Relay: A High-Level Compiler for Deep Learning

Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock

{jroesch, sslyu, jerry96, weberlo, joshpol, vegaluis, ziheng, tqchen, moreau, ztatlock}@cs.uw.edu
Paul G. Allen School of Computer Science & Engineering
University of Washington

Abstract

Frameworks for writing, compiling, and optimizing deep learning (DL) models have recently enabled progress in areas like computer vision and natural language processing. Extending these frameworks to accommodate the rapidly diversifying landscape of DL models and hardware platforms presents challenging tradeoffs between expressivity, composability, and portability. We present Relay, a new compiler framework for DL. Relay's functional, statically typed intermediate representation (IR) unifies and generalizes existing DL IRs to express state-of-the-art models. The introduction of Relay's expressive IR requires careful design of domain-specific optimizations, addressed via Relay's extension mechanisms. Using these extension mechanisms, Relay supports a unified compiler that can target a variety of hardware platforms. Our evaluation demonstrates Relay's competitive performance for a broad class of models and devices (CPUs, GPUs, and emerging accelerators). Relay's design demonstrates how a unified IR can provide expressivity, composability, and portability without compromising performance.

1. Introduction

Deep learning (DL) has radically transformed domains like computer vision and natural language processing (NLP) [36, 56]. Inspired by these successes, researchers and companies are continually experimenting with increasingly sophisticated DL models and developing specialized hardware backends. DL frameworks for writing, optimizing, and compiling DL models reduce the complexity of these tasks, which in turn accelerates DL research and product development.

Popular DL compiler intermediate representations (IRs) offer different tradeoffs between expressivity, composability, and portability [1, 33, 50, 52, 5, 38]. Early frameworks adopted IRs specialized for then-state-of-the-art models and/or emerging hardware accelerators. As a result, non-trivial extensions require patching or even forking frameworks [27, 47, 52, 41, 55, 38, 51]. Such *ad hoc* extensions can improve expressivity while maintaining backwards compatibility with existing execution mechanisms. However, they are difficult to design, reason about, and implement, often resulting in modifications that are mutually incompatible.

Let us consider a hypothetical scenario that exemplifies IR design tensions in DL compilers. Suppose a machine learning engineer wants to write an Android app that uses sentiment analysis to determine the moods of its users. To maintain privacy, the app must run completely on-device, i.e., no work can be offloaded to the cloud. The engineer decides to use a variant of TreeLSTM, a deep learning model that uses a tree structure [46]. Unfortunately, current frameworks' IRs cannot directly encode trees, so she must use a framework extension like TensorFlow Fold [26].

Suppose that after adapting the model to run on her phone, the out-of-the-box performance of her model on her particular platform is not satisfactory, requiring her to optimize it. She chooses to employ *quantization*, an optimization that potentially trades accuracy for performance by replacing floating-point datatypes with low-precision ones. Although researchers have developed a variety of quantization strategies, each of which makes use of different bit-widths, rounding modes, and datatypes, our engineer must use a strategy supported by existing frameworks [15, 14, 34]. Unfortunately, frameworks only provide support for a small number of strategies, and supporting new quantization strategies is non-trivial. Each combination of operator, datatype, bit-width, and platform requires unique operator implementations. Optimizations like operator fusion exacerbate this combinatorial explosion, further increasing the number of unique implementations required. Furthermore, if a framework doesn't have specific support for the target phone model she cannot take advantage of specialized deep learning instructions or coprocessors [3].

The scenario above highlights the three-pronged *extensibility challenge* for DL IRs:

1. *Expressivity*: It should be straightforward to write models involving control flow, first-class functions and data structures (e.g., trees, graphs, and lists).
2. *Composability*: It should be straightforward to add and compose new optimizations with existing ones (e.g., quantization, operator fusion, and partial evaluation).
3. *Portability*: It should be straightforward to add new hardware targets (e.g., TPU, Inferentia) [20, 2].

Previous IRs have struggled to address these challenges, treating each component of the framework as a disconnected

set of programming tasks. Operators are defined in low-level languages like C++, connected by a dataflow graph, and then scripted in a host language like Python. Consequently, program analyses cannot cross language boundaries between components, inhibiting optimization and deployment. Learning from previous IRs, we have designed Relay, which features a principled approach to addressing extensibility and improves expressivity, composability, and portability over previous frameworks. We make the following contributions:

- The Relay IR, a tensor-oriented, statically typed functional IR, which we describe in Section 3. Relay's design is motivated by the insight that functional IRs, used by languages from the ML family[1] can be readily adapted to support DL. With its *expressive* semantics, including control flow, data structures, and first-class functions, Relay can represent entire state-of-the-art models.
- The insight that common features in ML frameworks, such as quantization and shape inference, can be reframed as standard compiler passes. By using this reframing we can tap into decades of traditional compilers research to design *composable* optimization passes.
- A platform-agnostic representation of operators and domain specific optimizations which work in concert to provide *portability* across hardware backends.

We evaluate Relay on several systems and over a diverse set of vision and NLP workloads to demonstrate that (1) Relay enables *expressive* programs via a large breadth of models, (2) Relay supports *composition* of program-level optimizations such as quantization and fusion, and (3) Relay provides *portability* by targeting a number of hardware backends. Not only does Relay provide these three properties, we do so while also demonstrating competitive performance. Relay is an open-source academic project.[2] It has been deployed at a popular web service provider, a telecommunications and consumer electronics manufacturer, and a social media company, among others.

2. Related Work

The acceleration of deep learning is an active topic of research and is cross-disciplinary by nature. The dominant platforms for deep learning are TensorFlow, PyTorch, and MxNet. Research on these frameworks cuts across all abstraction levels and involves experts from machine learning, systems, architecture, and programming languages (PL). We first discuss the evolution of modern DL frameworks, then the lower-level components DL frameworks have incorporated to gain performance (i.e., low-level tensor compilers and DL compilers), and finally, we turn to approaches from the PL community.

[1] "ML" as in "Meta Language," not "Machine Learning"

[2] Relay is publicly available at [redacted for review].

2.1. Deep Learning Frameworks

In the early days of deep learning, practitioners and researchers would program in general-purpose languages like Python utilizing scientific computing libraries like NumPy, which provide low-level *operators* such as matrix multiplication. In order to accelerate model execution, frameworks supporting accelerators such as GPU were introduced [5] . Early frameworks represented models as directed "computation graphs", where each node represents an operator, and each edge represents the flow of data from one operator to another. Computation graphs provide a limited programming model, enabling straightforward mapping of operators onto GPUs. Large technology companies, such as Google, Facebook, and Amazon, drive the development of frameworks, and consequently, each company has its own stack consisting of the core framework (TensorFlow [1], PyTorch [8], MxNet [6]), compilers(XLA [55], Glow [38], TVM [7]), and hardware accelerators (TPU [20], GraphCore, Inferentia [2]). Frameworks can be roughly categorized into those which support *static* computation graphs and those which support *dynamic* computation graphs. Frameworks which use static graphs are said to be *define-and-run* frameworks, whereas frameworks which use dynamic graphs are said to be *define-by-run* frameworks.

Define-And-Run Frameworks TensorFlow, Caffe [19], and Theano [5] are define-and-run frameworks. Static graphs represent a whole-program, enabling optimization and simplified deployment, by removing the need for a host language like Python. TensorFlow (TF) extends pure dataflow graphs with *control edges* to emulate the functionality of `if` and `while`. TF's representation captures many state-of-the-art models, provides support for heterogeneous hardware back-ends, and enables reverse-mode automatic differentiation [4, 1]. TF's encoding of control has limitations, as control-flow structures do not clearly map to familiar control-structures, instead using specialized encodings which make adapting traditional optimizations challenging. Furthermore, unmodified TensorFlow does not support building models where the shape of the computation graph is dependent on the input, frustrating researchers who wish to experiment with complex models. TensorFlow Fold addresses this *particular* limitation [26] but offers no general and extensible solution. The crux of the problem is the lack of generic mechanisms for users to define new control flow combinators (e.g., `fold`) and data types.

Define-By-Run Frameworks PyTorch [33], Gluon [12], Chainer [50], and TensorFlow eager-mode [41] are define-by-run frameworks which attempt to address the challenges of previous work. The approach popularized by PyTorch is to use a host language (e.g., Python) to eagerly execute operations while simultaneously building a computation graph as a side effect. By using the full host language, its features may be used to provide a highly expressive programming model to users. However, dynamic frameworks construct a graph *per program trace* and must re-optimize when the graph topology

changes, costing CPU cycles and incurring communication overhead between the host machine and accelerators. Instead of just representing traces, Relay combines the advantages of both worlds by representing the whole program ahead of time, while supporting constructs like control flow, first-class functions, and data structures.

2.2. Low-Level Tensor Compilers

Low-level tensor compilers are focused on the production of high-performance operators which implement compute-intensive operations such as matrix multiplication or convolution. There are a number of competing approaches, both from academic and commercial entities, such as TVM [7], Halide [35], Tensor Comprehensions(TC) [53], and Diesel [11]. The most notable designs are either inspired by the compute-schedule split introduced by Halide and adapted by TVM, or the polyhedral framework, as used by TC and Diesel. Operator compilers perform code generation for sets of scalar loop nests, but only represent a restricted subset of a whole program, ignoring details such as memory allocation/management, data structures, closures, and arbitrary control flow. Relay focuses on composing generic operators, and the surrounding program into an efficiently orchestrated DL program.

2.3. Deep Learning Compilers

DL frameworks have adopted compilers to tackle both performance and portability for existing applications, most notably XLA [55], Glow [38], nGraph [10], ONNC [24], PlaidML [9], and ModelCompiler. These *graph compilers* use computation graph IRs and provide lowering onto a variety of targets. Often graph compilers only perform high-level optimizations and then offload to vendor-specific libraries.

Due to their limited programming model, they provide the same functionality as Relay with a more limited language. The most comparable points to Relay are recent developments in the TensorFlow and PyTorch ecosystems of MLIR and TorchScript, respectively. Google introduced MLIR as a path forward for unifying its myriad of IRs. Upon first examination MLIR might appear to be a replacement for XLA and related TF compiler efforts, but it is not that. MLIR is shared infrastructure for constructing a set of interoperating IR "dialects" which can be used to construct compilers. The MLIR project is working on IR dialects for TF's IR and a low-level polyhedral IR, but does not yet have an end-to-end solution for deep learning built upon MLIR, the insights in this paper can guide MLIR's dialect development.

TorchScript is a high-level Python-like IR developed as the first layer of PyTorch's JIT compiler. PyTorch (since v1.0) can rewrite a subset of user programs into TorchScript, an idealized subset of Python. TorchScript can then be executed by the TorchScript VM or JIT-compiled to a target platform. TorchScript sits many layers above code generation and must accommodate the flexible semantics of Python, which rules out entire classes of static analysis. In order to optimize away this dynamic behavior, TorchScript has a profiling JIT mode which identifies stable program traces during execution. These stable static traces can then be optimized by lower-level compilers such as Glow or Relay to perform the last level of code generation. Microsoft released ModelCompiler, a system for efficiently compiling RNNs defined in CNTK to CPU. ModelCompiler uses Halide to represent low-level operations, but lacks the expressivity of the Relay IR and only demonstrates support for CPUs.

2.4. Programming Languages for Deep Learning

In recent years, the design of new programming languages, or the augmentation of existing ones, has become a popular area of research. New languages designed for machine learning and related tasks include Lantern [54], Lift [43], Flux.jl [18] AutoGraph [30], Swift for TensorFlow [48], and JAX [25]. Lantern [54] is the most related work to Relay as it can be used as a code generator. Lantern is a deep learning DSL in Scala that uses lightweight modular staging (LMS) to lower code into C++ and CUDA. Lantern's defining feature is the use of delimited continuations to perform automatic differentiation. Delimited continuations provide an elegant algorithm for AD, only requiring local transforms, but incurs cost of heap allocated structures, and a less straightforward mapping to define-by-run frameworks. Lantern solves this problem by using a CPS transform which complicated further optimization and code generation. Lantern does not yet support hardware accelerators, and does not focus on full program optimizations. The alternative approach is the augmentation of languages to support deep learning, the most notable being systems like AutoGraph, Flux.jl, Swift for TensorFlow, and JAX. These systems are designed to be user-facing programming environments for deep learning and use a compiler IR to generate code. For all intents and purposes Relay could be the IR in question, therefore Relay complements these systems well by providing a more expressive IR to map computation onto.

3. Design

Relay's expressive high-level IR is designed to support complex models while abstracting over hardware-specific implementation details to enable hardware agnostic program analysis and optimization. Rather than invent an entirely new language, Relay's IR design is based on IRs used by the well-studied ML family of functional programming languages (e.g., SML and OCaml). These IRs are expressive enough to capture general-purpose programs (including control flow, first-class functions, and data types) and have clearly specified semantics (e.g., lexical scope and controlled effects). By borrowing from PL literature, we can apply program analysis and optimization techniques from decades of research [28].

Relay's IR takes a small functional core and enriches it with domain-specific additions—namely, the inclusion of tensors and operators as expressions and a novel tensor type system

```
Expr  e  ::=  %l                                  (local var)
          |   @g                                  (global variable)
          |   const((r | b),s,bt)                 (constant tensor)
          |   e(⟨τ, ..., τ⟩)?(e, ..., e)          (call)
          |   let %l (:τ)? = e; e                 (let)
          |   e; e                                (let %_ = e; e)
          |   %graph = e; e                       (graph let)
          |   fn (⟨T, ..., T⟩)?
          |       (x, ..., x) (→ τ)?              (function)
          |       {e}
          |   (e, ..., e)                        (tuple formation)
          |   e.n                                 (tuple proj.)
          |   if (e) {e} else {e}                 (if-else)
          |   match (e) {
          |     | p → e
          |     ⋮                                 (pattern match)
          |     | p → e
          |   }
          |   op                                 (operator)
          |   ref(e)                             (new ref)
          |   !e                                 (get ref)
          |   e:=e                               (set ref)

Type  τ  ::=  bt                                  (base type)
          |   s                                   (shape)
          |   Tensor[s,bt]                        (tensor type)
          |   tv                                  (type variable)
          |   fn ⟨T, ..., T⟩
          |       (τ, ..., τ) → τ                (function type)
          |       (where τ, ..., τ)?
          |   Ref[τ]                              (ref type)
          |   (τ, ..., τ)                        (tuple type)
          |   τ[τ, ..., τ]                       (type call)
          |   tn                                  (type name)
```

Figure 1: The BNF Grammar for the Relay language.

design to support tensor shapes. Our principled design enables the import of existing models from deep learning frameworks and exchange formats, the implementation of a number of domain-specific optimizations, and efficient deployment across a variety of targets. In the remainder of this section, we describe the IR design in further detail and explore the ramifications of this design on the compilation stack.

3.1. IR

The Relay IR is designed to subsume the functionality of computation graph-based IRs while providing greater faculties for abstraction and control flow. We present Relay's design by incrementally building up to the full IR starting from a subset that corresponds to a simple computation graph. Deep learning models fundamentally operate on tensors. Hence, Relay's primary value type is a tensor and operators are included as language primitives (see the `tensor constant` and `operator` rules in Figure 1). Relay leaves the implementation of each operator opaque; the operators are represented by a lower-level IR, which is optimized independently. A computation graph, in its simplest form, is a directed acyclic graph with multiple inputs and a single output. Relay uses three constructs to support these simple graphs: (1) `variable`, (2) function `call`, and (3) `operator`; see Figure 1 for the corresponding rules.

Multiple Outputs Computation graph IRs have primitive support for multiple outputs because many tensor operators require it. For example, the `split` operator separates a tensor along a given axis and returns each component. In Relay, multiple outputs can be modeled as tuples, requiring only two rules: `tuple formation` and `tuple projection`.

Let By construction, computation graphs enjoy implicit sharing of subcomputations via multiple outgoing dependency edges. Implicit sharing is often implemented via pointers that uniquely identify subgraphs, a property useful for both execution and analysis. Previous frameworks often obtain this sharing by using a host language's name binding to construct a graph (e.g., by binding a Python variable to a subgraph and using that variable to construct other subgraphs). General-purpose programming languages, on the other hand, provide *explicit* sharing via binding constructs, such as `let`. In programs free of scope, ordering, and effects, implicit sharing and explicit sharing are semantically equivalent. However, in practice, user programs rely on effects and ordering, requiring previous approaches to provide workarounds. For example, TensorFlow's Eager Mode inserts dummy control edges in its generated graphs to impose effect ordering. The lack of lexical scope in computation graphs complicates language features, like first-class functions and control flow, and reduces the precision of traditional analyses, such as liveness, because the high-level program structure is absent [32, 39]. The addition of a humble `let` binding, a central concept in functional languages, provides explicit sharing and a solution to the problems outlined above.

Control Flow Emerging models, particularly in the domain of natural language processing, increasingly rely on data-dependent control flow, forcing frameworks based on computation graph IRs to incorporate control flow, often through *ad hoc* and difficult-to-extend constructs. For example, TensorFlow Fold [27] extends TF with special combinators that dynamically compute a graph for each shape permutation; these high-level constructs are opaque to further optimizations. The functional programming community has demonstrated that recursion and pattern matching are sufficient to implement arbitrary combinators for control flow and iteration (e.g., maps, folds, and scans). To support the definition of functional combinators we enrich Relay with two more language features to implement arbitrary combinators: `if` and first-class

```
i = tf.constant(1)
j = tf.constant(1)
k = tf.constant(5)

def c(i, j, k):
  return
    tf.equal(
      tf.not_equal(
        tf.less(i + j, 10),
        tf.less(j * k, 100)),
       tf.greater_equal(k, i + j))
def b(i, j, k): return [i+j, j+k, k+1]
tf.while_loop(c, b, loop_vars=[i, j, k])
```

⇒

```
fn %while_loop(
  %lvar0: Tensor[(1,), int32], %lvar1: Tensor[(1,), int32],
  %lvar2: Tensor[(1,), int32]) {
  %0 = add(%lvar0, %lvar1)
  %1 = less(%0, meta[Constant][0])
  %2 = multiply(%lvar1, %lvar2)
  %3 = less(%2, meta[Constant][1])
  %4 = not_equal(%1, %3)
  %5 = add(%lvar0, %lvar1)
  %6 = greater_equal(%lvar2, %5)
  if (min(equal(%4, %6))) {
    %9 = add(%lvar0, %lvar1)
    %10 = add(%lvar1, %lvar2)
    %11 = add(%lvar2, meta[Constant][2])
    %while_loop(%9, %10, %11)
  } else { (%lvar0, %lvar1, %lvar2)
  }
}
%while_loop(meta[Constant][3], meta[Constant][4], meta[Constant][5])
```

Figure 2: A simple TensorFlow loop in the user-facing DSL and the Relay loop produced by automatically converting it. Note the TensorFlow while loop corresponds neatly to a tail recursive function. The Relay text format supports a "metadata" section which functions as a constant pool among other things. `meta[Constant][n]` represents the `n`-th constant in the pool.

recursive functions.

First-Class Functions A computation graph is a single computation from multiple inputs to multiple outputs. While it is tempting to reinterpret a graph as a function, graphs lack functional abstraction and named recursion. The addition of first-class named functions dramatically increases Relay's expressivity, allowing it to encode generic higher-order functions and thus capture higher-level program structure. First-class functions also enable simpler implementations of importers that map higher-level programs to our IR. For example, an instance of TensorFlow's looping construct `tf.while_loop` can be represented as a single specialized loop function or a generic fold over the loop state. See Figure 2 for an example of this conversion (via the Relay TensorFlow frontend).

Data Abstraction Many models make use of additional data types beyond tuples, such as lists, trees, and graphs [21, 46, 23]. Relay borrows from functional languages a generic and principled method of extension: algebraic data types (ADTs). To support them, we add mechanisms for (1) type declaration and (2) pattern matching. This final addition results in a strict functional language, closely resembling the core of languages like OCaml and SML. The increase in expressivity introduced by the Relay IR introduces new optimizations challenges, which we discuss in Sec. 4.

3.2. Type System

Relay's type system is essential to optimizations. Typing guarantees both well-formedness of the program and provides crucial tensor shape information to perform allocation, check correctness, and facilitate loop optimizations. Shape information is also valuable for data layout transformations and tensorization, two transformations often demanded by hardware accelerators. In computation graph IRs, only numeric data types and shapes are tracked for each operator. Symbolic shapes (i.e., shape polymorphism) are only handled dynamically, inhibiting certain types of optimizations.

It is possible to model arbitrarily complex static properties, such as shape information, with a dependent type theory [40], but such a design incurs significant user complexity. By incorporating shape analysis into a broader type system, Relay's type system balances the desire for static tensor shapes with usability. In this subsection, we describe how to extend a polymorphic type system with shape information and type inference with shape inference.

Tensor Types The primitive value in Relay is a tensor, which has a shape and a base type (`tensor type` in Figure 1). Base types describe the elements of tensors by tracking the bit width, the number of lanes (for utilizing vectorized intrinsics), and whether the type is floating point or integral. To ensure Relay can offload tensor computation to devices with greatly varying architectures, Relay tensors may only contain base types, preventing, for example, tensors of closures. The shape of a tensor is a tuple of integers describing the tensor's dimensions. A dimension may be a variable or arithmetic expression that indicates how the output shape of an operator depends on those of its inputs. Functions may be polymorphic over shapes, which results in shape constraints that must be solved during type inference. Sec. 3.2 describes the process. Relay also supports a special shape called `Any`, which is used to mark a dynamic shape when static relationships are not profitable to model.

Operators and Type Relations Operators are one of the key primitives that differs from those of general-purpose programming languages. Relay's use of opaque operators enables backends to choose different lowering strategies based on the hardware target. Relay's operator set is extensible, meaning that users may add new operations. Supporting common or user-defined tensor operators requires a type system that can adapt to complex shape relationships between input and output types (e.g., elementwise operators with broadcasting

semantics).

To handle the constraints between operators' argument shapes, Relay's type system introduces type relations. A type relation is implemented as a function in the meta-language and represents a symbolic relationship between the input and output types. When developers add a new operator to Relay, they may constrain its type with an existing relation or add their own. Function types may include one or more type relations over a subset of the argument types and the return type. The type checker enforces that these relationships hold at each call site.

Type Inference To incorporate type relations into Relay's type system, we enrich a Hindley-Milner-style type inference algorithm with a constraint solver. Relay's inference algorithm has three steps: first, it performs a pass over the AST, generating types and a set of relations, then it solves the incurred constraints, and finally annotates each sub-expression with its inferred type.

When the type inference algorithm visits a function call site, the function's type relations are instantiated with the concrete argument types at the call site. Each instantiated relation is added to the queue of relations to solve. The relationship between a call's type variables and relations is added as an edge to a bipartite dependency graph where the two disjoint sets are type variables and type relations. Traditional unification constraints are represented using a modified union-find structure that integrates with this dependency graph.

Once the queue is populated, the algorithm will dequeue a relation and attempt to solve it. There are two cases when solving a type relation:

1. If all the relation's type variables are concrete, we the relation function. If that function returns true, the constraint is discharged. Otherwise, type checking fails.
2. If any type is fully or partially symbolic, the algorithm will propagate existing concrete type information via unification. All relations affected by new assignments to type variables (as determined by the dependency graph) are moved to the beginning of the queue. If the current type relation is now completely solved, we discard it to avoid unnecessarily visiting it again.

We run this to fixpoint or until the queue is empty. If the queue is non-empty and no progress is made between iterations, then at least one variable is underconstrained and inference fails. Note that a type relation's implementation can compromise type soundness, as they are axiomatic descriptions of operations implemented outside of Relay. In practice, the number of type relations needed to express Relay's operators is small, and their implementations are straightforward and amenable to exhaustive testing.

3.3. Compiler Framework

The process for compiling Relay proceeds in three stages. First, the frontend converts input formats into the Relay IR. Next, the Relay compiler typechecks and optimizes the program to produce the final program. After performing optimizations, the Relay backend transforms the Relay program into a form that can be executed on the intended hardware, based on the specified execution mechanism. The backend additionally lowers Relay operators into a TVM expression, computes a schedule for the final TVM expression, and lowers it into native code.

Frontend There are several ways to write an Relay program. A user can build an in-memory representation of a program in C++ or Python, parse one written in the Relay text format, load one from the on-disk serialization format, or import one from popular frameworks and interchange formats (e.g., TensorFlow, MxNet, Keras, DarkNet, and ONNX). Many frameworks and interchange formats use static computation graph-based representations, which can easily be translated into Relay. A greater challenge is translating frameworks with a richer computation model such as TensorFlow (TF). TF supports control flow and includes `TensorArray`, a write-once tensor container. We can extract the loop structure out of the TF graph, converting it to an Relay loop, and transform the `TensorArray` into an Relay list. Once new deep learning languages and IRs under development are stable it is likely they can be translated into Relay (see Section 2.4). PyTorch provides an expressive programming model, and is a good fit for Relay, which has integration into PyTorch's JIT infrastructure, enabling users to transparently use Relay for improved performance.

Compiler Once an Relay abstract syntax tree (AST) is produced, the program is optimized by applying a series of Relay-to-Relay passes. Between each pass, Relay performs type inference and checking, rejecting malformed programs as well as populating shape and type information that passes can utilize. The Relay compiler supports traditional optimizations (e.g., constant folding, common subexpression elimination, and dead code elimination) and domain-specific optimizations (see Sec. 4).

Backends Relay produces machine-specific code by decomposing the problem of code generation into multiple distinct phases. Relay translates all operators into TVM expressions to produce dense linear algebra kernels [7, 53, 35]. TVM produces low-level operators that expect a fixed calling convention, as well as preallocated inputs and outputs. The result is an object file containing hardware-specific implementations of all operations. The remaining Relay program then is executed or compiled, with operator invocations replaced by calls to the optimized operators. By representing operators as TVM expressions, we can programmatically transform them and automatically generate new implementations for the transformed operators. Optimizations like fusion and quantization rely on this novel behavior. After primitive operators are lowered, the remaining Relay program ties together operator invocations, allocation, control-flow, recursion, and high-level data structures. There are multiple options for executing the combined full program: the Relay interpreter (with JIT compilation), an

Relay virtual machine, the TVM graph runtime, and an experimental Relay ahead-of-time compiler that converts programs to C++ to produce a target-specific binary.

4. Optimizations

A high-level IR by itself does not provide a path to high-performance code. Obtaining high-performance models requires domain-specific optimizations tailored to deep learning. In this section, we showcase the use of the Relay compiler framework to write general, domain-specific, and target-specific optimizations, enabling generation of high-performance code.

4.1. Operator Fusion

Operator fusion is an indispensable optimization in deep learning compilers. Fusion enables better sharing of computation, removal of intermediate allocations, and facilitates further optimization by combining loop nests. Fusion is known to be the most critical optimization in machine learning compilers, but existing fusion techniques are closed (working over a fixed set of ops) and target-dependent. Traditional operator fusion algorithms resemble instruction selection: A sequence of operators eligible for fusion is first identified and then replaced with a corresponding handwritten fused implementation, usually from a vendor-provided library. For example, if a fused implementation for a GPU operator does not exist in CuDNN, it will remain unfused. More advanced strategies, implemented in XLA, detect a closed set of statically shaped operators for fusion and generate code for CPU/GPU.

Relay's fusion algorithm addresses weaknesses of previous approaches by representing *all* operators in a secondary IR. Relay operators are backed by a TVM compute expression that describes operations in a high-level DSL that resembles Einstein notation but omits low-level scheduling details. TVM's separation of compute and scheduling provides many favorable qualities for Relay's fusion algorithm. It enables producing shape-specialized fused operators for an open set of operators, fusing arbitrary-length chains of operators (not just pairwise combinations), and handling operators with multiple outputs and nonlinear consumer-producer patterns. TVM is also able to reschedule after fusion and perform further optimization via auto-tuning. Relay performs fusion in two steps, detailed below.

Extraction First, Relay identifies subexpressions containing fusion-eligible and factors them into local functions that are marked as primitive. Primitive functions can later be lowered to platform-specific code. Fusion-eligible subexpressions are identified by constructing a directed acyclic graph (DAG) representing data flow between operators. As the dataflow DAG is acyclic, it allows for the simple construction of a post-dominator tree. Subexpressions are grouped into equivalence classes determined by their immediate post-dominator. The use of the post-dominator tree enables fusion between non-linear producer-consumer relationships; for example, Relay can fuse diamond-shaped data-flow relations, where an input is used by multiple parallel operator chains that are combined again by a later operator. Finally, Relay constructs an expression from each equivalence class, collects the expressions' free variables, constructs a function with the expression as the body and the free variables as parameters, and marks it as primitive.

Lowering In a second step, the Relay compiler converts the generated primitive function into platform and shape specific code. For each operator, Relay collects the high-level TVM expression that represents it, then combines them into an aggregate expression that represents the fused operation. Generating code using TVM also requires producing a schedule. It is possible to use TVM's default schedule to generate code for a single operation, but the default schedule does not support fusion. In order to generate code for the combined expression, we must generate a master schedule based on the set of operations being fused. The fusion algorithm analyzes the expressions to select a master schedule, the master schedule will perform the appropriate scheduling actions to generate fused code, such as inlining loops, or reorganizing computation. By combining the master schedule with the fused computation, Relay is able to produce an optimized version of the operator for any platform supported by TVM. For example, a related project by one of the co-authors implemented a RISC-V backend which immediately obtained full operator fusion with no new code. Due to the Relay compiler's integration with AutoTVM, we can further optimize fused operations by performing auto-tuning on the master schedule template to obtain the best performance.

4.2. Quantization Framework

Deep learning is constrained by memory, compute, and accuracy. Accuracy is often the only metric optimized by machine learning researchers, leading to compute- and memory-hungry models. The sheer number of parameters and the requisite compute makes deploying models to resource-limited devices, such as in mobile or IoT, challenging. Even in non-edge devices, the compute cost of using datatypes like FP32 is large and computing with mixed precision or reduced precision can aid performance. Unfortunately, reducing bit-width is not a silver bullet and can dramatically harm model accuracy. The tradeoffs between these quantities has lead to the study of quantized neural networks, the process by which NNs are modified to use a smaller precision or non-standard datatypes to improve throughput and memory usage. Quantization is particularly essential for supporting many accelerators due to their restricted set of datatypes.

State-of-the-art work on quantization demonstrates a number of tradeoffs between different quantization techniques, with the best often determined by platform and model type [22]. Most DL frameworks have chosen a specific set of fixed quantization schemes and datatypes due to the effort required to manually implement operators for each pairing.

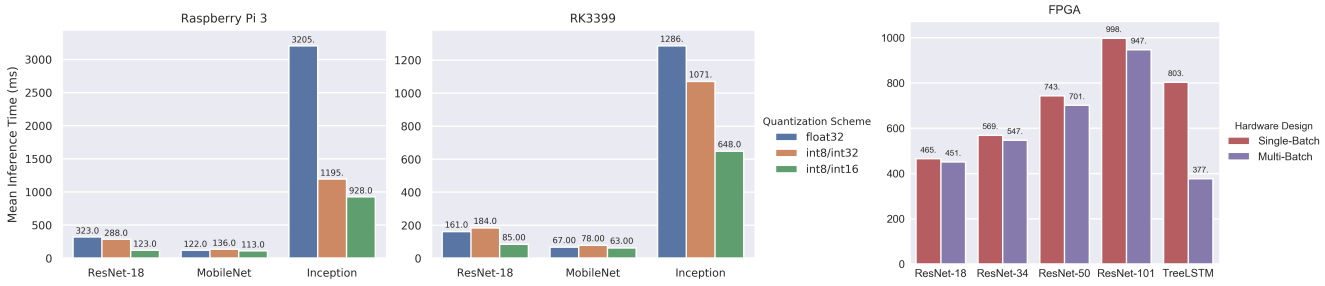

Figure 3: *(left)* Inference time of vision DNNs on low-power platforms using different data types. Relay allows us to reduce inference time on power-constrained devices by easily substituting `float32` multiplications with `int8` multiplications and `int16` or `int32` accumulations (denoted as `int8`/`int16` and `int8`/`int32`, respectively). We used 1000 trials for each model. *(right)* Batch-size-normalized inference time of vision DNNs and a TreeLSTM running on two DNN accelerator variants implemented on an edge FPGA. One accelerator performs single-batch inference, while the other implements multi-batch inference. The two hardware designs have the same number of compute units that are arranged differently to take advantage of different types of tensor computation. Relay applies a multitude of graph-level transformations required to run different workloads onto these DNN hardware designs. We used 12 trials for each model.

Instead, Relay includes a generic, compiler-based quantization flow that supports a diverse set of quantization schemes and can automatically generate code for each one. Relay provides a general-purpose program-rewriting framework that can be extended with per-operator rules, which can annotate inputs and outputs with a datatype and precision to quantize to. Users can overload Relay's existing quantization rewriting rules or add new ones to implement different quantization strategies, enabling users to choose between signed or unsigned integers or different rounding strategies, such as floor, ceiling, or stochastic rounding.

Figure 4 illustrates the rewriting process. Furthermore quantization is expanded to standard Relay operators, which perform the scaling. Due to this choice, Relay can then fuse these elementwise operations into the original operator, resulting in a brand-new quantized operation. Finally, Relay can subsequently apply further optimizations like layout transformation, accelerator-specific packing, or auto-tuning to further improve performance or portability. This enables the generation of customized quantized operators for user-provided schemes and operators, not limiting users to a single scheme.

We will now detail the three steps of the generic quantization flow: annotation, calibration, and realization.

Annotate Annotation rewrites the program to insert simulated quantization operations according to annotation rule for each operator. Each input or output to be quantized is passed to `simQ`, an operator that simulates the effect of quantization (for example, from a 32-bit floating point value to an 8-bit integer value). `simQ` has a set of parameters that must then be calibrated in order to correctly quantize the graph, namely the bits, the scale, and the range. `simQ` simulates quantization on the unquantized type; that is, it performs computation on the unquantized type and then scales it to the target type. By computing on the unquantized type, Relay can later calibrate the parameters to `simQ` a necessary step to preserve accuracy of the model.

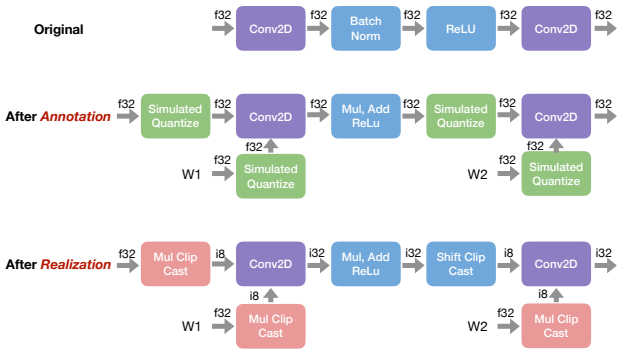

Figure 4: The top graph represents the dataflow graph of operators after annotation, and the bottom graph represents the result of quantization.

$$\texttt{simQ}(x,\beta,\sigma,\rho) = \frac{\texttt{clip}\left(\texttt{round}\left(x/\rho \cdot 2^{\beta-\sigma}\right)\right)\cdot\rho}{2^{\beta-\sigma}}$$

Calibrate As seen above `simQ` has an input x, as well as a number of parameters β, σ, and ρ. `simQ`'s parameters control the mapping between the quantized and unquantized type and must be calibrated, without calibration the model can be wildly inaccurate. We must perform an auxiliary optimization task to find the appropriate setting for these parameters. The Relay compiler supports a variety of strategies for setting these parameters. The first strategy implemented is a hyper parameter sweep of a single global scale until such a scale is found that does not result in overflow. Another approach is a vision specific scheme which uses a per-channel scale, and optimizes the scales using a simple mean-squared error loss. Finally an approach adopted from MxNet uses a KL-divergence based loss to optimize the quantization scales.

Realize Finally, after the algorithm has set the parameters appropriately, it applies realization, which transforms the `simQ` operator into the below quantization operator.

$$Q(x,\rho,\beta,\sigma) = \texttt{cast}\left(\texttt{clip}\left(\texttt{round}\left(x/\rho\cdot 2^{\beta-\sigma}\right),\texttt{qtype}\right)\right)$$

The produced operator performs the necessary scaling by realizing the operation as a sequence of finer-grained operators such as multiplication and rounding. The output of original operator is now immediately scaled by this new operation. Due to Relay's handling of fusion we are able fuse these scaling operations directly into to the original operator, transforming a convolution from `fp32` to a type such as `int4`.

4.3. Partial Evaluator

Existing deep learning IRs have relied on a mixture of staging and constant evaluation in order to optimize user programs. Partial evaluation is a generalized form of constant evaluation that can reduce partially constant programs. A partial evaluator (PE) allows the use of high-level abstractions without limiting code that *could* in practice be compiled to a particular target. Relay is the first compiler to apply partial evaluation techniques to deep learning, the core approach of which is based on [49]. Partial evaluation, when composed with other optimizations like fusion, yields a variety of useful optimizations without requiring a separate implementation of each. For example, the partial evaluator can be used to perform loop unrolling, which then enables further fusion, without any additional compiler passes.

Relay's partial evaluator works by defining a interpreter where the value domain is partially static values. The partially static domain represents simple values, such as constant tensors, as themselves. The representations of aggregate values mirror their structure; for example, tuples become a tuple of partially static values. The partially static domain represents dynamic values, which may not be known until execution time, alongside the static values traditionally supported by constant evaluators. Our partial evaluator must solve two important problems: managing effectful computations and handling references. In order to handle effects, the evaluator keeps the generated program in A-normal form to ensure effects are properly ordered and restrict duplication of effectful computations. The partial evaluator supports references by simulating the store at partial evaluation time. The explicit store is threaded throughout execution and provides a flow-sensitive PE. Finally, the evaluator constructs a new program with the static subcomputations evaluated away.

4.4. Accelerator-Specific Optimizations

This subsection focuses on a subset of optimizations necessary to compile Relay to deep learning hardware accelerators. Although DL accelerators form a diverse family of designs, one property they have in common is a restricted computing model. This means that some individual accelerators may not be able to solely execute many Relay programs. For example, many accelerators cannot execute unbounded loops, requiring some computation to be scheduled on a host device like the CPU.

Axis scale folding is an optimization that removes scaling operations that occur before or after convolution-like operators. The multiplication by a scalar is moved through a convolution towards its constant inputs, such as parameters. By moving the scaling operation to a constant weight, we are able to compute away the scale using the partial evaluator. This optimization is required for certain accelerators that lack scalar multipliers [31]. In order to target these accelerators, we must eliminate *all* scalar operations.

Parallel convolution combination is a specialized optimization that fuses multiple 2D convolutions that share the same input. The goal of this pass is to produce a larger kernel for the GPU, as each kernel launch on the GPU has overhead. It was designed with the Inception network [45] in mind, as it contains blocks of convolutions that share the same input. The entire parallel convolution combination pass, including documentation and tests, required fewer than 350 lines of code and was contributed by a non-Relay affiliated undergraduate student in their first contribution to our codebase.

5. Evaluation

We evaluate Relay's ability to provide expressivity, composability, and portability, without compromising on performance. In particular, our evaluation is composed of three parts:

1. **Relay expresses diverse workloads**: Despite increasing expressiveness, Relay's performance is competitive with the state of the art on popular models.
2. **Relay enables composable optimizations**: Relay supports composing program transformations to incrementally improve performance.
3. **Relay handles challenging backends**: Relay can compile models to execute efficiently on a variety of backends, such as FPGA accelerators, which require quantization, layout optimizations, and bit-packing transformations.

We evaluated the following vision models: *Deep Q-Network (DQN)*, a CNN that achieved state-of-the-art performance on 49 Atari games in 2015; *MobileNet*, a CNN designed for image recognition on mobile and embedded devices; *ResNet-18*, a CNN for image recognition that achieved state-of-the-art performance on ImageNet detection tasks in 2015; and *VGG-16* (named for the Visual Geometry Group at Oxford), a CNN for image recognition that achieved top-2 performance in the 2014 ImageNet Challenge. [29, 17, 16, 42]. We evaluated the following NLP models: *CharRNN*, a generator character-level RNN from a PyTorch tutorial; *TreeLSTM*, a generalization of LSTMs to tree-structured network topologies; and *RNN*, *GRU*, and *LSTM*, a selection of models from the Gluon Model Zoo [37, 46, 13].

5.1. Experimental Methodology

Because we only evaluate inference in this paper, we frequently make use of random inputs to models when measuring performance. There were two exceptions where we evaluated on real data because it was readily available: CharRNN and TreeLSTM. For each experiment, we run 10 untimed "warm-up" iterations to ensure any effects from caching and

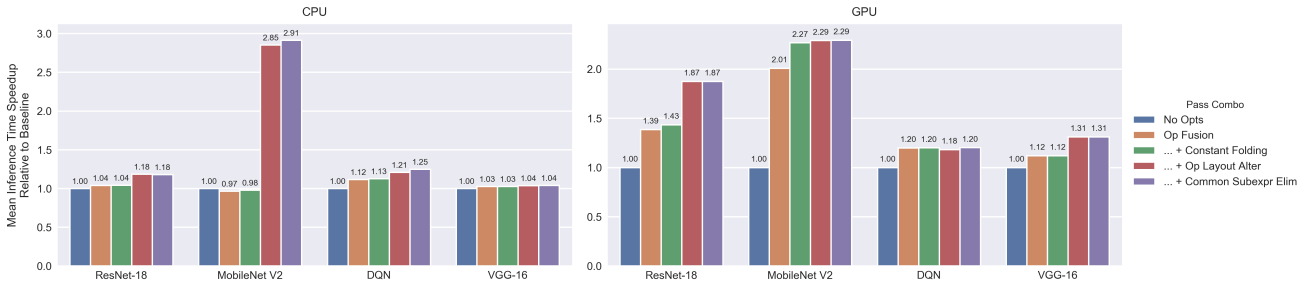

Figure 5: Speedup from successively layering compiler passes in Relay on CPU (AMD Ryzen Threadripper 1950X) and GPU (Nvidia Titan-V), relative to no optimizations at all. The "Op Fusion" bars represent the application of operator fusion, the "... + Constant Folding" bars represent the application of operator fusion *and* constant folding, and so on. The full list of passes used is as follows: *operator fusion*; *constant folding*; *operator layout alteration*, which transforms the data layouts of operators for better cache performance; and *common subexpression elimination*. We find that composing passes can steadily increase performance. The effectiveness of each pass is both model- and device-dependent. In particular, the most effective passes for CPU and GPU are operator layout alteration and operator fusion, respectively.

JIT compilation are excluded from the timed runs. The vision and NLP experiments (from Section 5.3 and Section 5.2) were run on a machine with an AMD Ryzen Threadripper 1950X 16-Core CPU, an Nvidia Titan-V GPU, and 64 GB of RAM. For the vision workloads, we used TVM's graph runtime as the executor, and for the NLP workloads, we used Relay's AoT compiler. The low-power vision experiments from Section 5.4 were run on multiple edge-class ARM development boards: a Raspberry Pi 3, a Firefly RK3399, and an Pynq-Z1 FPGA platform. We implement our DNN accelerators on a Zynq-7020 low-cost FPGA, and clock them at 100MHz. We used the following software versions: CUDA version 10.0, CuDNN version 7.5.0, TVM commit `e518fe1c`[3], MxNet version 1.5.0, PyTorch version 1.2.0, and TensorFlow version 1.14.0.

5.2. Relay Expresses Diverse Workloads

An age-old story in compilers literature is that increasing expressivity impacts the global performance of the system. We set out to build zero-cost abstractions for Relay, governed by Stroustrup's principle, "What you don't use, you don't pay for" [44]. We demonstrate that we achieve competitive performance on a wide set of CNNs that are well supported by existing frameworks. We evaluated inference time for two classes of workloads: computer vision and natural language processing. We compared Relay to NNVM, TensorFlow, TensorFlow-XLA (Accelerated Linear Algebra), PyTorch, and MxNet. Results are summarized in Figure 6.

Vision Evaluation We ran each model with batch size 1, a common setting in inference tasks. Relay achieves performance on par with NNVM and outperforms TensorFlow, TensorFlow-XLA, MxNet, and PyTorch on every benchmark. Relay's ability to apply aggressive inter-operator optimizations enables it to outperform existing frameworks. Operator fusion over long chains of operations is particularly effective, because it can generate *new* hardware-specific fused implementations.

[3]NLP experiments required extensions to the MxNet importer that will be made public later

NLP Evaluation Implementations of the NLP models were not available in all frameworks; we used MxNet baselines for RNN, GRU, and LSTM and PyTorch for CharRNN and TreeLSTM. NLP workloads feature control flow, which makes them more challenging to optimize. Relay performs better than MxNet on GRU and LSTM because they are implemented in Python using MxNet's looping constructs. However, MxNet outperforms Relay on the Gluon RNN, because it uses a hard-coded optimization to unroll the RNN, whereas Relay expresses it as a loop without any unrolling. PyTorch instead uses handwritten and heavily optimized C implementations of the recursive network cells. Despite this, our pure Relay implementation outperforms PyTorch by 1.4× on CharRNN and 2× on TreeLSTM. This speedup comes from Relay's ability to compile *entire* models with complex control flow (e.g., CharRNN) to a single lean binary.

5.3. Relay Enables Composable Optimizations

We demonstrate that Relay facilitates composable optimizations by evaluating vision workloads under both general-purpose and DL-specific compiler passes. Figure 5 shows mean inference speedup relative to no optimizations as Relay applies optimizations more aggressively. We find that performance gains vary significantly between each device-model pairing. Most networks benefit greatly from operator layout alteration on CPU and operator fusion on GPU. On both CPU and GPU, VGG-16 is resistant to most optimizations, because the architecture primarily consists of back-to-back convolutions, which are not fusable. Elementwise operations *can* be fused, so we see greater improvements on ResNet and MobileNet, which both feature elementwise adds from residual connections. It's unsurprising that MobileNet fares so well on CPU—it is *designed* to run well on CPU. Nature-DQN has simple operators, which don't benefit from layout alterations, whereas ResNet-18 and VGG-16 are dense convolutional neural networks, which *do* benefit from layout transformations. Overall, these results show that Relay lets us compose opti-

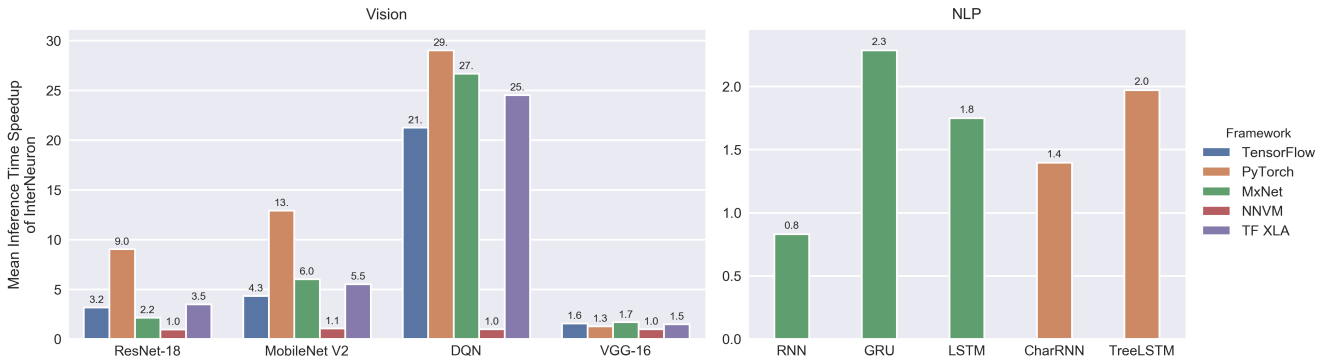

Figure 6: Inference speedup of Relay relative to popular frameworks on vision and NLP benchmarks. The vision benchmarks used an NVIDIA Titan-V GPU, and the NLP benchmarks ran on CPU only. We ran 1000 trials for each model, except for CharRNN, on which we used 100 trials. Relay matches the performance of NNVM on vision but additionally supports NLP, where Relay provides performance competitive to the state of the art (up to 2.3× speedup over MxNet on GRU).

mizations in a way that is beneficial to diverse workloads.

5.4. Relay Handles Challenging Backends

To demonstrate portability, we evaluate two sets of optimizations: those that are merely *beneficial* for low-power platforms and those that are *necessary* to target hardware accelerators.

Quantized Inference on ARM Platforms To demonstrate the effectiveness of our generic quantization (see Section 4.2), we use Relay to evaluate both *accuracy* and *performance* of different quantization schemes on vision workloads. To evaluate *accuracy*, we tested various quantization schemes (denoted m/n for m-bit quantization and n-bit accumulation) against a `float32` baseline on three vision models, as shown in the table below:

ResNet-18		MobileNet V2		Inception V3	
QS	Acc.	QS	Acc.	QS	Acc.
`fp32`	70.7 %	`fp32`	70.9 %	`fp32`	76.6 %
8/32	69.4 %	8/32	66.9 %	16/32	76.6 %
8/32	69.4 %	8/16	66.9 %	8/32	75.2 %

Figure 3 shows the results of different levels of quantization on *performance* when applied to the Raspberry Pi 3 and Firefly RK3399 ARM-based platforms. The numbers show that as we opt for a more aggressive quantization scheme (e.g., 8/16), we achieve much improved performance with hardly a drop in accuracy. Interestingly, on some model/platform pairs, the `int8/int32` scheme performs slightly worse than `float32` on both platforms, which likely stems from the existence of faster hardware intrinsics for 16-bit operations on these systems.

Targeting Deep Learning Accelerators on FPGAs We demonstrate that Relay can support specialized hardware by compiling vision and NLP workloads onto two DNN accelerator designs (`single-batch`, `multi-batch`) we generate in-house. The two DNN designs have the same number of MAC (multiply and accumulate) units which are arranged differently to expose different compute intrinsics to the compiler.

We evaluate batch-size-normalized inference time on the accelerator designs on a mix of vision and NLP workloads: ResNet-18, ResNet-34, ResNet-50, ResNet-101 [16]; and TreeLSTM in Figure 3. The ResNets have high arithmetic intensity due to 2D convolutions, while the NLP workload is memory bound due to the lower-intensity vector-matrix multiplication used in the LSTM cells.

We show that running the workloads on the `multi-batch` DNN accelerator improves throughput on all workloads at the cost of naturally increasing inference latency. Batching is not compelling on ResNet because it increases the arithmetic intensity of a workload that is already compute-bound. On the other hand, TreeLSTM presents a compelling target for batching due to the memory bound nature of the LSTM cell computation. While these two accelerator variants have the same peak throughput on paper, we show that Relay let us evaluate more nuanced end-to-end performance numbers across different workloads.

These experiments demonstrate Relay's ability to target current and future deep learning architectures, and make informed decision on what hardware design to choose across different workloads.

6. Conclusion

This paper introduced Relay, a high-level IR that enables end-to-end optimization of deep learning models for a variety of devices. In particular, Relay provides a design for *extensibility*. In addition to representing, optimizing, and executing models defined in popular frameworks, we use Relay's design to define a combination of traditional and domain-specific optimizations. Relay's approach can be adopted by other DL frameworks to implement IRs that can support extension *without* compromising on *performance*, *expressivity*, *composability*, or *portability*. With its extensible design and expressive language, Relay serves as a foundation for future work in applying compiler techniques to the domain of deep learning systems.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Amazon Web Services. Aws inferentia. https://aws.amazon.com/machine-learning/inferentia/, 2018.

[3] Apple. https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/, 2017.

[4] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767, 2015.

[5] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 3 – 10, 2010.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[8] PyTorch Contributors. Pytorch. https://pytorch.org/, 2018.

[9] Intel Corporation. Plaidml. https://www.intel.ai/plaidml/, 2017.

[10] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *CoRR*, abs/1801.08058, 2018.

[11] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod 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*, MAPL 2018, pages 42–51, New York, NY, USA, 2018. ACM.

[12] Gluon Team. Gluon. https://gluon.mxnet.io, 2018.

[13] Gluon Team. Gluon model zoo. https://gluon-nlp.mxnet.io/model_zoo/index.html, 2019.

[14] Google. Tensorflow lite supported datatypes, 2019.

[15] J.L. Gustafson. *The End of Error: Unum Computing*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2015.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[18] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018.

[19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[20] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan

Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[21] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/, May 2015.

[22] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018.

[23] Xiaodan Liang, Xiaohui Shen, Jiashi Feng, Liang Lin, and Shuicheng Yan. Semantic object parsing with graph LSTM. *CoRR*, abs/1603.07063, 2016.

[24] W. Lin, D. Tsai, L. Tang, C. Hsieh, C. Chou, P. Chang, and L. Hsu. Onnc: A compilation framework connecting onnx to proprietary deep learning accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 214–218, March 2019.

[25] Google LLC. Jax: Autograd and xla. https://github.com/google/jax, 2018.

[26] Moshe Looks, Marcello Herreshoff, and DeLesley Hutchins. Announcing tensorflow fold: Deep learning with dynamic computation graphs. https://research.googleblog.com/2017/02/announcing-tensorflow-fold-deep.html, February 2017.

[27] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *CoRR*, abs/1702.02181, 2017.

[28] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. Exploiting vector instructions with generalized stream fusio. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 37–48, New York, NY, USA, 2013. ACM.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[30] Dan Moldovan, James M Decker, Fei Wang, Andrew A Johnson, Brian K Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. Autograph: Imperative-style coding with graph-based performance. *arXiv preprint arXiv:1810.08061*, 2018.

[31] T. Moreau, T. Chen, L. Vega, J. Roesch, L. Zheng, E. Yan, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro*, pages 1–1, 2019.

[32] Joel Moses. The function of function in lisp or why the funarg problem should be called the environment problem. *SIGSAM Bull.*, (15):13–27, July 1970.

[33] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[34] PyTorch Team. Quantization in glow. https://github.com/pytorch/glow/blob/master/docs/Quantization.md, 2019.

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[36] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[37] Sean Robertson. Generating names with a character-level rnn. https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html, 2017.

[38] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018.

[39] Erik Sandewall. A proposed solution to the funarg problem. *SIGSAM Bull.*, (17):29–42, January 1971.

[40] Daniel Selsam, Percy Liang, and David L. Dill. Developing bug-free machine learning systems with formal mathematics. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of*

Machine Learning Research, pages 3047–3056, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[41] Asim Shankar and Wolff Dobson. Eager execution: An imperative, define-by-run interface to tensorflow. https://ai.googleblog.com/2017/10/eager-execution-imperative-define-by.html, October 2017.

[42] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[43] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 74–85, Piscataway, NJ, USA, 2017. IEEE Press.

[44] Bjarne Stroustrup. Abstraction and the C++ machine model. In *Embedded Software and Systems, First International Conference, ICESS 2004, Hangzhou, China, December 9-10, 2004, Revised Selected Papers*, 2004.

[45] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[46] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015.

[47] TensorFlow Team. Announcing tensorflow lite. https://developers.googleblog.com/2017/11/announcing-tensorflow-lite.html, November 2017.

[48] TensorFlow Team. Swift for tensorflow. https://www.tensorflow.org/community/swift, 2018.

[49] Peter Thiemann and Dirk Dussart. Partial evaluation for higher-order languages with state. 1996.

[50] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

[51] Torch Team. Torchscript documentation. https://pytorch.org/docs/stable/jit.html, 2018.

[52] Bart van Merrienboer, Dan Moldovan, and Alexander Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6256–6265. Curran Associates, Inc., 2018.

[53] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.

[54] Fei Wang, Xilun Wu, Grégory M. Essertel, James M. Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *CoRR*, abs/1803.10228, 2018.

[55] XLA Team. Xla - tensorflow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html, March 2017.

[56] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *CoRR*, abs/1708.02709, 2017.