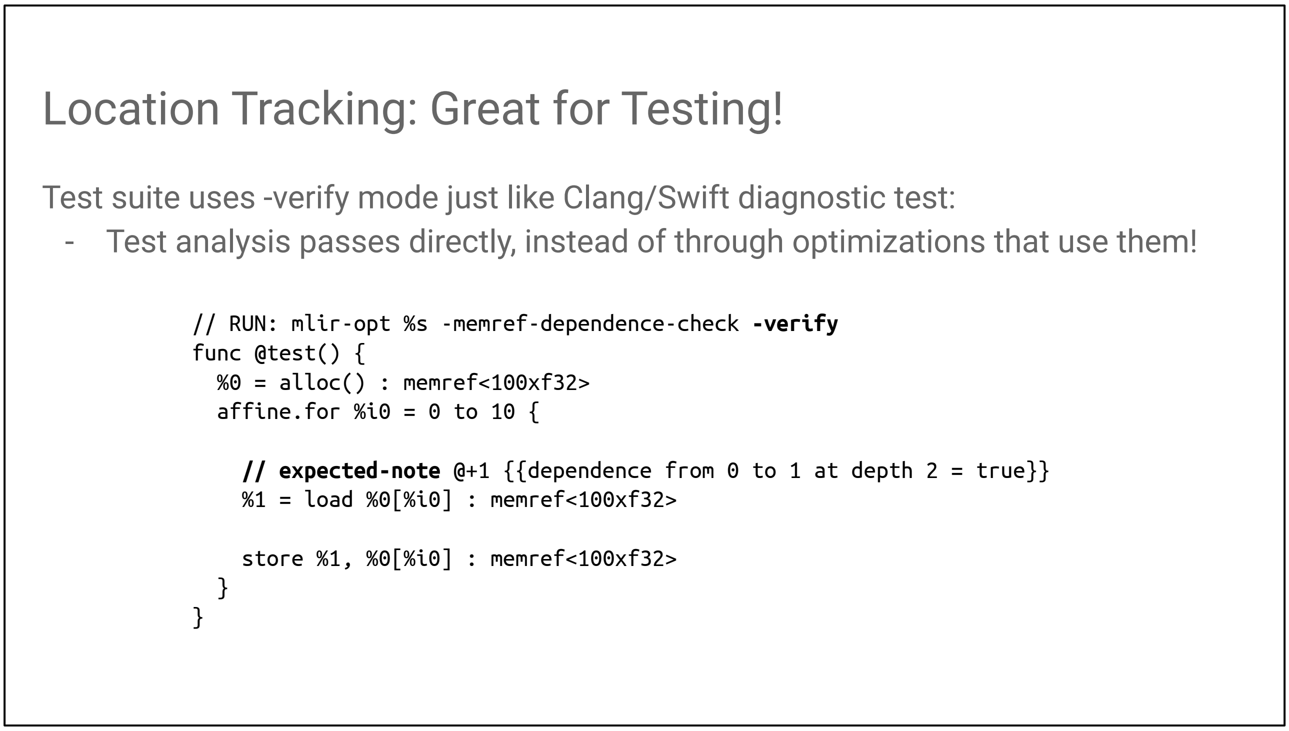
- File/line/column, op fusion, op fission
- “Unknown” is allowed, but discouraged and must be explicit.

Easy for passes to emit structured diagnostics:

```
$ cat test/Transforms/memref-dependence-check.mlir

// Actual test is much longer...
func @test() {
  %0 = alloc() : memref<100xf32>
  affine.for %i0 = 0 to 10 {
    %1 = load %0[%i0] : memref<100xf32>
    store %1, %0[%i0] : memref<100xf32>
  }
  return
}
```

```
$ mlir-opt -memref-dependence-check memref-dependence-check.mlir
…
m-d-c.mlir:5:10: note: dependence from 0 to 0 at depth 1 = false
    %1 = load %0[%i0] : memref<100xf32>
         ^
…
m-d-c.mlir:6:5: note: dependence from 1 to 0 at depth 1 = false
  store %1, %0[%i0] : memref<100xf32>
  ^
```

Location tracking is 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. Normally, a frontend would add source location info, but the MLIR asmparser adds location information pointing to the .mlir file if none is present, this is very convenient in practice.



Location Tracking: Great for Testing!

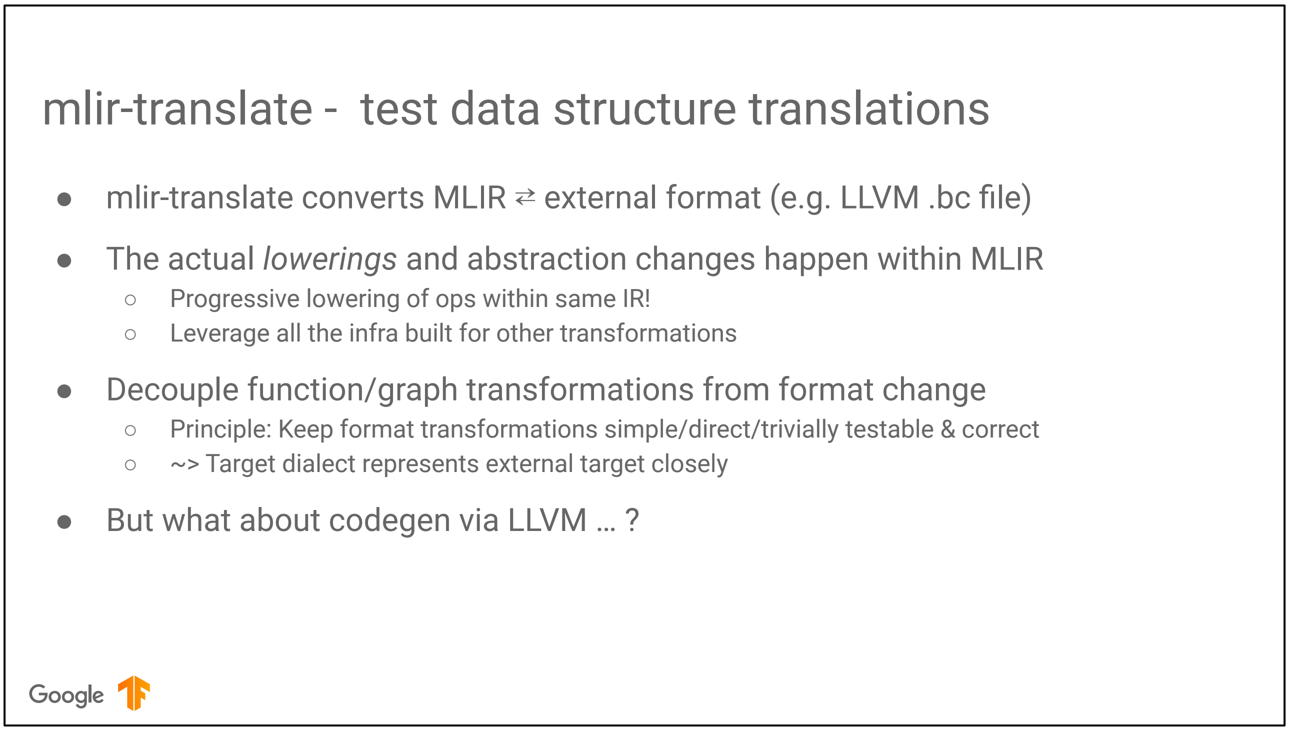
Test suite uses -verify mode just like Clang/Swift diagnostic test:

- Test analysis passes directly, instead of through optimizations that use them!

```
// RUN: mlir-opt %s -memref-dependence-check -verify
func @test() {
  %0 = alloc() : memref<100xf32>
  affine.for %i0 = 0 to 10 {

    // expected-note @+1 {{dependence from 0 to 1 at depth 2 = true}}
    %1 = load %0[%i0] : memref<100xf32>

    store %1, %0[%i0] : memref<100xf32>
  }
}
```

Because location tracking is integral, we can also build the testsuite to depend on it, and use it to test analysis passes. 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 an key part of our design, and we take it further than LLVM did.

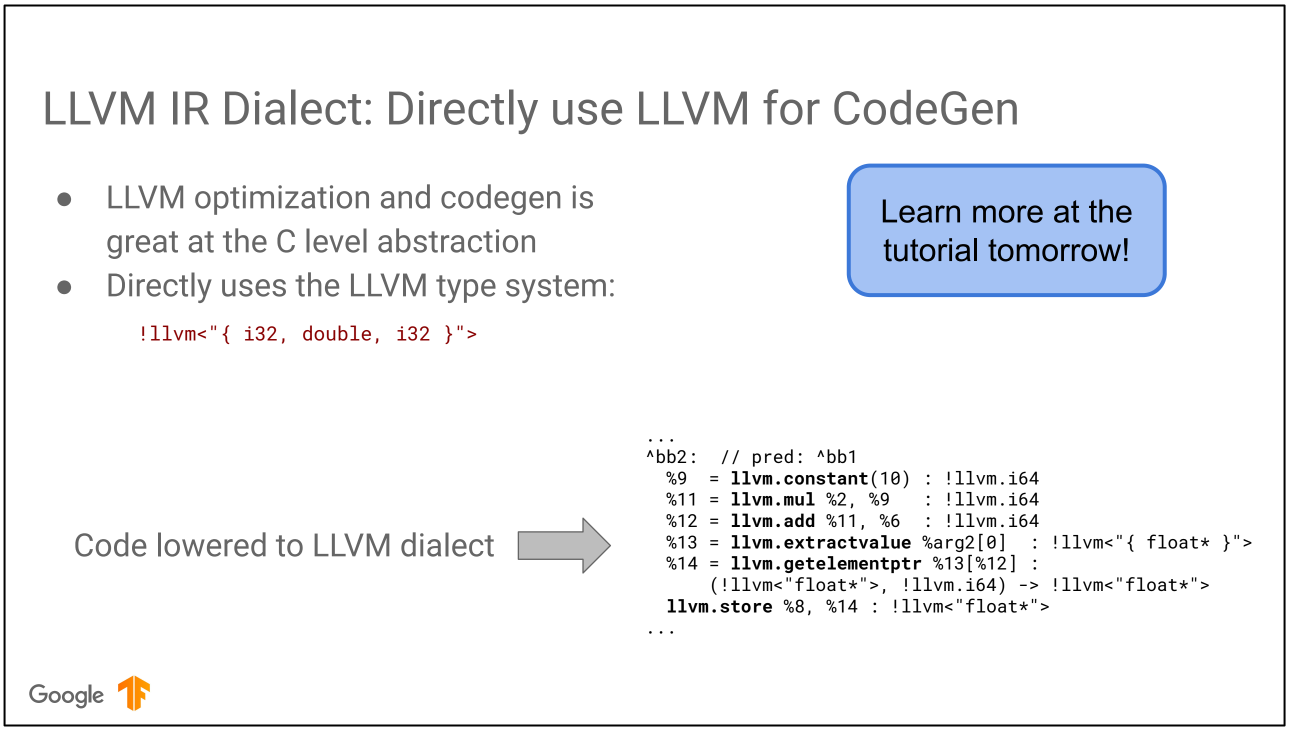


mlir-translate - test data structure translations

- mlir-translate converts MLIR ⇄ external format (e.g. LLVM .bc file)
- The actual *lowerings* and abstraction changes happen within MLIR
  - Progressive lowering of ops within same IR!
  - Leverage all the infra built for other transformations
- Decouple function/graph transformations from format change
  - Principle: Keep format transformations simple/direct/trivially testable & correct
  - ~> Target dialect represents external target closely
- But what about codegen via LLVM … ?

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 the do all lowering within MLIR 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 IR Dialect: Directly use LLVM for CodeGen

- LLVM optimization and codegen is great at the C level abstraction
- Directly uses the LLVM type system:

```
!llvm<"{ i32, double, i32 }">
```

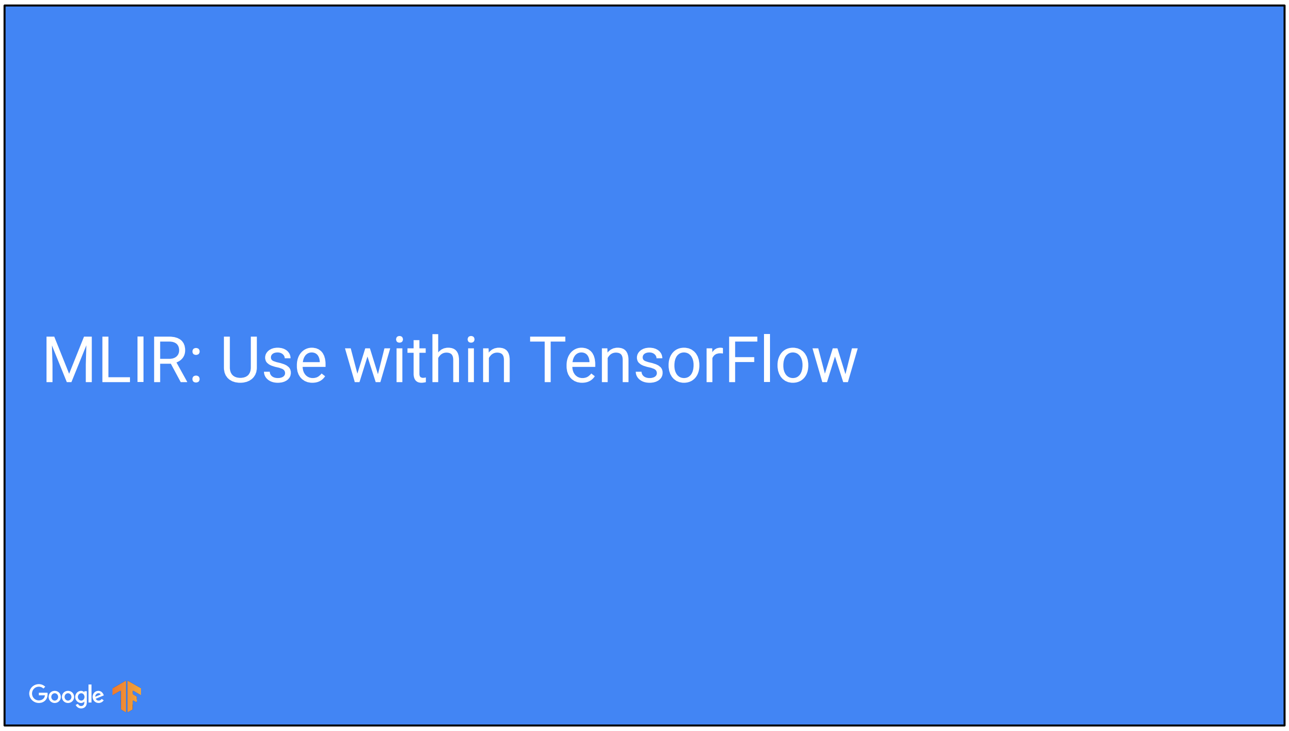
Learn more at the tutorial tomorrow!

Code lowered to LLVM dialect

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

Google

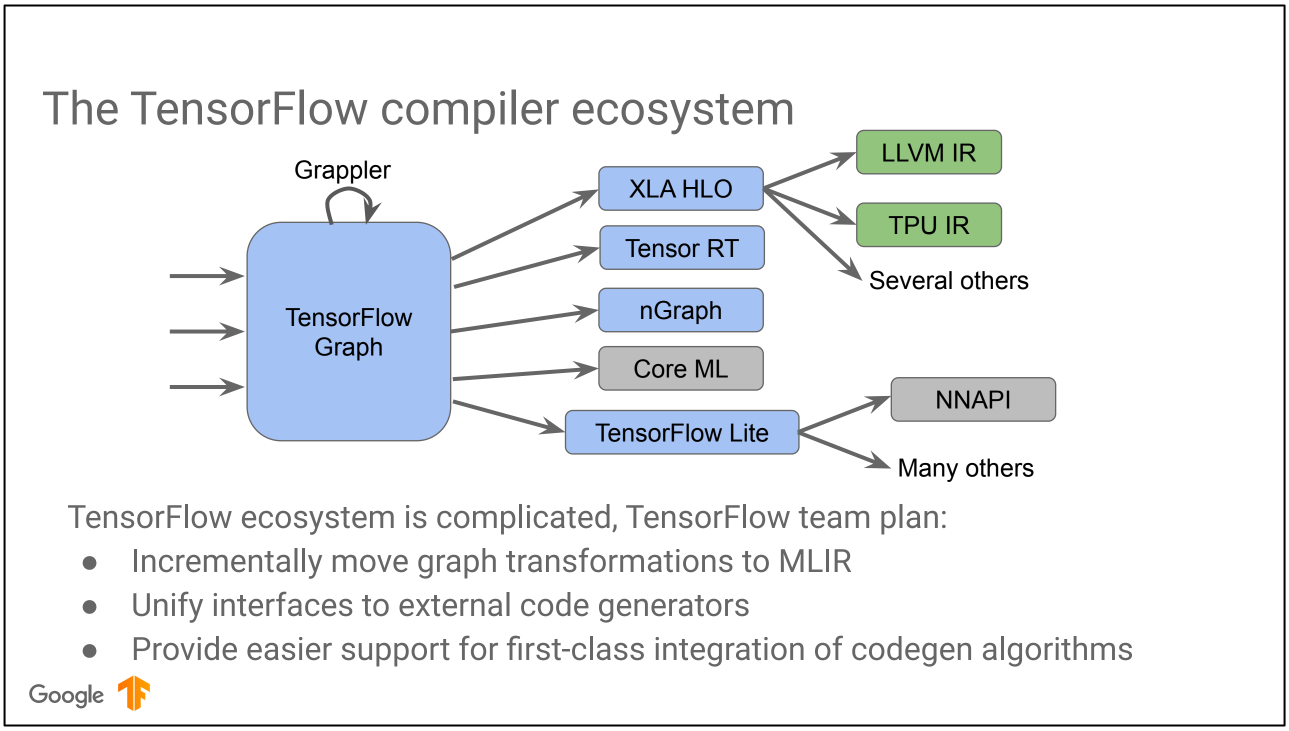
Of course, LLVM is great for C level optimization and code generation to CPUs and PTX, and as such we have an LLVM IR dialect in MLIR that we lower to, which is isomorphic to LLVM IR. This is only used as a lowering step, we don't expect people to be reimplementing existing LLVM IR optimizations on this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



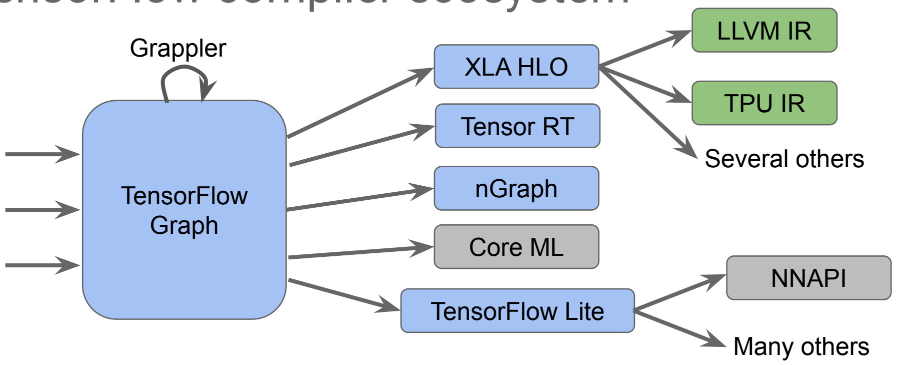
MLIR: Use within TensorFlow

Google

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



The TensorFlow compiler ecosystem



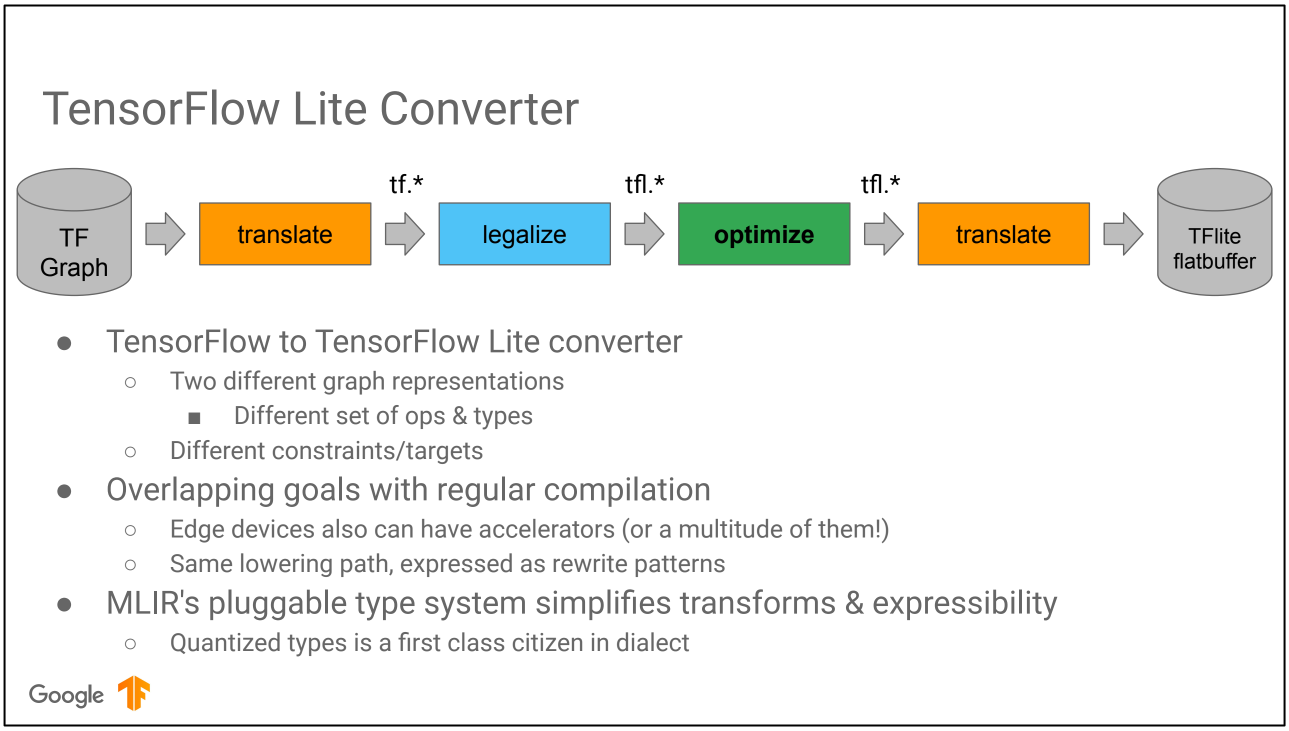
TensorFlow ecosystem is complicated, TensorFlow team plan:

- Incrementally move graph transformations to MLIR
- Unify interfaces to external code generators
- Provide easier support for first-class integration of codegen algorithms

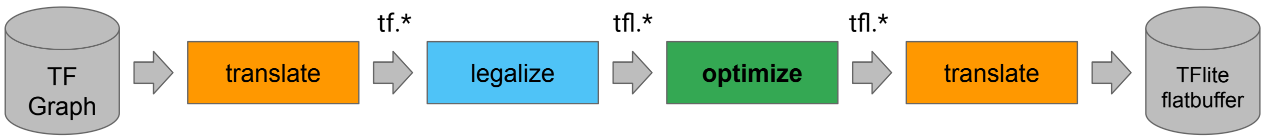
Google

<<intentionally the same diagram as before>>

Coming back to our discussion of TensorFlow's ecosystem, there is a lot going on here. MLIR is the path forward for TensorFlow infrastructure and we are now well on our way to merge together an integrate various subsystems in the ecosystem - let's talk about a few of those.



TensorFlow Lite Converter

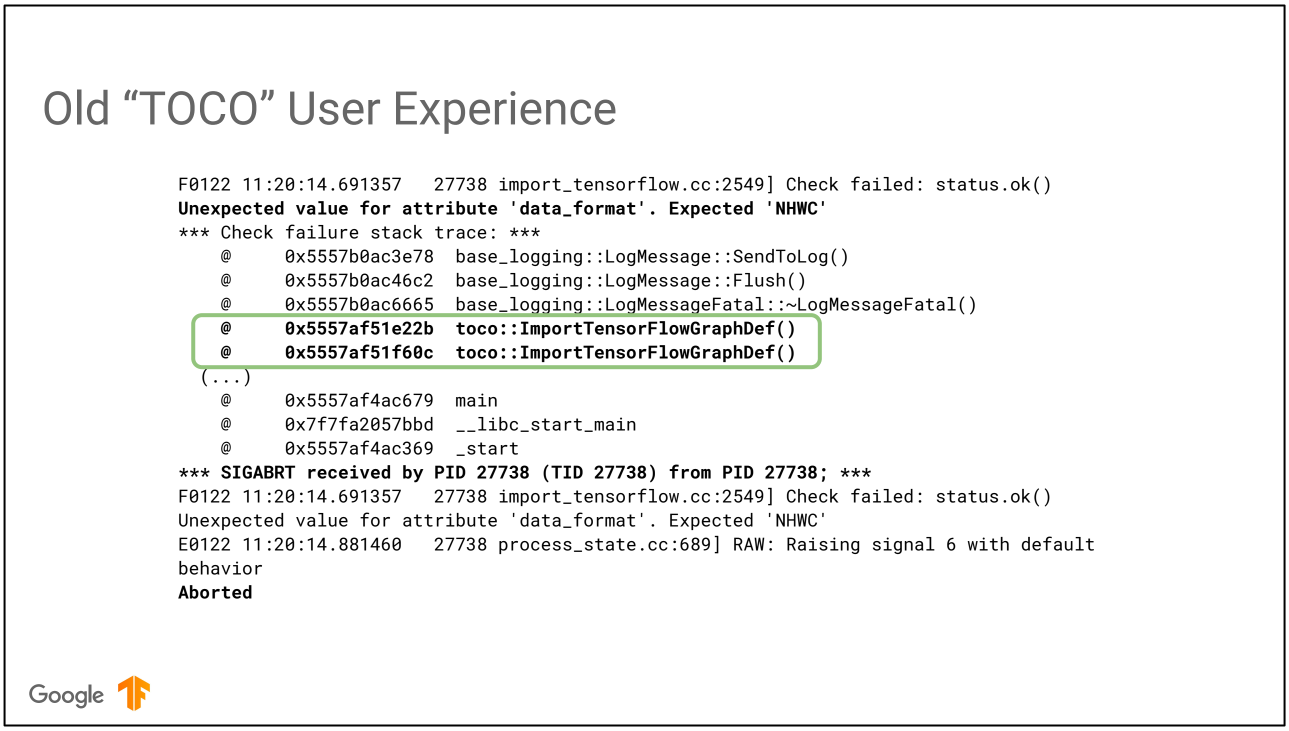


- TensorFlow to TensorFlow Lite converter
  - Two different graph representations
    - Different set of ops & types
  - Different constraints/targets
- Overlapping goals with regular compilation
  - Edge devices also can have accelerators (or a multitude of them!)
  - Same lowering path, expressed as rewrite patterns
- MLIR's pluggable type system simplifies 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.

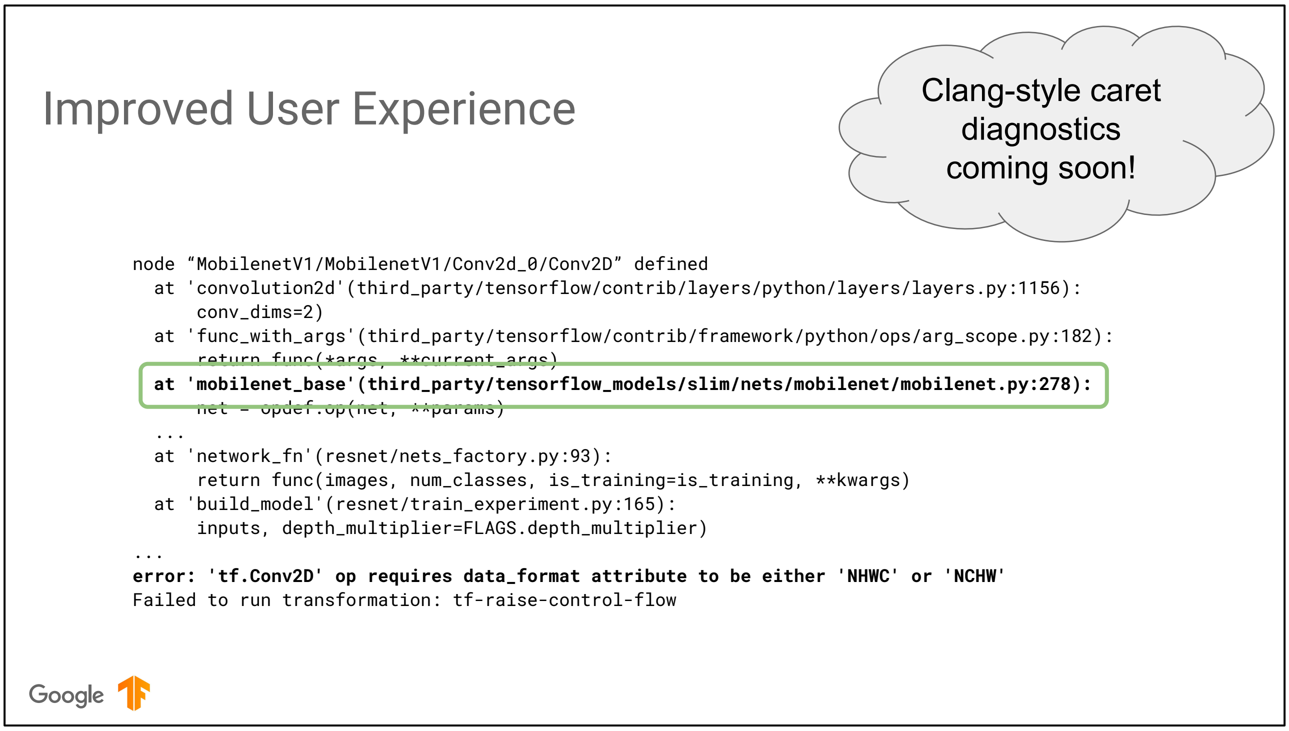
Moving it to MLIR makes it easier to test and develop, and the user experience is dramatically improved due to the location tracking in MLIR.



Old “TOCO” User Experience

```
F0122 11:20:14.691357   27738 import_tensorflow.cc:2549] Check failed: status.ok()
Unexpected value for attribute 'data_format'. Expected 'NHWC'
*** Check failure stack trace: ***
    @     0x5557b0ac3e78  base_logging::LogMessage::SendToLog()
    @     0x5557b0ac46c2  base_logging::LogMessage::Flush()
    @     0x5557b0ac6665  base_logging::LogMessageFatal::~LogMessageFatal()
    @     0x5557af51e22b  toco::ImportTensorFlowGraphDef()
    @     0x5557af51f60c  toco::ImportTensorFlowGraphDef()
  (...)
    @     0x5557af4ac679  main
    @     0x7f7fa2057bbd  __libc_start_main
    @     0x5557af4ac369  _start
*** SIGABRT received by PID 27738 (TID 27738) from PID 27738; ***
F0122 11:20:14.691357   27738 import_tensorflow.cc:2549] Check failed: status.ok()
Unexpected value for attribute 'data_format'. Expected 'NHWC'
E0122 11:20:14.881460   27738 process_state.cc:689] RAW: Raising signal 6 with default
behavior
Aborted
```

The current TOCO (TFLite) converter is crashing on valid (but unsupported) user input. The backtrace is hard to understand and to trace back to the faulty source line.

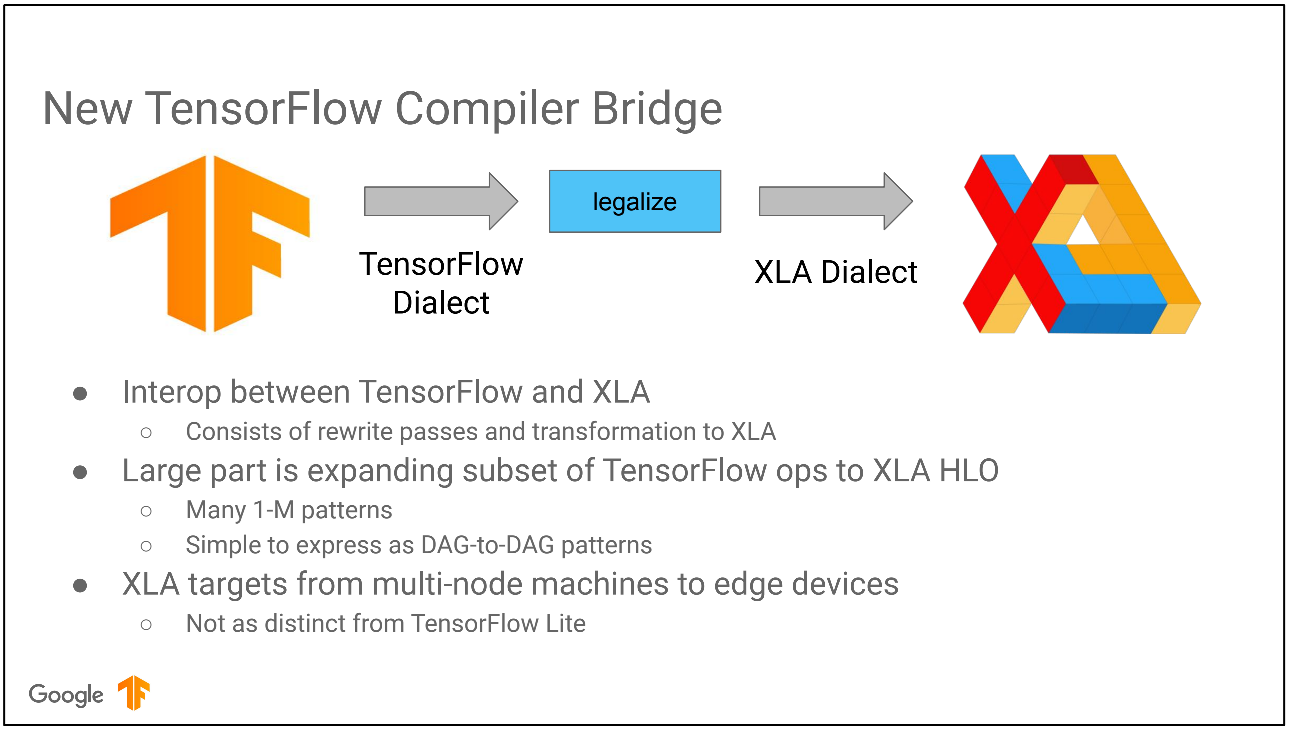


Improved User Experience

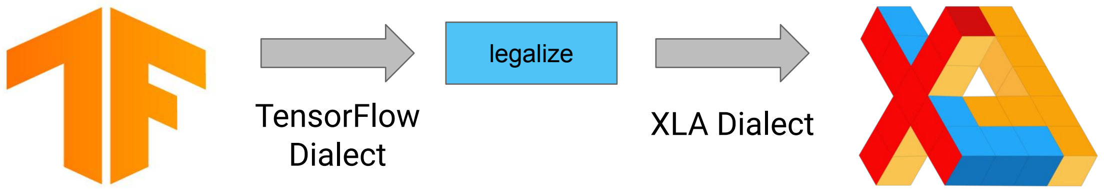
Clang-style caret diagnostics coming soon!

```
node “MobilenetV1/MobilenetV1/Conv2d_0/Conv2D” defined
  at 'convolution2d'(third_party/tensorflow/contrib/layers/python/layers/layers.py:1156):
      conv_dims=2)
  at 'func_with_args'(third_party/tensorflow/contrib/framework/python/ops/arg_scope.py:182):
      return func(*args, **current_args)
  at 'mobilenet_base'(third_party/tensorflow_models/slim/nets/mobilenet/mobilenet.py:278):
      net = opdef.op(net, **params)
  ...
  at 'network_fn'(resnet/nets_factory.py:93):
      return func(images, num_classes, is_training=is_training, **kwargs)
  at 'build_model'(resnet/train_experiment.py:165):
      inputs, depth_multiplier=FLAGS.depth_multiplier)
...
error: 'tf.Conv2D' op requires data_format attribute to be either 'NHWC' or 'NCHW'
Failed to run transformation: tf-raise-control-flow
```

Great source location tracking location, and stronger invariants in the compiler give us a much better user experience -- actually telling the user what is wrong **in their code**.

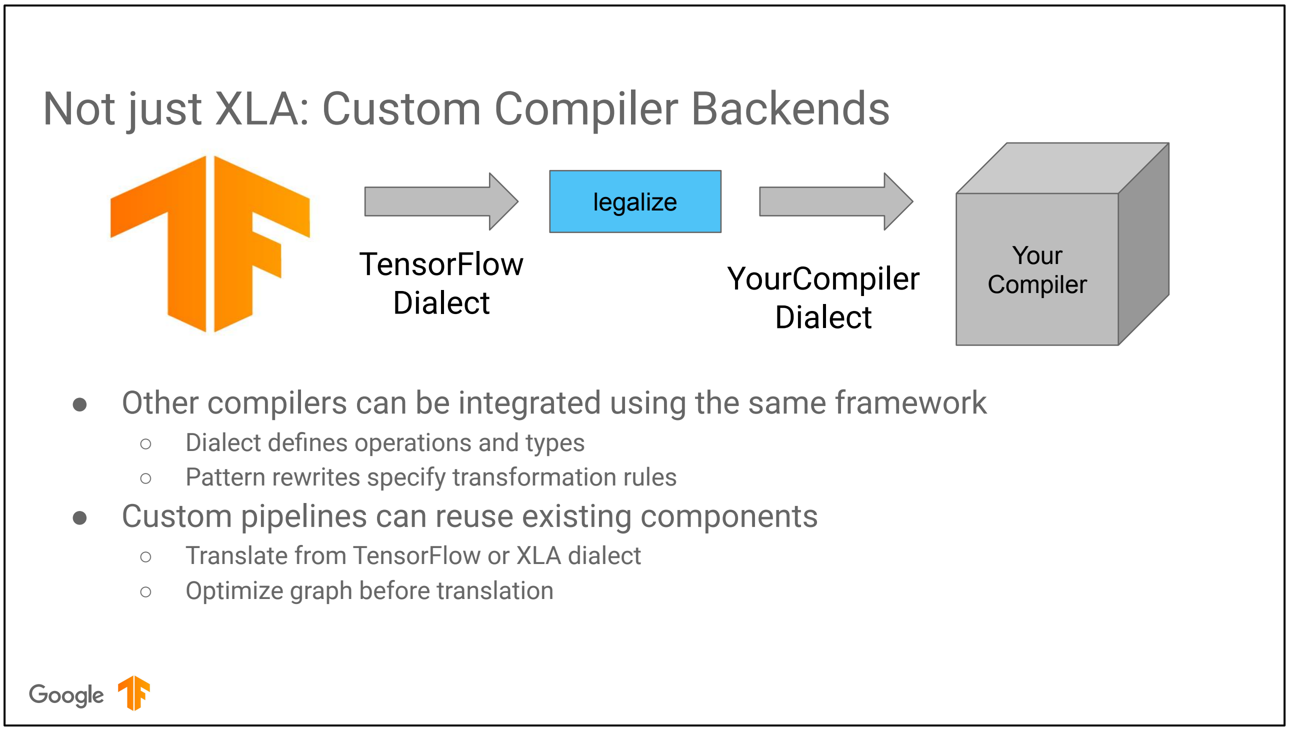


New TensorFlow Compiler Bridge

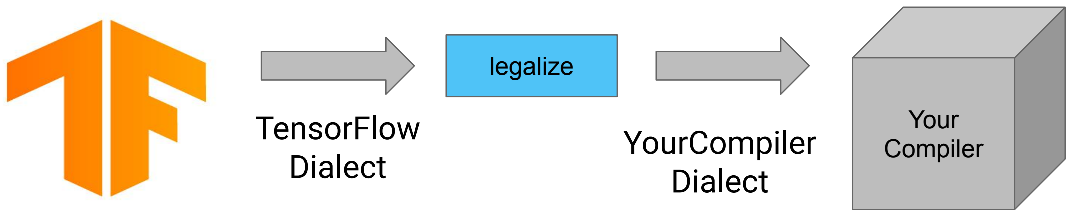


- Interop between TensorFlow and XLA
  - Consists of rewrite passes and transformation to XLA
- Large part is expanding subset of TensorFlow ops to XLA HLO
  - Many 1-M patterns
  - Simple to express as DAG-to-DAG patterns
- XLA targets from multi-node machines to edge devices
  - Not as distinct from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



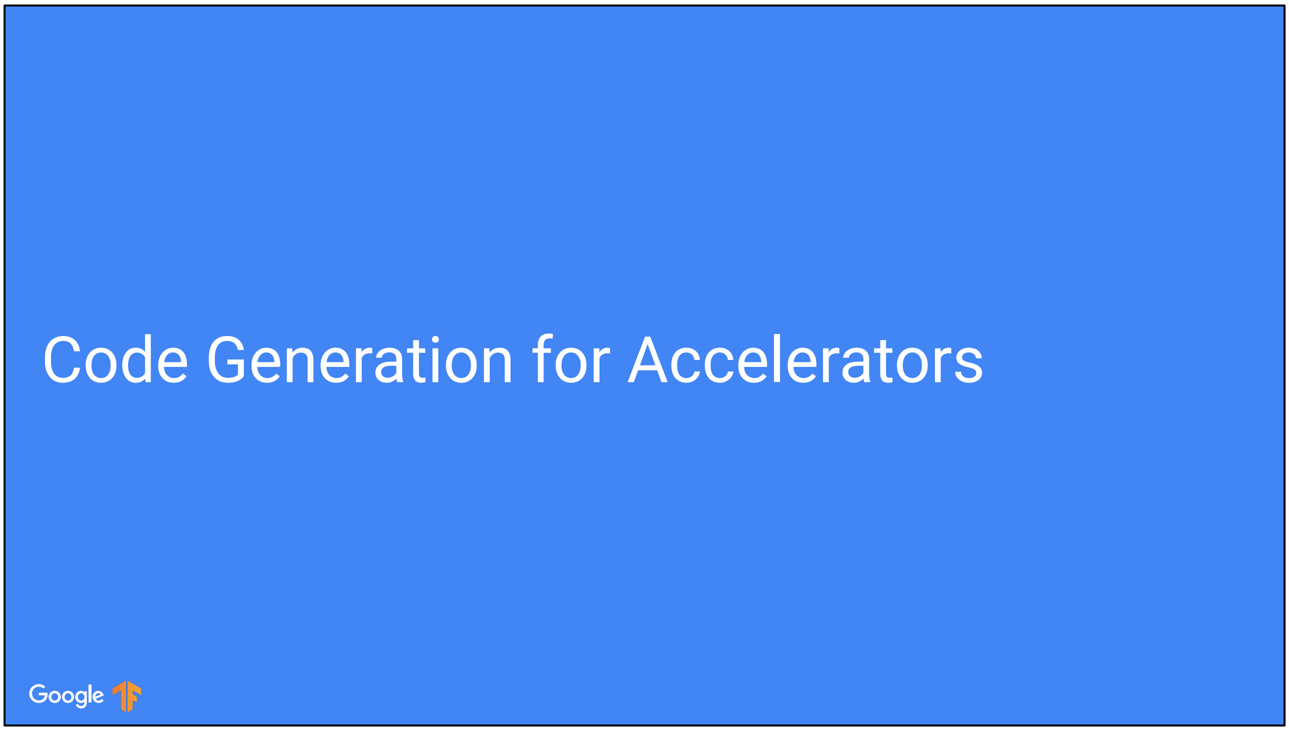
Not just XLA: Custom Compiler Backends



- Other compilers can be integrated using the same framework
  - Dialect defines operations and types
  - Pattern rewrites specify transformation rules
- Custom pipelines can reuse existing components
  - Translate from TensorFlow or XLA dialect
  - Optimize graph before translation

Google

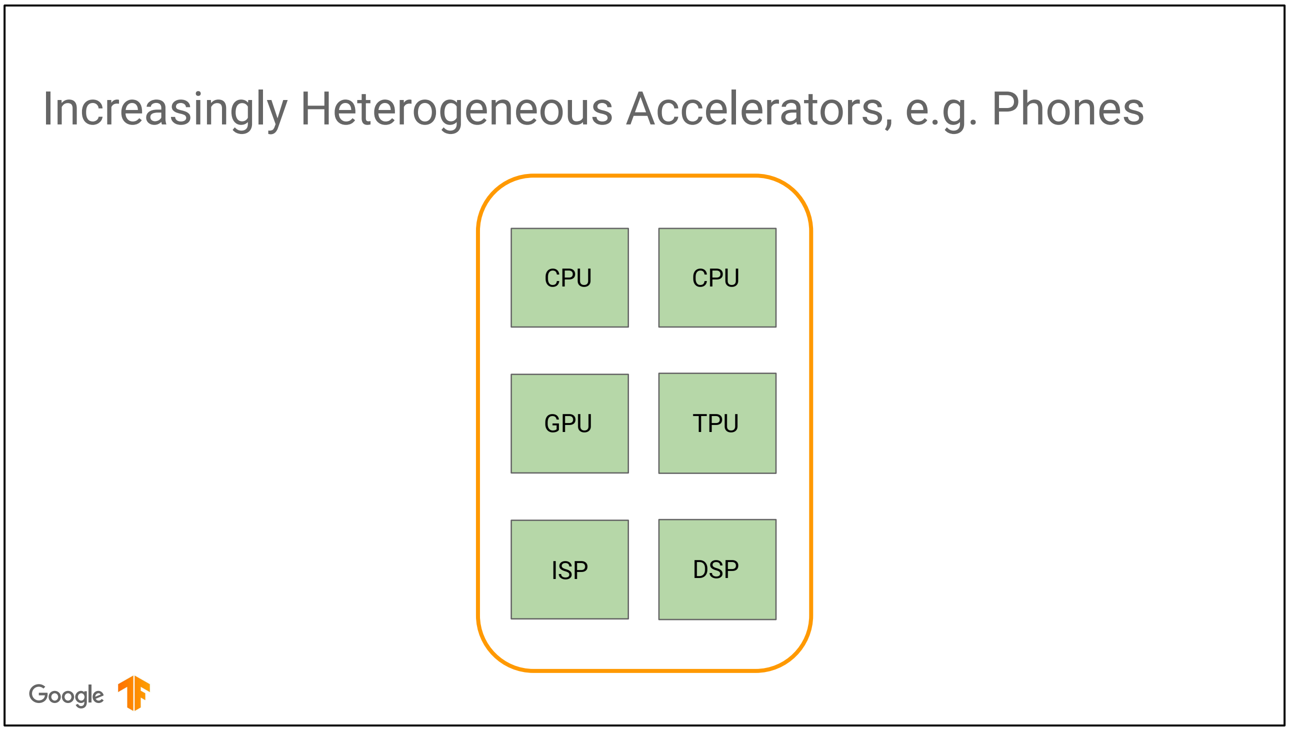
Adding support for a new backend compiler is not that different from TF/XLA. We can reuse the same infrastructure.



Code Generation for Accelerators

Google

Next, let’s talk about code generation algorithms for domain specific accelerators



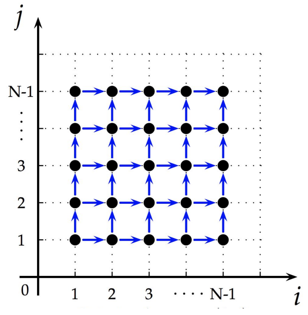
Increasingly Heterogeneous Accelerators, e.g. Phones

Consider a cell phone: it has a CPU, usually with multiple cores, sometimes with big/little architectures.
It has a GPU, and nowadays it is increasingly common to have an inference accessorator.
Phones also have ISPs that accelerate image and video decode, DSPs and other specialized processors.

TensorFlow is a great way to describe the high level computation that spans multiple accelerators, but each of these accelerators requires a specialized compiler.

Tensor Codegen: Investing in Two Approaches

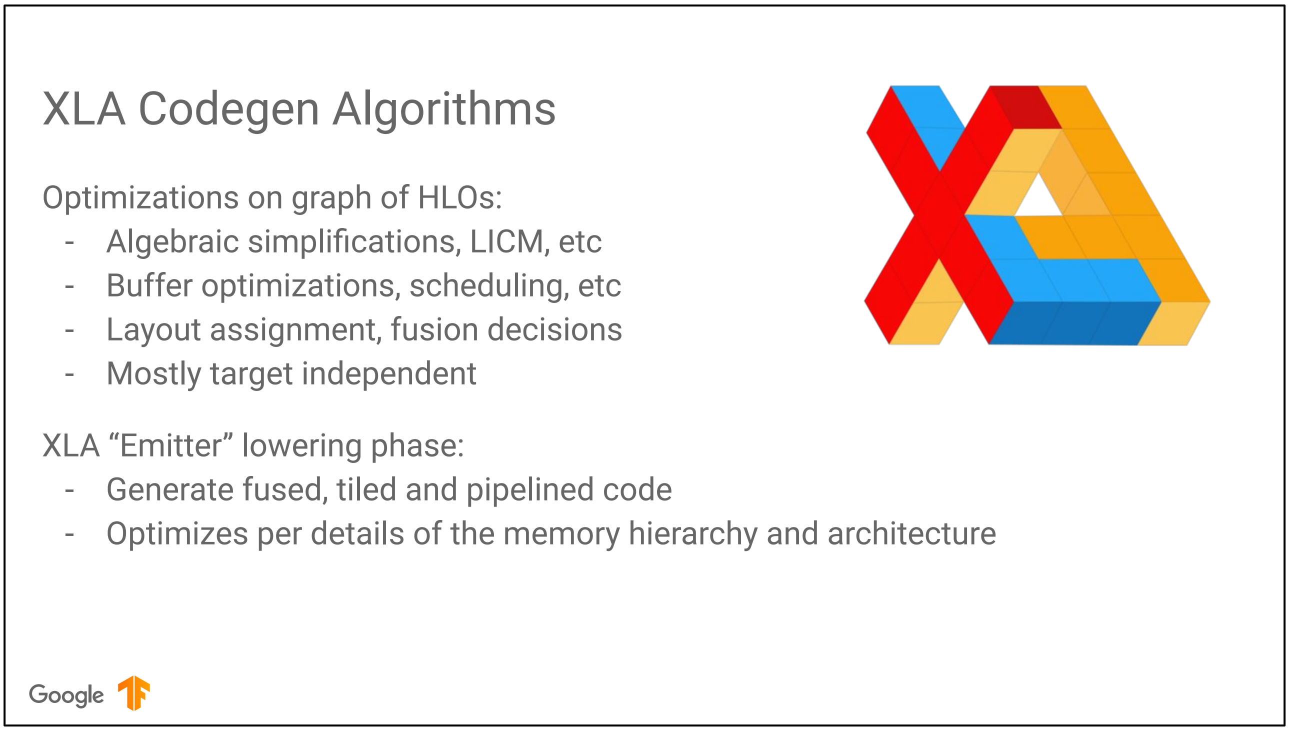
XLA Compiler Technology



Polyhedral Compiler Algorithms

Google

For the neural net accelerator part of the problem, we are investing in two different directions, which have different tradeoffs. Let’s talk about these a bit to understand how they work.



XLA Codegen Algorithms

Optimizations on graph of HLOs:

- Algebraic simplifications, LICM, etc
- Buffer optimizations, scheduling, etc
- Layout assignment, fusion decisions
- Mostly target independent

XLA “Emitter” lowering phase:

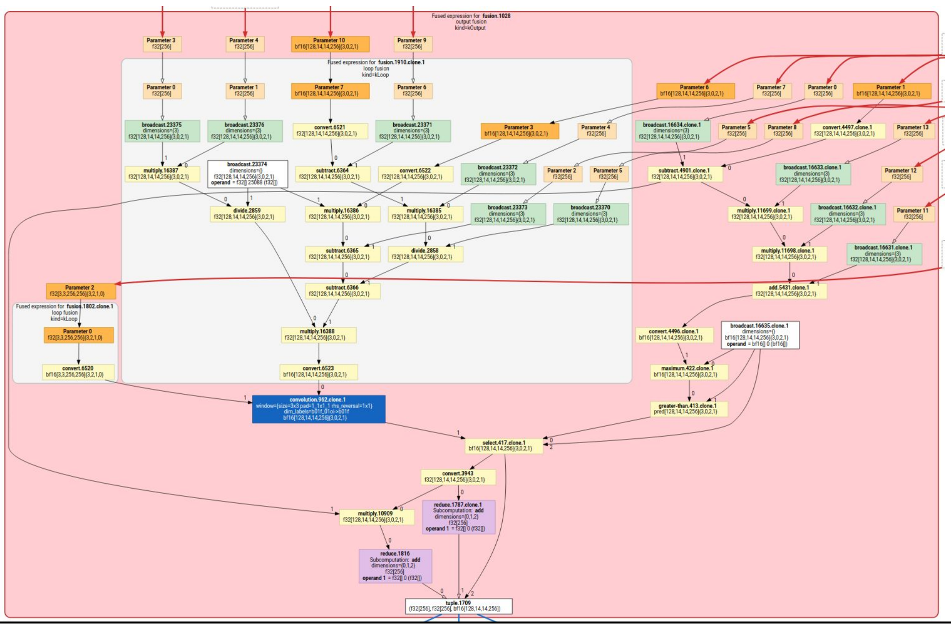
- Generate fused, tiled and pipelined code
- Optimizes per details of the memory hierarchy and architecture

Google

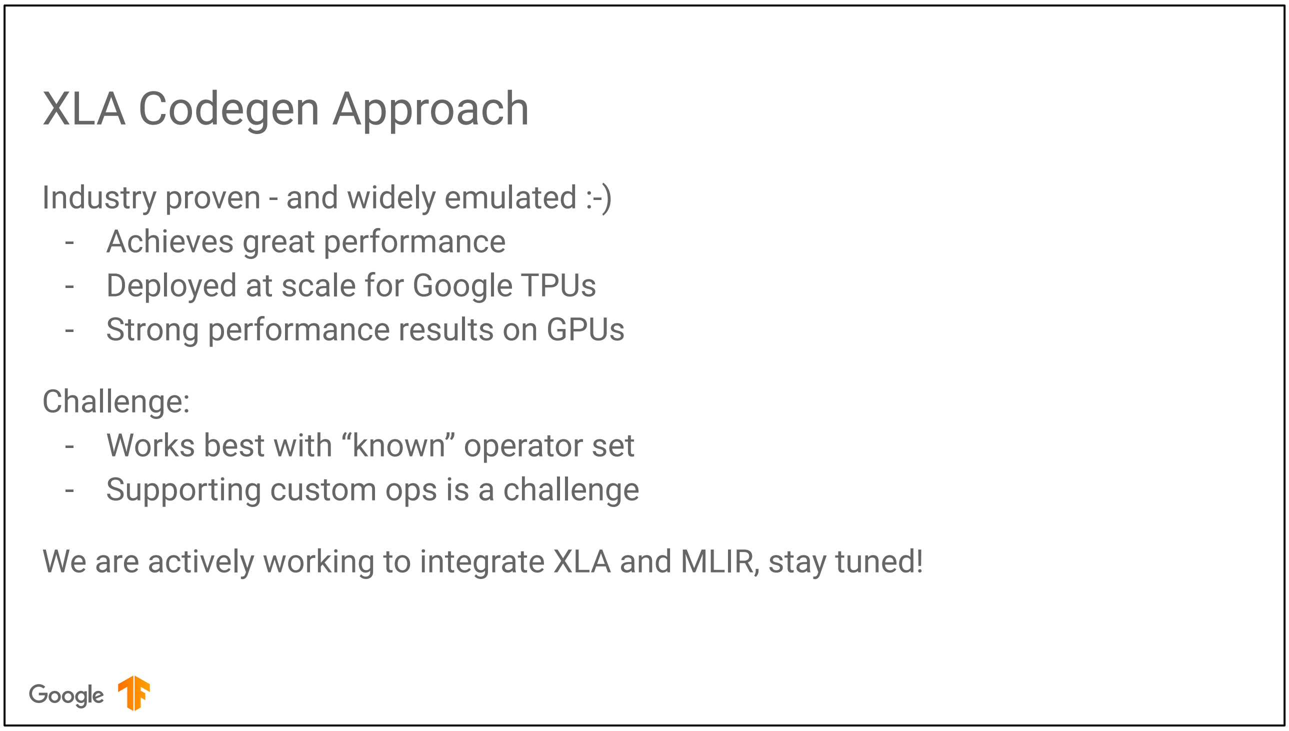
XLA is a critical part of the TensorFlow ecosystem. It’s codegen approach starts with a graph of “HLOs” in the tensor domain. It supports a number of well known compiler transformations in the Tensor domain, but its primary specialty is kernel fusion.

The core XLA algorithm of note here is the “expander” phase, which takes pre-fused XLA HLO graph and generates machine code.

Typical XLA Fusion Node



Fusion is a very important optimization, and XLA is capable of fusing very large and varied computations, which is critical for memory subsystem performance and balancing memory/compute ratios.



XLA Codegen Approach

Industry proven - and widely emulated :-)

- Achieves great performance
- Deployed at scale for Google TPUs
- Strong performance results on GPUs

Challenge:

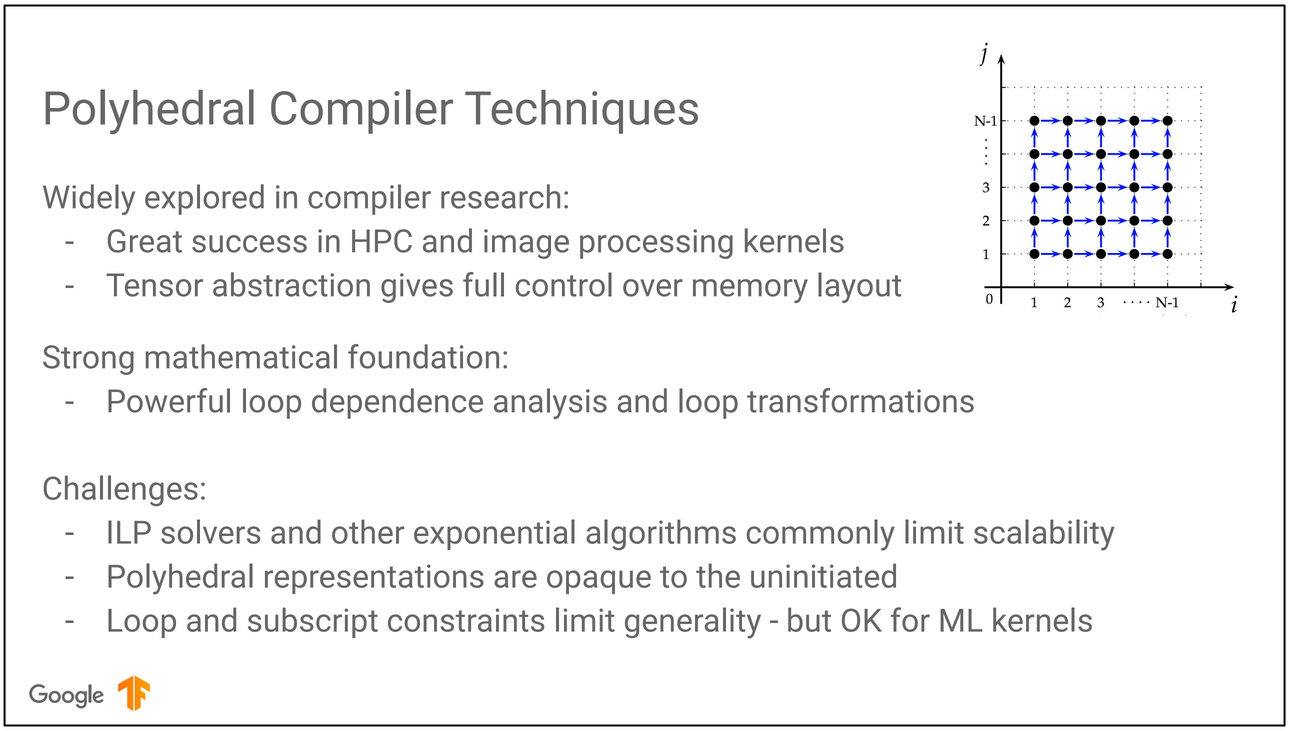
- Works best with “known” operator set
- Supporting custom ops is a challenge

We are actively working to integrate XLA and MLIR, stay tuned!

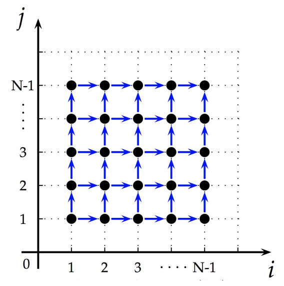
Google

XLA has been a huge success for us, particularly for TPUs. We are looking to generalize it to new architectures, and improve its flexibility.

One challenge is that we want researchers to be able to implement new operators, without having to hack up the compiler.



Polyhedral Compiler Techniques



Widely explored in compiler research:
- Great success in HPC and image processing kernels
- Tensor abstraction gives full control over memory layout

Strong mathematical foundation:
- Powerful loop dependence analysis and loop transformations

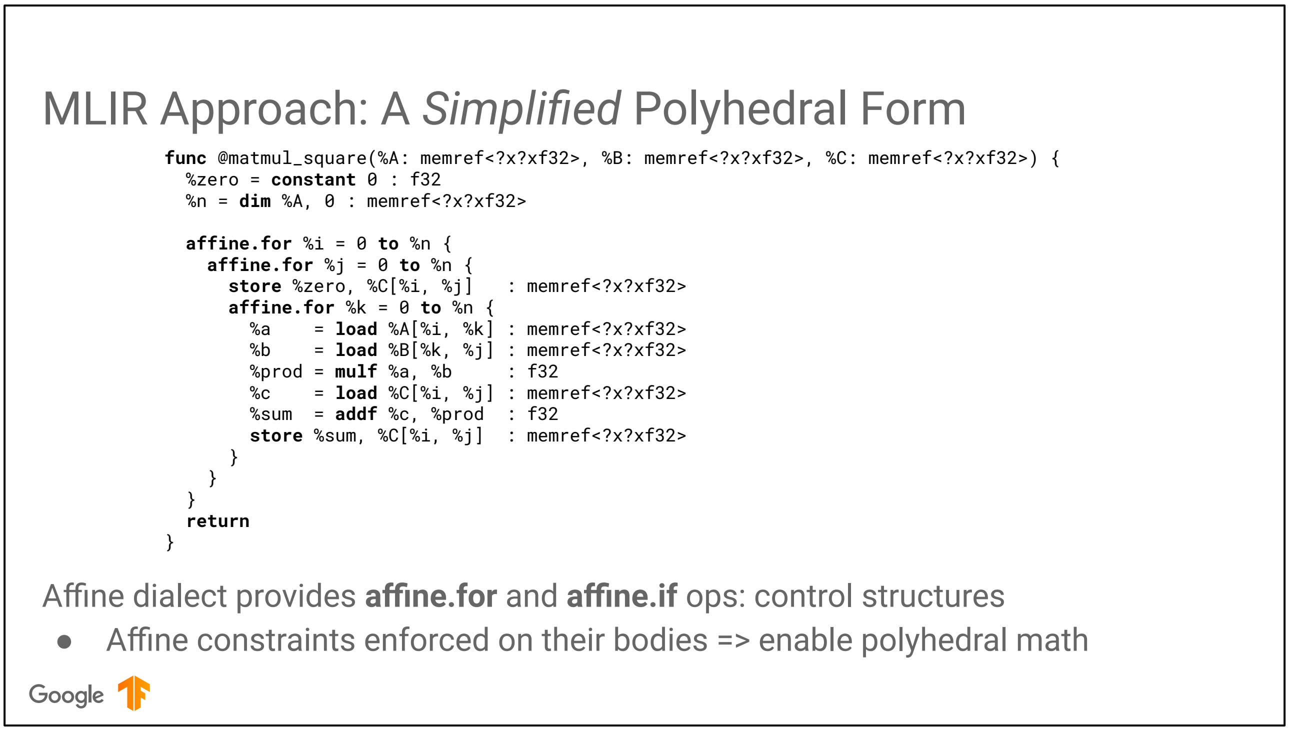
Challenges:
- ILP solvers and other exponential algorithms commonly limit scalability
- Polyhedral representations are opaque to the uninitiated
- Loop and subscript constraints limit generality - but OK for ML kernels

Google

As such, we're investing into something of a research direction, and hope to bring polyhedral compiler techniques mainstream.

If you aren't familiar, polyhedral techniques have been very successful as a compiler research direction for HPC and image processing applications. They have a lot of great attributes, including a strong theoretical basis.

Unfortunately, their adoption has been limited by scalability and some other issues.



MLIR Approach: A *Simplified* Polyhedral Form

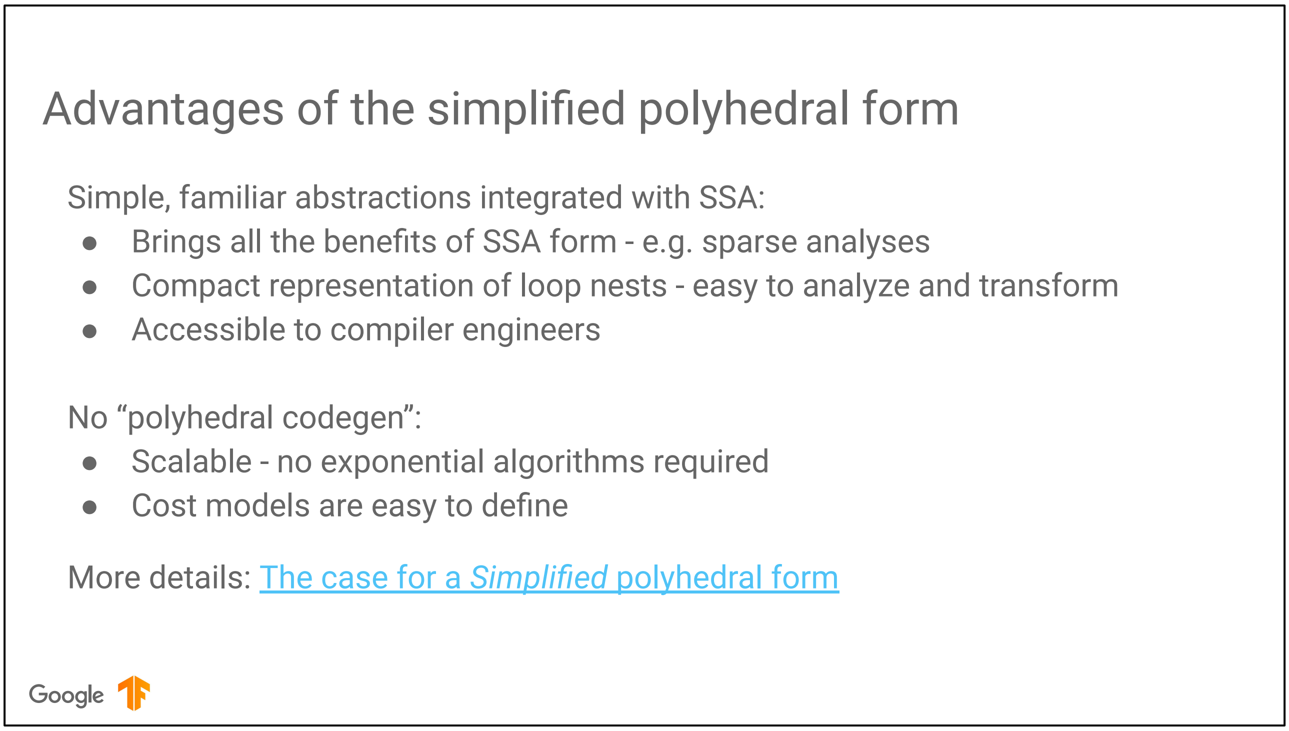
```
func @matmul_square(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %zero = constant 0 : f32
  %n = dim %A, 0 : memref<?x?xf32>

  affine.for %i = 0 to %n {
    affine.for %j = 0 to %n {
      store %zero, %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 dialect provides **affine.for** and **affine.if** ops: control structures

- Affine constraints enforced on their bodies => enable polyhedral math

Google

The MLIR approach to this is through the affine dialect, which defines some simple control structures. These provide polyhedral abstractions, and enforce the polyhedral requirements on their bodies. As you can see, this is very straightforward given our design.



Advantages of the simplified polyhedral form

Simple, familiar abstractions integrated with SSA:

- Brings all the benefits of SSA form - e.g. sparse analyses
- Compact representation of loop nests - easy to analyze and transform
- Accessible to compiler engineers

No “polyhedral codegen”:

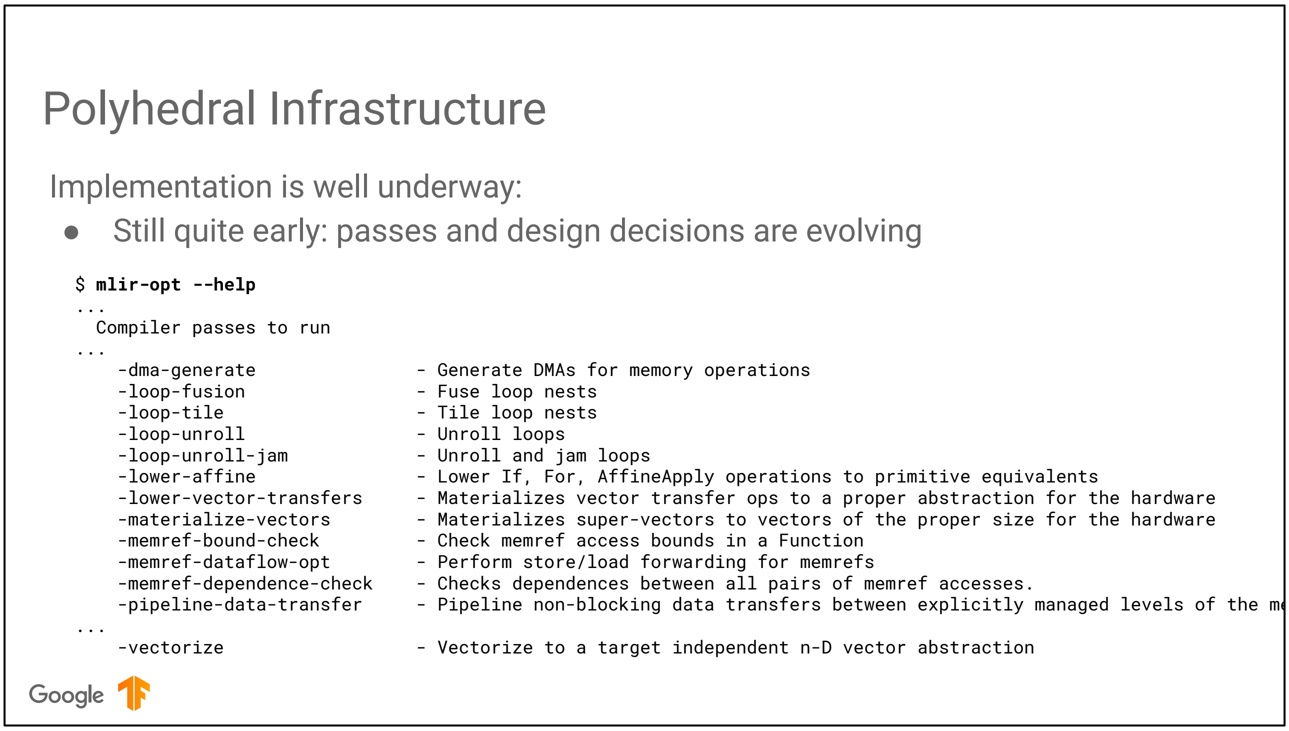
- Scalable - no exponential algorithms required
- Cost models are easy to define

More details: The case for a *Simplified* polyhedral form

Google

This approach has a number of advantages over traditional polyhedral forms - it is simple, SSA integrated (allowing standard SSA-based algorithms to apply to the same IR) and scalable. We don’t use ISL-style exponential time “polyhedral code generation”, which makes cost models easy to define in optimizer passes.

For more information, please see this whitepaper we have in our documentation directory.



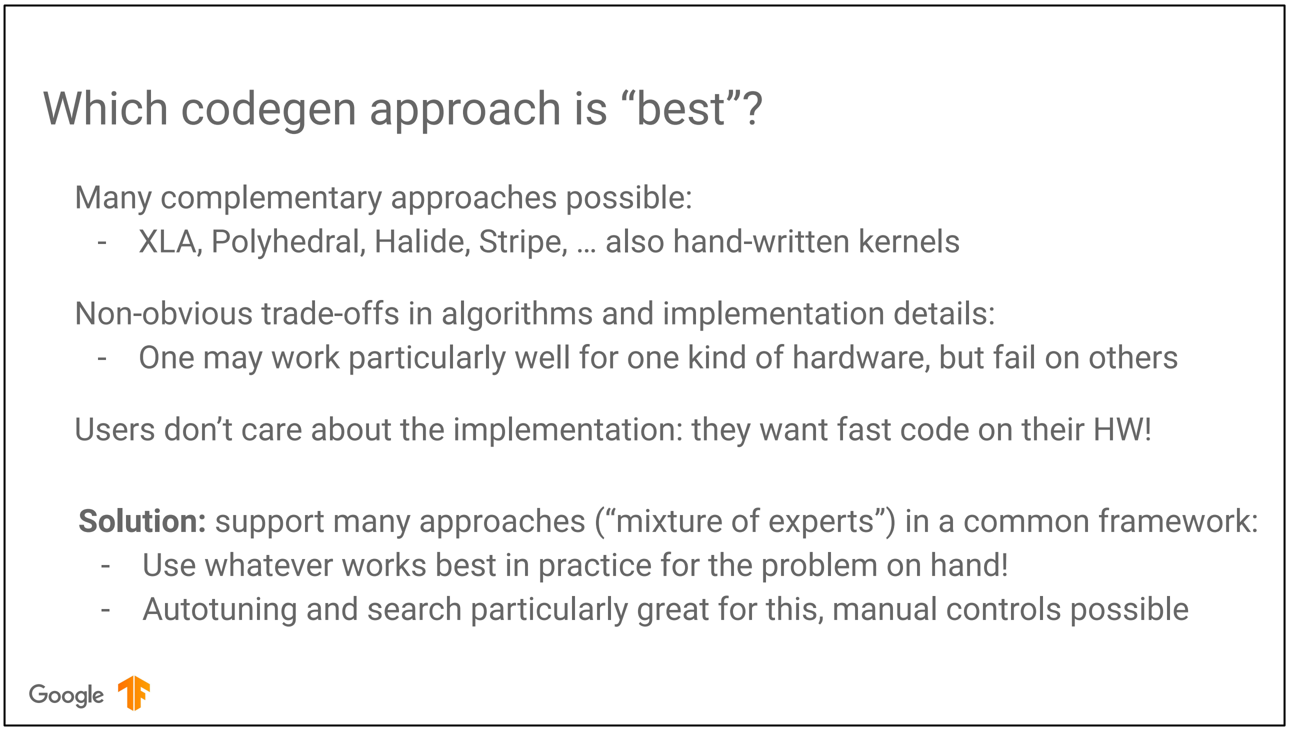
Polyhedral Infrastructure

Implementation is well underway:

- Still quite early: passes and design decisions are evolving

```
$ mlir-opt --help
...
  Compiler passes to run
...
    -dma-generate               - Generate DMAs for memory operations
    -loop-fusion                - Fuse loop nests
    -loop-tile                  - Tile loop nests
    -loop-unroll                - Unroll loops
    -loop-unroll-jam            - Unroll and jam loops
    -lower-affine               - Lower If, For, AffineApply operations to primitive equivalents
    -lower-vector-transfers     - Materializes vector transfer ops to a proper abstraction for the hardware
    -materialize-vectors        - Materializes super-vectors to vectors of the proper size for the hardware
    -memref-bound-check         - Check memref access bounds in a Function
    -memref-dataflow-opt        - Perform store/load forwarding for memrefs
    -memref-dependence-check    - Checks dependences between all pairs of memref accesses.
    -pipeline-data-transfer     - Pipeline non-blocking data transfers between explicitly managed levels of the me
...
    -vectorize                  - Vectorize to a target independent n-D vector abstraction
```

Our implementation is well underway, and we've built a number of passes using this representation. That said, we're still actively evolving and changing things here - this would be a great place to get involved if you are interested in contributing.



Which codegen approach is “best”?

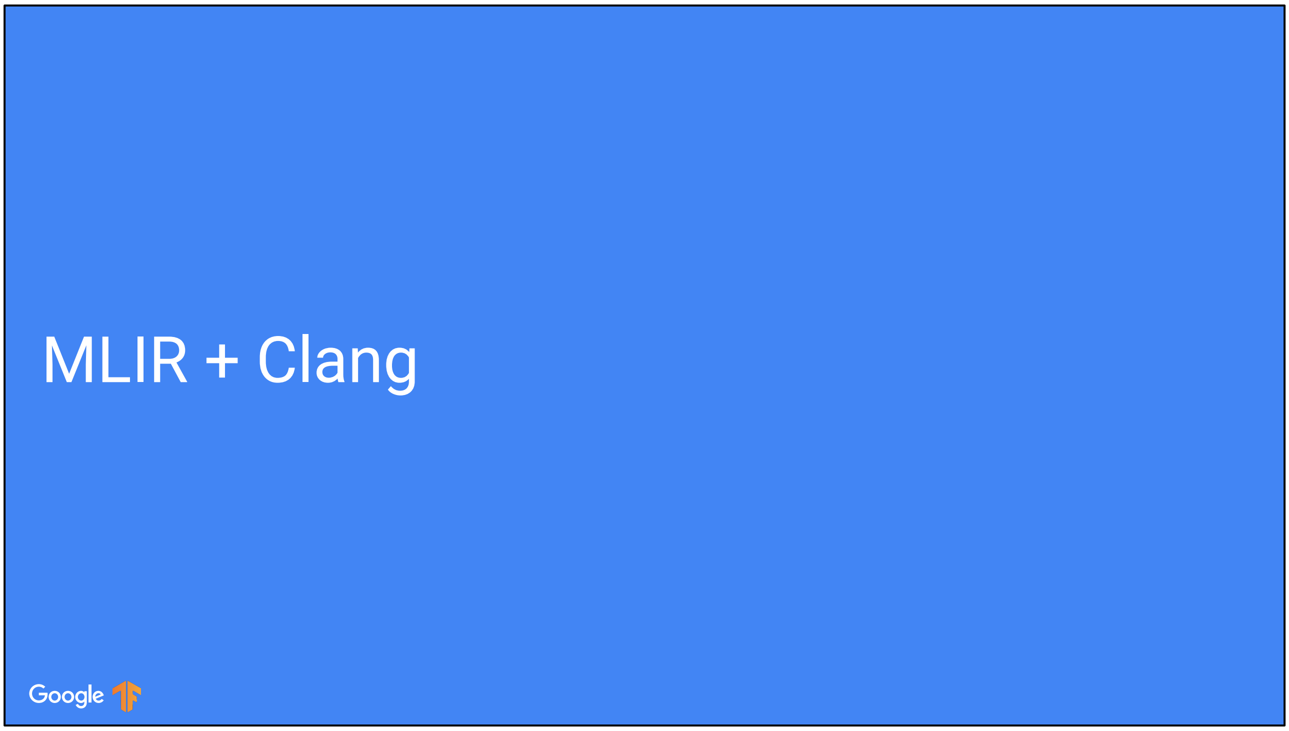
Many complementary approaches possible:
- XLA, Polyhedral, Halide, Stripe, … also hand-written kernels

Non-obvious trade-offs in algorithms and implementation details:
- One may work particularly well for one kind of hardware, but fail on others

Users don’t care about the implementation: they want fast code on their HW!

Solution: support many approaches (“mixture of experts”) in a common framework:
- Use whatever works best in practice for the problem on hand!
- Autotuning and search particularly great for this, manual controls possible

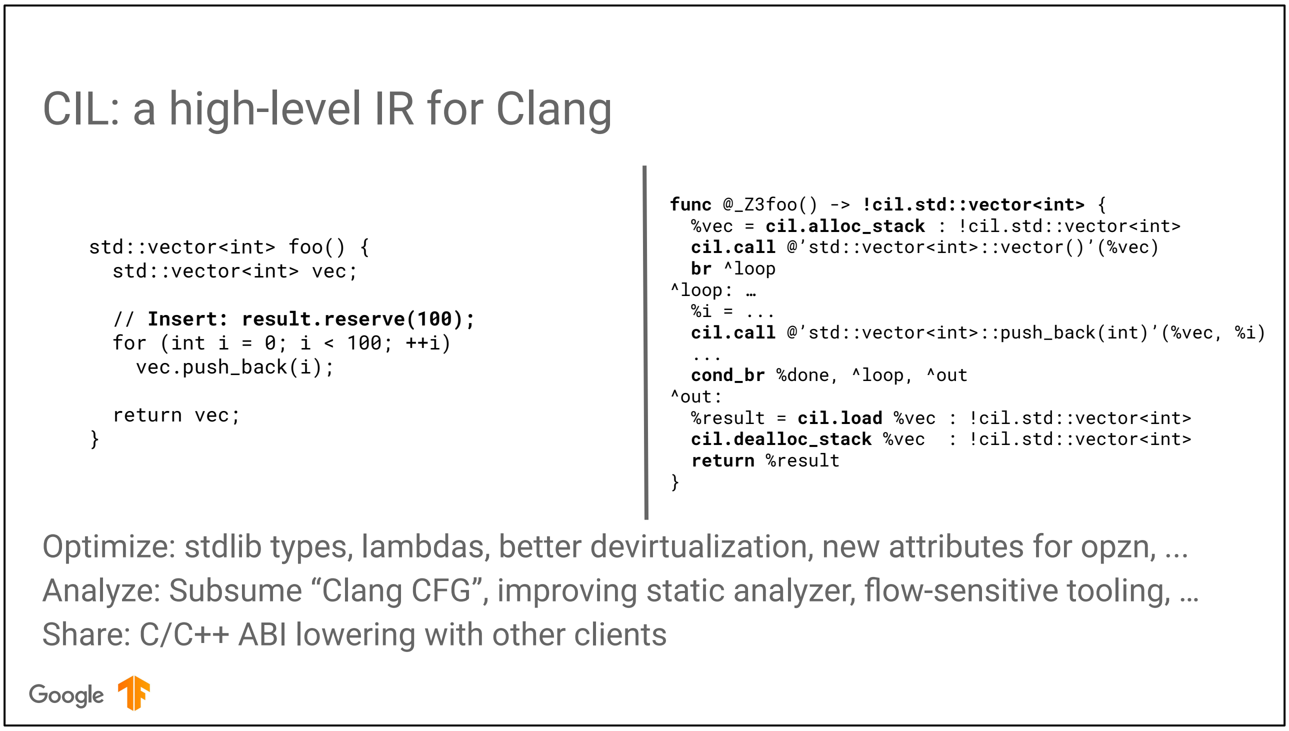
So we just described two very different approaches for generating code for tensors - which is best?
There are a lot more than two - there are lots of different approaches possible here.
We don’t know which will work out the best in practice, and we need to support lots of different kinds of hardware.
Our viewpoint is that these will all have pros and cons in different scenarios, so the best thing for users is to support many of them, and pick the best in practice.



MLIR + Clang

Google

We’ve spent a lot of time talking about tensors and machine learning, let’s turn now to talk about potential applications of MLIR to the Clang ecosystem.



CIL: a high-level IR for Clang

```
std::vector<int> foo() {
  std::vector<int> vec;

  // Insert: result.reserve(100);
  for (int i = 0; i < 100; ++i)
    vec.push_back(i);

  return vec;
}
```

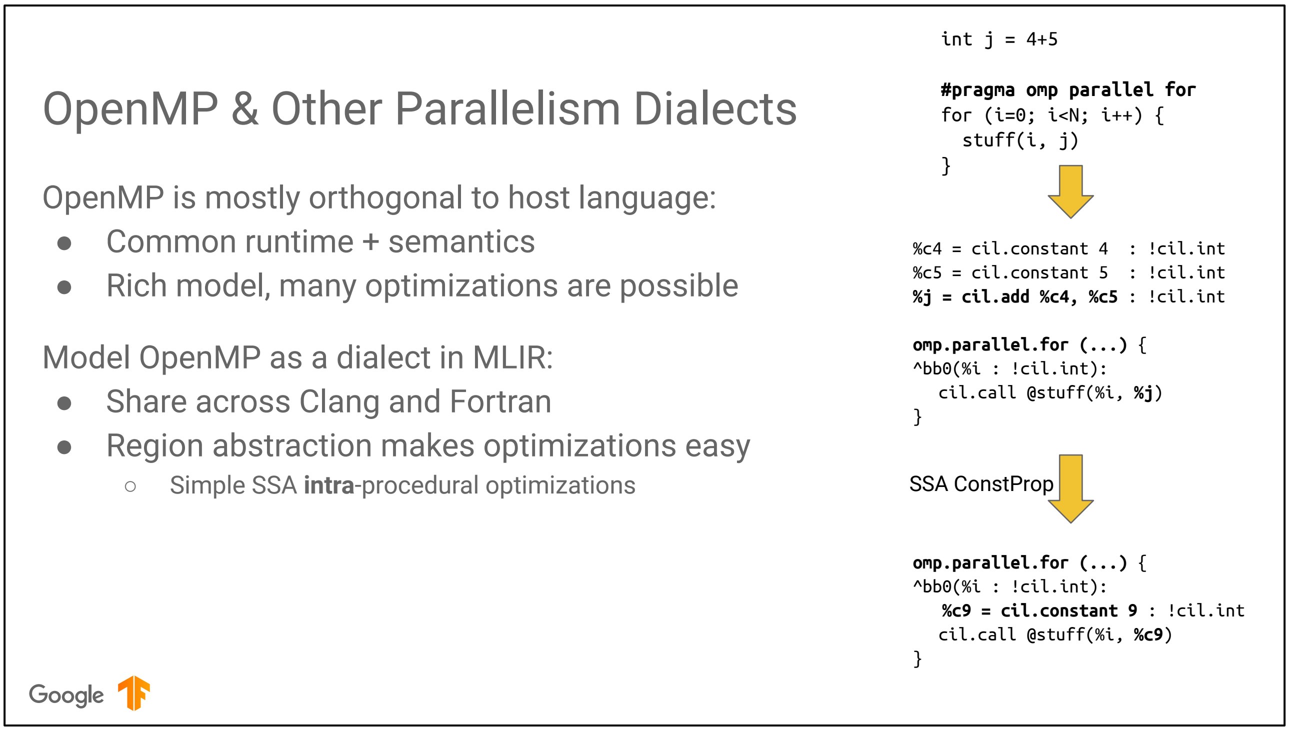
```
func @_Z3foo() -> !cil.std::vector<int> {
  %vec = cil.alloc_stack : !cil.std::vector<int>
  cil.call @'std::vector<int>::vector()'(%vec)
  br ^loop
^loop: …
  %i = ...
  cil.call @'std::vector<int>::push_back(int)'(%vec, %i)
  ...
  cond_br %done, ^loop, ^out
^out:
  %result = cil.load %vec : !cil.std::vector<int>
  cil.dealloc_stack %vec  : !cil.std::vector<int>
  return %result
}
```

Optimize: stdlib types, lambdas, better devirtualization, new attributes for opzn, ...
Analyze: Subsume “Clang CFG”, improving static analyzer, flow-sensitive tooling, …
Share: C/C++ ABI lowering with other clients

Google

We think it would make sense to introduce a high level CIL IR, just like Swift has in SIL. Why? There are lots of things that are just impractical to do right now on the Clang AST (e.g. because it requires lowering and dataflow analysis), and doing that on LLVM IR isn’t practical, because the ABI lowering makes the generated code target-specific.

Adding this would have a lot of benefits, including the ability to do library-level optimizations, to make IRGen simpler. This would also address the frequent request by frontend authors to reuse the Clang ABI lowering code.



OpenMP & Other Parallelism Dialects

OpenMP is mostly orthogonal to host language:

- Common runtime + semantics
- Rich model, many optimizations are possible

Model OpenMP as a dialect in MLIR:

- Share across Clang and Fortran
- Region abstraction makes optimizations easy
  - Simple SSA **intra**-procedural optimizations

```
int j = 4+5

#pragma omp parallel for
for (i=0; i<N; i++) {
  stuff(i, j)
}
```

```
%c4 = cil.constant 4  : !cil.int
%c5 = cil.constant 5  : !cil.int
%j = cil.add %c4, %c5 : !cil.int

omp.parallel.for (...) {
^bb0(%i : !cil.int):
   cil.call @stuff(%i, %j)
}
```

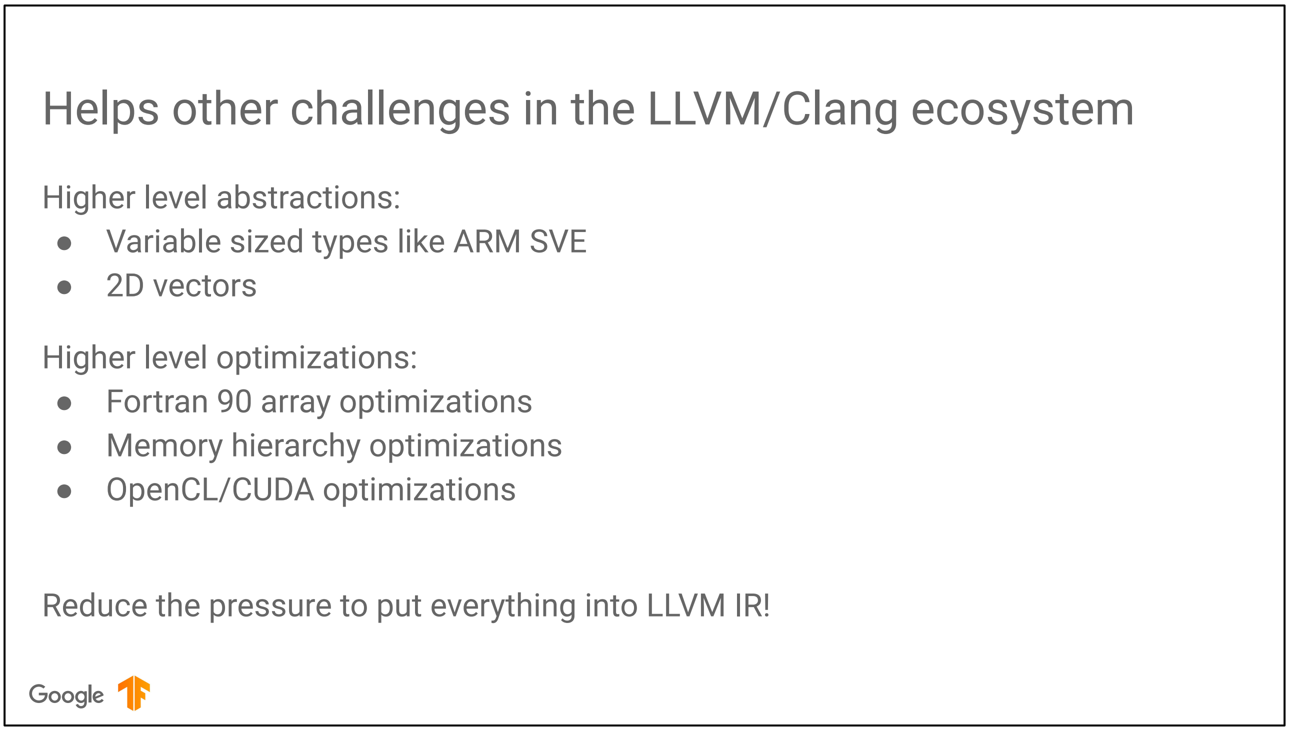
SSA ConstProp

```
omp.parallel.for (...) {
^bb0(%i : !cil.int):
   %c9 = cil.constant 9 : !cil.int
   cil.call @stuff(%i, %c9)
}
```

Google

Sub communities within Clang would also benefit. For example, OpenMP is not very well served with the current design. Very simple optimizations are difficult on LLVM IR, because function outlining has already been performed. This turns even trivial optimizations (like constant folding into parallel for loops) into interprocedural problems. Having a first class region representation makes these things trivial, based on standard SSA techniques.

This would also provide a path for Fortran to reuse the same code. Right now the Flang community either has to generate Clang ASTs to get reuse (egads!) or generate LLVM IR directly and reimplement OpenMP lowering.



Helps other challenges in the LLVM/Clang ecosystem

Higher level abstractions:

- Variable sized types like ARM SVE
- 2D vectors

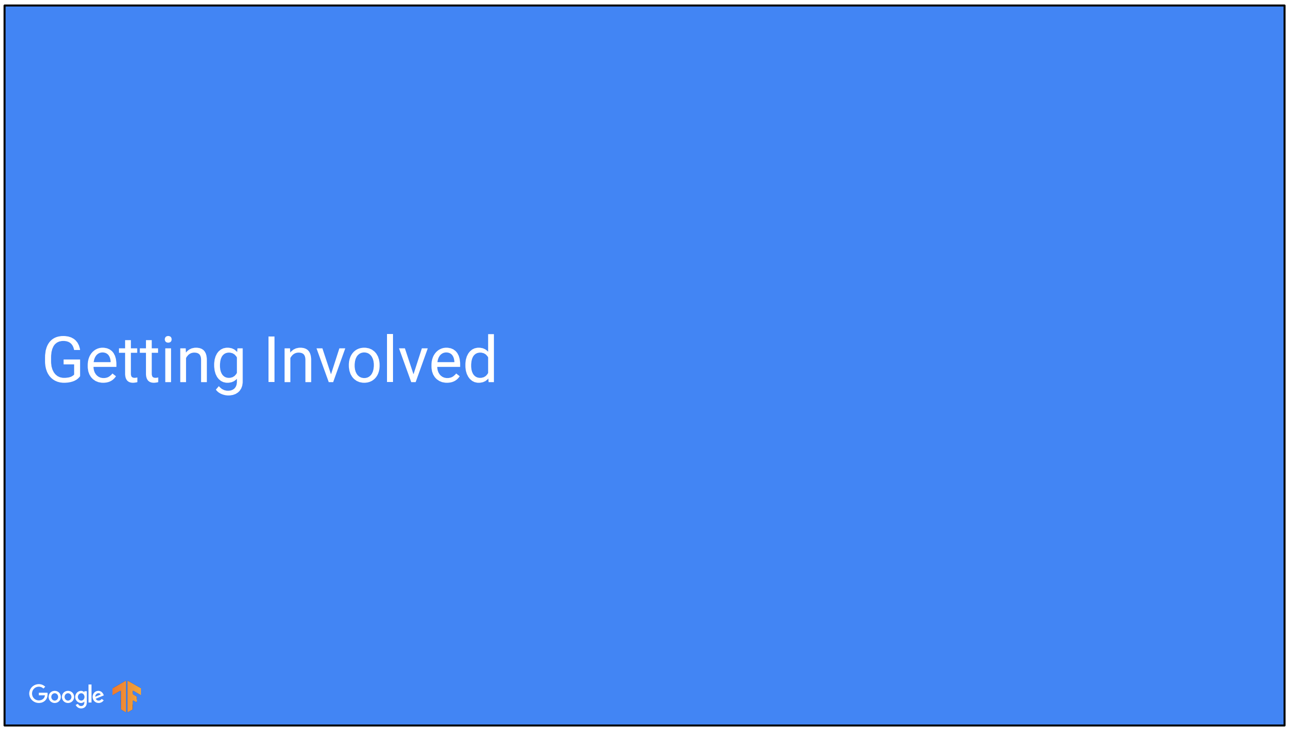
Higher level optimizations:

- Fortran 90 array optimizations
- Memory hierarchy optimizations
- OpenCL/CUDA optimizations

Reduce the pressure to put everything into LLVM IR!

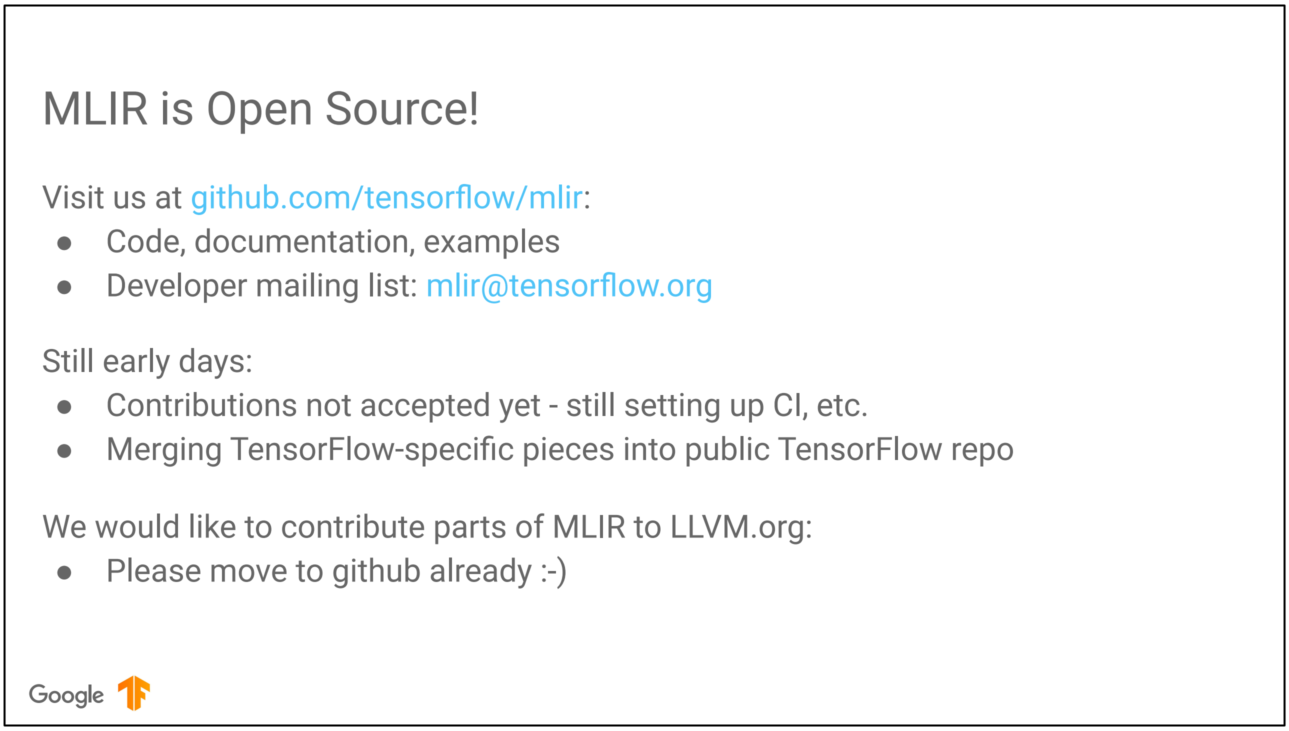
Google

There are also a bunch of other projects that are trying to cram high level concepts into LLVM IR, and struggling, because LLVM was never designed for them. ARM SVE and 2D vector types are examples of things that are no problem to model in MLIR (which already has variable size tensors and N-D vectors). Doing these optimizations and analyses on MLIR (optionally using the polyhedral stuff to make it all more powerful/efficient) and lowering to primitives in LLVM IR eliminates this sort of pressure.



Getting Involved

Ok, so that is a ton of information. If you are interested, how do you get involved?



MLIR is Open Source!

Visit us at github.com/tensorflow/mlir:

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

Still early days:

- Contributions not accepted yet - still setting up CI, etc.
- Merging TensorFlow-specific pieces into public TensorFlow repo

We would like to contribute parts of MLIR to LLVM.org:

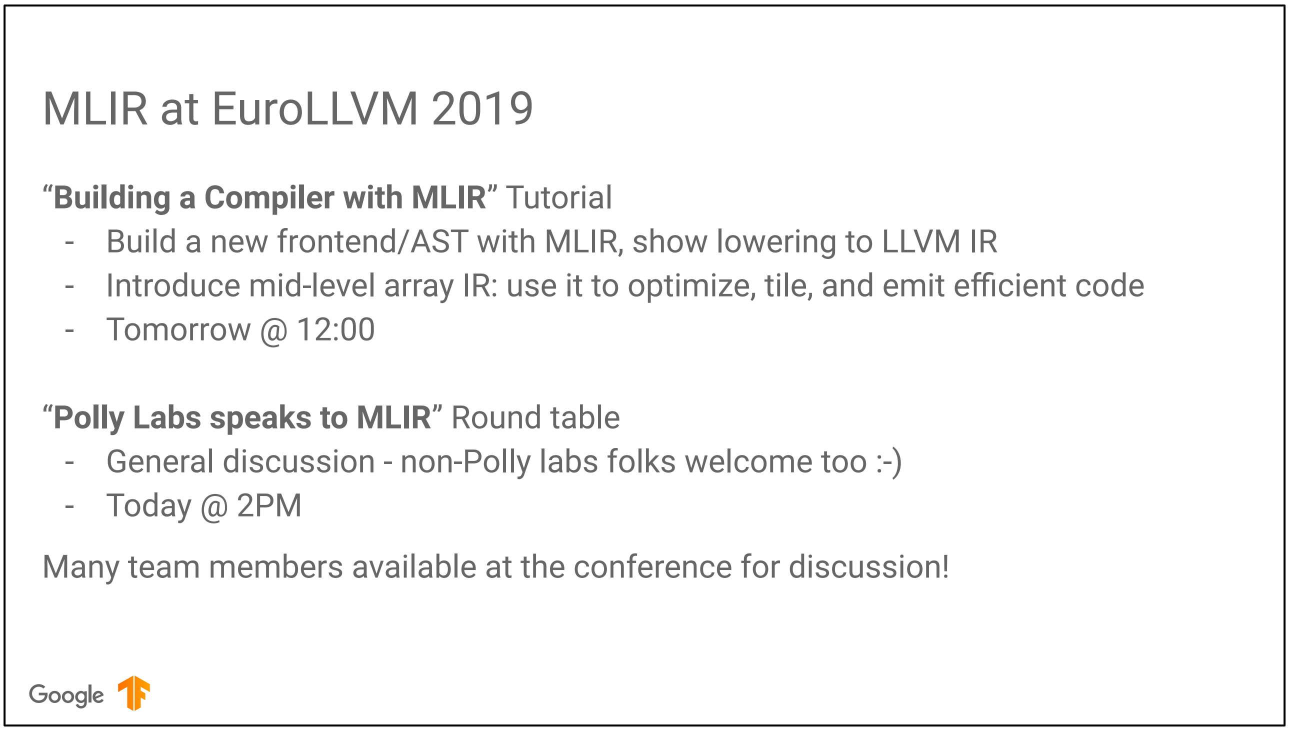
- Please move to github already :-)

Google

MLIR is open source, as of last wednesday. We have the code on github and you can join our public dev mailing list.

We're still setting up the infrastructure, but we expect this all to be in place over the next month.

We'd like to discuss contributing MLIR technology to LLVM.org as a subproject, and plan to start a thread on llvm-dev to see if there are any objections to that direction.



MLIR at EuroLLVM 2019

“**Building a Compiler with MLIR**” Tutorial

- Build a new frontend/AST with MLIR, show lowering to LLVM IR
- Introduce mid-level array IR: use it to optimize, tile, and emit efficient code
- Tomorrow @ 12:00

“**Polly Labs speaks to MLIR**” Round table

- General discussion - non-Polly labs folks welcome too :-)
- Today @ 2PM

Many team members available at the conference for discussion!

Google

We also have a great tutorial tomorrow, which goes into how to build a compiler frontend (even using MLIR as an AST), how to generate LLVM IR, and how to do advanced optimizations on array abstractions.

We have a round table later today and, despite the name, we’re willing to talk to people even if they aren’t from Polly Labs :-)

Thank you to the team!

Questions?

We are hiring!
mlir-hiring@google.com

Google

Anyway, that is all we have for today, thank you to everyone who has contributed to the team. If you are interested in joining us, please reach out to us on the mailing list. Any more questions?