MLIR in TensorFlow Ecosystem

C4ML II at CGO 2020

Jacques Pienaar
Google

Agenda

- Update on year past
  - MLIR announced at C4ML last year
- Brief introduction to MLIR
- MLIR in TensorFlow ecosystem
  - Uses current and future in TF
  - An aside on simple ML inference engine
- MLIR community
  - Excluding the other talks today …
  - … and Albert Cohen's talk yesterday or Chris Lattner and Tatiana Shpeisman's talk at CGO
- Getting involved

The past year

Year since C4ML in review

~ MLIR announced @ C4ML 2019, Feb 17

~ MLIR open sourced
- Core @ Mar 29th & TF/MLIR @ Jun 27

~ Partner announcement &
proposal to contribute to LLVM @ Sep 9

~ MLIR core moved to LLVM project Dec 23rd
- *“Landing as a great Christmas present for LLVM developers interested in heterogeneous hardware compilation ...”*

~ TF / TFLite converter replaced @ Feb 19 2020

Google

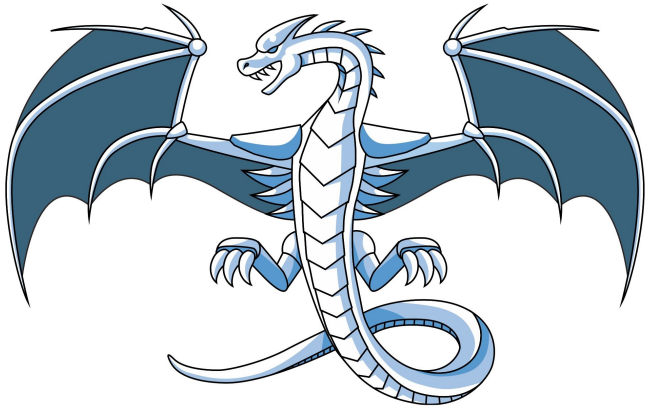
What is MLIR?

A collection of modular and reusable software components that enables the progressive lowering of high level operations, to efficiently target hardware in a common way

Multi-Level Intermediate Representation

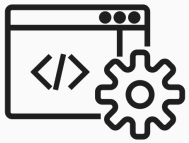
New compiler infrastructure

Originally built by TensorFlow team



Part of LLVM project

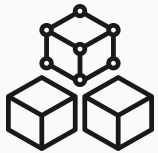
How is MLIR different?



State of Art Compiler Technology

MLIR is NOT just a common graph serialization format nor is there anything like it

New shared industry abstractions spanning languages ("OMP" dialect?)



Modular & Extensible

From graph representation through optimization to code generation

Mix and match representations to fit problem space

Not opinionated

Choose the level of representation that is right for your device

We want to enable whole new class of compiler research

Google

A toolkit for representing and transforming “code”

Represent and transform IR ⇄↺⇊

Represent Multiple Levels of

- tree-based IRs (ASTs),
- graph-based IRs (TF Graph, HLO),
- machine instructions (LLVM IR)

IR at the same time

While enabling

Common compiler infrastructure

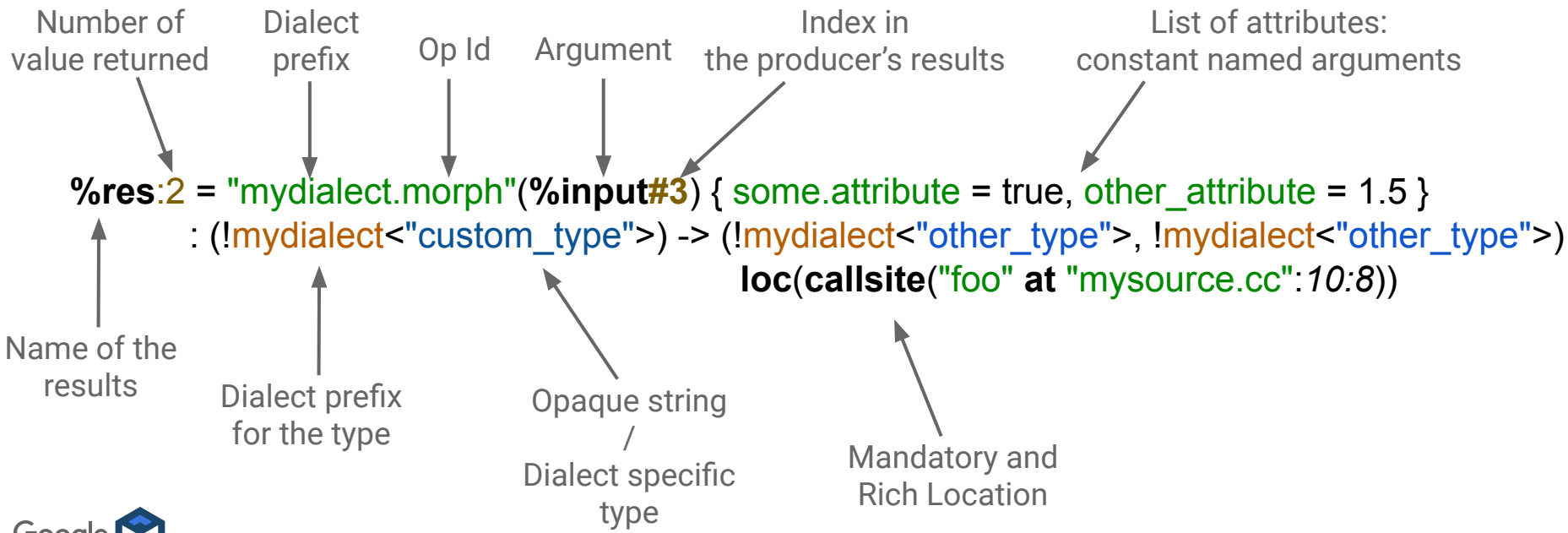
- location tracking
- richer type system
- common set of conversion passes

And much more

Google

Operations, Not Instructions

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



Google

(Operations→Regions→Blocks)

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

Dialects

A MLIR dialect is a logical grouping including:

- A prefix (“namespace” reservation)
- A list of custom types, each defined by a C++ class.
- A list of operations, each its name and C++ class implementation:
  - Verifier for operation invariants
  - Semantics (has-no-side-effects, constant-folding, CSE-allowed, ….)
- Possibly custom parser and assembly printer
- A list of passes (for analysis, transformations, and dialect conversions)

Interfaces

- Decouple transformations from dialect and operation definitions
    - LoopLike, Inlining
- Apply transformations across dialects
- Design passes to operate on characteristics/structure rather than specific ops
- Easily extend to new dialects/ops

Google

Interfaces

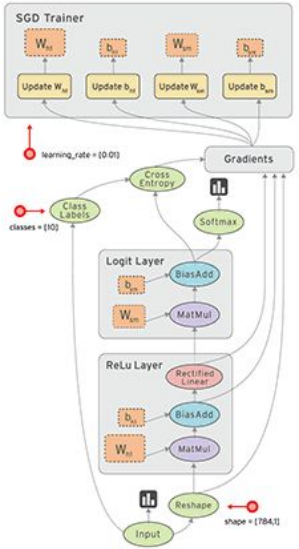
- Decouple tra
  operation de
  - LoopLike,
- Apply transf
- Design pass
  characterist
  specific ops
- Easily extend to new dialects/ops

Much more info
~ defining ops
~ declarative patterns (DAG -> DAG)
~ declarative ASM syntax
...

But that's a whole tutorial (see online and next iteration @ EuroLLVM!)

MLIR in TensorFlow ecosystem

TF Optimization & Compilation



Optimization & Legalization

Code Generation

CPU

GPU

TPU

TFLite

TensorFlow.js

EdgeTPU

?PU

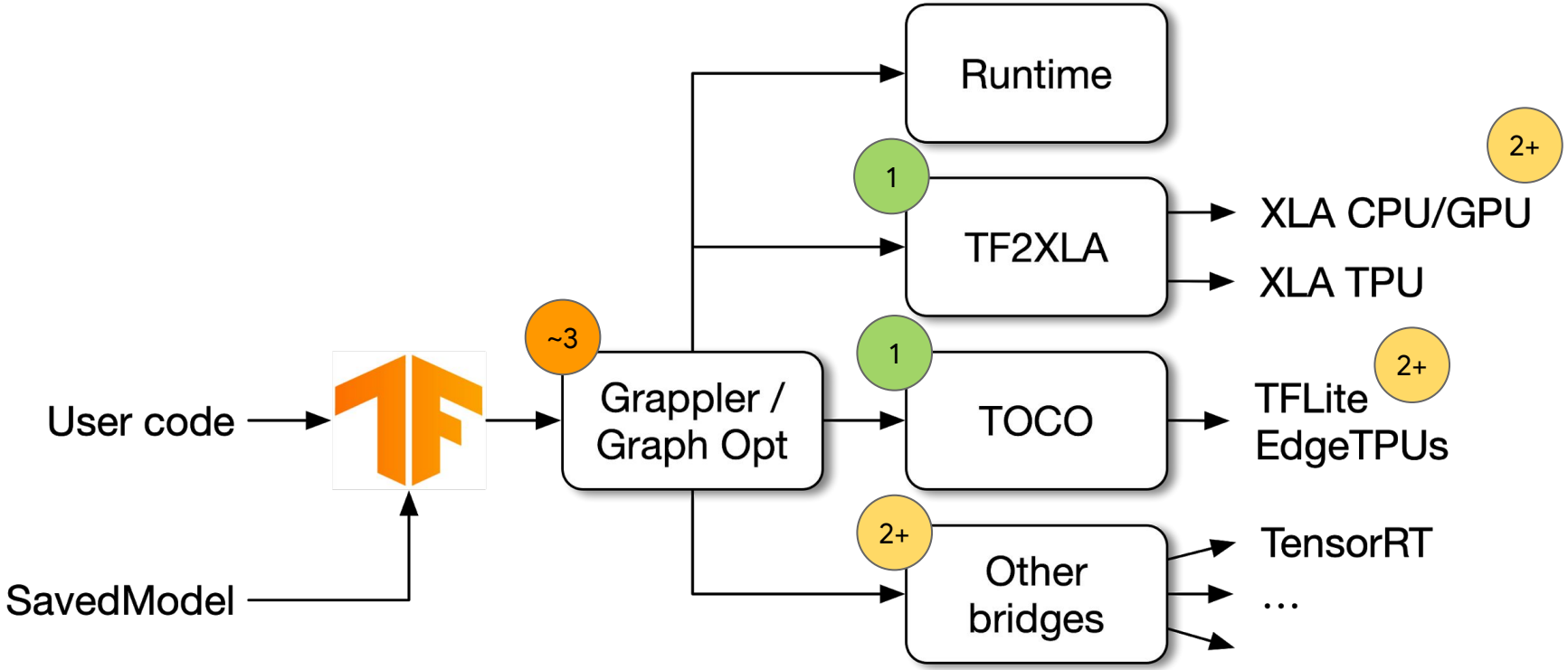
Google

Goal: Global improvements to TensorFlow infrastructure

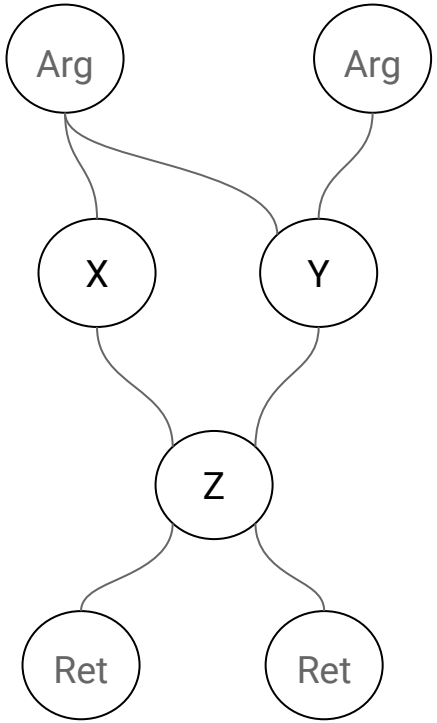
SSA-based designs to generalize and improve ML “graphs”:

- 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

TensorFlow usage (current/ongoing/future)



Computational Graph Dialect




```
func @foo( %arg0 : tensor<i1>, %arg1 : tensor<...>) ... {
  %X   = tf.X  %arg0 : tensor<...>
  %Y   = tf.Y  %arg0, %arg1 : tensor<...>, tensor<...>
  %Z:2 = tf.Z  %X, %Y : tensor<...>, tensor<...>
  return %Z#0, %Z#1 : tensor<...>, tensor<...>
}
```

Google

Control Flow and Concurrency

Control flow and dynamic features of TF1, TF2

- Conversion from control to data flow
- Lazy evaluation

Concurrency

- Sequential execution in blocks
- Distribution
- Offloading
- Implicit concurrency in `tf.graph` regions
  - Implicit **futures** for SSA-friendly, asynchronous task parallelism

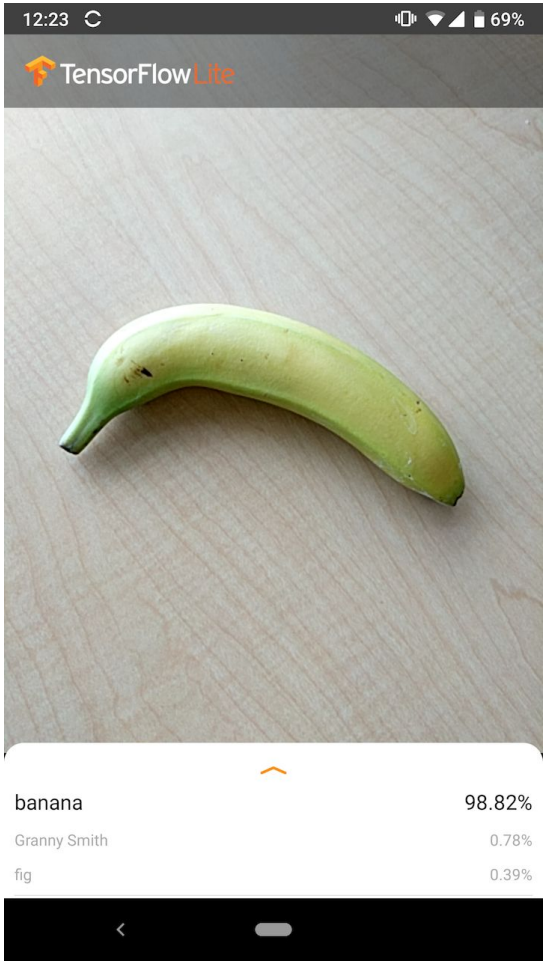
→ **Research: task parallelism, memory models, separation logic**

Control Flow and Concurrency

```
%0 = tf.graph (%arg0 : tensor<f32>, %arg1 : tensor<f32>,
               %arg2 : !tf.resource) {
 // Execution of these operations is asynchronous, the %control
 // return value can be used to impose extra runtime ordering,
 // for example the assignment to the variable %arg2 is ordered
 // after the read explicitly below.
 %1, %control = tf.ReadVariableOp(%arg2)
     : (!tf.resource) -> (tensor<f32>, !tf.control)
 %2, %control_1 = tf.Add(%arg0, %1)
     : (tensor<f32>, tensor<f32>) -> (tensor<f32>, !tf.control)
 %control_2 = tf.AssignVariableOp(%arg2, %2, %control)
     : (!tf.resource, tensor<f32>) -> !tf.control
 %3, %control_3 = tf.Add(%2, %arg1)
     : (tensor<f32>, tensor<f32>) -> (tensor<f32>, !tf.control)
 tf.fetch %3, %control_2 : tensor<f32>, !tf.control
}
```

TFLite : inference on the edge

- TensorFlow to TFLite converter
  - Adding control flow to TFLite
  - RNN support
- New quantization support
  - Forgot to mention: MLIR has quantized types!
  - Tooling to move state of the art forward
- Model optimization passes
  - Sparsity optimization

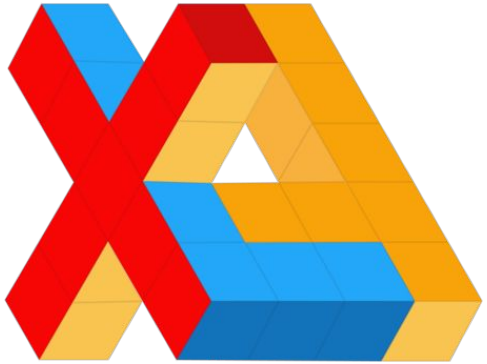


Google

CPU/GPU codegen

- Multiple collaboration on TensorFlow codegen

- XLA codegen (emitter style)
- Structured ops (e.g., LinAlg)
- (simplified) Polyhedral (e.g., Affine)



Albert's and other folks here's talks will look at different codegen!

MLIR community

MLIR is a community project

- Important takeaway from C4ML last year:
  - All solving the same problems over and over
  - Effort on common (but **very** important and not really common) parts take away from value add
- MLIR make it easy to add abstraction & compile down
- Community very important
  - Want to highlight some works
  - … but not those of folks already presenting here ;-)

Example: Stencil Computations

MLIR for accelerating climate modelling

Open Climate Compiler Initiative

VULCAN

MeteoSwiss

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

SPCL

ETH zürich

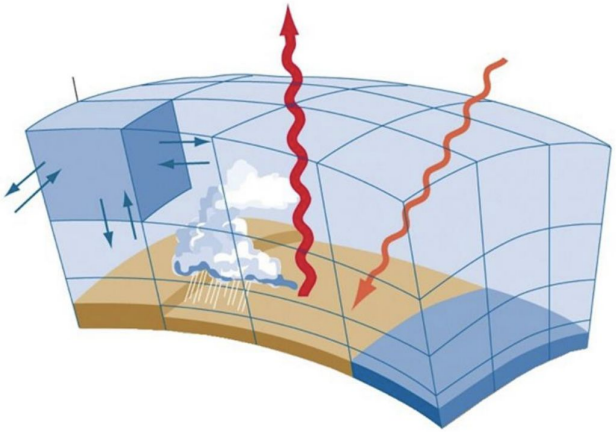
Google

A Compiler Intermediate Representation for Stencils

JEAN-MICHEL GORIUS, TOBIAS WICKY, TOBIAS GROSSER, AND TOBIAS GYSI

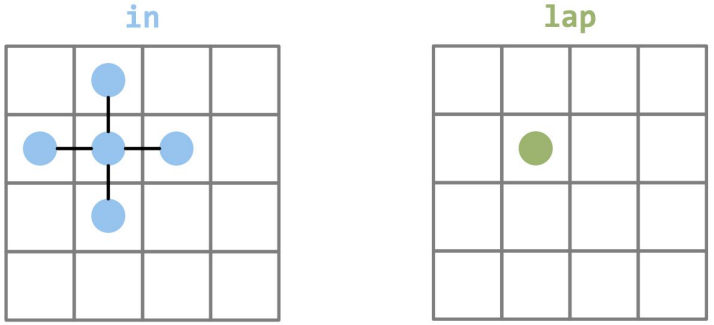
Domain-Science vs Computer-Science

- solve PDE
- finite differences
- structured grid



- element-wise computation
- fixed neighborhood

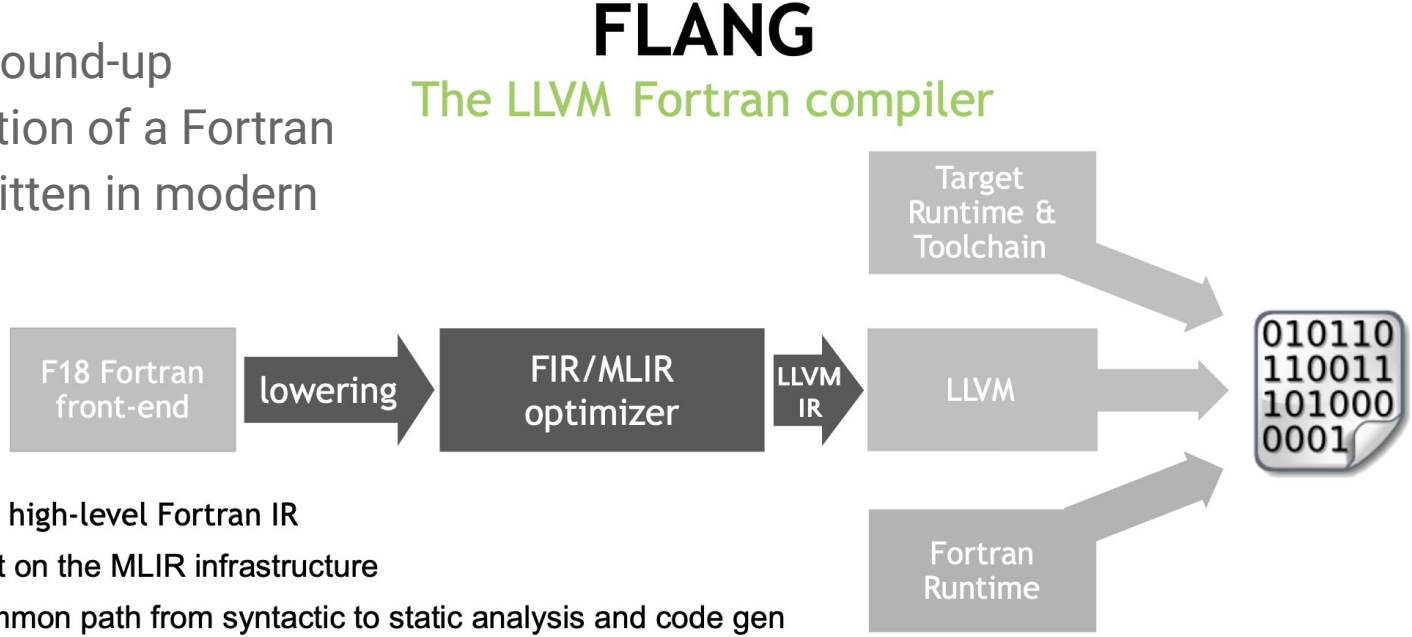
```
lap(i,j) = -4.0 * in(i,j) +
  in(i-1,j) + in(i+1,j) +
  in(i,j-1) + in(i,j+1)
```



Google

Example: Flang: the LLVM Fortran Frontend

- Flang is a ground-up implementation of a Fortran front end written in modern C++



FIR: high-level Fortran IR

Built on the MLIR infrastructure

Common path from syntactic to static analysis and code gen

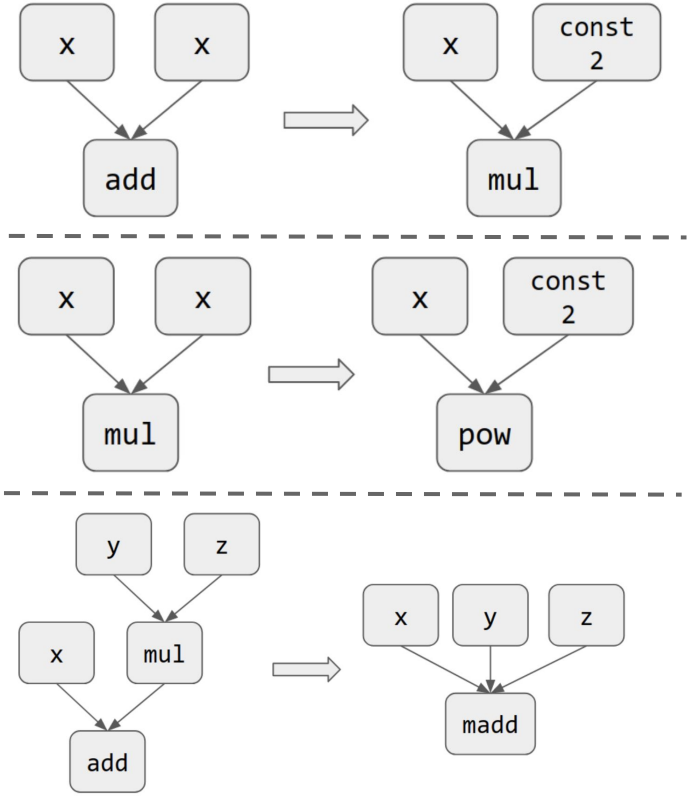
Shrink abstraction gap: core Fortran operational properties

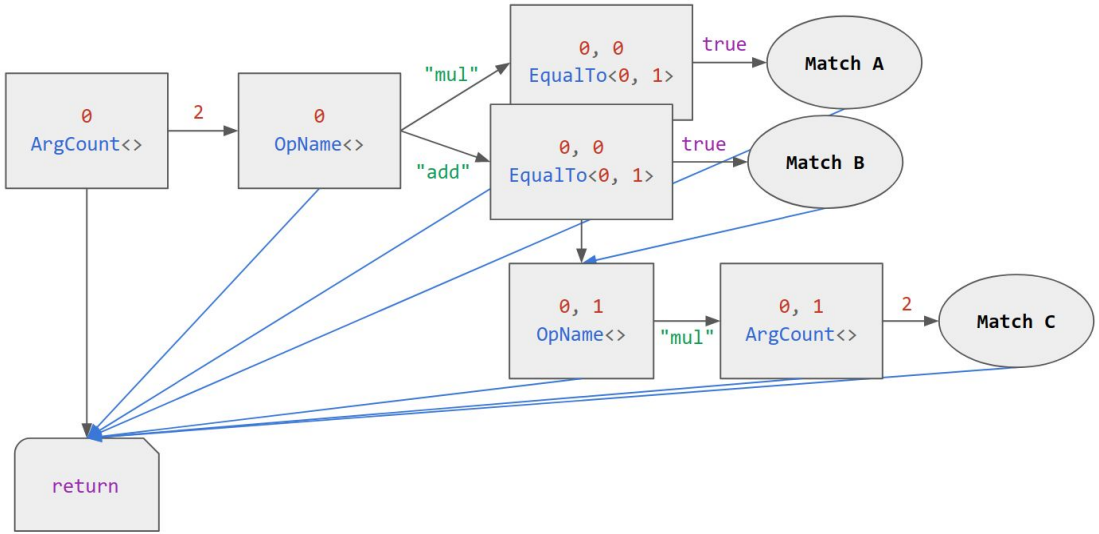
Focus on writing Fortran aware optimizations

Separation of concerns: constraints checking vs. optimizing computation

MLIR Pattern Matching and Rewrite

~ Instruction Selection problem.





Google

MLIR Pattern Matching and Rewrite

An MLIR dialect to manipulate MLIR IR!

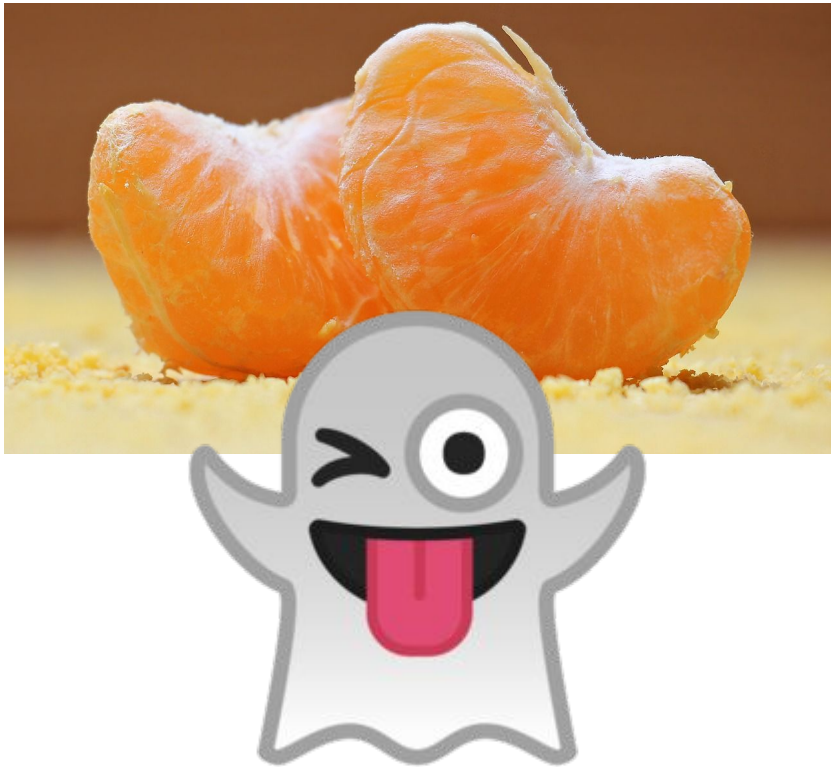
```
func @matcher(%0 : !Operation) {
^bb0:
  CheckArgCount(%0) [^bb1, ^ex0] {count = 2}
       : (!Operation) -> ()
^bb1:
  CheckOpName(%0) [^bb2, ^bb5] {name = "add"}
       : (!Operation) -> ()
^bb2:
  %1 = GetOperand(%0) {index = 0} : (!Operation) -> !Value
  %2 = GetOperand(%0) {index = 1} : (!Operation) -> !Value
  ValueEqualTo(%1, %2) [^rr0, ^bb3] : (!Value, !Value) -> ()
^rr0:
  // Save x
  RegisterResult(%1) [^bb3] {id = 0} : (!Value) -> ()
^bb3:
  %3 = GetDefiningOp(%2) : (!Value) -> !Operation
  CheckOpName(%3) [^bb4, ^bb5] {name = "mul"}
       : (!Operation) -> ()
^bb4:
  CheckArgCount(%3) [^rr1, ^bb5] {count = 2}
       : (!Operation) -> ()
```

```
^rr1:
  // Save x, y, and z
  %4 = GetOperand(%3) {index = 0} : (!Operation) -> !Value
  %5 = GetOperand(%4) {index = 1} : (!Operation) -> !Value
  RegisterResult(%1, %4, %5) [^bb5] {id = 1}
       : (!Value, !Value, !Value) -> ()
^bb5:
  // Previous calls are not necessarily visible here
  %6 = GetOperand(%0) {index = 0} : (!Operation) -> !Value
  %7 = GetOperand(%0) {index = 1} : (!Operation) -> !Value
  ValueEqualTo(%6, %7) [^bb6,  ^ex0] : (!Value, !Value) -> ()
^bb6:
  CheckOpName(%0) [^rr2, ^ex0] {name = "mul"}
       : (!Operation) -> ()
^rr2:
  // Save x
  RegisterResult(%6) [^ex0] {id = 2} : (!Value) -> ()
^ex0:
  return
}
```

Google

Example: Tiny C Inference Engine

- Problem:
  - Running ML models in highly resource constrained environments
  - On-device training in an end-to-end fashion with <1kB of on-device code
- End-to-end toolchain prototype
  - Stateful model, Multiple model entry points, Structured Python signatures with `@tf.function`
- Required less than 1 SWE-week of effort to implement



Conclusion

MLIR : Reusable Compiler Abstraction Toolbox

IR design involves multiple tradeoffs

- Iterative process, constant learning experience

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

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

No forced IR impedance mismatch

Fresh look at problems

Recap

MLIR is a great infrastructure for higher-level compilation

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

MLIR provides all the infrastructure to build dialects and transformations

- At each level it is the same infrastructure

Toy language tutorial available on github

Getting Involved

Get involved!

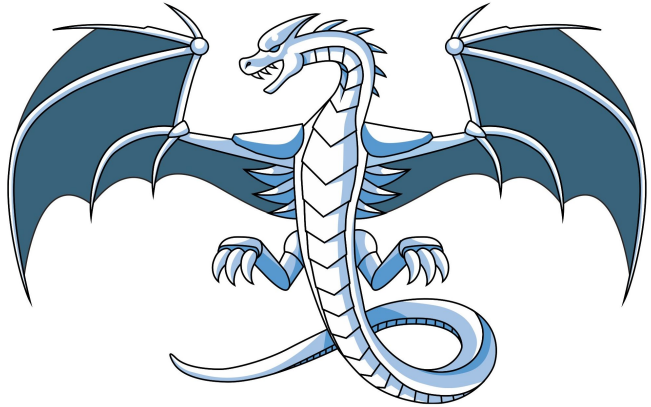
Visit us at mlir.llvm.org:

- Code, documentation, tutorial
- Developer forum/mailing list
  LLVM Discourse server
  mlir@tensorflow.org
- Open design meetings / TF MLIR SIG
- Contributions welcome!

Students: We have internship openings still

- Contact at mlir-hiring@google.com

And Google Summer of Code projects



Google

MLIR

Thank you to the team!

Questions?

We are hiring!
mlir-hiring@google.com