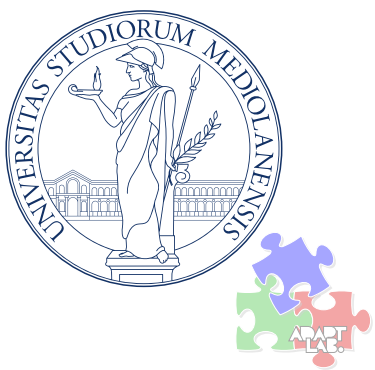
MLIR: Scaling Compiler Infrastructure for Domain Specific Computation



Federico Bruzzone,[1] PhD Candidate

Milan, Italy – 18 March 2026

Supervised by Prof. Walter Cazzola

[1]ADAPT Lab – University of Milan,
Website: federicobruzzone.github.io,
Github: github.com/FedericoBruzzone,
Email: federico.bruzzone@unimi.it
Slides: federicobruzzone.github.io/activities/presentations/MLIR.pdf

MLIR: Multi-Level Intermediate Representation

Part of the LLVM project, the MLIR is a novel approach to building **reusable**, **modular**, and **extensible** compiler infrastructure.

MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building **domain specific compilers**, and aid in connecting existing compilers together.

Why another compiler infrastructure?

Although the *one size fits all* approach of traditional compilers (e.g., LLVM [1] and JVM [2]) has been successful for general-purpose programming, it has shown limitations in the context of domain-specific applications.

Many problems are better modeled at a **higher-** or **lower-level abstraction** — e.g., source-level static analysis of C++/Rust is difficult on LLVM IR.

Hence, many languages and frameworks developed their own intermediate representations (IRs) to leverage the **semantic information** of their domain — including TensorFlow's XLA HLO, PyTorch's Glow, Rust's MIR, Swift's SIL, Clang's CIL, and so on.

While domain-specific IRs are well-understood, their *high engineering costs* often lead to compromised infrastructure quality. This results in *suboptimal compilers* plagued by bugs, latency, and a poor debugging experience [3].

MLIR to the rescue

MLIR directly addresses these issues by making it **cheap** to design and **introduce** new abstraction layers.

It achieves this by:

- standardizing the Static Single Assignment (SSA)-based IR data structures,
- providing a declarative system for defining IR *dialects*, and
- providing a wide range of common infrastructure including documentation, parsing and printing logic, location tracking, multithreaded compilation support, pass management.

Design Principles

- **Parsimony**: Apply *Occam's razor* to builtin semantics, concepts, and programming interface. Specify invariants once, but verify correctness throughout $\Longrightarrow$ *extensibility*.
- **Traceability**: Retain rather than recover information. Declare rules and properties to enable transformation, rather than step wise imperative specification $\Longrightarrow$ *composability*.
- **Progressivity**: Premature lowering is the root of all evil. Beyond representation layers, allow multiple transformation paths that lower individual regions on demand $\Longrightarrow$ *reusability*.

Little Builtin, Everything Customizable

[Parsimony]

- The system is based on a minimal number of fundamental concepts, leaving most of the intermediate representation fully **customizable**.
- A handful of abstractions—types, operations and attributes—should be used to express *everything else*, allowing fewer and more consistent abstractions that are easy to **comprehend**, **extend**, and **adopt**.
- A success criterion for customization is the possibility to express a diverse set of abstractions including **ML graphs**, ASTs, mathematical abstractions such as **polyhedral**, CFGs and instruction-level IRs such as **LLVM IR**, without hard-coding concepts.

SSA and Regions

[Parsimony]

- **SSA** [4] makes dataflow analysis *simple* and *sparse*. However, while many existing IRs use this flat, linearized CFG, representing higher level abstractions push introducing **nested regions**[2] as a first-class citizen — e.g., structured control flow, concurrency constructs, and closures.
- The (LLVM) normalization/canonicalization process is sacrificed due to the presence of multiple ways to represent the same semantics.
- The frontend is responsible for choosing the level of abstraction for the IR.

[2] A region is a single-entry, multi-exit CFG that can be nested inside an operation. It is a generalization of the concept of basic blocks and allows for more flexible control flow representation.

The Canonical Loop Structure

Pre-header, *header*, *latch*, and *body* is a prototypical loop structure.

```
; for (int i = 0; i < n; ++i) { ... }                    LLVM
entry:
  br label %header
header:
  %i = phi i32 [ 0, %entry ], [ %inc, %latch ]
  %cmp = icmp slt i32 %i, %n
  br i1 %cmp, label %body, label %multi-exit
latch:
  %inc = add i32 %i, 1
  br label %header
body:
  ; loop body
  br label %latch
multi-exit:
  ; code after the loop
```

```
// A simple loop from 0 to 10 with a step of 1          mlir
scf.for %i = %c0 to %c10 step %c1 {
  // Loop body goes here
  // %i is the induction variable
  "some.operation"(%i) : (index) -> ()
}
```

```
// An affine loop: optimized for polyhedral compilation  mlir
affine.for %i = 0 to 10 {
    %val = affine.load %buffer[%i] : memref<10xf32>
    // ... operations ...
}
```

Maintain Higher-Level Semantics

[Progressivity]

- Attempts to **recover** abstract semantics once lowered are fragile and often **fail** to capture the full semantics.
- The system should maintain the structure of computations and **progressively lower** to the hardware abstraction.
- Removing structured control flow — i.e. lowering to a CFG — essentially means no further transformations will be performed that exploits the structure.
- Previous compilers have been introducing multiple fixed levels of abstraction in their pipeline causing **phase ordering** issues.

Declaration and Validation

[Parsimony & Traceability]

- Defining representation modifiers should be as simple as introducing new abstractions.
- Common transformations should be implementable as **rewrite rules** expressed declaratively.
- Although rewriting systems are well-studied, the MLIR's extensibility opens up new challenges.
- While verification, testing, and translation validation [5] are useful a more robust approach to combining all these techniques for **extensible** and **modular** IRs.

Source Location Tracking

[Traceability]

- **Lack-of-transparency** in complex compilation systems is ubiquitous. This is particularly problematic when compiling safety-critical and sensitive applications (cf. `WYSINWYX` by Balakrishnan et al. [6]).
- Thus, the **provenance** of an operation — including its original location and applied transformations — should be easily traceable within MLIR.
- One indirect goal of accurately propagating high-level information to the lower levels is to help support **secure** and **traceable** compilation.

Intermediate Representation Design

MLIR has a *generic* textual representation that supports MLIR's extensibility and fully reflects the in-memory representation, which is paramount for **traceability**, manual IR **validation** and **testing**. Extensibility comes with the burden of verbosity, which can be compensated by the custom syntax that MLIR supports.

$$C_{i+j} \leftarrow C_{i+j} + (A_i * B_j)$$

```mlir
// 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) -> ()
```

IR: *Operations*

[Parsimony]

The unit of semantics in MLIR is an **operation** (Op): *instruction*, *function* and *module* are modeled as Ops.

MLIR does not have a fixed set of Ops, but allows user-defined extensions.

The infrastructure provides a declarative syntax for defining Ops based on the well-known **TableGen**.[3]

```
// An Op is a TableGen definition that inherits the "Op" class parameterized with the Op name
def LeakyReluOp: Op<"leaky_relu",
    // and a list of traits used for verification and optimization.
    [NoSideEffect, SameOperandsAndResultType]> {
  // The body of the definition contains named fields for a one-line
  // documentation summary for the Op.
  let summary = "Leaky Relu operator";
  // The Op can also a full-text description that can be used to generate
  // documentation for the dialect.
  let description = [{
    Element-wise Leaky ReLU operator x -> x >= 0 ? x : (alpha * x) }];
  // Op can have a list of named arguments, which include typed operands and attributes.
  let arguments = (ins AnyTensor:$input, F32Attr:$alpha);
  // And a list of named and typed outputs.
  let results = (outs AnyTensor:$output);
}
```
tablegen

[3] https://llvm.org/docs/TableGen/

IR: *Operations* (cont.)

[Parsimony]

Ops have a **unique** opcode: the operation and its dialect.

Ops take and produce zero or more SSA *operands* and *results*.

Values represent runtime data and are fully typed to ensure compile-time knowledge.

Ops may also have *Attributes*, *Regions*, *Successor Blocks*, and *Location* Information.

```mlir
// 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) -> ()
```

IR: *Attributes*

[Parsimony]

MLIR **attributes** contain compile-time information about Ops.

Attributes are typed (e.g., integer, string), and each Op instance has an open key-value dictionary from string names to attribute values.

Attributes derive their meaning either from the **Op semantics** or from the **dialect** they are associated with.

As with opcodes, there is no fixed set of attributes.

```mlir
// 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, %a1) : (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) -> ()
```

IR: *Location Information*

[Traceability]

MLIR provides a compact representation for **location information**, and encourages the processing and propagation of this information throughout the system, following the **traceability** principle.

It can be used to keep the source program stack trace that produced an Op, to generate debug information.

```mlir
// 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 loc("kernel.c":10:12)
    %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)} : (memref<?xf32>, index) -> f32 loc("kernel.c":11:12)
    %2 = "std.mulf"(%0, %a1) : (f32, f32) -> f32 loc(fused["kernel.c":12:15, "params.h":5:2])
    %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1} : (memref<?xf32>, index, index) -> f32 loc("kernel.c":13:8)
    %4 = "std.addf"(%3, %2) : (f32, f32) -> f32 loc("kernel.c":13:14)
    "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1} : (f32, memref<?xf32>, index, index) -> () loc("kernel.c":13:5)
    // Blocks end with a terminator Op.
    "affine.terminator"() : () -> ()
  // Ops have a list of attributes.
  }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":9:5)
  "affine.terminator"() : () -> ()
}) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":8:3)
```

IR: *Regions and Blocks*

[Progressivity]

An instance of an Op may have a list of attached **regions**.

A region contains a list of **blocks**, each of which contains a list of Ops.

As with *attributes*, the semantics of a region are defined by the operation they are attached to. However the blocks inside the region form a CFG through the use of **terminator** operations that specify the successor blocks.

```mlir
// 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 loc("kernel.c":10:12)
    %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)} : (memref<?xf32>, index) -> f32 loc("kernel.c":11:12)
    %2 = "std.mulf"(%0, %a1) : (f32, f32) -> f32 loc(fused["kernel.c":12:15, "params.h":5:2])
    %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1} : (memref<?xf32>, index, index) -> f32 loc("kernel.c":13:8)
    %4 = "std.addf"(%3, %2) : (f32, f32) -> f32 loc("kernel.c":13:14)
    "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1} : (f32, memref<?xf32>, index, index) -> () loc("kernel.c":13:5)
    // Blocks end with a terminator Op.
    "affine.terminator"() : () -> ()
  // Ops have a list of attributes.
  }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":9:5)
  "affine.terminator"() : () -> ()
}) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":8:3)
```

IR: *Regions and Blocks* (Cont.)

[Progressivity]

Regions and block allows **nesting** mechanisms that can be used to represent *structured control flow* etc.

Instead of using φ nodes, MLIR uses a **functional** form of SSA [7] — terminators pass values into *block arguments* defined by the successor block.

For the more attentive readers, there is a strong correlation with the **Sea of Nodes** IR [8], [9].

```mlir
// 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 loc("kernel.c":10:12)
    %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)} : (memref<?xf32>, index) -> f32 loc("kernel.c":11:12)
    %2 = "std.mulf"(%0, %a1) : (f32, f32) -> f32 loc(fused["kernel.c":12:15, "params.h":5:2])
    %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1} : (memref<?xf32>, index, index) -> f32 loc("kernel.c":13:8)
    %4 = "std.addf"(%3, %2) : (f32, f32) -> f32 loc("kernel.c":13:14)
    "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1} : (f32, memref<?xf32>, index, index) -> () loc("kernel.c":13:5)
    // Blocks end with a terminator Op.
    "affine.terminator"() : () -> ()
  // Ops have a list of attributes.
  }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":9:5)
  "affine.terminator"() : () -> ()
}) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":8:3)
```

IR: *Value Dominance and Visibility*

[Progressivity]

Ops can only use values that are in scope, i.e. **visible** according to SSA dominance, nesting, and semantic restrictions imposed by enclosing operations.

Region-based visibility is defined based on simple nesting of regions.

MLIR also allows operations to be defined as *isolated from above*, indicating that the operation is a scope barrier — e.g., `std.func` Op.

```mlir
// 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 loc("kernel.c":10:12)
    %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)} : (memref<?xf32>, index) -> f32 loc("kernel.c":11:12)
    %2 = "std.mulf"(%0, %a1) : (f32, f32) -> f32 loc(fused["kernel.c":12:15, "params.h":5:2])
    %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1} : (memref<?xf32>, index, index) -> f32 loc("kernel.c":13:8)
    %4 = "std.addf"(%3, %2) : (f32, f32) -> f32 loc("kernel.c":13:14)
    "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1} : (f32, memref<?xf32>, index, index) -> () loc("kernel.c":13:5)
    // Blocks end with a terminator Op.
    "affine.terminator"() : () -> ()
  // Ops have a list of attributes.
  }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":9:5)
  "affine.terminator"() : () -> ()
}) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":8:3)
```

IR: *Symbols and Symbol Tables*

[Traceability]

Ops can have a **symbol table** attached: a standardized way of associating *names* to *IR objects* (**symbols**) — e.g., global variables, functions or named modules.

The IR does not prescribe **what** symbols are used for, leaving it up to the Op definition.

Without this mechanism, it would have been **impossible** to define recursive function.

```mlir
module @kernel_module {
  func.func @compute_kernel(%arg0: index, %arg1: memref<?xf32>, %arg2: memref<?xf32>, %arg3: memref<?xf32>) {
    // 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 loc("kernel.c":10:12)
        %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)} : (memref<?xf32>, index) -> f32 loc("kernel.c":11:12)
        %2 = "std.mulf"(%0, %a1) : (f32, f32) -> f32 loc(fused["kernel.c":12:15, "params.h":5:2])
        %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1} : (memref<?xf32>, index, index) -> f32 loc("kernel.c":13:8)
        %4 = "std.addf"(%3, %2) : (f32, f32) -> f32 loc("kernel.c":13:14)
        "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1} : (f32, memref<?xf32>, index, index) -> () loc("kernel.c":13:5)
        // Blocks end with a terminator Op.
        "affine.terminator"() : () -> ()
      // Ops have a list of attributes.
      }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":9:5)
      "affine.terminator"() : () -> ()
    }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":8:3)
    return
  }
}
```

IR: *Dialects*

[Progressivity]

MLIR manages extensibility using **Dialects**, which provide a logical grouping of Ops, attributes and types under a unique namespace.

Dialects themselves do not introduce any new semantics but serve as a logical grouping mechanism that provides common Op functionality (e.g., constant folding).

This separation is conceptual and is akin to designing a set of **modular** libraries.

```mlir
module @kernel_module {
  func.func @compute_kernel(%arg0: index, %arg1: memref<?xf32>, %arg2: memref<?xf32>, %arg3: memref<?xf32>) {
  // 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 loc("kernel.c":10:12)
      %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)} : (memref<?xf32>, index) -> f32 loc("kernel.c":11:12)
      %2 = "std.mulf"(%0, %a1) : (f32, f32) -> f32 loc(fused["kernel.c":12:15, "params.h":5:2])
      %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1} : (memref<?xf32>, index, index) -> f32 loc("kernel.c":13:8)
      %4 = "std.addf"(%3, %2) : (f32, f32) -> f32 loc("kernel.c":13:14)
      "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1} : (f32, memref<?xf32>, index, index) -> () loc("kernel.c":13:5)
      // Blocks end with a terminator Op.
      "affine.terminator"() : () -> ()
    // Ops have a list of attributes.
    }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":9:5)
    "affine.terminator"() : () -> ()
  }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":8:3)
  return
  }
}
```

Ops from different dialects can *coexist* at any level of the IR at any time, they can use types defined in different dialects, etc.

Intermixing of dialects allows for greater *reuse*, *extensibility* and provides *flexibility* that otherwise would require developers to resort to all kinds of non-composable workarounds.

IR: *Type System*

[Parsimony]

Every value in MLIR has a **type**, which is specified in the Op that produces the value or in the block that defines the value as an argument — types encode compile-time information.

While the type system in MLIR is user-extensible, it enforces **strict** type equality checking and does **not** provide type conversion rules.

From the **type theory** point of view, MLIR does not support dependent types (but it can be encoded).

```mlir
module @kernel_module {
  func.func @compute_kernel(%arg0: index, %arg1: memref<?xf32>, %arg2: memref<?xf32>, %arg3: memref<?xf32>) {
    // 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 loc("kernel.c":10:12)
        %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)} : (memref<?xf32>, index) -> f32 loc("kernel.c":11:12)
        %2 = "std.mulf"(%0, %a1) : (f32, f32) -> f32 loc(fused["kernel.c":12:15, "params.h":5:2])
        %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1} : (memref<?xf32>, index, index) -> f32 loc("kernel.c":13:8)
        %4 = "std.addf"(%3, %2) : (f32, f32) -> f32 loc("kernel.c":13:14)
        "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1} : (f32, memref<?xf32>, index, index) -> () loc("kernel.c":13:5)
        // Blocks end with a terminator Op.
        "affine.terminator"() : () -> ()
      // Ops have a list of attributes.
      }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":9:5)
      "affine.terminator"() : () -> ()
    }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":8:3)
    return
  }
}
```

IR: *Functions and Modules*

[Parsimony]

Similarly to conventional IRs, MLIR is usually structured into **functions** and **modules**.

However, these are **not** separate concepts in MLIR: they are implemented as Ops in the `builtin` and `func` dialect.

A **module** is an Op with a single region containing a single block and does not transfer the control flow.

A **function** is an Op with a single region that may contain zero (in case of declaration) or more blocks.

```mlir
module @kernel_module {
  func.func @compute_kernel(%arg0: index, %arg1: memref<?xf32>, %arg2: memref<?xf32>, %arg3: memref<?xf32>) {
    // 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 loc("kernel.c":10:12)
        %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)} : (memref<?xf32>, index) -> f32 loc("kernel.c":11:12)
        %2 = "std.mulf"(%0, %a1) : (f32, f32) -> f32 loc(fused["kernel.c":12:15, "params.h":5:2])
        %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1} : (memref<?xf32>, index, index) -> f32 loc("kernel.c":13:8)
        %4 = "std.addf"(%3, %2) : (f32, f32) -> f32 loc("kernel.c":13:14)
        "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1} : (f32, memref<?xf32>, index, index) -> () loc("kernel.c":13:5)
        // Blocks end with a terminator Op.
        "affine.terminator"() : () -> ()
      // Ops have a list of attributes.
      }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":9:5)
      "affine.terminator"() : () -> ()
    }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":8:3)
    return
  }
}
```

But, a custom syntax?

Before

```mlir
module @kernel_module {
  func.func @compute_kernel(%arg0: index, %arg1: memref<?xf32>, %arg2: memref<?xf32>, %arg3: memref<?xf32>) {
    // 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 loc("kernel.c":10:12)
        %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)} : (memref<?xf32>, index) -> f32 loc("kernel.c":11:12)
        %2 = "std.mulf"(%0, %a1) : (f32, f32) -> f32 loc(fused["kernel.c":12:15, "params.h":5:2])
        %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1} : (memref<?xf32>, index, index) -> f32 loc("kernel.c":13:8)
        %4 = "std.addf"(%3, %2) : (f32, f32) -> f32 loc("kernel.c":13:14)
        "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1} : (f32, memref<?xf32>, index, index) -> () loc("kernel.c":13:5)
        // Blocks end with a terminator Op.
        "affine.terminator"() : () -> ()
      // Ops have a list of attributes.
      }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":9:5)
      "affine.terminator"() : () -> ()
    }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3} : (index) -> () loc("kernel.c":8:3)
    return
  }
}
```

After

```mlir
module @kernel_module {
  func.func @compute_kernel(%N: index, %A: memref<?xf32>, %B: memref<?xf32>, %C: memref<?xf32>) {



    // Outer loop: iterates from 0 to %N
    affine.for %i = 0 to %N {


      // Inner loop: iterates from 0 to %N
      affine.for %j = 0 to %N {

        // Load values from input memrefs using affine identifiers
        %val_a = affine.load %A[%i] : memref<?xf32>
        %val_b = affine.load %B[%j] : memref<?xf32>
        // Floating point multiplication using the Arith dialect
        %prod = arith.mulf %val_a, %val_b : f32
        // Accessing memory with a compound affine expression [%i + %j]
        %val_c = affine.load %C[%i + %j] : memref<?xf32>
        %sum = arith.addf %val_c, %prod : f32
        // Store the final computed value back into the destination memref
        affine.store %sum, %C[%i + %j] : memref<?xf32>
      }

    }
    return
  }
}
```

Putting it all together

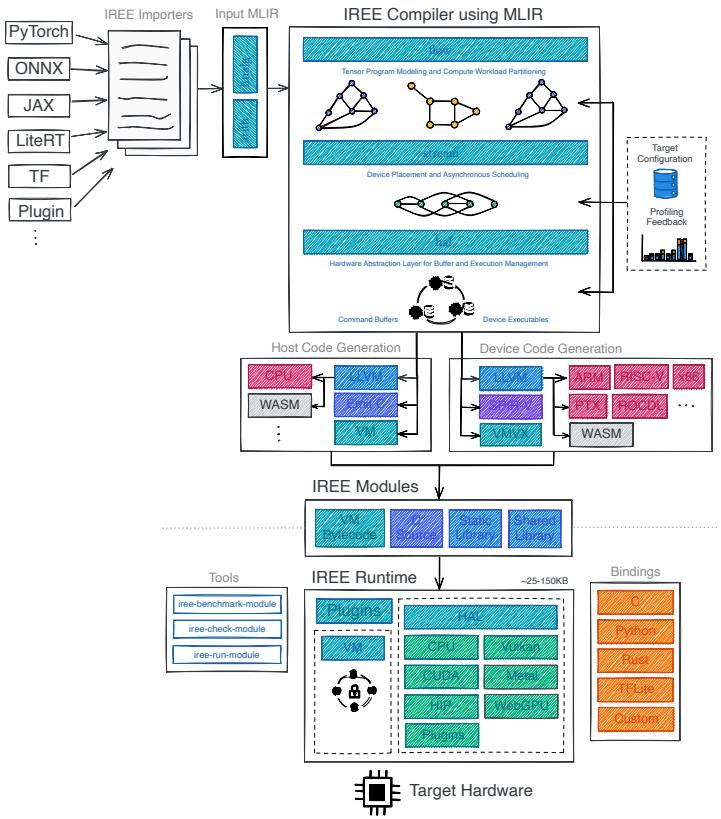
```mlir
module @kernel_module {
    func.func @compute_kernel(%N: index, %A: memref<?xf32>, %B: memref<?xf32>, %C: memref<?xf32>) {
        affine.for %i = 0 to %N {
            affine.for %j = 0 to %N {
                %val_a = affine.load %A[%i] : memref<?xf32>
                %val_b = affine.load %B[%j] : memref<?xf32>
                %prod = arith.mulf %val_a, %val_b : f32
                %val_c = affine.load %C[%i + %j] : memref<?xf32>
                %sum = arith.addf %val_c, %prod : f32
                affine.store %sum, %C[%i + %j] : memref<?xf32>
            }
        }
        return
    }
}
```

The `mlir-showcase` Repository

github.com/FedericoBruzzone/mlir-showcase

IREE: Intermediate Representation Execution Environment

IREE [10][4] is an MLIR-based end-to-end compiler and runtime that lowers ML models to a unified IR that scales to meet the needs of the *datacenter* and *mobile/edge* deployments.



[4] https://github.com/iree-org/iree

Export the TensorFlow Model

```python
import os
import tensorflow as tf
# Load the pre-trained MobileNetV2 model with ImageNet weights
model = tf.keras.applications.MobileNetV2(weights="imagenet")
# Export the model as a TF SavedModel (temporary, with default signatures)
model.export("mobilenet_v2_saved_model")
# Reload the exported SavedModel
loaded_model = tf.saved_model.load("mobilenet_v2_saved_model")
# Create a concrete function with a fixed batch size input signature
@tf.function(input_signature=[tf.TensorSpec([1, 224, 224, 3], tf.float32)])
def serve(input):
  return loaded_model.signatures["serve"](input)
# Re-save the model with the fixed input signature
tf.saved_model.save(
  loaded_model, "mobilenet_v2_saved_model", signatures={"serve": serve})
```

```
├── mobilenet_v2_saved_model/            # Exported TF SavedModel directory
│   ├── saved_model.pb                   #   Graph definition + signatures
│   ├── fingerprint.pb                   #   Model integrity hash
│   ├── assets/                          #   Extra assets
│   └── variables/                       #   Model weights
│       ├── variables.data-00000-of-00001 # Actual weight values (binary)
│       └── variables.index              # Index/lookup for the weight shards
```

We expect the model to have a **single** signature named `serve` that takes a single input tensor and produces a single output tensor.

```bash
python signatures.py
```

```
Signatures found: ['serve']
```

Import the Model into MLIR

Convert the TensorFlow SavedModel into MLIR

```bash
iree-import-tf \
  mobilenet_v2_saved_model \
  --tf-import-type=savedmodel_v1 \
  --tf-savedmodel-exported-names=serve \
  -o mobilenet_v2.mlirbc
```

From MLIR bitcode to a human-readable MLIR file

```bash
iree-ir-tool \
  copy mobilenet_v2.mlirbc \
  -o mobilenet_v2_readable.mlir
```

Compile to a VM FlatBuffer

```bash
iree-compile \
  mobilenet_v2.mlirbc \
  --iree-hal-target-backends=llvm-cpu \
  -o mobilenet_v2.vmfb
```

```mlir
module {
  ml_program.global public @"vars.block_10_depthwise/kernel_1"(dense<"..."> : tensor<3x3x384x1xf32>) : tensor<3x3x384x1xf32>
  ...
  ml_program.global public @"vars.block_10_project_BN/gamma_1"(dense<[1.99929357, ..., 1.9186883]> : tensor<96xf32>) : tensor<96xf32>
  ml_program.global public @"vars.block_11_project_BN/beta_1"(dense<[-0.00110509305, ..., 8.97456601E-4]> : tensor<96xf32>) : tensor<96xf32>
  ...
  func.func @session_initializer() { return }
  func.func @serve(%arg0: tensor<1x224x224x3xf32>) -> tensor<1x1000xf32> {
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
    %cst_0 = stablehlo.constant dense<6.000000e+00> : tensor<f32>
    ...
    %vars.block_10_depthwise2Fkernel_1 = ml_program.global_load @"vars.block_10_depthwise/kernel_1" : tensor<3x3x384x1xf32>
    %vars.block_10_depthwise_BN2Fmoving_variance_1 = ml_program.global_load @"vars.block_10_depthwise_BN/moving_variance_1" : tensor<384xf32>
    ...
    %vars.bn_Conv12Fgamma_1 = ml_program.global_load @"vars.bn_Conv1/gamma_1" : tensor<32xf32>
    %vars.bn_Conv12Fbeta_1 = ml_program.global_load @"vars.bn_Conv1/beta_1" : tensor<32xf32>
    ...
    %vars.predictions2Fbias_1 = ml_program.global_load @"vars.predictions/bias_1" : tensor<1000xf32>
    %vars.predictions2Fkernel_1 = ml_program.global_load @"vars.predictions/kernel_1" : tensor<1280x1000xf32>
    ...
    %0 = stablehlo.add %vars.block_10_depthwise_BN2Fmoving_variance_1, %cst_3 : tensor<384xf32>
    %1 = stablehlo.rsqrt %0 : tensor<384xf32>
    %2 = stablehlo.multiply %1, %vars.block_10_depthwise_BN2Fgamma_1 : tensor<384xf32>
    %3 = stablehlo.multiply %vars.block_10_depthwise_BN2Fmoving_mean_1, %2 : tensor<384xf32>
    %4 = stablehlo.subtract %vars.block_10_depthwise_BN2Fbeta_1, %3 : tensor<384xf32>
    %5 = stablehlo.add %vars.block_10_expand_BN2Fmoving_variance_1, %cst_3 : tensor<384xf32>
    %6 = stablehlo.rsqrt %5 : tensor<384xf32>
    %7 = stablehlo.multiply %6, %vars.block_10_expand_BN2Fgamma_1 : tensor<384xf32>
    %8 = stablehlo.multiply %vars.block_10_expand_BN2Fmoving_mean_1, %7 : tensor<384xf32>
    %9 = stablehlo.subtract %vars.block_10_expand_BN2Fbeta_1, %8 : tensor<384xf32>
    ...
    %596 = stablehlo.reduce(%595 init: %cst_1) applies stablehlo.add across dimensions = [1] : (tensor<1x1000xf32>, tensor<f32>) -> tensor<1xf32>
    %597 = stablehlo.reshape %596 : (tensor<1xf32>) -> tensor<1x1xf32>
    %598 = stablehlo.broadcast_in_dim %597, dims = [0, 1] : (tensor<1x1xf32>) -> tensor<1x1000xf32>
    %599 = stablehlo.divide %595, %598 : tensor<1x1000xf32>
    return %599 : tensor<1x1000xf32>
  }
}
```

Generate an Input Tensor and Inference with IREE

```python
img = Image.open(image_path).convert("RGB")
img = img.resize((224, 224), Image.BILINEAR)
input_data = np.array(img, dtype=np.float32)     # shape: (224, 224, 3)
input_data = input_data / 127.5 - 1.0            # scale to [-1, 1]
input_data = np.expand_dims(input_data, axis=0)  # shape: (1, 224, 224, 3)
np.save("input.npy", input_data)
```

input.npy


```
äãc¿ÒÑQ¿ÞÝ]¿ÞÝ]¿ÒÑQ¿ÜÛ[¿Ø×W¿ÐÏO¿Ø×W¿ÚÙY¿ÆÅE¿Ø×W¿ÜÛ
[¿ÈÇG¿ÞÝ]¿ÚÙY¿ÃÃC¿ÚÙY¿ÖÕU¿ÌËK¿ÖÕU¿äãc¿ÖÕU¿äãc¿óós¿ðï
o¿öõu¿úùy¿üû{¿öõu¿úùy¿ø÷w¿ø÷w¿îím¿ÒÑQ¿îím¿âáa¿º¹9¿æåe¿â
áa¿¾»;¿èçg¿îím¿ÒÑQ¿óós¿ø÷w¿òñq¿ø÷w¿üû{¿üû{¿úùy¿ø÷w¿üû{¿üû
{¿öõu¿úùy¿ø÷w¿úùy¿úùy¿úùy¿úùy¿ø÷w¿úùy¿üû{¿öõu¿óós¿úùy¿
öõu¿úùy¿úùy¿ø÷w¿úùy¿üû{¿ø÷w¿óós¿úùy¿öõu¿üû{¿üû{¿ø÷w¿úùy¿üû{
¿öõu¿úùy¿úùy¿öõu¿öõu¿ø÷w¿öõu¿öõu¿úùy¿ø÷w¿öõu¿úùy¿ø÷w¿ø÷w¿ú
ùy¿ø÷w¿ø÷w¿úùy¿úùy¿óós¿ðïo¿óós¿òñq¿èçg¿îím¿ðïo¿ìëk¿óós¿òñq¿
òñq¿ðïo¿úùy¿úùy¿óós¿ø÷w¿úùy¿öõu¿ø÷w¿öõu¿òñq¿îím¿ðïo¿ìëk¿îím
¿ìëk¿îím¿ìëk¿îím¿êéi¿êéi¿îím¿ìëk¿ìëk¿ðïo¿ðïo¿öõu¿öõu¿óós¿öõu¿
úùy¿ø÷w¿ðïo¿ø÷w¿öõu¿ÚÙY¿ìëk¿óós¿ðïo¿úùy¿öõu¿öõu¿úùy¿öõu¿óós¿ø÷
```


From the command line, we can now run the model with the generated input:

```bash
iree-run-module \
  --module=mobilenet_v2.vmfb \
  --function=serve \
  --input=@input.npy
```

```
EXEC @serve
result[0]: hal.buffer_view
1x1000xf32=[0.000165652 0.000151373 1.42592E-05 5.92925E-05 6.70149E-05 0.000341131
0.000198785 0.000122406 0.00149087 4.39386E-05 3.3403E-05 5.3721E-05 5.35976E-05
1.82398E-05 9.94686E-05 9.30952E-05 6.3427E-05 0.000141799 3.17409E-05 3.06959E-05
6.87135E-05 0.000216699 6.09083E-05 7.07725E-05 2.60562E-05 0.000153595 0.000109031
0.000178287 5.53506E-05 0.000147815 0.000209688 0.000162711 0.000243295 1.71809E-05
0.000100425 1.57999E-05 0.000125091 0.000178397 6.25113E-05 7.59963E-05 9.62788E-05
3.86099E-05 0.00012561 3.23711E-05 4.97341E-05 6.13783E-05 0.000138996 6.39329E-05
4.45096E-05 6.67697E-05 9.46544E-05 3.5926E-05 0.000204062 0.000292093 0.000488473
9.58151E-05 8.47943E-05 0.000123163 9.52857E-05 6.79532E-05 0.000132884 0.000117373
4.67249E-05 2.62787E-05 6.68047E-05 6.79969E-05 4.83555E-05 4.88604E-05 4.05947E-05
4.22143E-05 0.000371253 6.22883E-05 0.000116463 4.35457E-05 0.000192234 8.1999E-05
3.30845E-05 3.31993E-05 5.05881E-05 8.58546E-05 0.00010195 0.000208427 0.000207405
0.000130688 0.00011813 0.000120608 0.000313725 2.05671E-05 1.96285E-05 0.000157798
5.57028E-05 0.000126693 3.22056E-05 8.02654E-05 5.05778E-05 7.28497E-05 5.03175E-05
5.17303E-05 3.78772E-05 6.13501E-05 0.000115919 3.76777E-05 4.29638E-05 1.85399E-05
0.00484446 2.68151E-05 0.000148926 4.33687E-05 0.000137578 5.64227E-05 0.000435291
2.82973E-05 0.000133121 0.000222104 0.000252791 0.00011586 9.6491E-05 8.42967E-05
3.2464E-05 4.53697E-05 7.99469E-05 3.44823E-05 0.000181173 7.79511E-05 3.11126E-05
2.99579E-05 2.87822E-05 4.67909E-05 4.98542E-05 6.87359E-05 3.60754E-05 4.29585E-05
0.000123788 3.78884E-05 9.59466E-05 8.39355E-05 4.71987E-05 5.8172E-05 4.268E-05
4.00027E-05 3.32925E-05 8.46946E-05 9.10055E-05 4.89413E-05 7.85801E-05 4.35644E-05
3.85596E-05 7.80505E-05 0.000174066 0.000159893 0.000121992 0.000767081 0.00228256
0.000558238 0.000662587 1.34811E-05 3.92131E-05 0.0026327 9.529E-05 7.68284E-05
3.47142E-05 5.47353E-06 7.5111E-06 8.991E-06 3.45787E-05 1.31357E-05 1.15983E-05
4.08218E-06 1.22294E-05 6.40326E-05 5.27259E-05 4.69403E-05 9.65662E-06 8.58458E-05
0.000155262 3.94623E-05 1.61538E-05 1.289E-05 2.22543E-05 0.000125113 6.12213E-05
6.69415E-06 2.30811E-05 4.93018E-05 3.2618E-05 0.000193106 ... 0.000102141]
```

How to Interpret the Output? Let's Post-process it!

```python
# 1. Load ImageNet class names.
categories = load_imagenet_classes(os.path.join(SCRIPT_DIR, "imagenet_classes.txt"))
# 2. Configure the IREE runtime (CPU via the local-task driver).
config = ireert.Config(driver_name="local-task")
# 3. Load the compiled module (.vmfb).
ctx = ireert.SystemContext(config=config)
with open(args.model, "rb") as f:
  vm_module = ireert.VmModule.copy_buffer(ctx.instance, f.read())
ctx.add_vm_module(vm_module)
# 4. Preprocess the input image.
input_data = preprocess_image(args.image)
# 5. Invoke the "serve" function (synchronous).
serve = ctx.modules.module["serve"]
output = serve(input_data)
# 6. Convert logits to probabilities via softmax.
logits = np.asarray(output).flatten()
probabilities = softmax(logits)
# 7. Print results: top-N predictions with class names.
print(f"\nOutput shape: {list(np.asarray(output).shape)}")
top_k = min(args.top, len(probabilities))
top_indices = np.argsort(probabilities)[::-1][:top_k]
print(f"\nTop-{top_k} predictions:")
print(f"{'Rank':<6} {'Class':<30} {'Probability':>12}")
print("-" * 50)
for rank, idx in enumerate(top_indices, start=1):
  print(f"{rank:<6} {categories[idx]:<30} {probabilities[idx]:>11.4%}")
```

imagenet_classes.txt:

```
tench
goldfish
great white shark
...
Samoyed
...
bolete
ear
toilet tissue
```

The final output of the post-processing script should look like this:

```
Loaded image: dog.jpg  (original size: 1546×1213)
Running inference...

Output shape: [1, 1000]

Top-5 predictions:
Rank   Class                          Probability
--------------------------------------------------
1      Samoyed                           0.1990%
2      Arctic fox                        0.1081%
3      Pomeranian                        0.1070%
4      keeshond                          0.1015%
5      Persian cat                       0.1009%
```

Thank You!

Federico Bruzzone, PhD Candidate

ADAPT Lab – University of Milan
Website: federicobruzzone.github.io
Github: github.com/FedericoBruzzone
Email: federico.bruzzone@unimi.it

Slides: federicobruzzone.github.io/activities/presentations/MLIR.pdf

Bibliography

[1] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[2] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Addison-wesley, 2013.

[3] C. Lattner *et al.*, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14. doi: 10.1109/CGO51591.2021.9370308.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991, doi: 10.1145/115372.115320.

[5] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Steffen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 151–166.

[6] G. Balakrishnan and T. Reps, "WYSINWYX: What you see is not what you eXecute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, Aug. 2010, doi: 10.1145/1749608.1749612.

[7] A. W. Appel, "SSA is functional programming," *SIGPLAN Not.*, vol. 33, no. 4, pp. 17–20, Apr. 1998, doi: 10.1145/278283.278285.

[8] C. Click and M. Paleczny, "A simple graph-based intermediate representation," *SIGPLAN Not.*, vol. 30, no. 3, pp. 35–49, Mar. 1995, doi: 10.1145/202530.202534.

[9] C. Click and K. D. Cooper, "Combining analyses, combining optimizations," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 2, pp. 181–196, Mar. 1995, doi: 10.1145/201059.201061.

[10] H.-I. C. Liu, M. Brehler, M. Ravishankar, N. Vasilache, B. Vanik, and S. Laurenzo, "TinyIREE: An ML Execution Environment for Embedded Systems From Compilation to Deployment," *IEEE Micro*, vol. 42, no. 5, pp. 9–16, 2022, doi: 10.1109/MM.2022.3178068.