MLIR: Scaling Compiler Infrastructure for Domain Specific Computation

Chris Lattner
*Google, USA**
clattner@llvm.org

Mehdi Amini
Google, USA
orcid.org/0000-0002-2066-3106

Uday Bondhugula
Indian Institute of Science, India†
orcid.org/0000-0002-8297-6159

Albert Cohen
Google, France
orcid.org/0000-0002-8866-5343

Andy Davis
Google, USA

Jacques Pienaar
Google, USA
orcid.org/0000-0003-0443-7624

River Riddle
Google, USA

Tatiana Shpeisman
Google, USA

Nicolas Vasilache
Google, USA

Oleksandr Zinenko
Google, France
orcid.org/0000-0003-1978-0222

***Abstract*—This work presents MLIR, a novel approach to building reusable and extensible compiler infrastructure. MLIR addresses software fragmentation, compilation for heterogeneous hardware, significantly reducing the cost of building domain specific compilers, and connecting existing compilers together.**

MLIR facilitates the design and implementation of code generators, translators and optimizers at different levels of abstraction and across application domains, hardware targets and execution environments. The contribution of this work includes (1) discussion of MLIR as a research artifact, built for extension and evolution, while identifying the challenges and opportunities posed by this novel design, semantics, optimization specification, system, and engineering. (2) evaluation of MLIR as a generalized infrastructure that reduces the cost of building compilers—describing diverse use-cases to show research and educational opportunities for future programming languages, compilers, execution environments, and computer architecture. The paper also presents the rationale for MLIR, its original design principles, structures and semantics.

I. Introduction

Compiler design is a mature field with applications to code generation, static analysis, and more. The field has seen the development of a number of mature technology platforms which have enabled massive reuse, including systems like the LLVM compiler infrastructure [1], the Java Virtual Machine (JVM) [2], and many others. A common characteristic of these popular systems is their "one size fits all" approach—a *single abstraction level* to interface with the system: the LLVM Intermediate Representation (IR) is roughly "C with vectors", and JVM provides an "object-oriented type system with a garbage collector" abstraction. This "one size fits all" approach is incredibly valuable—and in practice, the mapping to these domains from ubiquitous source languages (C/C++ and Java respectively) is straightforward.

At the same time, many problems are better modeled at a higher- or lower-level abstraction, e.g. source-level analysis of C++ code is very difficult on LLVM IR. We observe that many languages (including e.g. Swift, Rust, Julia, Fortran) develop their own IR in order to solve domain-specific problems, like language/library-specific optimizations, flow-sensitive type checking (e.g. for linear types), and to improve the implementation of the lowering process. Similarly, machine learning systems typically use "ML graphs" as a domain-specific abstraction in the same way.

While the development of domain-specific IRs is a well studied art, their engineering and implementation cost remains high. The quality of the infrastructure is not always a first priority (or easy to justify) for implementers of these systems. Consequently, this can lead to lower quality compiler systems, including user-visible problems like slow compile times, buggy implementations, suboptimal diagnostic quality, poor debugging experience for optimized code, etc.

The MLIR project[1] aims to directly tackle these programming language design and implementation challenges—by making it cheap to define and introduce new abstraction levels, and provide "in the box" infrastructure to solve common compiler engineering problems. MLIR does this by (1) standardizing the Static Single Assignment (SSA)-based IR data structures, (2) providing a declarative system for defining *IR dialects*, and (3) providing a wide range of common infrastructure including documentation, parsing and printing logic, location tracking, multithreaded compilation support, pass management, etc.

This paper further presents the overarching principles underlying the design and implementation of MLIR. We will explore the essential design points of the system and how they relate to the overarching principles, sharing our experience applying MLIR to a number of compilation problems.

A. Contributions

Most of the MLIR system is built out of well known concepts and algorithms. Yet the objectives and design are sufficiently

*With SiFive at the time of publication.

†Visiting researcher at Google at the time of this work.

[1]https://mlir.llvm.org

novel that studying them offer vast opportunities for research, and even more so within the boundaries of the following overarching principles:

Parsimony: Apply Occam's razor to builtin semantics, concepts, and programming interface. Harness both intrinsic and incidental complexity by abstracting properties of operations and types. Specify invariants once, but verify correctness throughout. Query properties in the context of a given compilation pass. With very little builtin, this opens the door to extensibility and customization.

Traceability: Retain rather than recover information. Declare rules and properties to enable transformation, rather than step wise imperative specification. Extensibility comes with generic means to trace information, enforced by extensive verification. Composable abstractions stem from "glassboxing" their properties and separating their roles—type, control, data flow, etc.

Progressivity: Premature lowering is the root of all evil. Beyond representation layers, allow multiple transformation paths that lower individual regions on demand. Together with abstraction-independent principles and interfaces, this enables reuse across multiple domains.

While these principles are well established, one of them is often implemented at the expense of another; e.g., layering in network and operating system stacks aligns with the progressivity principle but breaks parsimony. This has also been the case in compilers with multiple layers of IR. Also, following these principles may hurt expressiveness and effectiveness; e.g., traceability in safety-critical and secure systems involves limiting optimizations and their aggressivity.

In a nutshell, we identify design and engineering principles for compiler construction to thrive in a narrow middle that support an open semantics ecosystem. We discovered complexity can be tamed without restricting expressivity, allowing for fast IR design exploration and consolidation across domains, both of which are severely lacking in production systems.

The contributions of this paper are: (1) positioning the problem of building scalable and modular compiler systems in terms of proven design and engineering principles; (2) a description of a novel compiler infrastructure that follows these principles, with important industrial and research applications; (3) exploration of selected applications to diverse domains, illustrating the generality of the approach and sharing experience developing systems that build on the MLIR infrastructure.

B. Where Did MLIR Come From?

Work on MLIR began with a realization that modern machine learning frameworks are composed of many different compilers, graph technologies, and runtime systems (see Figure 1)—which did not share a common infrastructure or design principles. This manifested in multiple user-visible ways, including poor error messages, failures in edge cases, unpredictable performance, and difficulty generalizing the stack to support new hardware.

We soon realized that the compiler industry as a whole has a similar problem: existing systems like LLVM are very

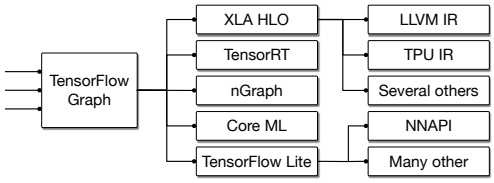

Fig. 1. TensorFlow execution spanning different frameworks.

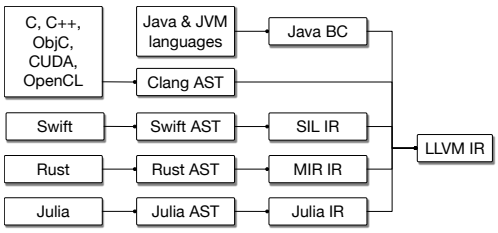

Fig. 2. Compilation pipeline with mid-level language IRs.

successful at unifying and integrating work across a range of different languages, but high-level languages often end up building their own high-level IR and reinventing the same kind of technology for higher levels of abstraction (see Figure 2). At the same time, the LLVM community struggled with the representation of parallel constructs, and how to share front-end lowering infrastructure (e.g. for C calling conventions, or cross-language features like OpenMP), with no satisfactory solutions.

Faced with this challenge, given we could not afford to implement N improved compiler instances, we decided to go for a more general solution: investing in a high quality infrastructure which would benefit multiple domains, progressively upgrading existing systems, making it easier to tackle pressing problems like heterogeneous compilation for specialized accelerators, and provide new research opportunities. Now that we gathered a significant amount of experience building and deploying MLIR-based systems, we are able to look back on its rationale and design and discuss why this direction was pursued.

II. Design Principles

Let us now explore the requirements that guided the design of MLIR and their relation with the overarching principles.

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, which are the most common in IRs—should be used to express everything else, allowing fewer and more consistent abstractions that are easy to comprehend, extend and adopt. Broadly, customizability ensures the system can adapt to changing requirements and is more likely to be applicable to future problems. In that sense, we ought to build an IR as a rich infrastructure with reusable components and programming abstractions supporting the syntax and semantics of its intermediate language.

A success criterion for customization is the possibility to express a diverse set of abstractions including machine learning graphs, ASTs, mathematical abstractions such as polyhedral,

Control Flow Graphs (CFGs) and instruction-level IRs such as LLVM IR, all without hard-coding concepts from these abstractions into the system.

Certainly, customizability creates a risk of internal fragmentation due to poorly compatible abstractions. While there is unlikely a purely technical solution, the system should encourage one to design reusable abstractions and assume they will be used outside of their initial scope.

SSA and Regions [Parsimony]: The Static Single Assignment (SSA) form [3] is a widely used representation in compiler IRs. It provides numerous advantages including making dataflow analysis simple and sparse, is widely understood by the compiler community for its relation with continuation-passing style, and is established in major frameworks. As a result, the IR enforces the value-based semantics of SSA, its referential transparency and algorithmic efficiency, all considered essential to a modern compiler infrastructure. However, while many existing IRs use a flat, linearized CFG, representing higher level abstractions push introducing *nested regions* as a first-class concept in the IR. This goes beyond the traditional region formation to lift higher level abstractions (e.g., loop trees), speeding up the compilation process or extracting instruction, or SIMD parallelism [4], [5], [6]. To support heterogeneous compilation, the system has to support the expression of structured control flow, concurrency constructs, closures in source languages, and many other purposes. One specific challenge is to make CFG-based analyses and transformations compose over nested regions.

In doing so, we agree to sacrifice the normalization, and sometimes the canonicalization properties of LLVM. Being able to lower a variety of data and control structures into a smaller collection of normalized representations is key to keeping compiler complexity under control. The canonical loop structure with its pre-header, header, latch, body, is a prototypical case of a linearized control flow representation of a variety of loop constructs in front-end languages. We aim at offering users a choice: depending on the compilation algorithm of interest, of the pass in the compilation flow, nested loops may be captured as nested regions, or as linearized control flow. By offering such a choice, we depart from the normalization-only orientation of LLVM while retaining the ability to deal with higher level abstractions when it matters. In turn, leveraging such choices raises questions about how to control the normalization of abstractions, which is the purpose of the next paragraph.

Maintain Higher-Level Semantics [Progressivity]: The system needs to retain the information and structure that are required for analysis or optimizing performance. Attempts to recover abstract semantics once lowered are fragile and shoehorning this information at low-level often invasive (e.g., all passes need to be revisited in the case of using debug information to record structure). Instead, the system should maintain the structure of computations and progressively lower to the hardware abstraction. The loss of structure is then conscious and happens only where the structure is no longer needed to match the underlying execution model. For example, the system should preserve the structured control flow such as loop structure throughout the relevant transformations; removing this structure, i.e. lowering to a CFG essentially means no further transformations will be performed that exploits the structure. The state of the art in modeling parallel computing constructs in a production compiler highlights how difficult the task may be in general [7], [8].

As a corollary, mixing different levels of abstraction and different concepts in the same IR is a key to allowing a part of the representation to remain in higher-level abstraction while another part is lowered. This would enable, for instance, a compiler for a custom accelerator to reuse some higher-level structure and abstractions defined by the system alongside with primitive scalar/vector instructions specific to the accelerator.

Another corollary is that the system should support *progressive lowering*,from the higher-level representation down to the lowest-level, performed in small steps along multiple abstractions. The need for multiple levels of abstractions stems from the variety of platforms and programming models a compiler infrastructure has to support.

Previous compilers have been introducing multiple fixed levels of abstraction in their pipeline—e.g. the Open64 WHIRL representation [9] has five levels, as does the Clang compiler which lowers from ASTs to LLVM IR, to SelectionDAG, to MachineInstr, and to MCInst. More flexible designs are required to support extensibility. This has deep implications on the phase ordering of transformations. As compiler experts started implementing more and more transformation passes, complex interactions between these passes started appearing. It was shown early on that combining optimization passes allows the compiler to discover more facts about the program. One of the first illustrations of the benefits of combining passes was to mix constant propagation, value numbering and unreachable code elimination [10].

Declaration and Validation [Parsimony and Traceability]: Defining representation modifiers should be as simple as introducing new abstractions; a compiler infrastructure is only as good as the transformations it supports. Common transformations should be implementable as rewrite rules expressed declaratively, in a machine-analyzable format to reason about properties of the rewrites such as complexity and completion. Rewriting systems have been studied extensively for their soundness and efficiency, and applied to numerous compilation problems, from type systems to instruction selection. Since we aim for unprecedented extensibility and incremental lowering capabilities, this opens numerous avenues for modeling program transformations as rewrite systems. It also raises interesting questions about how to represent the rewrite rules and strategies, and how to build machine descriptions capable of steering rewriting strategies through multiple levels of abstraction. The system needs to address these questions while preserving extensibility and enforcing monotonic and reproducible behavior.

The openness of the ecosystem also calls for an extensive validation mechanism. While verification and testing are useful to detect compiler bugs, and to capture IR invariants, the need

for robust validation methodologies and tools is amplified in an extensible system. The mechanism should aim to make this easy to define and as declarative as practical, providing a single source of truth. A long term goal would be to reproduce the successes of translation validation [11], [12], [13], [14] and modern approaches to compiler testing [15]. Both are currently open problems in the context of an extensible compiler.

Source Location Tracking [Traceability]: The provenance of an operation—including its original location and applied transformations—should be easily traceable within the system. This intends to address the lack-of-transparency problem, common to complex compilation systems, where it is virtually impossible to understand how the final representation was constructed from the original one.

This is particularly problematic when compiling safety-critical and sensitive applications, where tracing lowering and optimization steps is an essential component of software certification procedures [16]. When operating on secure code such as cryptographic protocols or algorithms operating on privacy-sensitive data, the compiler often faces seemingly redundant or cumbersome computations that embed a security or privacy property not fully captured by the functional semantics of the source program: this code may prevent the exposure of side channels or harden the code against cyber or fault attacks. Optimizations may alter or completely invalidate such protections [17]; this lack of transparency is known as WYSINWYX [18] in secure compilation. One indirect goal of accurately propagating high-level information to the lower levels is to help support secure and traceable compilation.

III. IR Design

Our main contribution is to present an IR that follows the principles defined in the previous section. This is what MLIR does and we review its main design points in this section.

MLIR has a *generic* textual representation (example in Figure 3) 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; for example, Figure 7 illustrates the user-defined syntax for Figure 3.

Operations: The unit of semantics in MLIR is an "operation", referred to as *Op*. Everything from "instruction" to "function" to "module" are modeled as Ops in this system. MLIR does not have a fixed set of Ops, but allows (and encourages) user-defined extensions, according to the parsimony and "everything customizable" principles. The infrastructure provides a declarative syntax for defining Ops based on TableGen [19], as illustrated in Figure 5.[2]

Ops (see Figure 4) have a unique opcode, a string identifying the operation and its dialect. Ops take and produce zero or more *values*, called *operands* and *results* respectively, and these are maintained in SSA form. Values represent data at runtime,

[2]Alternatives have been proposed, aiming for higher productivity, soundness guarantees, and better interoperability with high-level languages; this still a subject of active design discussions.

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

Fig. 3. MLIR *generic* representation for polynomial multiplication using affine and std dialects. The same IR is displayed with the custom syntax Figure 7.

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

Fig. 4. Operation (Op) is a main entity in MLIR; operations contain a list of regions, regions contain a list of blocks, blocks contains a list of Ops, enabling recursive structures

and have a *Type* that encodes the compile-time knowledge about the data. In addition to an opcode, operands and results, Ops may also have *Attributes*, *Regions*, *Successor Blocks*, and *Location Information*. Figure 3 illustrates values and Ops, `%`-identifiers are (packs of) named values, with "`:`" specifying the number in a pack if more than one and "`#`' a particular value. In the generic textual representation, operation names are quoted string literals followed by operands in parentheses.

Compiler passes treat unknown Ops conservatively, and MLIR has rich support for describing the semantics of Ops to passes through traits and interfaces as described in Section V-A. Op implementation has *verifiers* that enforce the Op invariants and participate in overall IR validation.

Attributes: MLIR attributes contain compile-time information about operations, other than the opcode. Attributes are typed (e.g., integer, string), and each Op instance has an open key-value dictionary from string names to attribute values. In the generic syntax, attributes are found in a brace-enclosed comma-separated list of pairs. Figure 3 uses attributes to define

```
// 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);
}
```

Fig. 5. Operation Definition Syntax (ODS) provides a concise way of defining new Ops in MLIR. Here, one defines the `LeakyRelu` Op taking a tensor and a floating-point value, and returning a tensor of the same type as the input one.

bounds of a loop that are known to be constant affine forms: `{lower_bound = () -> (0), step = 1 : index, upper_bound = #map3}` where `lower_bound` is an example of an attribute name. The `() -> (0)` notation is used for inline affine forms, in this case producing an affine function producing a constant `0` value. The `#map3` notation is used for attribute *aliases*, which allow associate attribute values with a label upfront.

Attributes derive their meaning either from the Op semantics or from the dialect (Section III) they are associated with. As with opcodes, there is no fixed set of attributes. Attributes may reference foreign data structures, which is useful for integrating with existing systems, e.g., the contents of (known at compile time) data storage in an ML system.

Location Information: 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. It standardizes the way to emit diagnostics from the compiler, and is used by a wide range of testing tools.

Location information is also extensible, allowing a compiler to refer to existing location tracking systems, high-level AST nodes, LLVM-style file-line-column address, DWARF debugging info, etc.

Regions and Blocks: An instance of an Op may have a list of attached regions. A *region* provides the nesting mechanism in MLIR: it contains a list of blocks, each of which contains a list of operations (that may contain further regions). As with attributes, the semantics of a region are defined by the operation they are attached to, however the blocks inside the region (if more than one) form a Control Flow Graph (CFG). For example, the `affine.for` operation in Figure 3 is a loop with the single-block body attached as a region, located between `({` and `})` delimiters. The Op specifies the flow of control across regions. In this example, the body is executed repeatedly until the upper bound is reached.

The body of each region is a list of *blocks*, and each block ends with a *terminator* operation, that may have *successor* blocks to which the control flow may be transferred. Each terminator (e.g. "switch", "conditional branch" or "unwind") defines its own semantics. It may chose to transfer the control flow to another block in the same region, or return it to the Op enclosing the region. The graph of successors defines a CFG, allowing standard SSA-based control flow within a region.

Instead of using ϕ nodes, MLIR uses a functional form of SSA [20] where terminators pass values into *block arguments* defined by the successor block. Each block has a (potentially empty) list of typed block arguments, which are regular values and obey SSA. The semantics of terminator Ops defines what values the arguments of the block will take after the control is transferred. For the first (entry) block of the region, the values are defined by the semantics of the enclosing Op. For example, `affine.for` uses the entry block argument `%arg4` as loop induction variable. Finally, this explicit graph design and the extensibility of Ops is reminiscent of the sea-of-nodes representation [21]: this connection is intentional and has been a major influence for the selection of MLIR's flavor of SSA.

Value Dominance and Visibility: Ops can only use values that are in scope, i.e. *visible* according to SSA dominance, nesting, and semantic restrictions imposed by enclosing operations. Values are visible within a CFG if they obey standard SSA dominance relationships, where control is guaranteed to pass through a definition before reaching a use.

Region-based visibility is defined based on simple nesting of regions: if the operand to an Op is outside the current region, then it must be defined lexically above and outside the region of the use. This is what allows Ops within an `affine.for` operation to use values defined in outer scopes.

MLIR also allows operations to be defined as *isolated from above*, indicating that the operation is a scope barrier—e.g. the "std.func" Op defines a function, and it is not valid for operations within the function to refer to values defined outside the function. In addition to providing useful semantic checking, a module containing isolated-from-above Ops may be processed in parallel by an MLIR compiler since no use-def chains may cross the isolation barriers. This is important for compilation to utilize multicore machines.

All these design choices highlight the progressivity principle, while erring on the side of parsimony when a concept does not appear to be generic and essential enough to be builtin.

Symbols and Symbol Tables: Ops can have a symbol table attached. This table is a standardized way of associating names, represented as strings, to IR objects, called *symbols*. The IR does not prescribe what symbols are used for, leaving it up to the Op definition. Symbols are most useful for named entities need that not obey SSA: they cannot be redefined within the same table, but they can be used prior to their definition. For example, global variables, functions or named modules can be represented as symbols. Without this mechanism, it would have been impossible to define, e.g., recursive function

referring to themselves in their definition. Symbol tables can be nested if an Op with a symbol table attached has associated regions containing similar Ops. MLIR provides a mechanism to reference symbols from an Op, including nested symbols.

Dialects: 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 behavior for all ops in the dialect). They organize the ecosystem of language- and domain-specific semantics while following the parsimony principle. The dialect namespace appears as a dot-separated prefix in the opcode, e.g., Figure 3 uses `affine` and `std` dialects.

The separation of Ops, types and attributes into dialects is conceptual and is akin to designing a set of modular libraries. For example, a dialect can contain Ops and types for operating on hardware vectors (e.g., shuffle, insert/extract element, mask), and another dialect can contain Ops and types for operating on algebraic vectors (e.g. absolute value, dot product, etc.). Whether both dialects use the same vector type and where does this type belong are design decisions left to MLIR user.

While it is possible to put all Ops, types and attributes in a single dialect, it would quickly become unmanageable due to the large number of simultaneously present concepts and name conflicts, amongst other issues. Although each Op, type and attribute belongs to exactly one dialect, MLIR explicitly supports a mix of dialects to enable progressive lowering. 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.

Type System: 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 about a value. The type system in MLIR is user-extensible, and may, for example, refer to existing foreign type systems. MLIR enforces strict type equality checking and does not provide type conversion rules. Ops list their inputs and result types using trailing function-like syntax. In Figure 3, `std.load` maps from the memory reference and index types to the type of the value it loads.

From the type theory point of view, MLIR only supports non-dependent types, including trivial, parametric, function, sum and product types. While it is possible to implement a dependent type system by combining Ops with symbols and user-defined types, such types will be opaque to the IR.

For convenience, MLIR provides a standardized set of commonly used types, including arbitrary precision integers, standard floating point types, and simple common containers—tuples, multi-dimensional vectors, and tensors. These types a merely a utility and their use is not required, illustrating parsimony.

Functions and Modules: 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 dialect, again an illustration of parsimony in the design.

A module is an Op with a single region containing a single block, and terminated by a dummy Op that does not transfer the control flow. Like any block, its body contains a list of Ops, which may be functions, global variables, compiler metadata, or other top-level constructs. Modules may define a symbol in order to be referenced.

Similarly, a function is an Op with a single region that may contain zero (in case of declaration) or more blocks. Built-in functions are compatible with "call" and "return" operations of the "std" dialect, which transfer the control to and from the function, respectively. Other dialects are free to define their own function-like Ops.

IV. Evaluation: Applications of MLIR

MLIR is a system that aims to generalize and drive a wide range of compiler projects, so our primary evaluation metric is to show that it is being adopted and used for diverse projects. By doing so we acknowledge the software engineering nature of the problem and contributions. We provide a summary of community activity and describe a few use cases in more detail to highlight the generality and extensibility of MLIR and evaluate how well compiler and domain experts experience the design principles of the IR.

Today, MLIR is a growing open source project with a community spanning academia and industry.[3] For example, the first academic workshop about the use of MLIR in High-Performance Computing was attended by individuals from 16 universities and involved 4 national laboratories from 4 different countries.[4] MLIR was also endorsed by 14 multinational companies and at the 2019 LLVM Developer Meeting more than 100 industry developers attended a roundtable event about MLIR. Community adoption and participation is a proxy measure for usability and need. More than 26 dialects are in development in public or private and 7 projects across different companies are replacing custom infrastructure with MLIR. We argue that this shows a real need for MLIR, as well as endorses its usability.

A. TensorFlow Graphs

While the other discussed representations are familiar to most compiler developments, one of key use cases for MLIR is to support the development of machine learning frameworks. Their internal representations is often based on a data flow graph [22] with a dynamic execution semantics.

TensorFlow [23] is an example of such framework. Its representation is a high-level dataflow computation where the nodes are computations which can be placed on various devices, including specific hardware accelerators.

MLIR is used in TensorFlow to model this internal representation and perform transformations for the use cases presented in Figure 1: from simple algebraic optimizations to retargeting

[3]https://www.c4ml.org.

[4]http://www.cs.utah.edu/~mhall/mlir4hpc.

```
%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
}
```

Fig. 6. SSA representation of a TensorFlow graph in MLIR.

graphs for parallel and distributed execution on data center clusters and asynchronous hardware acceleration, from lowering to a representation suitable for mobile deployment to generating efficient native code using domain-specific code generators like XLA [24]. The representation of a TensorFlow graph in MLIR is illustrated on Figure 6. It illustrates the modeling of asynchronous concurrency, where the dataflow graph is desynchronized via implicit futures and side-effecting Ops are serialized through explicit control signals (also following dataflow semantics). Despite the widely different abstractions, concurrency, asynchrony, delayed evaluation, MLIR offers the same infrastructure, analysis and transformation capabilities as for any other dialect or compiler pass. In particular, essential graph-level transformations implemented in Grappler [5] are expressible in MLIR for both TensorFlow models and low level LLVM IR: dead code/node elimination, constant folding, canonicalization, loop-invariant code motion, common subexpression/subgraph elimination, instruction/device-specific-kernel selection, rematerialization, layout optimization; while other transformations may be domain-specific: optimizations for mixed precision, op fusion, shape arithmetic.

B. Polyhedral Code Generation

One of the original motivations for MLIR was the exploration of polyhedral code generation for accelerators. The affine dialect is a simplified polyhedral representation that was designed to enable progressive lowering. While a full exploration of the design points here is out of scope for this paper, we illustrate aspects of the affine dialect to show the modeling power of MLIR and contrast the affine dialect with past representations [25], [26], [27], [28], [29].

1) Similarities: The MLIR affine dialect operates on a structured multi-dimensional type for all accesses to memory. In the default case, these structured types are *injective*: different indexings are guaranteed not to alias by construction, a common precondition for polyhedral dependence analyses.

Affine modeling is split in two parts. Attributes are used to model affine maps and integer sets at compile-time and Ops are used to apply affine restrictions to the code. Namely, `affine.for` Op is a "for" loop with bounds expressed as affine maps of values required to be invariant in a function. Thus loops have static control flow. Similarly, `affine.if` is a conditional restricted by affine integer sets. The bodies of loops and conditionals are regions that use `affine.load` and `affine.store` to restrict indexing to affine forms of surrounding loop iterators. This enables exact affine dependence analysis while avoiding the need to infer affine forms from a lossy lower-level representation.

[5]https://www.tensorflow.org/guide/graph_optimization

```
// 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>
  }
}
```

Fig. 7. Affine dialect representation of polynomial multiplication `C(i+j) += A(i) * B(j)`.

2) Differences with existing polyhedral: They are numerous:

(1) *Rich types*: the MLIR structured memory reference type contains a layout map connecting the index space of the buffer to the actual address space. This separation of concerns makes loop and data transformations compose better: changes to data layout do not affect the code and do not pollute dependence analysis. Such mixes of transformations have been explored previously [30] but are uncommon.

(2) *Mix of abstractions*: Bodies of affine loops in MLIR can be expressed with operations on typed SSA values. Therefore, all traditional compiler analyses and transformations remain applicable and can be interleaved with polyhedral transformations. On the contrary, polyhedral compilers often abstract such details away completely, making it challenging for a polyhedral compiler to manipulate, e.g., vector types.

(3) *Smaller representation gap*: One of the key features of the polyhedral model is its ability to represent the order of loop iterations in the *type system*. In this system, a large number of loop transformations compose directly and can be reasoned about using simple mathematical abstractions [26]. However, polyhedral transformations require *raising* into a representation often drastically different from the original [31], [32]. Furthermore, the conversion from transformed polyhedra to loops is computationally hard [33]. MLIR-based representation maintains high-level loop structure around lower-level representation, removing the need for raising.

(4) Compilation speed is a crucial goal for MLIR as discussed in Section V-D, but has not been a focus of most existing polyhedral approaches. These rely heavily on algorithms with exponential complexity: on integer linear programming to derive loop orderings automatically and on polyhedron scanning algorithms to convert the representation back to loops. The MLIR approach explicitly does not rely on polyhedron scanning since loops are preserved in the IR. In addition, code generation may take place ahead-of-time, e.g., when producing generic

code for dynamic shapes, or just-in-time when specializing tensor operations on static shapes. The latter puts stricter constraints on available resources, and both scenarios are important.

Experience with the affine dialect shows that first-class affine abstractions facilitate the design and implementation of domain-specific code generators, including the linalg dialect,[6] and declarative rewrite rules in RISE.[7] These developments and the affine dialect itself represent important explorations that the MLIR design made practical.

C. Fortran IR (FIR)

The LLVM Fortran frontend "flang" is currently under major development, led by NVIDIA/PGI. Similar to Swift, Rust, and others, flang needs a specialized IR in order to support advanced transformations for high-performance Fortran codebase, and is using MLIR to support these Fortran-specific optimizations [34]. These high-level optimizations—advanced loop optimizations, array copy elimination, call specialization, devirtualization—would be hard implement using only LLVM.

For example, FIR is able to model Fortran virtual dispatch table as a first class concept (see Figure 8).

```
// Dispatch table for type(u)
fir.dispatch_table @dtable_type_u {
  fir.dt_entry "method", @u_method
}

func @some_func() {
  %uv = fir.alloca !fir.type<u> : !fir.ref<!fir.type<u>>
  fir.dispatch "method"(%uv) : (!fir.ref<!fir.type<u>>) -> ()
  // ...
}
```

Fig. 8. FIR has first class support for dynamic virtual function dispatch tables.

The ability to model the high-level semantics of the programming language in a structured IR is very powerful. For example, first-class modeling of the dispatch tables allows a robust devirtualization pass to be implemented. While this could have been implemented with a bespoke compiler IR, the use of MLIR allowed the flang developers to spend their engineering resources focused on the IR design for their domain instead of reimplementing basic infrastructure.

The choice of MLIR also unlocks the reusability of other dialects that are not specific to Fortran: a language-independent OpenMP dialect could be shared between Fortran and C language frontends. Similarly, targeting a heterogeneous platform using OpenACC becomes tractable within MLIR through the sharing and reuse of the GPU-oriented dialects and passes. This is straightforward thanks to MLIR begin specifically designed to support a mix of composable dialects.

D. Domain-Specific Compilers

The applications above are within large workflows. But MLIR also helps building smaller domain specific compilers. A reusable and modular infrastructure makes these specialized paths feasible and relatively cheap to build.

Optimizing MLIR Pattern Rewriting: MLIR has an extensible system for pattern rewrites. In addition to statically declared patterns, we had applications where the rewrite patterns needed to be dynamically extensible at runtime, allowing hardware vendors to add new lowerings in drivers. The solution was to express MLIR pattern rewrites as an MLIR dialect itself, allowing us to use MLIR infrastructure to build and optimize efficient Finite State Machine (FSM) matcher and rewriters on the fly. This work includes FSM optimizations seen in other systems, such as the LLVM SelectionDAG and GlobalISel instruction selection systems.

Lattice Regression Compiler: Lattice regression [35] is a machine learning technique renowned for fast evaluation times and interpretability. The predecessor of the compiler was implemented using C++ templates. This allowed for high-performance code with metaprogramming, but expressing general optimizations on the end-to-end models was not straightforward. This particular lattice regression system is used in applications with multiple millions of users and hence performance improvements are critical.

MLIR was used as the basis for a new compiler for this specialized area, which was driven by a specialized search approach—effectively resulting in a machine learning problem being solved during compilation. The resultant compiler was developed by investing a 3 person-month effort, and resulted in up to 8× performance improvement on a production model, while also improving transparency during compilation.

V. Consequences of the MLIR Design

The MLIR design facilitates the modeling of new language and compiler abstractions while reusing existing, generic ones. Effectively, the solution to many problems is to "add new ops, new types", possibly collected into "a new dialect". This is a significant design shift for compiler engineering. It produces new opportunities, challenges, and insights. This section explores a few of them.

A. Reusable Compiler Passes

The ability to represent multiple levels of abstraction in one IR incentivizes the passes that operate across these levels. MLIR handles extensibility by inverting the common approach: since there are more Ops than passes, it is easier for Ops to know about passes. This also improves modularity as the dialect-specific logic is implemented within the dialects instead of the core transformations. Since the passes rarely need to know all aspects of an Op, MLIR relies on the following mechanisms to implement generic passes.

Operation Traits: Many common "bread and butter" compiler passes, such as Dead Code or Common Subexpression Elimination, rely on simple properties like "is terminator" or "is commutative". We define such properties as *Op Traits*. An Op exhibits a trait unconditionally, e.g., a "standard branch" Op is always a terminator. For many passes, it is sufficient to know that an Op has a set of traits to operate on it, for

[6]https://mlir.llvm.org/docs/Dialects/Linalg.
[7]https://rise-lang.org/mlir.

example by swapping the operands or removing Ops with no side effects and no users.

Traits can serve as verification hooks allowing to share the logic across multiple Ops that have the trait. For example, the "isolated from above" trait verifies that no regions in the Op use values defined in the regions enclosing the Op. It allows for generic processing of functions, modules and other self-contained structures.

Interfaces: When the unconditional, static behavior is insufficiently expressive, the processing can be parameterized through *interfaces*, a concept borrowed from object-oriented programming. An interface defines a view into the behavior of an IR object that abstracts away unnecessary details. Unlike traits, interfaces are *implemented* by IR objects, using arbitrary C++ code that can produce different results for different objects. For example, the "call" Op implements a "call-like" interface, but different instances of the Op call different functions.

MLIR passes can be implemented in terms of interfaces, establishing a contract with any Op that opts into being processed by a pass. Continuing the call-like example, consider the MLIR inlining pass that works on TensorFlow graphs, Flang functions, closures in a functional language etc. Such a pass needs to know: (1) whether it is valid to inline an operation into a given region, and (2) how to handle terminator operations that ended up in the middle of a block after inlining.

In order to query an Op about these properties, the pass defines a dedicated interface so that Ops may register their implementation with MLIR to benefit from inlining. The inlining pass will treat conservatively, i.e. ignore, any operation that does not implement the respective interface.

Constant folding is implemented through the same mechanism: each Op implements the "fold" interface by providing a function that may produce an attribute holding the value if the Op is foldable. More generic canonicalization can be implemented similarly: an interface populates the list of canonicalization patterns amenable to pattern-rewriting.This design separates generic logic from Op-specific logic and puts the latter in the Op itself, reducing the well-known maintenance and complexity burden of "InstCombine", "PeepholeOptimizer" and the likes in LLVM.

Interfaces can be implemented by dialects rather than specific Ops, which enables shared behavior or delegation to the external logic, for example when constant folding TensorFlow Ops. Interfaces are also supported on types and attributes, for example an addition operation may support any type that self-declares as "integer-like" with queryable signedness semantics.

B. Dialect-Specific Passes

Finally, it is valid and useful to define passes that are specific to particular dialects, which can be driven by full semantics of operations in the dialect(s) they are designed for. These passes are just as useful in the MLIR system as they are in other compiler systems. For example, code generators that want to do custom scheduling of machine instructions based on particular machine constraints or other tricks that do not fit into a broader framework. This is a simple and useful starting point for new transformations, where generalization isn't required.

C. Mixing Dialects Together

One of the most profound (but also most difficult to grok) aspects of MLIR is that it allows and encourages mixing operations from different dialects together into a single program. While certain cases of this are reasonably easy to understand (e.g. holding host and accelerator computation in the same module) the most interesting cases occur when dialects are directly mixed— because this enables an entire class of reuse that we have not seen in other systems.

Consider the affine dialect described in Section IV-B. The definition of affine control flow and affine mappings are independent of the semantics of the operations that are contained in affine regions. In our case, we combine the affine dialect with the "standard" dialect that represents simple arithmetic in a target independent form like LLVM IR, with multiple target-specific machine instruction dialects for internal accelerators. Others have combined it with abstractions from other problem domains.

The ability to reuse generic polyhedral transformations (using Op interfaces to get semantics of operations in specific transformations) is a powerful (and exciting to us) way of factoring compiler infrastructure. Another example is that an OpenMP dialect could be used and reused across a wide variety of source-language IRs.

D. Parallel Compilation

An important aspect of MLIR is the possibility to use multi-core machines to increase the compilation speed. In particular, the "isolated from above" trait (Section V-A) allows Ops such as functions to opt into the concurrent IR traversal mechanism supported by MLIR's pass manager. Indeed this trait guarantees that SSA use-def chain cannot cross the region boundaries and can be processed in isolation. MLIR also does not feature whole-module use-def chains, but instead references global objects through symbol tables (Section III) and defines constants as operations with attributes (Section III).

E. Interoperability

Our work involves interoperation with a large number of existing systems, e.g., machine learning graphs encoded as protocol buffers, compiler IRs including LLVM IR, proprietary instruction sets, etc. Often the representation has a number of suboptimal or unfortunate decisions that made sense in the context of an existing system, but capabilities of MLIR enable a more expressive representation. Because importers and exporters are notoriously difficult to test (test cases are often binary), we want to make sure their complexity is minimized.

The solution is to define a dialect that corresponds to the foreign system as directly as possible—allowing round tripping to-and-from that format in a simple and predictable way. Once the IR is imported into MLIR, it can be raised and lowered to a more convenient IR using all of the MLIR infrastructure, which allows those transformations to be tested similarly to all the other MLIR passes.

There are numerous examples of such dialects, including the LLVM dialect which maps LLVM IR into MLIR. This approach has worked well for us, and the MLIR tooling has also been useful to write tests for these foreign file formats.

F. Unopinionated Design Provides New Challenges

While MLIR allows one to define almost arbitrary abstractions, it provides very little guidance on what *should* be done: what works better or worse in practice? We now have some experience with a number of engineers and researchers applying the techniques and technologies to new problem domains, and have realized that the "art" of compiler IR design and abstraction design is not well understood in the compiler and languages field—many people work within the constraints of established systems, but relatively few have had the opportunity define the abstractions themselves.

This is a challenge, but is also another set of opportunities for future research. The broader MLIR community is building expertise with these abstraction design trade-offs, and we expect this to be a fertile area of study over time.

G. Looking Forward

The design of MLIR is different enough from other compiler infrastructures that we are still learning—even after building and applying it to many different systems. We believe that there is still a lot to discover, and several years of research will be required to better understand the design points and establish best practices. For example, the rise of out-of-tree dialects, increasing number of source language frontends using MLIR, possible application to Abstract Syntax Trees, and applications to structured data (like JSON, protocol buffers, etc) which are still very early and are likely to uncover interesting new challenges and opportunities. Better support for just-in-time compilation and precise garbage-collection would also be interesting, leveraging the modularity and programmability of the IR.

VI. Related Work

MLIR is a project that overlaps with multiple different domains. While the composed infrastructure provides a novel system, individual components have analogs in the literature. For references and discussion directly related to the IR design itself, please refer to Section II.

MLIR is a compiler infrastructure akin to LLVM [1], but where LLVM has been a great boon to scalar optimizations and homogeneous compilation, MLIR aims to model a rich set of data structures and algorithms as first-class values and operations, including tensor algebra and algorithms, graph representations, as well as heterogeneous compilation. MLIR allows mix-and-match optimization decomposing compilation passes into components and redefining lowering, cleanup roles. This is largely attributed to the pattern rewriting infrastructure, capturing full-fledged transformations as a composition of small local patterns and controlling which pattern rewrites are applied at the granularity of an individual operation. Extending, formalizing, and verifying the rewriting logic automatically would be an important next step [36], [37]. On the backend side, MLIR's DDR has an analogue to LLVM's instruction selection infrastructure, supporting extensible operations with multi-result patterns and specification as constraints [38].

Numerous programming languages and models tackle hardware heterogeneity. Originally a homogeneous programming model, OpenMP added support for offloading tasks and parallel regions to accelerators [39], based on earlier proposals such as StarSs and OpenACC [40], [41]. C++ AMP, HCC and SyCL leverage a conventional Clang/LLVM flow and modern C++ to provide a high-level abstraction for hardware acceleration [42]. Unfortunately, all these examples very quickly lower high-level constructs to calls to a runtime execution environment, relying on pre-existing optimizations in the host language (typically C++) to alleviate the abstraction penalty. Far fewer efforts target the heterogeneous compilation process itself. Parallel intermediate representations extending LLVM IR address part of the issue but traditionally focus on the homogeneous setting [7], [8]. The most ambitious effort to date may be Liquid Metal [43], with a co-designed Domain Specific Language (DSL) and compilation flow converting managed object semantics into static, vector or reconfigurable hardware; yet most of the effort in its Lime compiler reside in fitting round objects into square hardware (paraphrasing Kou and Palsberg [44]). MLIR provides a direct embedding for high level languages embracing heterogeneity through extensible set of operations and types, while providing a common infrastructure for gradually lowering these constructs with maximal reuse of common components across the different targets.

Tackling language heterogeneity has been a long-term promise of metaprogramming systems, and of multistage programming in particular. Lightweight Modular Staging (LMS) [45] is a state of the art framework and runtime code generator, providing a library of core components for generating efficient code and embedding DSLs in Scala. Delite [46] promises dramatic productivity improvements for DSL developers, while supporting parallel and heterogeneous execution. This approach is complementary to MLIR, providing a higher-level of abstraction to embed DSLs and implement optimizations through generic metaprogramming constructs.

One step further up into the language syntax, ANTLR [47] is among a class of parser generators that aim to facilitate the development of compiler frontends. MLIR currently does not have a general parser generator, no AST construction or modeling functionality. Combining MLIR with a system such as ANTLR could expand reusability upstream all the way to frontends and development environments.

More narrowly construed by their application to machine learning, XLA [24], Glow [48] and TVM [49], address similar heterogeneous compilation objectives. These frameworks provide domain-specific code generation instances, starting from a graph abstraction and targeting multi-dimensional vector abstractions for accelerators. All of these could leverage MLIR as infrastructure, taking advantage of the common functionality while using their current code generation strategies. Similarly, the loop nest metaprogramming techniques from Halide [50]

and TVM [49], earlier loop nest metaprogramming [26], [51], [52], [53], and automatic flows such as PolyMage [54], Tensor Comprehensions [29], Stripe [55], Diesel [56], Tiramisu [57] and their underlying polyhedral compilation techniques [25], [27], [58], [28] could co-exist as different code generation paths within an MLIR-based framework. This would greatly increase code reuse, defragmentation of the landscape, interoperability across domain, and portability. This is actually one of the motivations for the IREE project,[8] building on MLIR at multiple levels of abstraction, from tensor algebra and operator graphs down to the low-level orchestration of asynchronous coroutines and code generation for multiple CPU and GPU architectures (within the Vulkan/SPIR-V standard).

Finally, interoperability formats, such as ONNX [59], have a different approach towards addressing the diversity of ML frontends by providing a common set of ops that different frameworks could map on to. ONNX would be a candidate as a dialect in MLIR to and from which ops could be converted.

VII. Conclusion and Future Work

We presented MLIR, a concrete answer to the dual scientific and engineering challenge of designing a flexible and extensible infrastructure for compiler construction, ranging from backend code generation and orchestration of heterogeneous systems, to graph-level modeling for machine learning, and to the high-level language semantics of programming languages and domain-specific frameworks. We demonstrated its applicability to a range of domains and discussing research implications.

Motivated by the success of LLVM and looking ahead, we are eager to see how established communities in programming languages and high-performance computing, as well domain experts can benefit from the introduction of higher level, language-specific IRs. We also believe MLIR catalyzes new areas of research, as well as new approaches to teaching the art of compiler and IR design.

Acknowledgments

This paper and project would not have been possible without the contributions of numerous other individuals. We express our gratitude to all. We also acknowledge the Google Visiting Researcher Program for supporting the third author at the early times of MLIR.

Appendix

A. Abstract

The artifact for this paper includes the MLIR system, instructions on how to download and build it and link to MLIR-related source code in TensorFlow.

B. Artifact Check-List (Meta-Information)

- Program: MLIR
- Compilation: LLVM C++ toolchain
- Run-time environment: Recommended Linux
- Publicly available?: Yes
- Archived: DOI 10.5281/zenodo.4283090

[8]https://google.github.io/iree.

C. Description

1) How Delivered: To download MLIR please run

```
git clone \
  https://github.com/llvm/llvm-project.git
```

Instructions for downloading and building MLIR are also available at https://mlir.llvm.org/getting_started.

Additional information is available at mlir.llvm.org.

2) Software Dependencies: Downloading MLIR requires *git*. Building MLIR requires *Ninja* (https://ninja-build.org/) and a working C++ toolchain including *clang* and *lld*.

D. Installation

To build and test MLIR on Linux execute the following commands:

```
mkdir llvm-project/build
cd llvm-project/build
cmake -G Ninja ../llvm \
  -DLLVM_ENABLE_PROJECTS=mlir \
  -DLLVM_BUILD_EXAMPLES=ON \
  -DLLVM_TARGETS_TO_BUILD="X86;NVPTX;AMDGPU" \
  -DCMAKE_BUILD_TYPE=Release \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DCMAKE_C_COMPILER=clang \
  -DCMAKE_CXX_COMPILER=clang++ \
  -DLLVM_ENABLE_LLD=ON
cmake --build . --target check-mlir
```

E. Applications

MLIR use in TensorFlow can be observed in code located at https://github.com/tensorflow/tensorflow/tree/master/tensorflow/compiler/mlir/. Tests located in the *tensorflow/tests* subdirectory contain MLIR snippets illustrating TensorFlow graph representation and transformations. Instructions for building TensorFlow from source are available at https://www.tensorflow.org/install/source.

References

[1] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[2] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] 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. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[4] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 171–185. [Online]. Available: http://doi.acm.org/10.1145/178243.178258

[5] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treegion scheduling for wide issue processors," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, Las Vegas, Nevada, USA, January 31 - February 4, 1998*, 1998, pp. 266–276. [Online]. Available: https://doi.org/10.1109/HPCA.1998.650566

[6] G. Ramalingam, "On loops, dominators, and dominance frontiers," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 5, pp. 455–490, 2002. [Online]. Available: https://doi.org/10.1145/570886.570887

[7] D. Khaldi, P. Jouvelot, F. Irigoin, C. Ancourt, and B. Chapman, "LLVM parallel intermediate representation: Design and evaluation using OpenSHMEM communications," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2833157.2833158

[8] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into LLVM's intermediate representation," *SIGPLAN Not.*, vol. 52, no. 8, pp. 249–265, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3155284.3018758

[9] Open64 Developers, "Open64 compiler and tools," 2001.

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

[11] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, 1998, pp. 151–166. [Online]. Available: https://doi.org/10.1007/BFb0054170

[12] G. C. Necula, "Translation validation for an optimizing compiler," *SIGPLAN Not.*, vol. 35, no. 5, pp. 83–94, May 2000. [Online]. Available: http://doi.acm.org/10.1145/358438.349314

[13] J. Tristan and X. Leroy, "Formal verification of translation validators: a case study on instruction scheduling optimizations," in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, 2008, pp. 17–27. [Online]. Available: https://doi.org/10.1145/1328438.1328444

[14] ——, "Verified validation of lazy code motion," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, 2009, pp. 316–326. [Online]. Available: https://doi.org/10.1145/1542476.1542512

[15] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Z. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 197–208. [Online]. Available: https://doi.org/10.1145/2491956.2462173

[16] B. Schommer, C. Cullmann, G. Gebhard, X. Leroy, M. Schmidt, and S. Wegener, "Embedded Program Annotations for WCET Analysis," in *WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis*, vol. 63. Barcelona, Spain: Dagstuhl Publishing, Jul. 2018. [Online]. Available: https://hal.inria.fr/hal-01848686

[17] S. T. Vu, K. Heydemann, A. de Grandmaison, and A. Cohen, "Secure delivery of program properties through optimizing compilation," in *ACM SIGPLAN 2020 International Conference on Compiler Construction (CC 2020)*, San Diego, CA, Feb. 2020.

[18] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1749608.1749612

[19] "TableGen - LLVM 10 Documentation," Online, =https://llvm.org/docs/TableGen/, accessed Nov 22, 2019, 2019. [Online]. Available: https://llvm.org/docs/TableGen/

[20] A. W. Appel, "SSA is functional programming," *ACM SIGPLAN NOTICES*, vol. 33, no. 4, pp. 17–20, 1998.

[21] C. Click and M. Paleczny, "A simple graph-based intermediate representation," in *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, ser. IR '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 35–49. [Online]. Available: https://doi.org/10.1145/202529.202534

[22] A. Veen, "Dataflow machine architecture," *ACM Comput. Surv.*, vol. 18, pp. 365–396, 12 1986.

[23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[24] "XLA - TensorFlow, compiled," Google Developers Blog, https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html, Mar 2017. [Online]. Available: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html

[25] P. Feautrier, "Some efficient solutions to the affine scheduling problem. part II. multidimensional time," *Int. J. Parallel Program.*, vol. 21, no. 6, pp. 389–420, 1992.

[26] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Int. J. Parallel Program.*, vol. 34, no. 3, pp. 261–317, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10766-006-0012-3

[27] S. Verdoolaege, "ISL: An integer set library for the polyhedral model," in *Proceedings of the Third International Congress Conference on Mathematical Software*, ser. ICMS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 299–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=1888390.1888455

[28] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400713

[29] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated GPU kernels, automatically," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, pp. 38:1–38:26, Oct. 2019. [Online]. Available: http://doi.acm.org/10.1145/3355606

[30] C. Reddy and U. Bondhugula, "Effective automatic computation placement and data allocation for parallelization of regular programs," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2597652.2597673

[31] T. Grosser, A. Größlinger, and C. Lengauer, "Polly - performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 4, 2012. [Online]. Available: https://doi.org/10.1142/S0129626412500107

[32] L. Chelini, O. Zinenko, T. Grosser, and H. Corporaal, "Declarative loop tactics for domain-specific optimization," *TACO*, vol. 16, no. 4, pp. 55:1–55:25, 2020. [Online]. Available: https://doi.org/10.1145/3372266

[33] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 7–16. [Online]. Available: https://doi.org/10.1109/PACT.2004.11

[34] E. Schweitz, "An MLIR dialect for high-level optimization of fortran," LLVM Developer Meeting, Oct 2019.

[35] E. Garcia and M. Gupta, "Lattice regression," in *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, Eds. Curran Associates, Inc., 2009, pp. 594–602. [Online]. Available: http://papers.nips.cc/paper/3694-lattice-regression.pdf

[36] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/xt 0.17. A language and toolset for program transformation," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 52–70, 2008. [Online]. Available: https://doi.org/10.1016/j.scico.2007.11.003

[37] J. Meseguer, "Twenty years of rewriting logic," in *Proceedings of the 8th International Conference on Rewriting Logic and Its Applications*, ser. WRLA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 15–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1927806.1927809

[38] P. Thier, M. A. Ertl, and A. Krall, "Fast and flexible instruction selection with constraints," in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. New York, NY, USA: ACM, 2018, pp. 93–103. [Online]. Available: http://doi.acm.org/10.1145/3178372.3179501

[39] OpenMP ARB, "The OpenMP API specification for parallel programming," Online, https://www.openmp.org, accessed Feb 19, 2020.

[40] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *IJHPCA*, vol. 23, no. 3, pp. 284–299, 2009. [Online]. Available: https://doi.org/10.1177/1094342009106195

[41] "OpenACC application programming interface," Online, https://www.openacc.org, accessed Feb 19, 2020.

[42] "SyCL: C++ single-source heterogeneous programming for OpenCL," Online, https://www.khronos.org/sycl, accessed Feb 19, 2020.

[43] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla, "A compiler and runtime for heterogeneous computing," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 271–276. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228411

[44] S. Kou and J. Palsberg, "From oo to fpga: Fitting round objects into square hardware?" in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 109–124. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869470

[45] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," *Commun. ACM*, vol. 55, no. 6, pp. 121–130, 2012. [Online]. Available: https://doi.org/10.1145/2184319.2184345

[46] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4s, pp. 134:1–134:25, 2014. [Online]. Available: https://doi.org/10.1145/2584665

[47] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Softw. Pract. Exper.*, vol. 25, no. 7, pp. 789–810, Jul. 1995. [Online]. Available: http://dx.doi.org/10.1002/spe.4380250705

[48] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," 2018.

[49] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[50] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: Decoupling algorithms from schedules for high-performance image processing," *Commun. ACM*, vol. 61, no. 1, pp. 106–115, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3150211

[51] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame, "A programming language interface to describe transformations and code generation," in *Languages and Compilers for Parallel Computing*, K. Cooper, J. Mellor-Crummey, and V. Sarkar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 136–150.

[52] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul, "Opening polyhedral compiler's black box," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, 2016, pp. 128–138.

[53] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, and D. Padua, "In search of a program generator to implement generic transformations for high-performance computing," *Sci. Comput. Program.*, vol. 62, no. 1, pp. 25–46, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2005.10.013

[54] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic optimization for image processing pipelines," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 429–443.

[55] T. Zerrell and J. Bruestle, "Stripe: Tensor compilation via the nested polyhedral model," *CoRR*, vol. abs/1903.06498, 2019. [Online]. Available: http://arxiv.org/abs/1903.06498

[56] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: Dsl for linear algebra and neural net computations on gpus," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018. New York, NY, USA: ACM, 2018, pp. 42–51. [Online]. Available: http://doi.acm.org/10.1145/3211346.3211354

[57] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 193–205.

[58] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 2008, pp. 101–113. [Online]. Available: https://doi.org/10.1145/1375581.1375595

[59] The Linux Foundation, "ONNX: Open neural network exchange," Online, https://github.com/onnx/onnx, accessed Feb 19, 2020. [Online]. Available: https://github.com/onnx/onnx