# Contents

- Introduction (4 min) (daniel)
  - Motivation - problems that mlir addresses
    - pre-mature lowering, want to allow transformations at multiple levels
    - Domain-specific IRs: address fragmentation of domain specific hardwares
  - Overview:
    - Principles
- IR designs (3 min) (light)
  - Compare against LLVM IR
- Applications that exemplifies design principles (6 min) (jiyu jihong)
  - Parsimony, traceability, progressivity
- Evaluation (2 min) (light jihong)

# MLIR: Scaling Compiler Infrastructure for Domain Specific Computation

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, Oleksandr Zinenko

Group number: 16

Presented by: Jiyu Chen, Jihong Gan, Yuchen Jiang, Daniel Mishins

# Introduction

- What is MLIR?
- Why not LLVM?
- Why MLIR?
- How was MLIR designed?

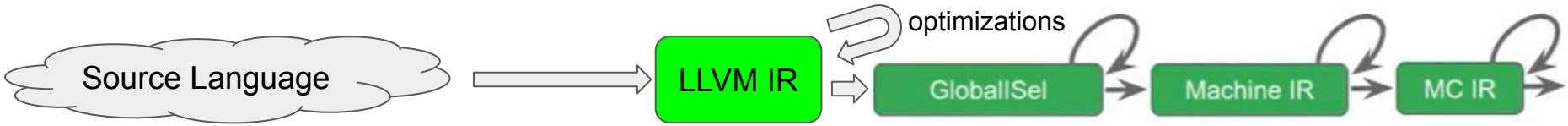
# What is MLIR?

- New (2021) general compiler framework, for building custom compilers
- "Hybrid" IR, supporting arbitrary instructions and abstraction levels

# Why can't we just use LLVM?

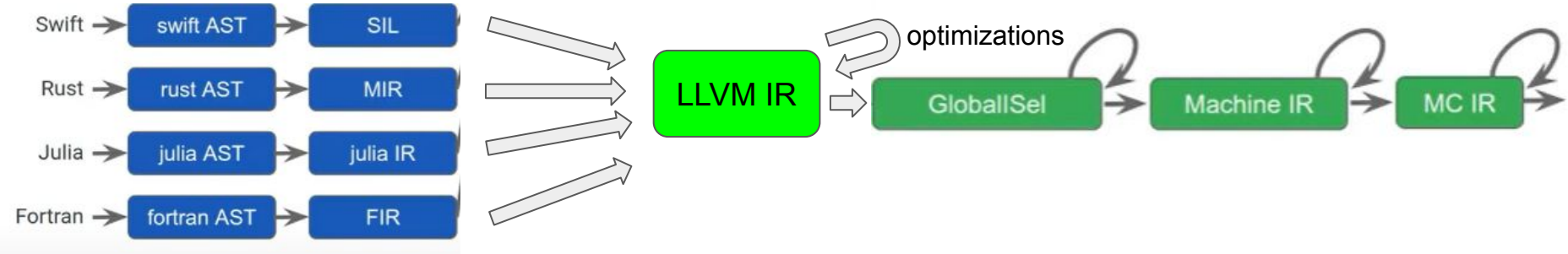

- LLVM is at a fixed abstraction level
  - similar to C with vectors and SSA

# Why can't we just use LLVM?



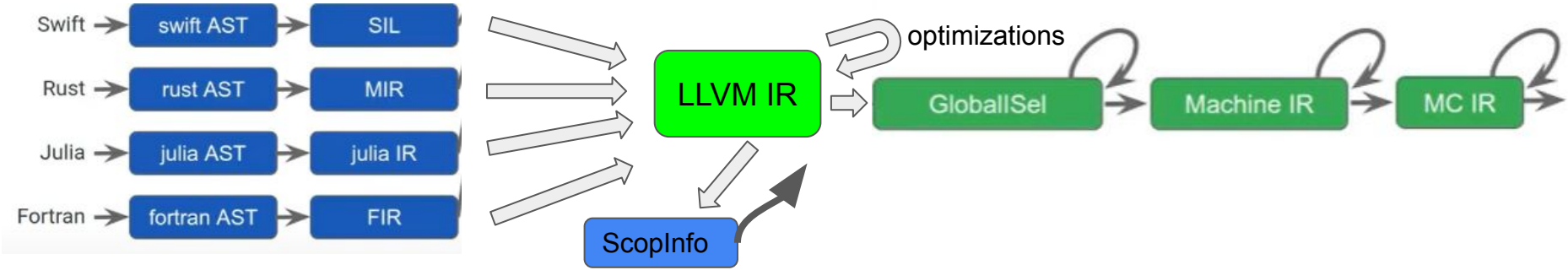

- LLVM is at a fixed abstraction level
  - similar to C with vectors and SSA
- LLVM IR is not enough for low-level representation
  - In reality, most toolchains define new low level intermediate representations for hardware level optimizations

# Why can't we just use LLVM?




- LLVM is at a fixed abstraction level
  - similar to C with vectors and SSA
- LLVM IR is not enough for low-level representation
  - In reality, most toolchains define new low level intermediate representations for hardware level optimizations
- LLVM IR is not enough for high-level representation
  - High level languages need to define their own intermediate IR's to do transformations that use higher level abstractions

# Why can't we just use LLVM?




- LLVM is at a fixed abstraction level
  - similar to C with vectors and SSA
- LLVM IR is not enough for low-level representation
  - In reality, most toolchains define new low level intermediate representations for hardware level optimizations
- LLVM IR is not enough for high-level representation
  - High level languages need to define their own intermediate IR's to do transformations that use higher level abstractions
  - Some optimizations, such as some based on the polyhedral representation we work on in our project, require taking LLVM IR backwards to higher level abstractions

# Value of MLIR

- Each IR needs to build:
  - Type system, syntax, structure, etc
  - Toolchain, pass manager, etc
- With MLIR, we can reuse aspects common to all IRs
- Rely on three design principles:
  - Use a minimal set of versatile built-in concepts to avoid complexity ("Parsimony")
    - Little buitin, everything customizable
    - SSA and regions
  - Preserve information across compiler levels ("Traceability")
    - Maintain original definitions when lowering abstraction
    - Support ability to validate code by preserving source locations
  - Support mixing abstraction levels and avoid premature lowering of abstraction ("Progressivity").

# IR Structure

- ❏ Dialect overview
- ❏ The components
- ❏ Dialect extensibility

# Dialects: families of attributes, operations, types

**Dialects ~ abstraction level**:

LLVM IR, Fortran FIR, Swift SIL, XLA HLO, TensorFlow Graph, ...

**Dialects can define:**

Sets of defined operations

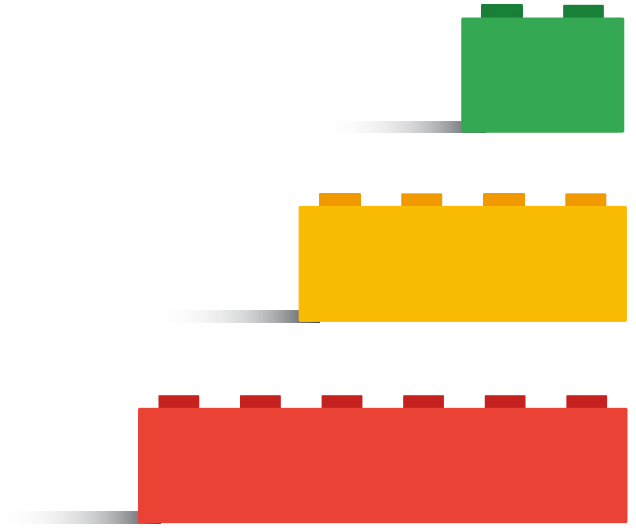
Entirely custom type system

**Operations can define:**

Invariants on # operands, results, attributes, etc

Custom parser, printer, verifier, ...

Constant folding, canonicalization patterns, ...



# IR Structure - Tree

**Operation**

**Region**

**Block**

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

# IR Structure - Tree

### Operation

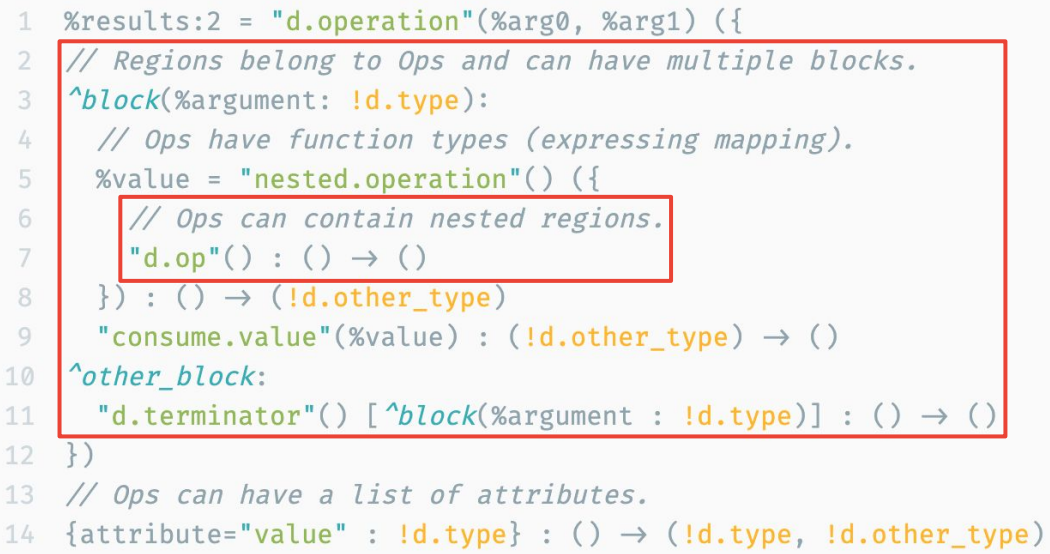
An Operation is the basic execution unit. An Operation can be as simple as an instruction, but can also be more complex and can represent any function.

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

# IR Structure - Tree

**Region**

A container attached to an operation that can contain other operations.



# IR Structure - Tree

**Block**

A list of operations contained in a region with no control flow. The last operation in a block is a terminator that can transfer control flow to blocks or regions.

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

# IR Structure - Data

### Value

Values are units of runtime data. They are defined and used by operations. Values obey static single assignment (SSA) rule.

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

# IR Structure - Data

### Type

Types describe compile-time information about a value. Each value has a type.
Operation specifies types of defined and used values.

### Attribute

Attributes describe compile-time information about an operation. They may be optional or mandatory as per operation semantics.

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

# Dialect syntax is customizable

```
// Attribute aliases can be forward-declared.
#map1 = (d0, d1) → (d0 + d1)
#map3 = ()[s0] → (s0)

// Ops may have regions attached.
"affine.for"(%arg0) ({
// Regions consist of a CFG of blocks with arguments.
^bb0(%arg4: index):
  // Block are lists of operations.
  "affine.for"(%arg0) ({
  ^bb0(%arg5: index):
    // Ops use and define typed values, which obey SSA.
    %0 = "affine.load"(%arg1, %arg4) {map = (d0) → (d0)}
       : (memref<?xf32>, index) → f32
    %1 = "affine.load"(%arg2, %arg5) {map = (d0) → (d0)}
       : (memref<?xf32>, index) → f32
    %2 = "std.mulf"(%0, %1) : (f32, f32) → f32
    %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1}
       : (memref<?xf32>, index, index) → f32
    %4 = "std.addf"(%3, %2) : (f32, f32) → f32
    "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1}
       : (f32, memref<?xf32>, index, index) → ()
    // Blocks end with a terminator Op.
    "affine.terminator"() : () → ()
  // Ops have a list of attributes.
  }) {lower_bound = () → (0), step = 1 : index, upper_bound = #map3}
    : (index) → ()
  "affine.terminator"() : () → ()
}) {lower_bound = () → (0), step = 1 : index, upper_bound = #map3}
  : (index) → ()
```

```
// Affine loops are Ops with regions.
affine.for %arg0 = 0 to %N {
  // Only loop-invariant values, loop iterators, and affine functions of
  // those are allowed.
  affine.for %arg1 = 0 to %N {
    // Body of affine for loops obey SSA.
    %0 = affine.load %A[%arg0] : memref<? x f32>
    // Structured memory reference (memref) type can have
    // affine layout maps.
    %1 = affine.load %B[%arg1] : memref<? x f32, (d0)[s0] → (d0 + s0)>
    %2 = mulf %0, %1 : f32
    // Affine load/store can have affine expressions as subscripts.
    %3 = affine.load %C[%arg0 + %arg1] : memref<? x f32>
    %4 = addf %3, %2 : f32
    affine.store %4, %C[%arg0 + %arg1] : memref<? x f32>
  }
}
```

# Dialects are easy to extend

**Operation**

Operations are open to customization while remaining high usability.

- LLVM IR intrinsics
- Integer arithmetic
- Tensorflow operations
- Affine loops and conditions
- …

**Type**

The type system is flexible and open.

- All of LLVM IR types
- Dependently typed nD vectors
- Ranked and unranked tensors
- …

**Attribute**

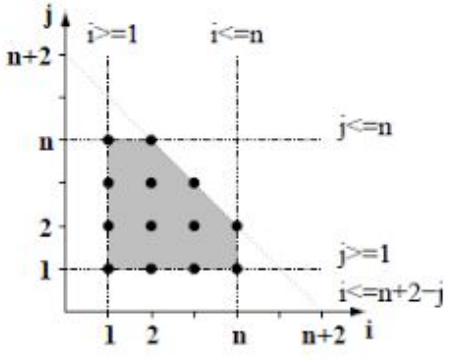
The attribute system is open.

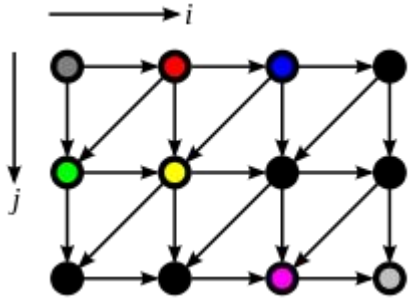
- integer or string values;
- file:line:col locations;
- affine maps;
- opaque AST node pointers;
- binary blobs;
- containers of other attributes, ...

# Applications

- Polyhedral code generation
- Partial lowering and mixed dialects
- TensorFlow graphs

# Applications: Polyhedral code generation




- The polyhedral model represents each dynamic instance of a statement in a loop nest as an integer point in a polyhedron
- Enabling optimizations in parallelism, locality, etc.
- Widely used in compilers for deep learning and scientific computing

# Applications: Polyhedral code generation

```
// Affine loops are Ops with regions.
affine.for %arg0 = 0 to %N {
  // Only loop-invariant values, loop iterators, and affine functions of
  // those are allowed.
  affine.for %arg1 = 0 to %N {
    // Body of affine for loops obey SSA.
    %0 = affine.load %A[%arg0] : memref<? x f32>
    // Structured memory reference (memref) type can have
    // affine layout maps.
    %1 = affine.load %B[%arg1] : memref<? x f32, (d0)[s0] -> (d0 + s0)>
    %2 = mulf %0, %1 : f32
    // Affine load/store can have affine expressions as subscripts.
    %3 = affine.load %C[%arg0 + %arg1] : memref<? x f32>
    %4 = addf %3, %2 : f32
    affine.store %4, %C[%arg0 + %arg1] : memref<? x f32>
  }
}
```

Affine dialect representation of polynomial multiplication C(i + j) += A(i) * B(j)

- Affine dialect: MLIR's built-in dialect for polyhedral compilation
- Note how the affine dialect chooses to represent nested loops with nested regions

# Applications: Polyhedral code generation

```
// Affine loops are Ops with regions.
affine.for %arg0 = 0 to %N {
  // Only loop-invariant values, loop iterators, and affine functions of
  // those are allowed.
  affine.for %arg1 = 0 to %N {
    // Body of affine for loops obey SSA.
    %0 = affine.load %A[%arg0] : memref<? x f32>
    // Structured memory reference (memref) type can have
    // affine layout maps.
    %1 = affine.load %B[%arg1] : memref<? x f32, (d0)[s0] -> (d0 + s0)>
    %2 = mulf %0, %1 : f32
    // Affine load/store can have affine expressions as subscripts.
    %3 = affine.load %C[%arg0 + %arg1] : memref<? x f32>
    %4 = addf %3, %2 : f32
    affine.store %4, %C[%arg0 + %arg1] : memref<? x f32>
  }
}
```

Affine dialect representation of polynomial multiplication C(i + j) += A(i) * B(j)

- Affine dialect: MLIR's built-in dialect for polyhedral compilation
- Note how the affine dialect chooses to represent nested loops with nested regions

# Applications: Partial lowering and mixed dialects

```
toy.func @main() {
  %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00],
[4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
  %2 = toy.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>
  %3 = toy.mul %2, %2 : tensor<3x2xf64>
  toy.print %3 : tensor<3x2xf64>
  toy.return
}
```

High-level abstract representation and tensor transpose and and multiplication



```
1  func.func @main() {
2    %cst = arith.constant 1.000000e+00 : f64
3    // %cst_0, ..., %cst_4 are constants that are used in the loop body
4
5    %0 = memref.alloc() : memref<3x2xf64>
6    %1 = memref.alloc() : memref<2x3xf64>
7
8    affine.store %cst, %1[0, 0] : memref<2x3xf64>
9    // load %cst_0, ..., %cst_4 from the constant pool
10
11   affine.for %arg0 = 0 to 3 {
12     affine.for %arg1 = 0 to 2 {
13       %2 = affine.load %1[%arg1, %arg0] : memref<2x3xf64>
14       %3 = arith.mulf %2, %2 : f64
15       affine.store %3, %0[%arg0, %arg1] : memref<3x2xf64>
16     }
17   }
18
19   toy.print %0 : memref<3x2xf64>
20   memref.dealloc %1 : memref<2x3xf64>
21   memref.dealloc %0 : memref<3x2xf64>
22   return
23 }
```

- Progressively lowered into a mix of *arith, memref, affine, toy* dialects
- Apply affine loop fusion

# Applications: Partial lowering and mixed dialects

- Compose dialects together and reuse passes in other dialects
- Dialect rewrite rules defined declaratively – no need to handwrite C++ pattern matching
- Retain high-level structure
  - It would have been harder to perform loop fusion if we lower to LLVM IR immediately

High-l
transp

- Progressively lowered into a mix of *arith, memref, affine, toy* dialects
- Apply affine loop fusion

# Applications: TensorFlow graphs

```
%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, %arg0, %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
}
```

Figure. SSA representations of a TensorFlow graph in MLIR.

- Resources have “memory-like” semantics
- Tensors are SSA values ⇒ CSE, etc.

Graph is an operation with region

# Applications: TensorFlow graphs

```
%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, %arg0, %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
}
```

Figure. SSA representations of a TensorFlow graph in MLIR.

- Resources have “memory-like” semantics
- Tensors are SSA values ⇒ CSE, etc.

Graph is an operation with region

# Evaluations

- Strengths
- Weaknesses

# Strengths

- **Wide industry adoption**: Torch-MLIR, OpenXLA, OpenAI/Triton
- **Strong expressiveness**: custom syntax and type systems, custom operations and optimizations
- **Flexible abstraction level**: opportunities to further optimizations such as parallelism

# Weaknesses/Challenges

- **Inadequate design consideration**: the creators do not compare their design choices or justify why they picked them over alternatives.
  - For example, could the creators have used a (named/unnamed) region as the unit of fundamental execution, instead of defining the secondary abstraction of an operation?
- **Too much freedom**: by creating such a flexible compiler architecture, they allow too much freedom, and users can create internally fragmented dialects within MLIR, reducing its impact.

# Questions?