PRINCIPLED OPTIMIZATION OF DYNAMIC NEURAL NETWORKS

JARED ROESCH

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Zachary Tatlock, Chair

Luis Ceze

Carlos Guestrin

Program Authorized to Offer Degree:

Computer Science & Engineering

UNIVERSITY OF WASHINGTON

ABSTRACT

PRINCIPLED OPTIMIZATION OF DYNAMIC NEURAL NETWORKS

Jared Roesch

Chairs of the Supervisory Committee:
Associate Professor Zachary Tatlock
Computer Science & Engineering

In the past decade *deep learning* has revolutionized many areas of computer science. Deep learning computations mostly consist of expensive linear algebra kernels defined over a mixture of large sparse and dense tensors. From the early days of deep learning framework development, researchers realized the potential for applying compiler optimizations to accelerate neural networks. As deep learning continues to grow in popularity the diversity of models also continues to grow. Due to the early success of deep learning in computer vision, early deep learning systems were were focused on static, feed-forward networks processing fixed sized images. First-generation deep learning compilers have also been similarly overfit for static model compilation, with strong assumptions of static control-flow, static tensor dimensions and no complex data structures. A focus on static models has created challenges for deep learning practitioners, as dynamic models introduce input-dependent graph topology, violating key invariants of existing systems and invalidating optimizations designed for purely static data flow graphs. This lack of support has manifested as series of ad-hoc extensions to both frameworks, deep learning runtimes and compilers. Choosing to ignore dynamic behaviors has allowed deep learning compilers to make significant strides in optimizing common deep learning workloads, but existing techniques miss increasing generality without sacrificing performance.

This dissertation in particular focuses on an under served, yet important problem: the representation, optimization, differentiation, and execution of *dynamic neural networks*. In this thesis I propose generalizing overspecialized compilation techniques applied to static dataflow graphs, the predominant programming model of deep learning, to fully dynamic neural networks. These generalizations are powered by a simple insight: dynamic neural networks are just programs which manipulate tensors. The challenge is constructing a representation that captures this generality in a principled manner, while not sacrificing state-of-the-art performance or the programming model. In particular, the contributions include:

- An intermediate representation which can represent dynamic behaviors.
- A new automatic differentiation technique for dynamic neural networks.
- A set of general optimizations which work on all programs, as well as specialized dynamic optimizations.
- An efficient runtime for dynamic neural networks.

The efforts of my thesis now exists in Apache TVM, a deep learning compiler framework. Apache TVM is deployed in production at multiple leading companies including Amazon, Facebook, and Microsoft and is a critical piece of the technology stack at OctoML: a company I co-founded around the TVM project. One notable impact is its use in Amazon Alexa, Amazon's AI assistant which executes on a variety of devices such as "smart speakers" which include digital assistants. Amazon engineers used Relay to optimize Alexa's wake word model, executed each time a user interacts with Alexa.

Contents

Introduction 13

Background 21

Relay: An IR for deep learning 33

Automatic Differentiation 51

Optimizations 59

Executing Relay 75

Future Work 91

Bibliography 93

List of Figures

3.1 The BNF Grammar for the Relay language expressions and types. 36

3.2 A simple TensorFlow loop in the user-facing DSL. 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. 39

3.3 Examples of Relay's typing inference rules, namely the rules for function definitions and function calls, where Δ is the environment for types and Γ is the environment for variables. These demonstrate that type relations must hold at each call site. 40

3.4 Inference slowdown of popular frameworks relative to Relay on vision benchmarks running on NVIDIA GTX 1080 Ti GPUs. Relay provides performance competitive to the state of the art. We ran 1000 trials for each model and used the AoT compiler. 48

3.5 Inference slowdown relative to Relay on NLP benchmarks running on NVIDIA Titan-V GPUs. NLP workloads feature control flow, which makes them more challenging to optimize. Relay provides performance competitive to state of the art (up to 2.4× speedup over MxNet on GRU). We ran 1000 trials for each model, except for CharRNN, on which we used 100 trials. 48

3.6 Inference time (ms) of vision DNNs on Ultra-96 FPGA-enabled SoC. We compare vision workloads that Relay compiles onto the embedded Cortex A53 CPU vs. a DNN accelerator implemented on the integrated FPGA fabric. Targeting DNN accelerators can unlock up to 11x speedups, but requires a multitude of graph-level transformations. We used 10 trials for each model. 49

4.1 Transformation Rules for Automatic Differentiation in Relay. The most interesting case is for function calls. The backpropagator Δ is initialized to `ref(fn() { () })` at the top level of each ADTerm call. Successive update closures δ are then composed with Δ to form a chain. Syntactic sugar is used for some constructs which are not available as primitives in Relay. 55

4.2 Example of running the compiler pass pipeline for AD on the identity function. First, we run the base AD pass on the original function (described in Section 4.0.2). Then, we run the partial evaluator, which primarily optimizes away the reads and calls in `%x2` and `%x3` in post-AD. Since it conservatively determines whether a subexpression is effectful, it generates many bindings which are dead code. At this point, we run the dead code elimination pass to crunch the code back down. 56

5.1 The top graph represents the dataflow graph of operators after annotation, and the bottom graph represents the transformed graph. SimQ simulates the rounding error and saturating error of quantizing. Its argument will get tuned during calibration. 62
5.2 The simulated quantization operation. 62
5.3 An example of overloading the annotation function for 2-d convolution. In this example we treat both input, and the weights as unsigned integers, applying rounding to the input, and stochastic rounding to the weights. 64
5.4 Some heterogeneous device placement rules. (a) The inputs and outputs of shape functions are placed on CPU. (b) `device_copy` changes the device of output accordingly. (c) The device of all arguments to `invoke_mut` must be the same. 70

6.1 An excerpt from the VM dispatch loop. 82
6.2 Relative latency comparison between symbolic codegen and static codegen of 3 dense operators on ARM CPU. The latency of kernel compiled with static shapes is used as the baseline. "dispatch/k" indicates that we generate k symbolic kernels to be dispatched at runtime. "no dispatch" means that only one symbolic kernel is generated and therefore no dispatching is needed. 88

List of Tables

5.1 This table shows the accuracy of various quantized models. `float32` refers to the non-quantized model. Notation as in "8/16" refers to 8-bit quantization, 16-bit accumulation 64

6.1 The opcode and the description of the Relay instruction set 81
6.2 LSTM model inference latency of Relay, PyTorch (PT), MXNet (MX), and TensorFlow (TF) on Intel CPU, Nvidia (NV) GPU, and ARM CPU. 87
6.3 Tree-LSTM model inference latency on Intel CPU and ARM CPU. TensorFlow Fold was not built successfully on ARM CPU. 87
6.4 BERT model inference latency on Intel CPU, Nvidia GPU, and ARM CPU. 88

0.1 Acknowledgements

As I was writing my thesis I spent a lot of time thinking about all the people who helped me along the way and feeling an immense amount of gratitude. During this process it seems silly that we write such short preface to such a long document when the people, their time, energy, thoughtfulness, friendship, and roles seem so much larger than the things that ended up in this document. I will inevitably forget to thank some people, if you feel that you helped me you probably did; here it goes.

First of all thanks to my family, Kevin, Adrienne, and Ian who have always encouraged me to be my own thinker and let me do it differently even if they don't always fully understand my intensity and insanity. To my undergraduate mentors without whom I would probably be an unemployed jazz musician somewhere. Murat Karaoraman who believed in me from day one even though I didn't have much to show yet. If you hadn't bet on my potential I wouldn't be a computer scientist today. To Jay Freeman, who showed me what a true hacker is. To Ben Hardekopf and Tim Sherwood who taught me how to do research and seek out interesting questions with a PL spin. To my UCSB labmates who helped me grow into a researcher and provided inspiration, wisdom, and friendship. To Joseph who has remained a close friend after failing to convince me to go to the hospital during a deadline push, in the end I only lost an Appendix. To the UCSB crew, Pete, Tristan, Andrew, Chang, and more who helped me grow as a hacker, and not settle for less. To Daniel for staying up at all hours of the night talking about research, startups, or whatever stupid argument we had that week.

To UW PLSE, an amazing second home and group of people to go through graduate school. The lab crew, and our times in the PLSE lab playing loud music, pranks, games, and banter. Everyone was a friend and important part of my graduate school experience but specifically the people who were my closest friends (and sometimes roommates) during this time Max Willsey, Chandrakana Nandi, Sam Elliot, Luke Nelson, Shumo Chu, Chenglong Wang, and Luis Vega. Most importantly to all the amazing people I got to work with as collaborators across all my projects. Especially the Relay crew of Steven Lyubomirsky, Josh Pollock, Logan Weber, Marisa Kirisame, and Altan Haan.

To all my partners and friends during this time, who tolerated my obsession with work and shipping just one more 4k PR. To Pavel Panchekha who told me "doing verification sucks", I eventually got the message a few years later. To my co-founders at OctoML who brought me along for the ride even though I wasn't quite ready

yet. Thierry for grinding through the end of PhD together, fighting fires and "growing up" together. Tianqi for diverting my life into an amazingly fun and different direction thanks to a random email. Jason, for reminding me why betting on people is worth doing. Luis for having a grand vision and pushing us to dream bigger. Thanks to Amanda Robles for having our backs both in CSE and OctoML, you helped me numerous times even during the writing of this last section of my thesis. To all my co-workers at OctoML its been amazing to build a company with you. I appreciate the opportunity to learn and grow with such great people, and appreciate you taking a bet on us co-founders.

To Leo De Moura who was an amazing mentor and friend for the start of my PhD and taught me things about research, life, and writing lots of code. To Zachary L. Tatlock my devious advisor who was always there to pull the chair out from under me both literally and figuratively while laughing about it. I learned how to run a group, mentor, invest in people, attempt balance in a busy life, and do big things. To Xi Wang who was my co-advisor for the early part of my PhD and always someone who had wisdom and advice for me. I learned the importance of not being dogmatic in your approach or tools, and giving up early. Finally to Eunice who has put up with me as I have struggled to have a social life, relationship, found and run a startup, while finishing this little thing called a PhD on the side. Thanks for being my late night editor, sounding board, and partner in crime. You were always the taste tester for my real job of cooking way too often in the past few years. This document should really be titled the PhD Pigout, as I probably cooked and ate more then I did research. Finally the most important star of the show that last 6 years: coffee. This PhD wouldn't of been possible without the copious amount of coffee, and hours programming at coffee shops. I think one place in particular may of been the location of nearly 100,000 lines of Coq proofs early in my PhD.

I will always remember everyone named and unnamed as part of this awesome journey, thanks again to all for being in my life and helping me achieve my goals.

Introduction

Traditionally, machine learning algorithms were carefully engineered by humans in order to leverage limited data to obtain sufficient task performance. During the Big Data era, spanning the better part of the last two decades, more complex algorithms were displaced by simpler algorithms applied to the abundantly available data [Halevy et al., 2009][1]. In order to scale these algorithms a number of frameworks were developed, most notably streaming and graph processing systems such as MapReduce and Spark. In the early 2010s the emergence of deep learning changed the paradigm again replacing simple regular computation over streams with large batched-models composed of compute-hungry tensor computations. The paradigm shift towards deep learning has started a trend of deploying models which are applied to more complex data (i.e., full text documents, images, videos, audio), process a greater quantity of data (millions of data points), and are more computationally and memory expensive to evaluate on a single datum (i.e., terra-flops of compute per sample).

[1] "We don't have better algorithms. We just have more data." From "The Unreasonable Effectiveness of Data"

This new generation of machine learning applications demanded a corresponding era of innovation in systems. The first innovation in deep learning systems were "deep learning frameworks" such as TensorFlow and PyTorch. These monolithic systems can execute a subset of deep learning models efficiently on a subset of the most popular hardware devices. These systems provide good baseline performance, but often fail when straying from popular mainstream models, and only support limited deployment strategies. In particular TensorFlow's design was optimized for the distributed setting but provides a sub-optimal experience on a single node[2].

[2] In fact, Google engineers told me they would jokingly refer to TensorFlow as TensorSlow in the early days.

Scaling frameworks to span the scope of all models and hardware targets is a monumental endeavor requiring deep expertise often not found in conjunction with the skills of a machine learning engineer. Even with the aid of a machine learning framework a machine learning engineer cannot write code that scales across the growing sea of models, operators, optimizations, and devices while maintaining state-of-the-art performance. Frameworks only support a subset of programs, optimizations, and devices requiring large teams to extend

frameworks for even relatively simple deviations from standard models. Even when a team is able to get a model running, the resulting performance often is lacking relative to the performance of more popular workloads. Large companies are able to mitigate this cost with raw capital expenditure but this puts smaller organizations and researchers on unequal footing. In the years after the introduction of frameworks it has become clear that automation is needed to both help smaller groups of engineers more effectively develop and deploy deep learning, and allow larger organizations to work effectively and efficiently.

From the early days of deep learning framework development researchers realized the potential for apply compiler optimizations to accelerate neural networks. [3] Modern frameworks borrowing ideas from the research community began to introduce compilers into frameworks. Deep learning compilers, systems for accelerating the execution deep learning via compilation, have made similar tradeoffs as frameworks by narrowly focusing on executing a subset currently popular of models on a subset of devices. Designing a deep learning compiler even for this constrained subset is challenging. For example, a popular state-of-the-art DL compiler, Google's XLA, can famously slow down programs on CPU instead of speeding them up. Although non-intuitive, this is a general challenge of applying compilation: it is not a silver bullet and may not provide uniform speedups across all input programs. Compilers for deep learning [XLA Team, 2017, LLC, 2018, Rotem et al., 2018, Chen et al., 2018b, Breuleux and van Merriënboer, 2017, Innes, 2018b, Lattner et al., 2020] are a rapidly evolving area explored by both industrial and academic researchers.

[3] Theano was applying compilation to deep learning as early as 2007.

Deep learning compilers have been narrowly focused on the handling of static, feed-forward neural networks. Due to this design first-generation compilers have been overfit for static model compilation, with strong assumptions of static control-flow, static tensor dimensions and no complex data structures. Specifically, these models are represented as static data flow graphs where the size of each input and output (i.e. tensors or n-dimensional arrays) are known a priori, ensuring the execution path remains unchanged on every invocation. We refer to models with such fixed runtime behavior as *static models*. Continued advances in neural networks, especially those in natural language processing, have introduced new dynamism in models, such as control flow [Hochreiter and Schmidhuber, 1997, Sutskever et al., 2014], dynamic data structures [Tai et al., 2015, Liang et al., 2016], and dynamic shapes [Devlin et al., 2018]. We refer to models exhibiting these behaviors as *dynamic models*. At the time of their design, deep learning was revolutionizing computer vision but had not yet changed areas such as natural language processing (NLP).

Unfortunately the design of frameworks have severely limited the performance of programs that fall outside of the well-optimized static footprint.

This lack of support has manifested as series of ad-hoc extensions to both deep learning runtimes, and ML compilers. Many existing approaches to dynamic model optimization apply or extend existing deep learning frameworks [Xu et al., 2018, Gao et al., 2018, Yu et al., 2018, Jeong et al., 2018, 2019, Neubig et al., Looks et al., 2017b]. Existing work which builds on frameworks extends the programming model either via sophisticated additions [Yu et al., 2018] leading to designs that only work for a single framework or significant runtime overhead [Looks et al., 2017b, Jeong et al., 2019].

In this thesis I propose that overspecialized compilation techniques applied to static dataflow graphs, the predominant programming model of deep learning, can be generalized to fully dynamic neural networks. These generalizations are powered by a simple insight: dynamic neural networks are just programs which manipulate tensors. In particular I have spent the last several years focused on an under explored problem: the representation, optimization, differentiation, and execution of *dynamic neural networks*. The challenge is constructing a representation that captures this generality in a principled manner, enabling state-of-the-art performance without limiting the programming model. We realize a unified interface in the form of Relay: a set of systems and APIs designed to compose techniques across the stack to achieve state of the art performance.

Instead of building a single IR and compiler to rule them all[4] we carefully split the program optimization problem into a series of phases each focused on a specific optimization task. This approach balances the tension between expressivity, composition and performance portability. Part of my thesis work in graduate school lead to significant contributions to the general design and implementation of production quality deep learning compilers. Much of this work now exists in Apache TVM, a deep learning compiler framework. Relay and our work on dynamic neural networks was merged into Apache TVM and has catalyzed a major redesign of TVM, in order to embrace and extend the ideas of this thesis. Relay is deployed in production at multiple leading companies including Amazon, Facebook, and Microsoft and is a critical piece of the technology stack at OctoML a company I co-founded around the TVM project. One notable impact is its use in Amazon Alexa, Amazon's AI assistant which executes on a variety of devices such as "smart speakers". Amazon engineers used Relay in particular to optimize Alexa's wake word model, which is executed each time a user interacts with Alexa.

[4] and in the darkness bind them

1.1 State of the Art

Popular DL compiler intermediate representations (IRs) offer different tradeoffs between expressivity, composability, and portability [Abadi et al., 2016, Paszke et al., 2017, Tokui et al., 2015, van Merrienboer et al., 2018, Bergstra et al., 2010, Rotem et al., 2018]. 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 [Looks et al., 2017b, TensorFlow Team, 2017, van Merrienboer et al., 2018, Shankar and Dobson, 2017, XLA Team, 2017, Rotem et al., 2018, Torch Team, 2018]. 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. Furthermore extensions which introduce high-level semantics such as control-flow, data structures, or data dependent kernels are not explicitly represented by deep learning compiler IRs and are hard or impossible to effectively optimize.

Let us consider a hypothetical scenario that exemplifies IR design tensions in deep learning 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 [Tai et al., 2015]. Unfortunately, current frameworks' IRs cannot directly encode trees, so she must use a framework extension like TensorFlow Fold [Looks et al., 2017a].

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 [Gustafson, 2015, Google, 2019, PyTorch Team, 2019]. 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 [Apple, 2017].

Deep learning frameworks only supported a limited subset of these flows and compilers an even smaller subset. The critical challenge inhibiting optimization via compilation was an insufficiently expressive intermediate representation. Compilers must be able to internalize the constructs that they intended to optimize. Although many frameworks provide end-users with general, turing complete programming models the IRs used to lower them to specific backends can be extremely limited. Originally these IRs were designed for lowering a directed-acyclic graph of operations to high-level libraries provided by device vendors. Incrementally adapting these IRs to support a richer programming model has introduced a semantic gap between the source and target, often leading to complex or incomplete compilation processes. Due to this semantic gap many approaches focused on removing dynamic features in order to hang on to a simple static compilation model. For example TorchScript admits arbitrary Python code in the IR, code which can not be desugared into CUDA operations without employing a full Python to GPU compiler. TensorFlow makes use of Python to do meta-programming with graph fragments, using the outer language as a staged-DSL.

One view on the work described in this thesis is evolving IR and runtime mechanisms to better match the source language users are programming in. Existing compilers have maintained their current IRs as it is difficult to adapt domain specific transformations to an enriched IR. There are challenges around many important transformations and optimizations in a richer IR such as efficient automatic differentiation, optimizations around reduced or absent static information such as shape sizes, and code generation and execution around dynamic features such as control flow, function calls, or allocating and manipulating data structures. My thesis applies the simple observation that tensor languages can be transformed, compiled, optimized and executed just as we would with traditional programming languages. The central contribution is a methodology for not only growing the IR but also adapting domain specific innovations from the computation graph domain to a full programming language. We can do this by building on ideas from decades of previous work in compilation and previous trends in compilers in order to solve this problem for an enriched language.

1.2 Dynamic Neural Networks

In this thesis we explore the four key aspects of compiling dynamic neural networks: their representation, optimization, differentiation, and execution. We do this by building an enriched compiler framework based on TVM by introducing a new representation Relay which increases expressivity to handle the features we introduced in this section. We define a new approach to higher-order, higher-order automatic differentiation which is able to handle computing derivatives of the complete IR and can be uniformly optimized as a Relay program. We further extend this compositional framework for compilation to handle dynamism and are able to show state-of-the-art performance across a variety of devices.

In particular this thesis explores growing compiler support from a limited subset of programs to a more general programming model. There are multiple analogies to be made to historical compilation. For example early Fortran compilers did not support recursion as the compiler statically allocated all activation records[5]. It required evolving the compiler, IR, runtime data structures and even hardware to support new features. Another analogous process was adapting ideas from traditional compilation to dynamic languages such as Smalltalk and Self in the 80s and 90s. These pushes grew existing compilation techniques to support new features and programming paradigms. Ahead-of-time compilation based on static type information and layout could not be straightforwardly applied to dynamic languages, leading to generation of just-in-time compilation advances. This is strongly paralleled to the continued development of deep learning compilers. This first generation of compilers have not yet captured the spectrum of user programs, requiring us to build new systems which grow techniques to apply to the new domain-specific deep learning challenges.

[5] Fortran recursion notes, 2020. URL http://www.ibiblio.org/pub/languages/fortran/ch1-12.html

Most existing representations have been organically adapted from computation graphs a useful but simplistic IR that organically evolved from AD and ML literature. A representation must capture concepts like control flow, iteration or recursive operations like fold or scan, allocation, device placement, scoping, effects, function calls, data structures, and shape information. Previous IRs solve a subset of these problems and often only in partial or restricted domains.

Even if we have such a representation there are challenges specific to machine learning: we must be able to differentiate the programs. Deep learning has often relied on backpropagation or simple AD algorithms defined over computation graphs. The lack of support for a richer programming model has lead to numerous attempts to define both what is differentiable and how to differentiate it as we

discuss in Chapter 2.

Even if these problems are solved there is still the final hurdle of being able to optimize and execute the code on target platforms. Traditional compiler optimizations are focused on low-level optimizations for traditional scalar, general purpose architectures. For example LLVM is not focused on optimizing aggregate allocations, or is away of accelerate memory semantics, scratch pads, or explicit cache management. Finally once programs are optimized we must be able to provide the correct runtime support that is efficient and cross-platform.

1.3 Thesis Organization

This thesis is organized around four pillars: representation, differentiation, optimization, and execution. We first explore related work and background material in Chapter 2. We discuss the design of the Relay intermediate representation in Chapter 3. We then demonstrate how automatic differentiation and training support can be adapted to Relay in Chapter 4. We present numerous program optimization in Chapter 5, followed by how to lower, execute, and compile these programs in Chapter 6. Finally we look to the future in Chapter 7.

Background

The acceleration of deep learning is an active topic of research and is cross-disciplinary by nature. Much of this research happens around, inside of, or on top of dominant deep learning frameworks such as 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). Below we discuss various related topics necessary for understanding the work presented in this thesis, including the frameworks themselves, their programming model, compilation, execution and deployment.

2.0.1 Deep Learning

Deep learning (DL) has been used to achieve state-of-the-art results in applications ranging from computer vision to game playing to natural language processing. Deep learning provides a collection of techniques for learning functions that are difficult to program directly. For example, suppose one wants to construct a function $f(x)$ to extract building addresses from images of a street. One approach to writing $f(x)$ explicitly might be to write programs to identify regions of the image that contain information pertinent to addresses (like street names or house numbers), partition those regions into individual characters, and identify each character. One may then write a program that chains together many such modules into a working system. It would require hundreds, if not thousands, of human-hours of work and dozens of heuristics to write any of those programs by hand, and there is no guarantee it will perform well. By contrast, deep learning has, with relatively little code, been used to approximate this entire recognition task with a single model. The learned system outperforms all hand-crafted solutions [Goodfellow et al., 2013]. In practice, DL engineers write procedural programs that manipulate *tensors*, multi-dimensional arrays of numbers. These restrictions allow DL practitioners to take advantage of statistics, linear algebra, and real analysis. A programmer solving a problem with DL performs three steps. First, she specifies a parametric function

(often called a *model*, *neural network*, or simply *network*) $F(\theta, x)$ of the function that computes the desired value. She then *trains* the model by applying an optimization algorithm to find a set of parameters θ that results in an acceptable approximation of the function on a collection of input-output pairs. In order to search for an assignment of parameters that produces a good approximation, we must first define what a "good" approximation of f is and then find an algorithm that optimizes this metric. The quality of an approximation is typically defined in terms of some ground truth to compare against $F(\theta, x)$. This ground truth is usually provided in the form of a *training set*, a set of input-output pairs that has either been manually collected or extracted from an existing data set. The function that scores the output of a network with respect to a ground-truth label is called the *loss function*,

$$L\colon (\text{input}, \text{output}) \to \text{network} \to \mathbb{R}^{\geq 0},$$

which typically takes in an input-output pair and network and produces a non-negative real number.

One method to optimize the network is to take the gradient of the loss function composed with the candidate network for some input-output pair. The gradient of a function points in the direction of steepest change, so subtracting some proportion of the gradient from the current parameters reduces the loss of the network. This process is known as *stochastic gradient descent (SGD)*, and it is the basis for many deep learning optimization algorithms used today.

More formally, training updates the model parameters using the following algorithm:

$$\theta_{i+1} := \theta_i - \varepsilon \nabla [L((\text{input}, \text{output})) \circ F]\Big|_{\theta_i}$$

for some small, positive ε. This update cycle is repeated in training until the error is acceptable. The final values of the parameters are then used for inference. Finally, she uses her learned function to produce outputs for new inputs in a process called *inference*.

While training and inference were once performed on the same machine, it is more common today to train models on a fleet of high-powered devices, since training is very computationally intensive. The values learned from training can then be deployed for inference on less powerful systems, such as a single GPU, a mobile phone, or an FPGA.

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. Frameworks such as Theano and Torch introduced a model of computation graphs, dataflow graphs composed of nodes of variables, and operations on data. These computation graphs represent the AST of a small embedded DSL that could be easily compiled and deployed to platforms which implement the correct operators. Operators, also called kernels, are dense linear algebra primitives like matrix multiplication, element-wise functions like `tanh`, and complex, domain-specific operations like image convolution. Operator execution dominates the execution time of a deep learning model: many operators are asymptotically slow and their input is large.

In order to accelerate model execution, frameworks supporting accelerators such as GPU were introduced [Bergstra et al., 2010] . 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 [Abadi et al., 2016], PyTorch [Developers, 2018], MxNet [Chen et al., 2015]), compilers(XLA [XLA Team, 2017], Glow [Rotem et al., 2018], TVM [Chen et al., 2018b]), and hardware accelerators (TPU [Jouppi et al., 2017], GraphCore, Inferentia [Amazon Web Services, 2018]). 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.

As a framework, TensorFlow [Abadi et al., 2016] supports dynamism via the addition of control flow primitives such as *switch* and *merge* in its graph representation [Yu et al., 2018]. Similarly, MXNet [Chen et al., 2015, Zheng, 2018] uses operators such as *foreach*, *cond*, and *while_loop* to support control flow. The main drawback of this approach is requiring a runtime that uses an inefficient and complex control flow encoding such as TesorFlow, or a hybrid runtime such as MxNet which is inflexible for deploying to edge devices. As a separate effort, TensorFlow Fold [Looks et al., 2017a] adds front-end syntactic sugar to TensorFlow for simplifying dynamic models via dynamic batching. Specifically, TensorFlow Fold conducts an analysis of the user's provided computation graphs and identifies dynamic operations that can be batched together. Once such operations are

found, it transforms them into an intermediate representation (IR) that can be ingested by TensorFlow for evaluation.

The nature of this approach can introduce large overhead as each path must be executed as a different sub-computation graph, as well as limiting further optimizaiton.

Dynet [Neubig et al., 2017] and PyTorch [Developers, 2018] use host language features to dynamically (i.e., Python's control flow) to construct a dynamic model architecture. However, it requires the creation of new static data flow graph for each path through the program introducing control flow overhead, and limiting the scope of optimization to a single trace through the program, a trace which often only executed once [Xu et al., 2018]. These challenges were similarly faced by tracing-JIT based systems in the JIT compiler literature. JAX [Bradbury et al., 2018] also supports both control flow and dynamic features in its programming model, but is fundamentally limited by the ability of XLA, TensorFlow's deep learning compiler to optimize and compile dynamic code. To the best of our knowledge XLA still requires all loop nests to be static at code generation time. JANUS [Jeong et al., 2019] optimizes the dynamic model execution in TensorFlow via speculative execution. It converts an imperative program that contains control flow defined in the host language (Python) into aTensorFlow graph for better performance. However, the conversion is incomplete due to certain dynamic features such as typing, control flow, not being fully supported by the runtime. In these cases Janus falls back to imperative execution in the host environment when execution assumptions are violated leading to overheads in these cases. It works well when models have portions that can be executed speculatively successfully but is not ideal in truly dynamic cases. Furthermore because deep learning frameworks rely on third-party libraries to implement operators with different data shapes, JANUS and other framework based techniques still don't solve the portability or performance portability challenges. For example if a model has a dynamically shaped operator which is not implemented by the corresponding high-performance third-party library the Janus nor the framework can help. Therefore, framework based techniques generally perform poorly on devices, if at all, on devices such as ARM CPU, which are not in the first tier of device support such as Google's TPU or Nvidia GPU in TensorFlow.

2.2 *Define-And-Run Frameworks*

TensorFlow, Caffe [Jia et al., 2014], and Theano [Bergstra et al., 2010] 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 [Baydin et al., 2015, Abadi et al., 2016]. 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 [Looks et al., 2017a] but offers no general and extensible solution. The encoding also requires *ad hoc*, special-purpose operators such as `NextIteration` and the addition of special control edges to the graph. 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. TensorFlow's programing model is relatively restricted and has led to a number of solutions including TF Eager, AutoGraph, and JAX. Modifying frameworks in this manner is a considerable engineering effort and does not scale.

2.3 *Define-By-Run Frameworks*

PyTorch [Paszke et al., 2017], Gluon [Gluon Team, 2018], Chainer [Tokui et al., 2015], and TensorFlow eager-mode [Shankar and Dobson, 2017] 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.

2.4 *Low-Level Tensor Compilers*

Low-level tensor compilers are focused on the production of high-performance operators which implement compute-intensive oper-

ations such as matrix multiplication or convolution. There are a number of competing approaches, both from academic and commercial entities, such as TVM [Chen et al., 2018b], Halide [Ragan-Kelley et al., 2013], Tensor Comprehensions(TC) [Vasilache et al., 2018], and Diesel [Elango et al., 2018]. 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.

2.5 *Deep Learning Compilers*

DL frameworks have adopted compilers to tackle both performance and portability for existing applications, most notably XLA [XLA Team, 2017], Glow [Rotem et al., 2018], nGraph [Cyphers et al., 2018], ONNC [Lin et al., 2019], PlaidML [Corporation, 2017], 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 similarly uses Halide to represent low-level operations, which provides fusion and other standard optimizations. Their high-level IR is not clearly described but appears to lack arbitrary control-flow, first-class functions, or datatypes. They employ quantization but do not demonstrate how/if it can be extended, or if low-bit (< 8) quantization is supported. They have shown support for CPU execution but provide no story for GPUs or accelerators.

Unlike the deep learning frameworks, Relay is designed as an end-to-end deep learning compiler. The existing deep learning compilers, including XLA [XLA Team, 2017], TVM [Chen et al., 2018b], and Glow [Rotem et al., 2018], offer a means to compile deep learning models to run on multiple hardware platforms, but only focus on static models and fail to support models with dynamism. MLIR [Lattner et al., 2020] provides compiler infrastructure for deep learning workloads which allows developers to implement new dialects and combine them into a sing application. Its graph-level dialect has the support for dynamism, but has not yet produced work demonstrating how to achieve good performance in dynamic scenarios. Relay enables a deep learning compiler to support dynamism in an efficient and flexible way by adding a dynamic type system, a dynamic code generator, a VM-based runtime and a series of dynamic-specific optimizations.

Relay's compilation and VM design is largely inspired by production compilers and VMs, such as LLVM [Lattner and Adve, 2004], GCC [Developers, 2019], and JVM [Oracle, 2013]. These generic solutions are able to easily handle dynamic behaviors, uch as control flow and variable-length input arrays. However, the design of these compilers are heavily tailored t o the optimization and execution profile of traditional programs. These programs manipulate small scalar values and consist of a large number of low level instructions. These VMs have a large number of instructions, which requires instruction execution and dispatch to be extremely efficient.

Instead, deep learning compilers like Relay manipulate primarily tensor values using a relatively small number of coarse-grained instructions, i.e. 16 instructions are sufficient for the Relay VM. The execution hotspots of DL workloads are the compute-intensive operators (i.e. convolution) over large tensors. Thus, the micro-optimization of VM instruction dispatch is not essential to overall

performance.

2.5.1 *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 [Wang et al., 2018], Lift [Steuwer et al., 2017], Flux.jl [Innes, 2018b] AutoGraph [Moldovan et al., 2018], Swift for TensorFlow [TensorFlow Team, 2018], and JAX [LLC, 2018]. Lantern [Wang et al., 2018] 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.

2.6 *Hardware Backends*

Even if a model can execute on a particular hardware device, developers often manually design model variants for each target platform to achieve the best performance. If a programmer wants to experiment with a new hardware device, she must manually account for variations in hardware intrinsics, data types, and data layout. This is only possible if the device is supported by her framework of choice. Even with the addition of device-specific code, there is no guarantee performance will be acceptable, let alone optimal (or even that there will not be regression). Engineers design these platform specific model variants through tedious experimentation and framework-specific tweaks to achieve acceptable performance [Howard et al., 2017]. For example, engineers may need to *quantize* models by manually reduc-

ing numerical precision for a particular platform, sacrificing some accuracy for better performance [Rastegari et al., 2016]. Manually tuning models requires that engineers learn the details of each platform's unique data types, intrinsics, and memory systems [Johnson, Jouppi et al., 2017, Fowers et al., 2018, Tu, 2018].

The landscape of specialized hardware accelerators is also rapidly growing [Moreau et al., 2019, UCSB ArchLab, Jouppi et al., 2017]. Accelerators provide an attractive way to decrease the execution time, memory and power consumption of models, as well as enable deep learning in new contexts. Many deployment targets, such as mobile phones, are a heterogenous system consisting of a CPU, GPU, and customized machine learning accelerator. Automatically targeting accelerators requires sophisticated compilers which must perform a variety of optimizations and schedule computation across heterogenous devices. Hardware vendors need software stacks, often extending a deep learning compiler with support for their devices in the form of operations, optimizations, and datatypes. It is essential that frameworks can adapt to new hardware devices with minimal changes to applications. Relay provides an extensible framework for hardware accelerators, and implements a backend for VTA an emerging hardware accelerator [Moreau et al., 2019]. We demonstrate how we can use an expressive general-purpose deep learning IR, plus a few platform specific optimizations

2.6.1 High Performance DSLs for Tensors

My work is focused on the end-to-end compilation of deep learning, and for this reason we do not directly focus on the generation of efficient low-level code for tensor operations. Instead my work is focused on optimizing programs which make use of low-level operations and optimizations around programs which combine them with higher-level abstractions, like closures and datatypes. In particular Relay is designed as an extension to TVM [Chen et al., 2018b], a tensor expression language used for specifying dense array operations, to support full models. Previous work on the TVM focused on producing efficient operators (dense linear algebra kernels), such as generalized matrix multiplication (GEMM) and convolutions. In theory Relay could use other high performance DSLs such as Halide [Ragan-Kelley et al., 2013], which TVM derived from its IR, and its split computation and schedule model. Tensor Comprehensions (TC) shares common goals with the TVM framework, but achieves its goal through different techniques, such as polyhedral compilation rather than algorithmic schedules. TC could replace TVM in Relay's compilation.

2.6.2 *Programming Language based Deep Learning*

My work is not the first attempt to apply programming language techniques to machine learning or problems such automatic differentiation. Lantern [Wang et al., 2018] is a deep learning DSL in Scala that uses lightweight modular staging (LMS) to lower code into C++. LMS takes a graph as input from the user and converts it to an AST representation, similar to Relay's graph mode. LMS removes unnecessary abstractions. Going one step further than Lantern, Relay supports accelerators.

`Flux.jl` [Innes, 2018b] is a DL library written in Julia [JuliaLang Team, 2018], a numerical computing language. Like Relay, `Flux.jl` adds shapes to the type system; however, unlike Relay, it uses a mixture of compile-time inference and runtime checks to enforce shape constraints [Innes et al., 2017] and cannot perform platform-specific optimizations. Unlike `Flux.jl`, which tightly coupled with Julia, Relay is language agnostic and decouples frontend and IR considerations. Relay's features, such as type inference and deployment to multiple back-ends, can be easily reused by new frameworks written in arbitrary programming languages. Additionally, previous work in higher-order differentiation from the PL community has informed Relay's design. In particular, we draw inspiration from various implementations of automatic differentiation [Elliott, 2009, Baydin et al., 2015, Kmett et al., 2008, Pearlmutter and Siskind, 2008, Wang and Pothen, 2017, ThoughtWorks Inc., 2018b,a], with particular attention to techniques that can compute higher-order gradients of higher-order programs. `Zygote.jl` [Innes, 2018a], like Relay, uses source code transformations to implement automatic differentiation.

2.6.3 *Runtime systems*

To address frameworks' challenges with dynamism, e.g. prohibitively high overhead in reconstruction of computation graphs and difficulties in batching together computations with similar input shapes recent works has proposed deep learning runtime systems [Xu et al., 2018, Gao et al., 2018]. These techniques exhibited effectiveness in improving the performance of dynamic models. However, these approaches suffer from a common limitation–they all heavily rely on the third party libraries, such as cuDNN [Chetlur et al., 2014] and MKL-DNN [Intel, 2018], raising concerns about portability. In addition, they usually only support a narrow subset of dynamic models on each platform.

2.6.4 *Dynamic Neural Networks*

Systems for machine learning is an active research area with a variety of techniques for ML inference engines which support dynamic features, cross-platform execution, and per-platform code generation. We discuss related work below. One area in which deep learning has made significant advances is natural language processing (NLP), such as finding keywords in a research paper, determining the sentiment of a tweet, or summarizing a news article. Reasoning about text requires context-sensitive analysis and data of non-fixed dimensions, unlike in many vision tasks. To allow for context-sensitive analysis, DL researchers have developed networks with persistent state, known as *recurrent neural networks* (RNNs). Like the simple networks described earlier, these networks have an input and an output; however, they take an additional input and produce an additional output known as the *hidden state*. Beginning with an initial hidden state and a list of inputs (for example, characters from a source text), one can produce a list of outputs (for example, a translation of the source text). Recurrent neural networks have found use not only in NLP, but also in speech recognition, music transcription, eSports, and other areas [Hochreiter and Schmidhuber, 1997, Graves et al., 2013, OpenAI, 2018]. Unfortunately, since most machine learning frameworks rely on computation graphs, which cannot represent recursion, RNNs are usually finitely unrolled to a fixed depth. This may be acceptable if the depth can be determined statically and the loop unrolled ahead of time; however, if the depth depends on runtime values or complex control flow, unrolling must be performed dynamically.

Relay: An IR for deep learning

Popular DL compiler intermediate representations (IRs) offer different tradeoffs between expressivity, composability, and portability [Abadi et al., 2016, Paszke et al., 2017, Tokui et al., 2015, van Merrienboer et al., 2018, Bergstra et al., 2010, Rotem et al., 2018]. 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 [Looks et al., 2017b, TensorFlow Team, 2017, van Merrienboer et al., 2018, Shankar and Dobson, 2017, XLA Team, 2017, Rotem et al., 2018, Torch Team, 2018]. 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. Nearly all popular deep learning representations were designed for static computation graphs, leading to numerous extensions designed to support dynamic neural networks.

This thesis present Relay, a new compiler IR for deep learning. 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. Relay's extensible compiler can eliminate abstraction overhead and target new hardware platforms.

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. Relay's design takes inspiration from traditional compiler literature where many of the challenges facing machine learning compilers have been well studied in the scalar setting. We analyzed

previous deep learning IRs finding ways to obtain the desirable properties of these IRs with a principled approaches, for example using references to split pure and in-pure fragments, or the use of closures to represent complex control operators.

Relay is also designed to abstract over platform specific behaviors but not prevent representing or optimizing for them. Given a known target, a user can schedule a new optimization, and all necessary platform optimizations and code generation will occur. Target independence might seem like a property already enjoyed widely, but in many frameworks each operator is implemented per platform, and often models only work on a single well-supported platform (i.e Nvidia GPU). Previous IRs are either designed to be tethered to a specific end-user programming model or low-level operator library which enables the programs to be executed on specific platforms such as GPU. Leveraging these features leads to powerful use cases, for example we are able to easily get best in class performance on many devices by mixing and matching TVM, with native kernel libraries to obtain the best performance, without the end user needing to adapt their program in anyway. The rest of this chapter describes Relay's IR and type system design, and presents some preliminary performance results. We focus on the following contributions:

- The Relay IR, a tensor-oriented, statically typed functional IR, which we describe in this chapter. 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. Collections of *ad hoc* extensions in previous frameworks that patched shortcomings in expressiveness are subsumed by a handful of well-known language constructs like let expressions, ADTs, first-class functions, and references. In addition to improving expressivity, incorporating these features as language constructs allows optimizations to more readily compose.
- By representing DL models as functional programs, we reframe traditional deep learning framework features as compiler problems. Backpropagation becomes a source code transformation, transforming an arbitrary Relay function into its gradient function; *ad hoc* shape inference becomes principled type inference; graph rewriting becomes program optimization; and the executor becomes (depending on what the context demands) an interpreter, virtual machine, or ahead-of-time compiler. 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*

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

across hardware backends.

3.1 The Relay IR

The Relay IR is a high-level, functional, differentiable language. Relay 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 [Mainland et al., 2013]. 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 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. The Relay IR is designed to subsume the functionality of computation graph-based IRs while providing greater faculties for abstraction, and dynamism. We present Relay's design by incrementally building up to the full IR starting from a subset that corresponds to a simple computation graph.

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 3.1). 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 presentation, is a directed acyclic graph with multiple inputs and a single output. The syntax of an equivalent computation graph is realized by a language with three rules (1) `variables`, (2) function `calls`, and (3) `operators`, see Figure 3.1 for the corresponding rules.

3.1.1 Operators

Operators are the primitive computation of Relay and represented using an opaque function signature, and backed by fragments of TVM's TE or TIR which can be lowered to low-level device specific code to realize the abstract operations with specialized implementations.

Figure 3.1: The BNF Grammar for the Relay language expressions and types.

Expr	e	::=	`%`l (local var)
		|	`@`g (global variable)
		|	`const(`(r | b)`,`s`,`bt`)` (constant tensor)
		|	e`(`$\langle \tau$`, ..., `$\tau \rangle$`)?(`e`, ..., `e`)` (call)
		|	`let %`l `(:`τ`)? =` e`;` e (let)
		|	e`;` e (`let %_ =` e`;` e)
		|	`%graph =` e`;` e (graph let)
		|	`fn (`$\langle$`T, ..., T`$\rangle$`)?` `(x, ..., x) (`$\rightarrow \tau$`)?` `{`e`}` (function)
		|	`(`e`, ..., `e`)` (tuple formation)
		|	e`.`n (tuple proj.)
		|	`if (`e`) {`e`} else {`e`}` (if-else)
		|	`match (`e`) {` `|` $p \rightarrow e$ ⋮ `|` $p \rightarrow e$ `}` (pattern match)
		|	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` $\langle$`T, ..., T`$\rangle$ `(`τ`, ..., `τ`)` $\rightarrow \tau$ `(where `τ`, ..., `τ`)?` (function type)
		|	`Ref[`τ`]` (ref type)
		|	`(`τ`, ..., `τ`)` (tuple type)
		|	τ`[`τ`, ..., `τ`]` (type call)
		|	tn (type name)

3.1.2 *Multiple Outputs*

Many common operators like `split`, which splits a tensor along a particular axis, require multiple outputs. In order to handle these programs, computation graph IRs have added primitive support for multiple outputs. Multiple outputs can be modeled as tuples, which can be added with just two rules (1) `tuple formation` and (2) `tuple projection`. Instead of using multiple outputs as a builtin concept the use of tuples allow us to represent operations which not only return one level of data structure, but can also use nested structures such as tuples of tuples.

3.1.3 *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 [Moses, 1970, Sandewall, 1971]. The addition of a humble `let` binding, a central concept in functional languages, provides explicit sharing and a solution to the problems outlined above.

3.1.4 *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 [Looks et al., 2017b] extends TF with special combinators that dynamically compute a graph for each shape permutation; these high-level constructs are opaque to further opti-

mizations. Even in the presence of control flow-free models, looping constructs are necessary to implement optimization algorithms such as SGD. The central challenge is a flexible and extensible encoding of control flow operations. It is well known in functional programming literature 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 recursive functions.

3.1.5 First-Class Functions

A computation graph is a single expression from multiple inputs (i.e. its free variables) to multiple outputs. While it may be tempting to reinterpret a graph as a function, it lacks functional abstraction and named recursion. Adding the ability to name functions and pass them as first-class values dramatically increases Relay's expressivity, allowing it to encode generic higher-order functions and readily use techniques used in functional compilers like automatic deforestation. First-class functions enable passes such as automatic differentiation, and simplify the framework importers which map higher-level programs to our IR [Breuleux and van Merriënboer, 2017]. 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 full loop state. See Figure 3.2 for an example of this conversion (via the Relay TensorFlow frontend).

3.1.6 Data Abstraction

Many models make use of additional data types beyond tuples, such as lists, trees, and graphs [Karpathy, 2015, Tai et al., 2015, Liang et al., 2016]. 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 Chapter 2.

The resulting language is a familiar strict functional language, resembling the core of languages like OCaml and SML. A functional language provides a few notable advantages. Its pure fragment represents idealized computation graphs free from effects. This fragment can be easily optimized by end users who can reason about it as pure dataflow. For this reason, Relay is pure by default but exposes a limited form of mutation via ML-style references that we

have primarily used for automatic differentiation.

Relay is more expressive than many previous frameworks and this expressivity introduces new challenges. Previous essential functionality such as shape inference and automatic differentiation must be adapted for our new IR. How does one reason about the shapes of operators when the input is unknown? How does one backpropagate over pattern-matching, control, data types, and mutation? In the following subsection we demonstrate how one can adapt techniques from type inference and checking to Relay.

```
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])
```

```
let %while_loop =
fn (%loop_var0: Tensor[(1,), int32],
    %loop_var1: Tensor[(1,), int32],
    %loop_var2: Tensor[(1,), int32]) {
    %0 = add(%loop_var0, %loop_var1)
    %1 = less(%0, meta[Constant][0])
    %2 = multiply(%loop_var1, %loop_var2)
    %3 = less(%2, meta[Constant][1])
    %4 = not_equal(%1, %3)
    %5 = add(%loop_var0, %loop_var1)
    %6 = greater_equal(%loop_var2, %5)
    if (min(equal(%4, %6))) {
    %9 = add(%loop_var0, %loop_var1)
    %10 = add(%loop_var1, %loop_var2)
    %11 = add(%loop_var2, meta[Constant][2])
    %while_loop(%9, %10, %11)
    } else {
    (%loop_var0, %loop_var1, %loop_var2)
    }
}
%while_loop(meta[Constant][3],
            meta[Constant][4],
            meta[Constant][5])
```

Figure 3.2: A simple TensorFlow loop in the user-facing DSL. 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.

$\boxed{\Delta; \Gamma \vdash e : \tau}$ Expression e has type τ in type context Δ and variable context Γ.

Relation-T

$$\frac{\Delta, T_1 : \texttt{Type}, \ldots, T_n : \texttt{Type} \vdash (Rel(T_1, T_2, \ldots, T_n) \in \{\top, \bot\})}{\Delta; \Gamma \vdash Rel : \texttt{Relation}}$$

Type-Func-Def

$$\frac{\begin{array}{c}\forall i \in [1, r]\, \Delta; \Gamma \vdash R_i(T_1, \ldots, T_n, O) \\ \Delta; \Gamma, a_1 : T_1, \ldots, a_n : T_n, \quad f : \texttt{fn}(T_1, \ldots, T_n) \to O \ \texttt{where}\ R_1, \ldots, R_r \vdash body : O\end{array}}{\begin{array}{c}\Delta; \Gamma \vdash \texttt{def}\ \texttt{@}f\texttt{(}a_1\texttt{:}T_1\texttt{,} \ldots a_n\texttt{:}T_n\texttt{)} \to O\ \texttt{where}\ R_1, \ldots, R_r\ \texttt{\{}\ body\ \texttt{\}} : \\ \texttt{fn}(T_1, \ldots, T_n) \to O\ \texttt{where}\ R_1, \ldots, R_r\end{array}}$$

Type-Call

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash f : \texttt{fn}(T_1, \ldots, T_n) \to O\ \texttt{where}\ R_1, \ldots, R_r \\ \Delta; \Gamma \vdash a_1 : T_1, \ldots, a_n : T_n \quad \forall i \in [1, r]\, \Delta; \Gamma \vdash R_i(T_1, \ldots, T_n, O)\end{array}}{\Delta; \Gamma \vdash f(a_1, \ldots, a_n) : O}$$

Figure 3.3: Examples of Relay's typing inference rules, namely the rules for function definitions and function calls, where Δ is the environment for types and Γ is the environment for variables. These demonstrate that type relations must hold at each call site.

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. Computation graph IRs rely on typing in the form of datatype and shape inference. Datatype and shape inference is the process of computing the concrete datatypes (e.g., `float32`, `int32`) and shapes (e.g., $(10, 5)$, $(100, 1, 32)$) of all tensors in a computation graph. Deep learning frameworks and compilers use static shape information to perform allocation, check correctness, and facilitate optimization. Precise static shape information is also valuable for traditional loop optimizations, data layout transformations, tensorization, and optimizations that are necessary to map to hardware accelerators' unique ISAs. 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. Shape inference is usually formulated as a simple analysis over the dataflow graph that propagates shape information. Shape inference looks remarkably similar to type inference. Unlike type inference, though, shape inference is separate from the type system and does not provide types for functions or data structures. Handling shape inference at compile time is desirable, because it allows optimizations to take advantage of this information even though certain shapes may be symbolic. Can shape information be encoded in static types? If we type Relay as simply typed lambda calculus, we gain a simple system, but one that

can not represent polymorphism, and lacks shape information. Even with the addition of polymorphism, there is no representation of static shape information. It is possible to model arbitrarily complex static properties, such as shape information, with a dependent type theory [Selsam et al., 2017], but such a design incurs significant user complexity. Adopting a well known type system allows the application of classic techniques, but standard type systems do not provide a solution for tracking static shape information. 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.

3.2.1 *Tensor Types*

The primitive value in Relay is a tensor, which has a shape and a base type (`tensor type` in Figure 3.1). 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. 3.3 describes the process.

In the context of dynamic models, many data shapes can only be determined at runtime. Therefore, the previous assumption of static data shapes does not hold. In order to support dynamic data shapes, Relay VM introduces a special dimension called Any to represent statically unknown dimensions. For example, we can represent a tensor type as `Tensor[(1, 10, Any), float32]`, where the size of the third dimension in this tensor is unknown while the other two dimensions have concrete values. This concept has been introduced in other frameworks. We do not support tensor types with dynamic ranks given the relatively limited use cases and optimization opportunities.

3.2.2 *Operators and Type Relations*

A difference between general purpose programming models and those tailored to deep learning is the use of operators as the primitive unit of computation. Relay's use of opaque operators enables back-

ends 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). For example some operators have result types which can be written as a function of the input types. For example, we use a relation that describes the broadcasting rule for all element-wise operations. Unfortunately some are not only functions, but also relations that specify constraints between input and output shapes. To handle the constraints between operators' argument shapes, Relay's type system introduces a concept of 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 (including those of operators) may include one or more type relations over an arbitrary subset of the argument types and the return type. The type checker enforces that these relationships hold at the call site. These relations may be viewed as a verification condition induced at a function call site, where the formula is a conjunction of the relations. For example, primitive operators are assigned types that are universally quantified over both the input and output types. We can then use a type relation to encode a constraint that must hold later when type checking observes specific input and output types. We first describe these relation in depth, then discuss how they are used in type inference.

3.2.3 Operator Type Relation

A type relation describes the relationship between the types of operator inputs and outputs. The type system of TVM's Relay IR relies on these type relations to infer and bidirectionally propagate type and shape relationships between inputs and outputs of operators across the whole program. For example, the type relation for broadcasting operators (e.g. `broadcast_add`) performs the shape calculation following the NumPy broadcasting rules[2].

[2] https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

The type relation must be generalized to properly handle dynamic shapes. For example, a program which grows a tensor on each loop iteration (a case existing in the decoder of many NLP models) is both impossible to type and compile without proper type system support. With the introduction of Any, we are able to improve the existing type relations to support dynamic models.

There are two cases that must be handled after we introduce

Any to the type relations. First, operators such as `arange`[3] and `unique`[4] have dynamic output shapes depending on the input data, which will be described in Any. We can now describe the type relation using the enhanced type relation functions with Any but not before. For example, type relation of `arange` can be expressed as follows

[3] `arange` generates a range of values in a (start, stop, step) interval the arrays output size is a function of input arguments.

[4] `unique` selects the unique elements of a tensor.

```
arange: fn(start: fp32, stop: fp32, step: fp32) -> ([Any], fp32)
```

Second, when input shapes of a type relation contain an `Any` dimension, the type relation needs to propagate Any correctly to the output types and relax typing constraints that hold in the static cases when necessary. For example, type relation for `broadcasting` operators determines the compatibility between corresponding dimensions from both inputs if they are equal or one of them is 1 according to the Numpy broadcasting rule. For example, the rules for broadcast type relation given the matching dimension from two inputs when having Any are defined as follows:

$$\begin{aligned} \text{broadcast\_rel}(Any, 1) &\rightarrow Any \\ \text{broadcast\_rel}(Any, d) &\rightarrow d \qquad (d > 1) \\ \text{broadcast\_rel}(Any, Any) &\rightarrow Any. \end{aligned}$$

Note that due the presence of dynamic shapes these type relation rules can no longer rule out all type errors at compile-time. For example, for the second rule shown above, when Any is neither 1 nor d at runtime, it then violates the broadcast type constraints. To address this, we take the gradual typing [Siek and Taha, 2006] approach and leave certain type checking at runtime after Any is instantiated by a concrete value (see subsection 6.3.5 for more details). One could eliminate these errors using a more advanced type system, but at increased complexity.

3.3 *Type Inference*

The most interesting parts of the type system are where shape computation occurs. We highlight a few examples of Relay's inference rules in Fig. 3.3. In this subsection we focus on design decisions behind Relay's type system and the implementation of 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 subexpression 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 run 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.

Our fine-grained dependence graph provides the transitive dependencies between relations and unification variables. The use of fine-grained dependencies enables our algorithm to only retry a minimal number of relations when we learn a new variable assignment. We run this to fixpoint or until the queue is empty. If the queue is not empty and no progress is made between iterations, then at least one variable is under constrained 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.

One caveat of introducing an Any dimension is that unknown dimensions will propagate during type inference, reducing chances for shape specialization. For example, if we use an operator such as `arange` to produce a tensor with dynamic shape (i.e., `Tensor[(Any,), float32]`) and later `broadcast_add` to a tensor with static shape (i.e., `Tensor[(5, 1), float32]`), the output shape will also contain an Any dimension (i.e., `Tensor[(5, Any), float32]`).

To limit the contamination of Any, we further introduce *sub-shaping* to improve the precision of types computed by type-inference. Much like sub-typing used in popular programming languages [Liskov and Wing, 1994, Amadio and Cardelli, 1993], our extension

enables values with more specific shape information to be passed in contexts which require less specific shapes. Further, we perform extra analysis that repeatedly replace one Any dimension by a symbolic variable followed by a type inference pass to identify if other Any dimensions share the same symbolic expression. This information is passed to downstream compilation and can reduce the search space during the symbolic codegen. We can use this analysis in the downstream compilation to generate shape-specialized code during codegen. We treat concrete dimension and symbolic dimension as a sub-shape of Any.

3.3.1 Shape Function

Existing deep learning compilers only deal with static shapes, enabling all shape computation to occur at compile-time. Therefore, it is easy to allocate memory for the tensors using static shape information to compute the precise amount of storage needed for each tensor. However, introducing Any invalidates this pre-allocation mechanism as the tensors may now contain dynamic dimensions. The introduction of Any dimension invalidates the pre-allocation mechanism adopted in the existing deep learning compiler. Instead, we now have to track the amount of memory required to be allocated in parallel to computing. Furthermore, static type checking cannot eliminate all type errors at compile-time due to dynamic tensor shapes. Consequently, we define a *shape function* to compute the output shape for storage allocation and verify the type relation in accord with the semantics of every operators. The shape function is similar in structure to the type relations described in 3.2 but are present at runtime instead of compile-time. The shape function enables compiling and embedding the computation of output shapes into the program.

According to the characteristics of the operator, we divide the shape functions in three different modes: data independent, data dependent, and upper bound.

- *Data independent* shape functions are used for operators in which the output shape only depends on the shapes of inputs such as normal 2-D convolution.
- *Data dependent* shape functions require the concrete input values to compute the output shapes. For example, the output shape of `arange` depends on the value of start, stop, and step.
- In addition, there are certain operators such as Non Maximum Suppression (`nms`) where the complexity of computing the output shapes is on par with the complexity of executing the operator itself. In order to avoid the redundant computation, we use an

upper bound shape function to quickly estimate an upper bound shape for the output. We also require such operators to return the output shape along with output value, so as to use the real shape to slice the output tensors into precise output shape and layout.

It is worth noting that in the presence of dynamic shape functions, operator fusion needs to be specially taken care of. However, we only define the shape function as TIR expressions. As a result, we also must fuse shape functions in parallel with the operator fusion. The compiler can easily connect the shape functions of basic operators to form the shape function for a composite operator when all shape functions are data independent. However, a basic operator with a data dependent or upper bound shape function cannot be fused to other operators, i.e., taking the outputs of other operators as its inputs to fuse together, as the shape function requires to access to the intermediate result within a composite operator. As a result, we explicitly define the fusion policy to prevent this from happening. Due to our definition of shape functions we are able to perform fusion using a generalized version of the algorithm used for standard operator fusion which handles passing the appropriate input shape or input depending on the shape function. For example imagine that the fuse operator has a single input that was originally used an operator that requires the input, and the second fused operator requires the input shape. Depending on the type of the shape function defined above, we handle operator fusion in the following manner. Any data independent shape functions could be fused with other operators. Any fused group can only have at most one shape function and it should be fused together with the operator that needs shape function.

3.4 *Evaluation*

We evaluated Relay on several systems (x86 CPUs, ARM CPUs, NVIDIA GPUs, Xilinx FPGAs) and over diverse vision and NLP workloads to demonstrate that (1) Relay enables *composability* of graph-level optimizations, (2) Relay delivers *performance* on inference tasks competitive with state-of-the-art frameworks (TensorFlow, PyTorch, MxNet), and (3) Relay provides *portability* over difficult-to-compile-to hardware backends such as FPGAs

We evaluated the following vision models: *Deep Q-Network (DQN)*, a DNN that achieved state-of-the-art performance on 49 Atari games in 2015; *MobileNet*, a DNN designed for image recognition on mobile and embedded devices; *ResNet-18*, a DNN for image recognition that achieved state-of-the-art performance on ImageNet detection tasks in 2015; *VGG-16* (named for the Visual Geometry Group at Oxford),

a CNN used for image recognition tasks [Mnih et al., 2013, Howard et al., 2017, He et al., 2015, Simonyan and Zisserman, 2014].

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; *RNN, GRU, and LSTM*, a selection of models from the Gluon model zoo [Robertson, 2017, Tai et al., 2015, Gluon Team, 2019].

3.4.1 *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 models where the input is random, we run 1000 timed iterations. Before the timed runs, we run 8 untimed "warm-up" iterations to ensure any caching and JIT compilation employed in lower levels of the stack are included in the 1000 timed runs. This way,st the timed runs reflect the *stable* performance of the system. For our purposes, "performance" refers to end-to-end framework time on inference tasks (i.e., the time it takes to run a trained model) in a single-machine setting. Our vision experiments from Chapter 2 and Section 3.4.2 were run on a machine with an AMD Ryzen Threadripper 1950X 16-Core CPU, an NVidia 1080 Ti GPU, and 64 GB of RAM. Our NLP experiments from Section 3.4.2 were run on a machine with an AMD Ryzen Threadripper 1950X 16-Core CPU, an NVidia Titan-V GPU, and 64 GB of RAM. We evaluated Relay's handling of accelerators on a VTA design with a 16×16 matrix-vector 8-bit tensor core clocked at 333MHz on the Ultra-96 platform. In terms of software, we used Cuda version 10.0, CuDNN version 7.5.0, TVM commit `cefe07e2a`[5], MxNet version 1.4.0, Pytorch version 1.0.1post2, and TensorFlow version 1.13.1. The Relay vision experiments utilized aggressively tuned TVM schedules on the GTX 1080 Ti GPU, improving performance significantly.

[5] NLP experiments required custom modifications that may be made public later

3.4.2 *Vision 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" [Stroustrup, 2004]. We demonstrate that we can achieve competitive performance on both CPUs and GPUs 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 (using our AoT compiler) to NNVM, TensorFlow,

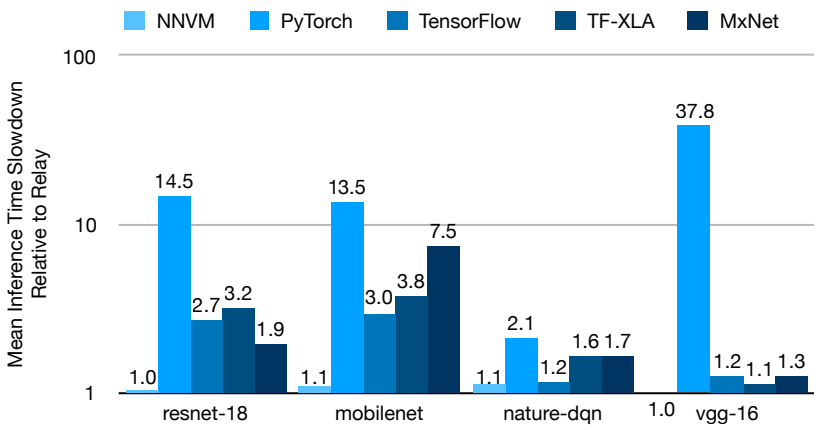

Figure 3.4: Inference slowdown of popular frameworks relative to Relay on vision benchmarks running on NVIDIA GTX 1080 Ti GPUs. Relay provides performance competitive to the state of the art. We ran 1000 trials for each model and used the AoT compiler.

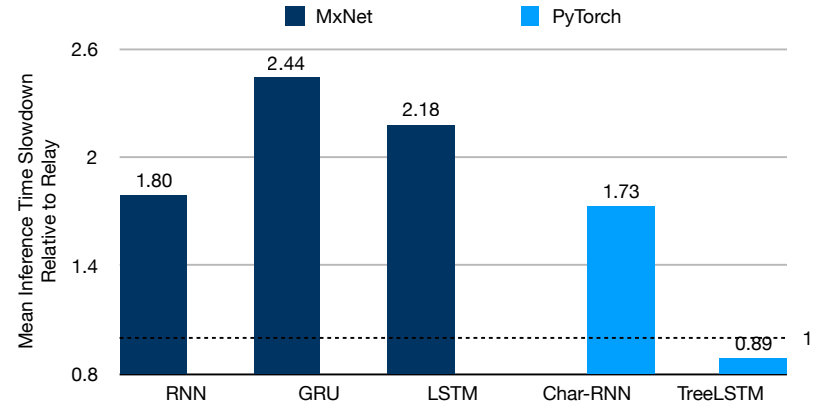

Figure 3.5: Inference slowdown relative to Relay on NLP benchmarks running on NVIDIA Titan-V GPUs. NLP workloads feature control flow, which makes them more challenging to optimize. Relay provides performance competitive to state of the art (up to 2.4× speedup over MxNet on GRU). We ran 1000 trials for each model, except for CharRNN, on which we used 100 trials.

TensorFlow-XLA (Accelerated Linear Algebra), PyTorch, and MxNet. We ran the vision and NLP workloads on GTX 1080 Ti and Titan-V GPUs, respectively.

3.4.3 *Vision Evaluation*

We ran each model with batch size 1, a common setting in inference tasks. Relay achieves performance on par with NNVM, an existing deep learning graph compiler in use at Amazon. Relay outperforms TensorFlow, TensorFlow-XLA, MxNet and PyTorch on every benchmark. Relay's ability to do aggressive optimizations like operator fusion on long chains of operations, generating hardware specific implementations, enables it to outperform existing frameworks that don't perform inter-operator optimizations.

3.4.4 *NLP Evaluation*

Figure 3.4 compares Relay against state-of-the-art NLP models on a Titan-V GPU. Implementations of the NLP models were not available in all frameworks; we used MxNet baselines for RNN, GRU, and LSTM and PyTorch for Char-RNN and TreeLSTM. Relay performs better than MxNet on recursive models due to the fact they are implemented in Python using MxNet's looping constructs. PyTorch instead uses handwritten and heavily optimized C implementations of the recursive network cells. Due to this we perform slightly *worse* than PyTorch. It is interesting to note that our pure Relay implementation performs competitively against the hand-optimized version.

3.4.5 *Targeting Deep Learning Accelerators on FPGAs*

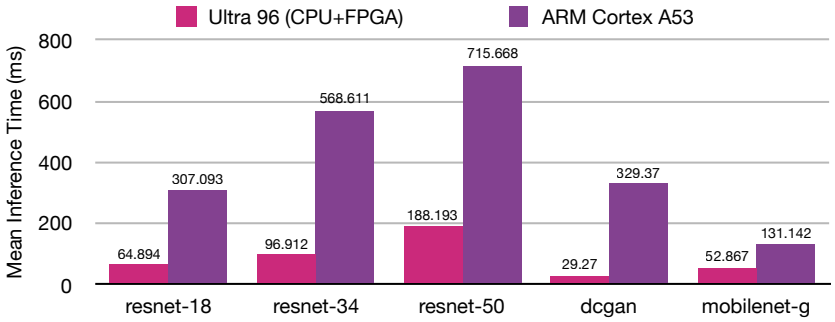

Figure 3.6: Inference time (ms) of vision DNNs on Ultra-96 FPGA-enabled SoC. We compare vision workloads that Relay compiles onto the embedded Cortex A53 CPU vs. a DNN accelerator implemented on the integrated FPGA fabric. Targeting DNN accelerators can unlock up to 11x speedups, but requires a multitude of graph-level transformations. We used 10 trials for each model.

We evaluated inference time on five models including MobileNet-G [Howard et al., 2017], a grouped variant of the MobileNet architecture; ResNet-18, ResNet-34, and ResNet-50[He et al., 2015]; and Deep Convolutional Generative Adversarial Networks [Radford et al., 2015], a generative DNN used in unsupervised learning. Overall, Relay helps us efficiently offload deep learning operators onto specialized accelerators like VTA. Our results in Figure 3.6 show that we can achieve between 2.5 to 11.7× reduction in single-batch inference latency by offloading critical operators to the FPGA accelerator. These experiments demonstrate Relay's ability to target current and future deep learning architectures:

1. *Heterogeneous FPGA/CPU offloading*: Relay lets us define the rules for offloading specific operators to the FPGA-based accelerator.

2. *Push-button quantization*: Relay can take a `fp32` model and convert its parameters to `int8` in order to enable inference on specialized accelerators.

3. *Accelerator-friendly data packing:* Relay reorganizes data so it can be effortlessly consumed by a specialized TPU-like accelerator [Jouppi et al., 2017].

Automatic Differentiation

In deep learning merely specifying tensor computations is one only one piece of the puzzle, as one must be able differentiate networks in order to perform optimization. Many popular optimization algorithms, such as stochastic gradient descent rely on the ability to take the gradient of the function with respect to some parameters. Early frameworks provided heavily structured optimization frameworks which directly implemented the backpropagation algorithm for training deep neural networks. The backpropagation algorithm combines computing the gradient of a loss function defined over the output of a network with respect to some parameters, and an optimization step which updates each parameter based on its gradient. Modern deep learning frameworks have realized that the structured imposed by backpropagation is not fundamental and model definitions can be expressed more uniformly. For example we can break down a deep neural network training process into (1) a network definition, a function defined over tensor inputs, (2) a loss function, which computes a scalar score from tensor inputs, and (3) an update step which modifies each of the parameters using their gra gradient with respect to the loss function. All of these pieces can be written as programs which manipulate tensors, except for transforming one computation into another which computes its gradient.

Early systems such as Theano demonstrated that automatic differentiation, the process of automatically computing gradients, accelerates research in machine learning by removing the need to manually derive gradients. There are many ways to approximate or compute the gradient of a function but automatic differentiation is favored due to its precise gradient values and runtime efficiency. Early automatic differentiation work made use of runtime data structures which store a trace of all operations invoked and computes the gradient dynamically by just applying chain rule. This approach is very flexible as it allows users to implement a small number of primitive gradients and handle a large variety of programs which combine these primitives. Unfortunately this purely runtime based approach means that even if a program is known a priori we must dynamically construct and ex-

ecute its gradient. The dynamic nature induces both static overhead, but also limits ahead of time optimization, or deployment to resource limited devices.

In order to address these weaknesses many have reformulated automatic differentiation as a source code transformation where AD transforms a function into one which computes its gradient. The generated gradient function is just a standard tensor programming enabling uniform optimization and compilation. This approach has enjoyed recent popularity due its conceptual elegance and uniformity. For example a quantization technique defined as a program transform can then be applied to the forward and backward passes of your network without needing a special version of quantized AD. Unfortunately these source code transformations do not cleanly generalize to all language features or dynamic behaviors such as closures, mutation, control-flow, data structures or dynamic allocations. There have been numerous previous approaches attempting to handle a large number of features as described in Chapter 2. The rest of this chapter explores how to build on the simplicity of tape-based AD, by adapting it higher-order imperative languages, resulting in a source code transformation which supports dynamic features without giving up the ability to compile and optimize. We close the chapter focusing on how partial evaluation is essential to this approach.

4.0.1 Automatic-Differentiation

As discussed in Chapter 2 Automatic Differentiation has a long history and numerous applications across many fields. The style of automatic differentiation popular when we began this work was the what is known as a Wengert-list, a runtime data structure which tracks all operations performed, as well as its inputs and outputs, recording them on tape that can be later replayed to compute the partial derivative with respect to each variable. Approaching automatic differentiation in this style leads to a relatively simple implementation which can handle arbitrary language features. When performing differentiation we need not consider data structures, control, scope, or any other language features as they have been computed away. We only see a trace of operations of unrolled in time, allowing us to simply apply the chain rule from basic calculus. This approach is beautifully simple as we do not need to formulate clever ways to account for each language feature. Unfortunately the dynamic nature of this solution requires tracking extra runtime state and performing computation for each operation, as well as providing an incomplete view on the computation which often limits available optimizations.

A design goal of Relay is to provide a target agnostic substrate

on which to build more complex compilers. When starting to work on training and automatic differentiation we had the same design goal, we wanted to enable training frameworks which could use a single code base to provide state-of-the-art performance without compromising expressivity. For example train a model on a cloud GPU or a micro-controller without needing to change your workflow. Exploring training demanded an automatic differentiation method that can execute in multiple execution environments while handling Relay's increased expressivity. We can work backwards from our requirements to eliminate some courses of action. We need an algorithm which handles Relay's dynamic features, maintains key properties such as static typing, enables ahead of time optimization, and be compiled through standard flows. During the period we did this work there were numerous groups working on variations of this problem for numerous systems as detailed in Chapter 2. We ruled out existing approaches as many of them violated our requirements, some required dynamic typing and/or reflection, others required complex AD rules per language feature, were only defined on SSA, or used staging to remove features not supported by less expressive IRs. After many iterations we discovered a design which uniquely achieved these three goals with relative simplicity:

- Handles all language features including closures, references, control flow, and data structures.
- Maintains static typing of programs, and by extension shape inference.
- Can be performed as an ahead of time source code transformation allowing program to be uniformly optimized.

4.0.2 Higher-Order, Higher-Order Automatic Differentiation

Previous automatic differentiation (AD) techniques used on computation graphs cannot be directly applied to Relay due to new language features such as closures, recursion, and control flow. Furthermore, it has become increasingly important to compute not only first-order gradients of functions but potentially nth-order gradients [Chen et al., 2018a, Liu et al., 2018]. These two challenges were our original motivation to pursue a new automatic differentiation technique in Relay. Our AD algorithm is conceptually a source code transformation version of the Wengert List or tape-based method of AD. We make a few core changes. First, our algorithm is defined as a source code transformation. Given an expression, Relay produces a corresponding expression that computes its gradient. Figure 4.1 provides a denotation from Relay expression to Relay expression that defines

our AD algorithm. Second, our algorithm eschews ideas such as delimited continuations in favor of an approach using closures and references. Our AD algorithm lifts each tensor value into pairs of the original value, and a reference which tracks its partial derivative with respect to its output. This form of reverse mode AD is similar to how one would implement forward mode AD using dual numbers, except we also reverse the program and connect the backwards computations via stable references. Relay lifts all tensor-typed values to a pair, an expression of some tensor type `T` becomes a tuple of `(T, Ref<T>)` where the second component contains the sensitivity variable needed to compute the partial derivative. For each gradient function generated, Relay allocates a single reference which stores the "backpropagator," a closure which propagates gradients from the output to the input. Each subcomputation affecting the gradient updates this closure; when it is finally executed, the built-up closure returns the final derivatives with respect to to the arguments. As described in Figure 4.1, only computations involving tensors contribute to the gradient. For example, we support mutability for free because mutation does not affect the gradients. In this sense, our design is simple. All tracing needed to compute derivatives is done at run time, enabling support for higher order functions, higher order gradients, data-dependent control flow, and mutability without requiring changes to the algorithm. Finally, Relay exposes this transformation as an operator, allowing users to compute the gradient of a function `f` simply by writing `grad(f)`.

Many other variants of AD, including algorithms with different complexity bounds (e.g., forward-mode AD), exist. Forward-mode AD is useful for computing the Hessian vector product, which is necessary for techniques like differentiable architecture search (DARTS) [Liu et al., 2018]. Because our AD algorithm just another Relay pass, it is possible for users to implement and experiment with different AD techniques without changing the system. To this end, we also implemented a forward-mode AD algorithm using the traditional method of dual numbers [Baydin et al., 2015]. Both forward-mode and reverse-mode AD are higher-order and extensible: they support closures, abstract data types, control flow, and recursion. Although we have not investigated composing forward and reverse modes, it is possible to mix gradient functions because they are regular Relay functions. Because our algorithm enjoys a closure property, we can perform AD over the composition of the gradient functions.

4.0.3 Partial Evaluator

Existing deep learning IRs have relied on a mixture of staging

$$
\begin{aligned}
&\mathsf{ADType}\,(\texttt{Tensor}\ t) && = && \texttt{(}t\texttt{,Ref[}t\texttt{])}\\
&\mathsf{ADType}\,(\texttt{(}t_0\texttt{,}\cdots\texttt{,}t_n\texttt{)}) && = && \texttt{(}\mathsf{ADType}(t_0)\texttt{,}\cdots\texttt{,}\mathsf{ADType}(t_n)\texttt{)}\\
&\mathsf{ADTerm}\,(\texttt{Var}\ x) && = && x\\
&\mathsf{ADTerm}\,(\texttt{Ref}\ e) && = && \texttt{ref(}\mathsf{ADTerm}(e)\texttt{)}\\
&\mathsf{ADTerm}\,(\texttt{Lit}\ l) && = && \texttt{(}l\texttt{,ref(0))}\\
&\mathsf{ADTerm}\,(\texttt{(}e_0\texttt{,}\cdots\texttt{,}e_n\texttt{)}) && = && \texttt{(}\mathsf{ADTerm}(e_0)\texttt{,}\cdots\texttt{,}\mathsf{ADTerm}(e_n)\texttt{)}\\
&\mathsf{ADTerm}(\texttt{match (}e\texttt{) \{} && = && \texttt{match (}\mathsf{ADTerm}(e)\texttt{) \{}\\
&\quad\texttt{|}\ p_0 \to e_0 && && \quad\texttt{|}\ p_0 \to \mathsf{ADTerm}(e_0)\\
&\quad\vdots && && \quad\vdots\\
&\quad\texttt{|}\ p_n \to e_n && && \quad\texttt{|}\ p_n \to \mathsf{ADTerm}(e_n)\\
&\texttt{\})} && && \texttt{\}}\\
&\mathsf{ADTerm}(\texttt{fn (} && = && \texttt{fn (}\\
&\quad x_0\texttt{:}\ t_0\texttt{,} && && \quad x_0\texttt{:}\ \mathsf{ADType}(t_0)\texttt{,}\\
&\quad\vdots\texttt{,} && && \quad\vdots\texttt{,}\\
&\quad x_n\texttt{:}\ t_n\texttt{)} && && \quad x_n\texttt{:}\ \mathsf{ADType}(t_n)\texttt{)}\\
&\qquad\to\ t_{\text{ret}} && && \qquad\to\ \mathsf{ADType}(t_{\text{ret}})\\
&\texttt{\{}\ e\ \texttt{\})} && && \texttt{\{}\ \mathsf{ADTerm}(e)\ \texttt{\}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ADTerm}\,(f\texttt{(}e_0\texttt{,}\cdots\texttt{,}e_n\texttt{)}) && = \\
&\quad\texttt{let}\ \overleftarrow{e}_0\ \texttt{=}\ \mathsf{ADTerm}(e_0)\texttt{;}\\
&\quad\vdots\\
&\quad\texttt{let}\ \overleftarrow{e}_n\ \texttt{=}\ \mathsf{ADTerm}(e_n)\texttt{;}\\
&\quad\texttt{let}\ v\ \texttt{=}\ f\texttt{(}\overleftarrow{e}_0\texttt{.0,}\cdots\texttt{,}\overleftarrow{e}_n\texttt{.0);}\\
&\quad\texttt{let}\ \overline{v}\ \texttt{=}\ \texttt{ref(0);}\\
&\quad\texttt{let}\ \delta\ \texttt{=}\ \texttt{fn () \{}\\
&\qquad\texttt{let (}\overleftarrow{e}_0\texttt{,}\cdots\texttt{,}\overleftarrow{e}_n\texttt{) =}\\
&\qquad\quad\overleftarrow{f}\texttt{(}v\texttt{,!}\overline{v}\texttt{,!}\overleftarrow{e}_0\texttt{.0,}\cdots\texttt{,!}\overleftarrow{e}_n\texttt{.0);}\\
&\qquad\overleftarrow{e}_0\texttt{.1 +=}\ \overleftarrow{e}_0\texttt{;}\\
&\qquad\vdots\\
&\qquad\overleftarrow{e}_n\texttt{.1 +=}\ \overleftarrow{e}_n\texttt{;}\\
&\qquad\texttt{()}\\
&\quad\texttt{};}\\
&\quad\Delta\ \texttt{:=}\ \texttt{!}\Delta\ \circ\ \delta\texttt{;}\\
&\quad\texttt{(}v\texttt{,}\overline{v}\texttt{)}
\end{aligned}
$$

Figure 4.1: Transformation Rules for Automatic Differentiation in Relay. The most interesting case is for function calls. The backpropagator Δ is initialized to `ref(fn() { () })` at the top level of each ADTerm call. Successive update closures δ are then composed with Δ to form a chain. Syntactic sugar is used for some constructs which are not available as primitives in Relay.

Identity Function

```
fn <s, bt>(%d: Tensor[s, bt]) {
  %d
}
```

Post-AD

```
fn <s, bt>(%d: Tensor[s, bt]) {
  let %x = ref(fn () { () });
  let %x1 = (%d, ref(zeros_like(%d)));
  let %x2 =
    (fn <s, bt>(
      %d1: (Tensor[s, bt],
            ref(Tensor[s, bt]))) {
      %d1
    })(%x1);
  %x2.1 := ones_like(%x2.0);
  let %x3 = read(%x)();
  (%x2.0, (read(%x1.1),))
}
```

Post-PE

```
fn <s, bt>(%d: Tensor[s, bt]) {
  let %x = fn () {
    let %x1 = ();
    %x1
  };
  let %x2 = ref(%x);
  let %x3 = zeros_like(%d);
  let %x4 = ref(%x3);
  let %x5 = (%d, %x4);
  let %x6 =
    fn <s, bt>(
      %d1: (Tensor[s, bt],
            ref(Tensor[s, bt]))) {
      %d1
    };
  let %x7 = ones_like(%d);
  %x4 := %x7;
  let %x8 = ();
  let %x9 = (%x7,);
  let %x10 = (%d, %x9);
  %x10
}
```

Post-DCE

```
fn <s, bt>(%d: Tensor[s, bt]) {
  (%d, (ones_like(%d),))
}
```

Figure 4.2: Example of running the compiler pass pipeline for AD on the identity function. First, we run the base AD pass on the original function (described in Section 4.0.2). Then, we run the partial evaluator, which primarily optimizes away the reads and calls in `%x2` and `%x3` in post-AD. Since it conservatively determines whether a subexpression is effectful, it generates many bindings which are dead code. At this point, we run the dead code elimination pass to crunch the code back down.

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 [Thiemann and Dussart, 1996]. 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.

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 [Thiemann and Dussart, 1996]. 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.

In order to handle differentiating the full IR, our AD algorithm makes use of closures and references. However many of the programs are effectively first-order and do not require allocating references or a backpropagator closure. It is essential we remove unnecessary uses of closures and references as they inhibit optimizations like operator fusion. Previous approaches have used staging to manually phase computation, but this requires modifications to the language itself. A partial evaluator (PE) allows the use of high-level abstractions without limiting code that *could* in practice be compiled to a particular target. The benefits of partial evaluation do not only extend to code generated by AD but for all of Relay. 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. This makes the partial evaluator more powerful than a

constant-folding pass. The appendix presents an implementation of PE.

There are two important features of our partial evaluator: managing effectful computations and handling references. In order to handle effects, we keep the generated program in A-normal form to ensure effects are properly ordered and to avoid the 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 is managed to achieve flow sensitivity. After evaluation we construct a new program with static sub-computations evaluated away. The reconstructed program contains all original expressions, as well as evaluated expressions, because interleaving dead-code elimination (DCE) is non-trivial. Afterwards, we separately apply DCE. The result of this entire process is illustrated in Figure 4.2.

Optimizations

Building a compiler can be likened to building a bridge between two distant points. The semantic gap between your starting point and end point imply the complexity of the structure needed to bridge the gap. Optimizations are the building blocks of such a structure. If one chooses an input representation sufficiently close to the target representation compilation is relatively straight forward and often little to no optimization is needed. Unfortunately as the semantic gap and complexity are inextricably linked and as it grows so does the difficulty of writing effective optimizations. It is not sufficient for a compiler author to be able to perform an optimization by hand but they must be able to generalize it into a repeatable, inductive process. The challenge introduced by using a sufficiently abstract and high-level representation is designing the transformations and optimizations to get us to our end goal. Given that our goal is state-of-the-art performance in competitive area, it is critical we get our optimizations right. Over the course of my PhD thesis we have designed more than 50 passes for Relay, enabling TVM to reach state-of-the-art performance on numerous real world models [Fromm et al., 2020]. This chapter focuses on the design and implementation of selected passes. The remainder of which exist in Apache TVM's source tree and are open to all to read and inspect.

5.1 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 im-

plementation 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.

5.1.1 Extraction

First, Relay identifies subexpressions containing fusion-eligible operators 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.

5.1.2 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.

5.2 Generic 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. An emerging area in deep learning is performing training and inference on non-standard numeric types to improve throughput and memory usage. For example, a single neural network may have more than one million floating-point values as its parameters. The sheer quantity of parameters and their datatypes may limit the ability to execute these networks on hardware accelerators. Accelerators often support fixed point or other non-standard datatypes, at lower precision. In order to target these devices, Relay must map the computation to the appropriate domain. 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 suggests that there exist a number of tradeoffs between different quantization techniques, with the best often determined by platform and model type [Krishnamoorthi, 2018]. Current deep learning frameworks support a limited number of quantization schemes, and options because quantization

requires framework support in the form of custom platform-specific operators. Importantly, there are many different choices of quantization mechanisms. Each type of quantization has different running time and accuracy properties depending on the model as well as the target hardware. Existing frameworks manually choose a fixed quantized data format, which might be suboptimal. Instead, Relay includes a generic, compiler-based quantization flow that supports a diverse set of quantization mechanisms and automatically generate code for each of them. Relay's generalizable and flexible quantization workflow can support customization in both standard devices and acceleration schema and address various constraints across different hardware platforms. The pipeline that we designed can compress and accelerate neural networks with low-precision quantization to enable running the deep learning models on edge devices. 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.

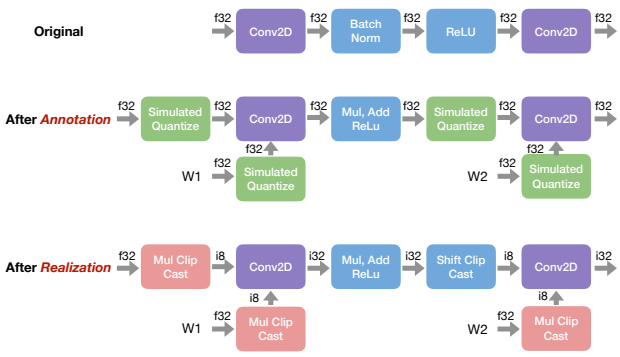

Figure 5.1: The top graph represents the dataflow graph of operators after annotation, and the bottom graph represents the transformed graph. SimQ simulates the rounding error and saturating error of quantizing. Its argument will get tuned during calibration.

Figure 5.2: The simulated quantization operation.

$$Q(x, r, bit, sign) = cast(clip(round(x/r * 2^{bit-sign}), int8)) \tag{5.1}$$

$$simQ(bits, sign, range) = \frac{clip(round(\frac{x}{r} * 2^{bit-sign})) * r}{2^{bit-sign}} \tag{5.2}$$

The generic quantization flow proceeds in three steps: annotation, calibration, and realization. We can apply this pass to convolution-like operators which have a quantized schedule available. Figure 5.1

shows a graphical visualization for this.

5.2.1 *Annotate*

Annotation rewrites the program by inserting simulated quantization operations a according to an annotation rule of each operator. The annotation rule describes how to transform an unquantified operation into a quantized one. 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).

See the definition of simulated quantize, as in Figure 5.1. We annotate the inputs to this operation with an operator $sim\mathcal{Q}$, which simulates the effect of quantization (for example, from a 32-bit floating point value to an 8-bit integer value). However, it is computed with a 32-bit float data type, which is convenient for the calibration pass and debugging. $sim\mathcal{Q}$ 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. Finally, after the algorithm has selected appropriate setting for these parameters, it applies realization, which transforms the simulated quantization operator into the numerator.

5.2.2 *Calibrate*

The simulated quantized operations have a set of parameters which must be calibrated in order to correctly quantize the graph to achieve the minimal decrease in accuracy. 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.

5.2.3 *Realize*

The realization pass transforms the simulated quantized graph (which uses 32-bit floats) into a real low-precision computation graph. The simulated quantized operator is transformed into several fine-grained operations like multiplication and addition. This transforms

ResNet-18		MobileNet V2		Inception V3	
Quant.	Accuracy	Quant.	Accuracy	Quant.	Accuracy
`float32`	70.7 %	`float32`	70.9 %	`float32`	76.6 %
8/16	69.4 %	8/32	66.9 %	16/32	76.6 %
8/32	69.4 %	8/16	66.9 %	8/32	75.2 %

Table 5.1: This table shows the accuracy of various quantized models. `float32` refers to the non-quantized model. Notation as in "8/16" refers to 8-bit quantization, 16-bit accumulation

the `simQ` operator into the below quantization operator.

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

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`.

To avoid the overflow, it also would be better to use larger global scale. With flexible configure, our workflow can be customized as best strategy for different networks and hardwares. Quantized models require low-precision operators for efficiency. Due to our operator representation, and the ability to leverage AutoTVM, TVM's auto tuner, we can automatic generate kernels for arbitrary types without effort.

Developers can customize quantization with very little code. For example, the quantization annotation function may be overloaded for any operation, making use of signed or unsigned integers, or different rounding strategies, such as floor, ceiling, or even stochastic rounding (Figure 5.3).

To demonstrate the effectiveness of our generic quantization, we use Relay to explore different choices of input and accumulation bits. The results are shown in Table 5.1. We find that no single strategy fits all use cases. For certain network architectures such as MobileNet and ResNet, 16-bit to 32-bit quantization provides good accuracy, but 8-bit to 16-bit provides the best speedup, assuming the input does not overflow. The current framework demonstrates the importance of applying various quantization schemes based on networks and hardware platforms.

```
@register_annotate_function("nn.conv2d", override=True)
def annotate_conv2d(ref_call, new_args, ctx):
    lhs, rhs = new_args
    lhs = attach_simulated_quantize(lhs, sign=False, rounding='round')
    rhs = attach_simulated_quantize(
      lhs, sign=False, rounding='stochastic_round')
    return expr.Call(ref_call.op, [lhs, rhs], ref_call.attrs)
```

Figure 5.3: An example of overloading the annotation function for 2-d convolution. In this example we treat both input, and the weights as unsigned integers, applying rounding to the input, and stochastic rounding to the weights.

5.2.4 *Quantized Inference on ARM Platforms*

To demonstrate the effectiveness of our generic quantization (see Section 5.2), 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 %

The above table 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.

5.3 *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. We describe the implementation and some concrete applications of partial evaluation in Chapter 4.

5.4 *Dynamic Memory Allocation, Device Placement, and Code Generation*

A key challenge preventing existing deep learning compilers from handling dynamism is the lack of a uniform and dynamic representation. For example, optimizations and runtime of existing IRs, e.g., TVM's NNVM [Chen et al., 2018b] assume the presence of static shape information. These assumptions introduce quite a few challenges for optimizing dynamic behaviors. This section describes how we transform standard TVM programs into our dynamic dialect

which enables us to easily apply static optimizations to dynamic programs, much as we do in traditional compiler optimization. Particularly, we detail three key components required to compile dynamic models.

- An extended type system which enables static tracking of dynamic shapes.
- A series of optimization passes that make dynamic output shapes, allocation, and device placement explicit.
- A set of code generation techniques for producing code of kernels with dynamic input and output shapes.

5.4.1 Dynamic Memory Planning

Deep learning workloads are dominated by two key factors (1) compute-intensive kernels and (2) memory consumption and allocation. Many deep learning compilers employ a form of static memory planning which tries to coalesce memory allocations and minimize memory consumption. For devices such as GPUs these optimizations are essential for reducing memory fragmentation and ensuring allocation does not hamper kernel performance. Existing deep learning compiler IRs hide memory allocation behind a functional interface, where each operator implicitly allocates their output storage. Then before execution, the system performs static-memory planning on the data-flow graph enabling efficient pre-allocation of the required memory. Due to this "out-of-band" nature of memory allocation, it is challenging to customize, modify or compose memory optimizations with other passes. For example, if one needs to adjust memory allocation for heterogeneous execution, modifications to the runtime are required. TVM's graph runtime is one such example of static memory planning.

Alternatively, some systems lower the entire program to low-level IRs such as LLVM [Lattner and Adve, 2004] in order to perform optimizations. Due to the coarse-grained memory semantics of deep learning models, it is essential that memory optimizations occur at a suitably high-level of abstraction before essential program facts are lost. Moreover, as discussed in subsection 6.3.5, we have introduced new ways to account for the handling of dynamic allocations, which further complicate memory analysis. In order to perform dynamic memory planning we have extended Relay's IR with a set of new primitives we dub the "memory dialect". The memory dialect is a subset of Relay programs where all memory allocations are explicit. The key to this transformation is an interprocedural change of calling convention, with each operator now

taking its outputs explicitly it is possible to track and transform allocations. In particular, we have introduced four new IR constructs, (a) `invoke_mut(op, inputs, outputs)` which takes outputs as mutable in-out arguments, (b) `alloc_storage(size, alignment, device)` which allocates a region of memory of a particular size, (c) `alloc_tensor(storage, offset, shape, dtype, attrs)` which allocates a tensor at a particular storage offset with a shape and data type, and (d) `kill(tensor)` which frees a tensor before its reference count becomes zero due to exiting the frame. Note that in the below code examples `Tensor<d1, ..., dn>` is shorthand for a tensor of shape `(d1, ..., dn)` containing floating point values.

We can demonstrate how to transform a single statically shaped operation such as broadcasting addition.

```
fn main() -> Tensor<10> {
  let t1, t2 : Tensor<10> = ...;
  add(t1, t2)
}
```

Here we only must allocate a single buffer, the return buffer for the addition operation.

```
fn main() -> Tensor<10> {
  let t1 = ...; let t2 = ...;
  let storage = alloc_storage(40, 64, cpu(0));
  let out1= alloc_tensor(storage, 0, (10), f32);
  invoke_mut(add, (t1, t2), (out1));
  out1
}
```

The above transformation replaces all operator invocations with a call to `invoke_mut`, a corresponding allocation of backing storage, and a single tensor placed at offset zero. The key insight is to internalize a notion of memory allocation into the IR, enabling static optimization of both static and dynamic allocations presence of control and dynamic shapes. We realize our shape functions as fragments of TVM's tensor expression language which computes the output shape for a particular operator. As detailed in subsection 6.3.5 our shape functions may require the input, the input *shape*, or both. Our uniform treatment of shape functions as standard tensor expressions enables them to be fused and optimized like normal, but one challenge is that we must now manifest allocations in a fixed point until we allocate for both the compute and necessary shape functions. We illustrate this below with a single dynamic concatenation.

```
fn (x: Tensor<?,2>, y: Tensor<1,2>)->Tensor<?,2> {
```

```
  concat((%x, %y))
}
```

This is the same transformation as the previous example with the addition of carefully inserting invocations to the shape function to compute output buffers sizes for the dynamically sized kernel.

```
fn (x: Tensor<?,2>, y: Tensor<1,2>)->Tensor<?,2> {
  let in_sh0 = shape_of(x);
  let in_sh1 = shape_of(y);
  let storage_0 = alloc_storage(16, 64, ...);
  let out_sh0 = alloc_tensor(storage_0, ...);
  invoke_shape_func(concat,
      (in_sh0, in_sh1), (out_sh0,), ...);
  let storage_01 = alloc_storage(...);
  let out_0 = alloc_tensor(
      storage_01, shape_func_out_0, ...);
  invoke_mut(concat, (x, y), (out_0));
  out_0
}
```

After the transformation you may notice we have introduced calls to `shape_func` which invokes a shape function corresponding to the kernel. The shape function requires input shapes as arguments which further require us to invoke `shape_of` for both `%x` and `%y`. `shape_of` will be directly mapped to a VM instruction to retrieve the shape of a tensor at runtime. More description of it will be provided in subsection 5.4.2.

Now that the all allocations are explicit in the IR we can provide analogous optimizations in the static case on dynamic programs, for example we have implemented a storage coalescing pass which groups storage into a larger region which we can then multiplex tensor allocations on to.

The key insight is to internalize a notion of memory allocation into the IR, enabling static optimization of both static and dynamic allocations in the presence of control and dynamic shapes. Now that all allocations are explicit in the IR, we can provide analogous optimizations in the static case on dynamic programs, for example we have implemented a storage coalescing pass to group storage into a larger region which allows the multiplexing of multiple tensor allocations to a single piece of storage. Further optimization like liveness analysis and graph coloring algorithm can be applied to the program to reuse storage.

5.4.2 *Heterogeneous Device Placement*

As discussed in Section subsection 6.3.5, shape functions are executed at runtime to calculate the output shape of an operator. These functions must execute on the CPU due to the host-interaction model of GPU like devices. In the case of heterogeneous execution (i.e., CPU and GPU) it is essential to carefully schedule the execution of shape functions and kernels as improper scheduling creates memory copies which can be disastrous for performance. For instance copying the inputs to shape functions from the GPU results in considerable performance penalty. To minimize overhead we analyze allocations and memory transfers so we can place each sub-expression on the correct device.

We introduce a unification based analysis for computing the correct device placement and allocation based on the previous scheduling of the compute kernels. The goal of our device analysis is assigning each IR node in a way that minimizes the number of cross-device copies. We introduce a concept of `DeviceDomain` to represent the domain of a device, including source and destination. Each expression in the IR defaults to the empty domain, meaning there are no constraints on its device placement. In addition, two new IR constructs are introduced to facilitate the heterogeneous execution of VM, namely `device_copy` and `shape_of`. The former performs a data transfer between different devices and is inserted when a cross-device data copy is mandatory. The latter is used to retrieve the shape of a tensor at runtime and is used to efficiently compute the input shapes for shape functions. Our analysis is formulated as a set of device placement rules which describe how device constraints flow, and then we use unification, a technique common in type inference and compilers in order to compute precise device placement.

As discussed in subsection 6.3.5, shape functions are executed at runtime to calculate the output shape of an operator. These functions should execute on the CPU as their outputs are used to compute the size of allocated memory. In the case of heterogeneous execution (i.e., CPU and GPU), it is essential to carefully place the execution of IR nodes to proper devices. Otherwise, considerable overhead from data transfers and device synchronization will occur if the inputs to shape functions and kernels need to be copied from or to GPU. To minimize the performance penalty, we analyze the program to place sub-expressions on the most suitable devices:

- `shape_of`. Defaults to the CPU domain because we can access a Tensor's shape regardless of which device it is placed on.

- Shape functions. These IRs take the output of one or multiple

`shape_of` and then derive the shape of an operation according to predefined type inference rules. The output of a shape function is used to compute the amount of memory that this operator requires at runtime, which only needs a few cheap scalar arithmetic computation. Therefore, the inputs and outputs would be better on a CPU domain as well.

- `device_copy`. The input and output of this IR are on different domains as it copies data from one domain to another. The device domains of the input and output are propagated in the opposite directions to other IR nodes that are reachable to/from the device copy node.
- Memory operations. The device domain of storage from `alloc_storage` is designated in the expression, and later is propagated to the device domain of the tensors allocated from this storage via `alloc_tensor`. in the instruction computation of the allocation size and the shape for memory operations, such as `alloc_storage` and `alloc_tensor`, should be on CPU due to low computation intensity.
- `invoke_mut`. All arguments used in the `invoke_mut` must have the same device domain.
- Other common IR nodes. The device domain of other common IR nodes, e.g. variables, constants, operators, etc., can be directly propagated from the above nodes.

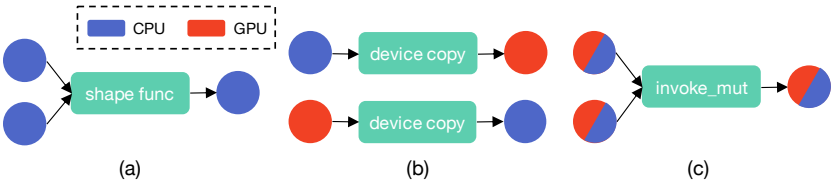

Figure 5.4: Some heterogeneous device placement rules. (a) The inputs and outputs of shape functions are placed on CPU. (b) `device_copy` changes the device of output accordingly. (c) The device of all arguments to `invoke_mut` must be the same.

Based on the rules defined above, we use a union-find data structure to bidirectionally propagate and unify the device placement of each IR node. We introduce two operations, `union(s, t)` and `find(s)`, to achieve `DeviceDomain` unification throughout the entire program. `union(s,t)` unions the equivalence device domains of `s` and `t` into one equivalence domain when the device types match. `find(s)` returns the representative of the device domain that `s` belongs to. These two operations are applied until all IR nodes are annotated. The result of the heterogeneous device placement composes with memory planning and shape function insertion resulting in correctly placed allocations.

5.4.3 *Dynamic Kernel Code Generation*

Deep learning compilers [Chen et al., 2018b, Ragan-Kelley et al., 2013] have demonstrated competitive performance compared to manually tuned kernels on multiple platforms. Recent trends apply machine learning based search to further reduce or eliminate complex manual performance tuning, existing work applies both template based [Chen et al., 2018c, Zheng et al., 2020] and beam search based [Adams et al., 2019] ones.

However existing work which focuses on tuning in the presence of fully static shape information falls short in the presence of symbolic or dynamic shapes. There are two inherent challenges with regards to performing code generation for symbolic shapes.

- How to achieve the same performance of kernels generated with symbolic shapes as that with static shapes when applying the same schedule?
- How to extend the machine learning based auto-tuning to kernels with symbolic shapes?

Loop parallelism and loop tiling are common optimization techniques that exploit multi-core capabilities by achieving data access patterns which are memory hierarchy aware for both CPUs and GPUs. However, the combination of these techniques often introduce complex loop boundary conditions. In many static cases, it is possible to prove these conditions always hold, and thus eliminate checks which hamper further optimizations such as unrolling. While straightforward to handle with static shapes, it becomes a non-trivial challenge when performing symbolic code generation. If not carefully handled, the boundary condition checks will stay, leading to poor performance.

To address this issue, we generate multiple kernels according to the residues modulo of the tiling factor and then dispatch based on the actual shape at runtime. For example, suppose a symbolic dimension x is divided by a factor of 8, we then generated eight duplicated kernels replacing the symbolic var x by $8k + r$ in each copy, where $k = \lfloor x/8 \rfloor$ and $r \in [0..7]$. Lastly, we automatically generate a dispatch function which invokes the corresponding kernel based on the residue. By applying this technique in conjunction with an enhanced symbolic expression simplification pass, we can eliminate most boundary checks to achieve performance that is nearly identical to kernels compiled with a single static shape. Lastly, we automatically generate a dispatch function that invokes the corresponding kernel based on the residue. This strategy can be viewed as a form of polymorphic inline caching [Hölzle et al., 1991],

where the cache is not keyed by type, but shape. Both the dispatch logic and kernels are represented as TIR code enabling them to be optimized and compiled using standard mechanisms.

In addition, the dispatch function can be extended to invoke either compiler generated kernels or third party library whichever is faster from the profiling results. The increased kernel size is relatively small compared to the overall deep learning models. In extreme cases where resources are extremely limited, we can either generate fewer number of kernels than the tiling factor or reduce the tiling factor to find an acceptable trade-off between code size and performance. In case where resources are extremely limited, we can either generate fewer number of kernels than the tiling factor or reduce the tiling factor to find an acceptable trade-off between code size and performance.

A known issue to machine learning based tuning is that it may take a long time (usually hours) to find the best schedule for a single kernel. When it comes to symbolic shapes, the tuning time may be exponentially longer if we naively tune for every possible shape. In this paper, we extend the template based tuning approach for symbolic shapes in order to make tuning time tractable. The template based tuning approach takes a human-defined code template and a search space, and searches the best configure within the search space by using machine learning algorithms. We observe that a good configuration for one shape usually performs well on other shapes. Based on this observation, we devise the following mechanism to tune the kernel for symbolic shapes.

1. First replace the symbolic dimensions by a large enough value (e.g., 64) such that the search space can cover most possibilities, and run the tuning algorithm on the static shape for a sufficient number of iterations.

2. Pick top k configurations, apply them to a selection of other shapes, and evaluate their performance.

3. Pick the configuration that performs best on average among shapes previously evaluated.

We found that $k = 100$ covers most of the best configurations for other shapes. Current popular dynamic models usually only require kernels with one symbolic variable. As a result, we choose the values of power of two up to 256 in the cross evaluation of other shapes. If there is more than one symbolic variable, a more sophisticated selection approach might be required to limit the evaluation time of step 2. We leave this to the future work. Further, if the workload

distribution is known, we could adjust the weighting of known shapes when picking the best configuration for step 3.

Though we address both challenges, we admit that our approach has limitations when all dimensions are unknown. In these cases symbolic codegen cannot completely replace manually tuned 3rd party libraries yet, but is complimentary when partial shapes are known.

5.5 Accelerator-Specific Optimizations

Many accelerates provide non-traditional programming abstractions to both compilers and users. In order to implement code generation for a traditional platform, compiler authors could simply expose an LLVM backend an leverage all LLVM infrastructure. Many accelerators are not programmable in this style and require a mixture of driver interactions and static and dynamic code generation to drive a program to completion. Even on very popular accelerators such as GPU violate the typical CPU programming model requiring custom transformations, runtimes, and optimizations. In order to lower generic deep learning programs to these types of accelerators it is essential we are able to optimize and transform programs to run on this hardware. For example some accelerators only support low-bit datatypes converting quantization from an optimization to a necessary program transformation. Although DL accelerators form a diverse family of designs, one property they have in common is a restricted computing model. The consequence of this is that individual accelerators can rarely execute entire Relay programs. For example, some accelerators cannot execute unbounded loops, requiring individual computations to be scheduled via host device (often a CPU) which interacts with the device runtime as well as programming it.

Below we highlight a few of these optimizations that were used to implement support for the VTA [Moreau et al., 2019] accelerator discussed in Chapter 6.

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 [Moreau et al., 2019]. 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 [Szegedy et al., 2015] 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.

5.6 Dataflow Rewriting

A key portion of compiler optimization is the selection and application of rewrite rules for a variety of use cases. Many of optimizations, especially those around program partitioning can be phrased a sequence of rewriting rules over a program or set of programs. We have explored two very different rewriting approaches in TVM the first was specifying dataflow patterns for end users to match dataflow graph properties and select subgraphs. The second was the use of Egg a library for utilizing equivalence graphs to perform saturated rewriting. We describe both approaches in the context of the TVM stack below.

Executing Relay

Once a Relay program has gone through all optimization and lowering passes we must execute it. In theory a naive interpretation strategy could be applied to the entire Relay program. In fact the Relay definitional interpreter implements a naive recursive AST traversal which applies JIT compilation to each kernel invocation contained in the program. A definitional interpreter is sufficient for specifying the behavior of a language, but not necessarily for efficiently executing one. After applying generic optimizations, most compilers have per target compilation pipelines which lower programs to a specific backend. Many of the behaviors contained in these target specific lowered programs require not only program transformations but also runtime support such as memory allocation, device selection, or scheduling. The remainder of this chapter focuses on the backend of the Relay compiler and its runtime mechanisms. We describe the general backend compilation strategy, lowering Relay to the graph runtime, the virtual machine, ahead of time compiler, and hardware accelerators. In this chapter we evaluate Relay's performance specifically on state-of-the-art NLP applications and demonstrate state-of-the-art performance out performing leading industry standards such as TensorFlow and PyTorch.

6.1 Compiler Framework

To begin we first refresh the reader on the end-to-end dataflow of the Relay compiler. First, a frontend converts its input format into the Relay IR. Next, the Relay compiler typechecks and optimizes the program. Processes such as automatic differentiation can be performed during this step. Next we extract primitive functions from the Relay program, functions which may be lowering to TE or TIR, the process for selecting these is defined in Section 5.1. We then schedule and lower these expressions to produce low-level target-specific versions of these functions. We then further transform the code which cannot be lowered into TIR via a sequence of passes depending on which backend we are targeting. Finally we execute

the remaining Relay code via a TVM runtime either the interpreter, which directly interprets the AST, the virtual machine, graph runtime, or native compiler all of which requires separate compilation phases. The focus of this chapter is describing this process in detail for each compilation and runtime target.

6.1.1 *Frontend*

There are several ways to write an Relay program. A user can build an in-memory representation of a program in C++, Rust 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 a TensorFlow graph, converting it to an Relay loop, and transform the `TensorArray` into an Relay list. Many "importers" struggle to translate TensorFlow's full graph as their intermediate representation is not rich enough to capture the full IR, and often result in ad-hoc hacks to replicate TensorFlow's behavior. Once new deep learning languages and IRs under development are stable it is likely that they can all be translated into Relay (see Section 2.5.1). PyTorch provides an expressive programming model, and is a good fit for Relay, which has been previously integrated into PyTorch's [1] [2] JIT infrastructure, enabling users to transparently use Relay for improved performance.

[1] `https://github.com/pytorch/tvm`

[2] PyTorch engineers built an integration which connects PyTorch's backend to TVM.

6.1.2 *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 Chapter 2 for more details.

6.1.3 *Runtimes*

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 [Chen et al., 2018b, Vasilache et al., 2018, Ragan-Kelley et al., 2013]. 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.

6.2 Interpreter & Graph Runtime

In the tradition of definitional interpreters we introduced a simple interpreter for Relay which implements its formal semantics, which we have separately formalized. Relay's interpreter can execute the full language but has notable limitations that make it unsuited for production deployments. It is structured as an inefficient interpreter that performs AST traversal to execute the program. Each time we want to run a sub-expression we must traverse each child node a large cost that can be easily avoided. This approach is conceptually simple but inefficient, as the AST traversal heavily relies on indirection. Furthermore we perform JIT compilation with concrete observed shapes in the interpreter, a flexible, but also costly choice. For example the initial Relay prototype reused the existing "graph runtime", to obtain acceptable performance for vision tasks. The graph runtime is heavily over engineered for completely static models. The graph runtime can only execute simple control-free, DAGs of operations. We can optimize Relay programs and map a subset of them to the graph runtime, but any use of new Relay features are unsupported. The graph runtime does not support control-flow, dynamic shapes, recursion, closures, or data structures.

6.3 Virtual Machine

Existing approaches to dynamic model optimization apply or extend existing deep learning frameworks [Xu et al., 2018, Gao et al., 2018, Yu et al., 2018, Jeong et al., 2018, 2019, Neubig et al., Looks et al., 2017b]. As discussed in Section 2.1 frameworks are large monolithic pieces of software with both portability and performance challenges. Existing work which builds on frameworks extends the programming model either via sophisticated additions [Yu et al., 2018] or significant runtime overhead [Looks et al., 2017b, Jeong et al., 2019]. Other work [Xu et al., 2018, Gao et al., 2018, Looks et al., 2017b] which is focused on optimizing specific types of models is hard to generalize to new models, or generalize over all models. Moreover, approaches which inherit from frameworks rely on third-party kernel libraries such as OpenBLAS [Zhang et al., 2014], cuDNN [Chetlur et al., 2014], and MKL-DNN [Intel, 2018] to achieve competitive performance. These libraries expose a fixed set of operators for the corresponding hardware, compromising the portability of dynamic models which require a large number of i specialized operators with varying data types and shapes. Designing a new interface independent of existing frameworks provides a clean programming model but often at the cost of performance, due to dynamic interpretation of the model [Neubig et al.]. Due to our ability to design a new IR and extensions for dynamic features we solve the majority of these challenges via the use of compilation. Even though we can elide dynamism in many cases, there are still programs which need truly dynamic features, especially in more advanced scenarios like training. To this end, we designed Relay VM, a high-performance and portable system for executing compiled dynamic neural networks on multiple platforms.

6.3.1 A Tensor Virtual Machine

TVM although named Tensor Virtual Machine, follows in the lineage of LLVM where its name is technically inaccurate. In fact this is the first abstract or virtual machine introduced to the TVM compiler stack. The Relay VM is a realization a tensor abstract machine, where each operation corresponds to high-level tensor operations such as allocating a tensor, invoking an operation like conv2d, or performing a device copy. Conventional deep learning runtimes, those which apply interpretation of the computation graph by walking each node in topological order, are non-optimal for dynamic neural networks. These interpreters are reminiscent of the earliest languages interpreters where the input language is directly processed to execute

the program. Due to the introduction of Relay programs containing control flow, recursion, dynamic shapes, and dynamic allocation, we must change how execution works. The interpreter offers simple solutions for these, but none is sufficiently compelling or optimized. The simplicity of the graph runtime provides attractive properties such as simple serialization, straightforward optimal memory layout, and ease of deployment. An alternative solution to the VM is ahead of time compilation, we discuss below.

Virtual machine (VM) design is a well-studied area in programming languages and systems, and there have been various virtual machine designs for both full-fledged and embedded programing languages. VMs provide are a sweet spot between poorly optimized naive interpreters and full ahead of time compilers. The key advantage of VMs is the flexibility provided by having fine grained control over program execution. For example dynamic scheduling or customized observability is much easier in the virtual machine setting vs. ahead of time compilation.

The Relay VM's design differs from traditional language VMs. Traditional language VM designs have been heavily tailored to the execution profile of traditional programs. Traditional programs manipulate small scalar values and consist of a large number of low-level instructions. The sheer quantity of instructions requires instruction execution and dispatch to be extremely efficient. Any instructions that don't directly correspond to the high-level instruction, such as managing VM state, or dynamic type tests are overhead. In the context of machine learning we manipulate primarily tensor values, using a (relatively) low number of high level instructions. ML programs' cost centers are expensive operator invocations, such as GEMM or convolution, over a large input. In this setting invoking the wrong kernel, or invoking a kernel inefficiently is the essential overhead. Due to the execution profile exhibited by ML programs, micro-optimizations present in scalar VMs are dramatically less important, and thus the removal of dispatch overhead from ahead of time compilation is also less important.

6.3.2 VM Compiler

In order to execute on the VM we wrote a new compiler which can lower Relay directly on to the VM bytecode, and then executed. The compiler performs a set of transformations on the high-level Relay program before generating code:

- A-Normal Form, converts program in to a limited single-assignment form.

- Lambda Lift, converts inline functions into top-level definitions, ensuring that capture lists are now explicit.
- Inline Primitives, ensures that fused functions are inlined into the program to enables simplified code generation.
- Inliner, general function inlining.
- Constant Pool Layout, traverse program collecting all constant values and layout them out in memory.
- ADT Tag Allocation, allocate the tag assignment for compilation to the VM.

6.3.3 VM ISA

After performing the above optimization pipeline the VM compiler itself is relatively straightforward. The transformed program has a nearly one to one correspondence with the VM ISA. The VM ISA is detailed in detail in the above figure.

The Relay dialect we transform the program into is designed to closely match the ISA, as discussed in Chapter 5. The VM's ISA is motivated by our previous observation that kernel execution dominates neural network execution time. If we treat kernel invocation as a single instruction, the cost of surrounding instructions is negligible in the total execution. As a result, our design is quite different from traditional language virtual machines, which contain many instructions that perform little work, leading to a profile where the cost of each instruction executed matters. Our ISA is composed of CISC-style instructions in which each instruction corresponds to a primitive IR expression on tensors, such as allocation and kernel invocation, which in turn may correspond to executing multiple "low-level" operations. For example, `LoadConst idx, $reg` is capable of multiple addressing modes as it first reads the index `idx` and then loads the data from a constant pool to the destination register `$reg`. A complete list of instruction set can be found in the appendices. We naturally select a register-based virtual machine design [Davis et al., 2003] for compact a bytecode, which is easy for users to read and modify. We provide the abstraction of an infinite set of virtual registers as it significantly simplifies optimizations and allocation (similar to SSA) and minimizes conceptual barriers to rapid prototyping and modification.

Instructions are represented using a traditional tagged union containing the op-code and the data payload. This representation enables both efficient serialization and instruction decoding and dispatch. Relay uses variable-length instruction format due to the

Instruction	Description
Move	Moves data from one register to another.
Ret	the object in register result to caller's register.
Invoke	Invokes a function at an index.
InvokeClosure	Invokes a Relay closure.
InvokePacked	Invokes the function including operator kernel.
AllocStorage	Allocates a storage block.
AllocTensor	Allocates a tensor value of a certain shape.
AllocTensorReg	Allocates a tensor based on a register.
AllocADT.	Allocates a data type using the entries from a register.
AllocClosure	Allocates a closure.
GetTag	Gets the tag of an Algebraic Data Types (`ADT`) cstr.
GetField	Gets the value at a certain index from an VM object.
ReshapeTensor	Changes the shape of a tensor without altering its data.
If	Jumps to the true or false offset with a condition.
Goto	Unconditionally jumps to an offset.
LoadConst	Loads a constant at an index from the constant pool.
LoadConsti	Loads a constant immediate.
DeviceCopy	Copies a chunk of data from one device to another.
ShapeOf	Retrieves the shape of a tensor.
Fatal	Raises fatal in the VM.

Table 6.1: The opcode and the description of the Relay instruction set

inclusion of variable sized operands such as data shapes in the instructions. This design has the following benefits. First, both CISC instructions and variable length encoding contribute to better code density. This is a significant advantage for edge devices that only have limited resources. Second, allowing multiple addressing modes to execute a single instruction can reduce the amount of data fetched from cache hierarchy and main memory. It may also lead to better spatial locality as the data (e.g. the tensor value) may remain in the cache. Third, a variable-length instruction encoding paves the way for extending extra information to instructions, e.g. debugging and

even branch prediction. Last but not least, the instruction designed in Relay effectively separates hardware-dependent kernels from model control logic. The Relay bytecode is hardware-independent which eases bytecode serialization, and can be paired with hardware-dependent kernels being invoked by the `InvokePacked` instruction.

6.3.4 VM Interpreter

```
void RunLoop() {
  this->pc = 0;
  Index frame_start = frames.size();
  while (true) {
  main_loop:
    auto const& instr = this->code[this->pc];
    switch (instr.op) {
      case Opcode::LoadConst: {
        auto constant_obj = constants[instr.kidx];
        // ...
        RegWrite(instr.dst, const_pool_[instr.kidx]);
        pc++;
        goto main_loop;
      }
      case Opcode::Invoke: {
        // Prepare args and then invoke.
        InvokeGlobal(functions[instr.func_idx], args);
        frames.back().caller_ret_reg = instr.dst;
        goto main_loop;
      }
      case Opcode::InvokePacked: {
        // Invoke primitive functions
        const auto& func = packed_funcs[instr.pidx];
        const auto& arity = instr.arity;
        // Read args from the register file.
        InvokePacked(instr.pidx, func, arity,
                     instr.osize, args);
        // Write outputs to the register file.
        pc++;
        goto main_loop;
      }
      // Other opcodes are omitted
    }
  }
}
```

Figure 6.1: An excerpt from the VM dispatch loop.

The VM compiler generates a VM executable, a serialized combination of both target specific kernels and the target independent bytecode. We can load a VM interpreter (which interprets the bytecode, not the program) from the executable. A VM interpreter can then be used to invoke any compiled VM functions directly. When a VM function is invoked, execution begins, and enters the dispatch loop. The dispatch loop checks the op-code and executes the appropriate logic, then repeats. As our instructions are coarse-grained (i.e. they can be viewed as super-instructions), the number of branches

generated by the dispatch-loop is lower than traditional programming language VMs, adding negligible overhead compared to ahead of time compilation.

VM uses a tagged object representation reminiscent of those used by programming languages such as Haskell, and OCaml. The tagged object representation smoothly integrates with various data structures, including tensors, algebraic data types, and closures. Due to the specialized object representation, VM instructions only need to interact with the coarse-grained data (i.e. tensors) requiring infrequent memory allocation in chunks.

In sum, the interpreter handles instructions in the following categories.

- Register-to-Register Operations. Register-to-Register operations, e.g. `Move`, i transfers data between different offset of the register file. Objects are reference counted, make use of copy-on-write and passed by reference ensuring register operations are cheap even if the size of underlying container is large.

- Memory Operations. Memory operations can allocate space for tensors, load constant tensors, and so on. Due the design of our constant pool, weights (which are constant during inference) can remain in-memory with no specialized support they can be referenced by the`LoadConst` instruction.

- Call Operations. Call operations are the most frequently executed instructions. The ISA has specialized call instructions for invoking a global function, a kernel primitive, closure, copying data across devices, reshaping runtime tensors, and calculating the shape of tensors. Kernel primitives are ahead-of-time compiled through and can leverage both compiler-generated kernels and the third-party libraries.

- Control Flow Operations. Unconditional jump instructions, e.g. `ret`, are used by both static and dynamic models to jump to a specific program point. Only dynamic models need conditional control operations to determine the direction of branching. The interpreter updates the PC using the offset from either the true branch or false branch based on the conditional value.

6.3.5 Shape Function

The introduction of Any dimension invalidates the pre-allocation mechanism adopted in the existing deep learning compiler. Instead, we now have to track the amount of memory required to be allocated in parallel to computing. Furthermore, static type checking cannot

eliminate all type errors at compile-time due to dynamic tensor shapes. Consequently, we define a *shape function* to compute the output shape for storage allocation and verify the type relation in accord with the semantics of every operator. The shape function is similar in structure to the type relations described in section 3.2 but are present at runtime instead of compile-time. It enables compiling and embedding the computation of output shapes into the program.

According to the characteristics of the operator, we divide the shape functions in three different modes: data independent, data dependent, and upper bound. *Data independent* shape functions are used for operators in which the output shape only depends on the shapes of inputs such as normal 2-D convolution. *Data dependent* shape functions require the concrete input values to compute the output shapes. For example, the output shape of `arange` depends on the value of start, stop, and step. In addition, there are certain operators such as Non Maximum Suppression (`nms`) where the complexity of computing the output shapes is on par with the complexity of executing the operator itself. In order to avoid the redundant computation, we use an *upper bound* shape function to quickly estimate an upper bound shape for the output. We also require such operators to return the output shape along with output value, so as to use the real shape to slice the output tensors into precise output shape and layout.

It is worth noting that in the presence of dynamic shape functions, operator fusion needs to be specially taken care of. Operator fusion, which combines *basic operators* into a *composite operator*, is a critical technique for performance optimization as it reduces unnecessary memory copies and improves the cache locality. However, we only define the shape function at elementary operator level. As a result, we also must fuse shape functions in parallel with the operator fusion. The compiler can easily connect the shape functions of basic operators to form the shape function for a composite operator when all shape functions are data independent. However, a basic operator with a data dependent or upper bound shape function cannot be fused to other operators, i.e., taking the outputs of other operators as its inputs to fuse together, as the shape function requires to access to the intermediate result within a composite operator. As a result, we explicitly define the fusion policy to prevent this from happening.

6.3.6 Ahead-of-time

Given we s from our abstract machine into machine code. But due to the granularity of the operations, dispatch time makes up a very small portion of the execution time. More importantly, the VM provides flexibility traditionally attributed to virtual machines and

a clear compiler/runtime split. We see the potential of VM to be integrated as a runtime module into a larger system. For example, VM can provide resource isolation where multiple inference instances share the same hardware in the cloud. Furthermore, a Quality of Service (QoS)-aware system, e.g., [Kang et al., 2018, Yachir et al., 2009], could leverage VM to pause the current model execution for a higher priority or time-critical model. Last, because of the simplicity of the VM design, one can verify the implementation of VM for security and privacy purposes. Although we don't see AoT as an essential ingredient yet there have are some desire to explore AoT compilation for devices such as micro controllers and is a future direction to explore.

6.4 Evaluation

This section evaluates the performance of Relay on dynamic models against existing state-of-the-art solutions, as well as discussing the role of the optimizations performed by Relay. Specifically, the section seeks to answer the following questions:

1. What is the overall performance of Relay for dynamic models when compared against state-of-the-art alternatives on various hardware platforms?
2. How much overhead does Relay VM introduce for handling dynamism at runtime?
3. How effective are the proposed optimization techniques, such as memory planning and symbolic codegen?

6.4.1 Experiment setup

All experiments were conducted on Amazon EC2 instances. We evaluated Relay on three hardware platforms: Intel Skylake CPUs (c5.9xlarge, 18 physical cores, hereinafter called *Intel CPU*), Nvidia Tesla T4 GPUs (g4dn.4xlarge, 1 card, 2,560 CUDA cores, hereinafter called *Nvidia GPU*), and ARM Cortex A72 (a1.4xlarge, 16 physical cores, hereinafter called *ARM CPU*). Although all tests are done on the cloud, our results of ARM CPU are portable to the edge devices, e.g. Raspberry Pi, due to the same architecture.

To study the efficiency of Relay in handling dynamic models, we compared it with mainstream deep learning frameworks, including TensorFlow (v1.15), MXNet (v1.6), PyTorch (v1.5) [3], as well as dynamic-specific systems TensorFlow Fold based on TensorFlow v1.0. We were unable to compare Relay with Cavs [Xu et al., 2018], JANUS [Jeong et al., 2019], or Jeong et al.[Jeong et al., 2018] as

[3] We use PyTorch v1.4 on ARM CPU because PyTorch v1.5 fails to build on ARM instance.

none of them is open-source. No public deep learning compiler has claimed support for dynamic models.

Three popular models that represent different classes of dynamism were chosen in this experiment, viz. LSTM [Hochreiter and Schmidhuber, 1997] (dynamic control flow), Tree-LSTM [Tai et al., 2015] (dynamic data structure), and BERT [Devlin et al., 2018] (dynamic data shape). The input size / hidden size used in the LSTM and Tree-LSTM model are 300/512 and 300/150, respectively. We used BERT base implementation. For LSTM and BERT, we used Microsoft Research's Paraphrase Corpus (MRPC) [Dolan et al., 2005] with variable input lengths as our input dataset. For Tree-LSTM, we used the Stanford Sentiment Treebank (SST) [Socher et al., 2013] with various tree structures as the input dataset.

6.4.2 Overall performance

We compare the overall performance of Relay against baselines for each dynamic models. Relay successfully accomplished inference for all models on all platforms. However, not all baseline systems could perform inference for these models. For instance, TensorFlow Fold was not designed to process LSTM and BERT hence no result was obtainable, and Tree-LSTM only runs on PyTorch and TensorFlow Fold as other frameworks cannot handle dynamic data structures. Finally the model inference of Tree-LSTM on Nvidia GPU was omitted as it's hard to saturate GPU compute capability due to too many control flows and its model size, making GPUs less favorable deployment targets.

The baseline systems all make use of third-party kernel libraries to achieve high-performance by leveraging the heavily hand-optimized operators. We observe that dynamic models are often well-optimized on a single platform but perform poorly in other frameworks or on other targets. However, Relay has the ability to select either the self-compiled kernels or the ones provided by third-party library based on which one maximizes performance. It uses dynamic dispatch logic to invoke the selected kernels using platform-independent bytecode at runtime. This enables Relay to deliver portable and consistent results as many compiler optimizations are platform agnostic.

First, the latency results of Relay, MXNet, PyTorch, and TensorFlow on LSTM are shown in Table 6.2. Relay consistently outperforms the baseline on both 1- and 2-layer cases. For example, it reduces the latency of 1-layer LSTM model inference by 1.7×, 4.5×, and 6.3× over PyTorch, MXNet, and TensorFlow on Intel CPU, and 1.2×, 1.5×, 3.3× on Nvidia GPU, respectively. On ARM CPU, Relay decreases the latency numbers even more remarkably, i.e. 9.5× over

Unit: μs/token	1 layer Intel	NV	ARM	2 layers Intel	NV	ARM
Relay	**47.8**	**93.0**	**182.2**	**97.2**	**150.9**	**686.4**
PT	79.3	110.3	1729.5	158.1	214.6	3378.1
MX	212.9	135.7	3695.9	401.7	223.8	7768.0
TF	301.4	304.7	978.3	687.3	406.9	2192.8

Table 6.2: LSTM model inference latency of Relay, PyTorch (PT), MXNet (MX), and TensorFlow (TF) on Intel CPU, Nvidia (NV) GPU, and ARM CPU.

Unit: μs/token	Intel	ARM
Relay	**40.3**	**86.3**
PyTorch	701.6	1717.1
TF Fold	209.9	–

Table 6.3: Tree-LSTM model inference latency on Intel CPU and ARM CPU. TensorFlow Fold was not built successfully on ARM CPU.

PyTorch, 20.3× over MXNet, and 5.4× over TensorFlow, respectively. The similar trend applies to 2-layer case of the LSTM model. We observe that latency on Nvidia GPU is higher than Intel CPU. This is because the size of LSTM model is relative small so that it cannot fully utilize the massive parallelism in the GPU. The significant performance improvement is due to Relay encoding the control flow into platform-independent instructions that have minimal overhead while deep learning frameworks use control flow specific primitives to process the sequence, which introduces a large performance penalty.

Next, we inspect the performance of model inference on Tree-LSTM as exhibited in Table 6.3 by comparing Relay with PyTorch and TensorFlow Fold. The table shows that Relay runs substantially faster than the baselines. On PyTorch, the performance speedups are 17.4× on Intel CPU and 19.8× on ARM CPU as PyTorch uses Python to handle the tree data structure. TensorFlow Fold is 5.2× slower than Relay on Intel CPU because it has to re-compile upon every input.

Third, Table 6.4 summarizes the performance of BERT for Relay, MXNet, and TensorFlow. The results indicate that Relay outstrips the baselines for all frameworks on all platforms in the experiment. The reduction in latency compared to the best framework on each platform is 1.5×, 1.05×, and 1.3× on Intel CPU, ARM CPU, and Nvidia GPU, respectively. The reasons are two-fold: (a) similar to frameworks, Relay is also able to use the well-tuned third-party libraries on Intel CPU (MKL) and Nvidia GPU (cuDNN). (b) Relay can further enjoy the benefit of powerful operator fusion brought by the deep learning compiler. One can observe that we obtained more speedups on the ARM CPU for PyTorch and MXNet as the third-party libraries performed less favorable. However, Relay is only slightly faster than TensorFlow on the ARM CPU. This is because the dense operators (contributing to more than 90% of the overall latency in Bert) on the ARM CPU was not well optimized by the underlying compiler. Therefore, the performance of the combination of operators

Unit: μs/token	Intel	Nvidia	ARM
Relay	**307.0**	**95.2**	**2862.6**
PyTorch	479.5	220.4	11851.2
MXNet	455.8	152.9	8628.0
TensorFlow	768.7	125.2	2995.4

Table 6.4: BERT model inference latency on Intel CPU, Nvidia GPU, and ARM CPU.

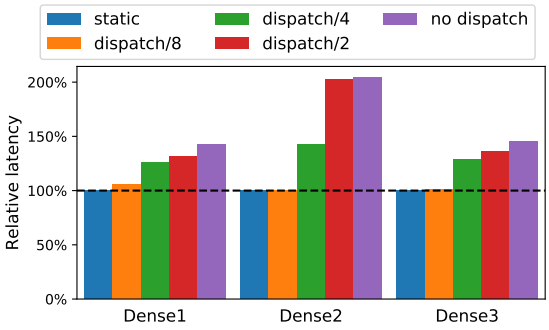

Figure 6.2: Relative latency comparison between symbolic codegen and static codegen of 3 dense operators on ARM CPU. The latency of kernel compiled with static shapes is used as the baseline. "dispatch/k" indicates that we generate k symbolic kernels to be dispatched at runtime. "no dispatch" means that only one symbolic kernel is generated and therefore no dispatching is needed.

it selected is on par with the ones used by TensorFlow.

In sum, the evaluation results demonstrate that Relay produces more portable performance for all dynamic models on different platforms. Instead, the performance of frameworks is more platform dependent and varies from model to model.

6.4.3 *Microbenchmarks*

This section analyzes the performance gain of Relay by using BERT as the microbenchmark. Three studies will be conducted to examine (a) the overhead introduced by the VM, (b) the advantage of the proposed memory planning pass, and (c) the performance discrepancy between symbolic and static codegen.

Overhead in handling dynamism: In order to understand the overhead that Relay spends to take care of dynamism, we compared it to TVM where static sequence length and TVM static runtime is used to execute BERT.

6.5 *Supporting Hardware Accelerators*

Although not the main focus of this thesis, much our work on Relay was focused on supporting hardware accelerators. Below we briefly touch on two important use cases of Relay in accelerator compilers.

6.5.1 *Bring Your Own Code Generation*

In conjunction with my collaborators at AWS we implemented an extensible framework for hooking hardware accelerators into TVM.

Bring Your Own Code Generation (BYOC) is a mechanism introduced into Relay for offloading specific sub-programs to hardware accelerators which operate outside of the traditional TVM programming model. This framework is under active development and we plan on writing a paper detailing its design and implementation after the publication of this thesis. The key contributions are a framework which uses "dataflow" patterns to describe hardware capabilities that can be offloaded to hardware accelerators such as ARM's EthosN device, or Amazon's Inferentia. For example these patterns can be used to describe fusion capabilities or simply what is supported by a device enabling smooth heterogenous execution for devices which don't have a low-level TVM backend.

6.5.2 VTA

Hardware specialization is a powerful way to accelerate a known set of applications and workloads. A component of Relay is lowering high-level programs down to the bespoke semantics of emerging hardware accelerators. Unfortunately, deep learning (DL) is anything but a static field, and the machine learning (ML) community rapidly changes how they use to write models, the architecture of models themselves, the operators used by said models, and the data types they operate over. Initial programmable accelerators [Jouppi et al., 2017] offer potentially huge performance improvements at the cost of complex specialized compilation. Furthermore the churn of machine learning has lead to an interest in customizable designs, with features such as new numeric representations, new hardware engines, and more. In order to customize the behavior of accelerators designs, even when open-sourced, there is a need for the availability of a transparent and modular software stack. An end-to-end approach requires integration of frameworks, systems, compilers, and architecture in order to execute state-of-the-art ML using hardware acceleration. Peak FLOPs provide value only if a programmer can access them. In order to tackle this problem I have collaborated on the design for VTA (Versatile Tensor Accelerator), an explicitly programmed accelerator paired with a compiler and runtime that can evolve in tandem with deep learning models without sacrificing the advantages of specialization.

VTA makes following contributions:

- *A programmable accelerator design* that exposes a two-level programming interface: a high-level task ISA to allow explicit task scheduling by the compiler stack, and a low-level microcode ISA to provide software-defined operational flexibility. In addition, the VTA architecture is fully parameterizable: the hardware in-

trinsics, memories, and data types can be customized to adapt the hardware backend requirements.

- *An extensible runtime system* for heterogeneous execution that performs JIT compilation of microcoded kernels to provide operational flexibility. For example, the VTA runtime lets us extend the functionality of VTA's original computer-vision-centric design to support operators found in style transfer applications without requiring any hardware modifications.
- *A schedule auto-tuning platform* that optimizes data access and reuse in order to rapidly adapt to changes to the underlying hardware and to workload diversity.

My collaborators and I published a paper on VTA in the IEEE Micro Journal Special Issue on Deep Learning Acceleration [Moreau et al., 2019]. VTA and its children projects are a critical part of the research agenda at UW SAMPL lab, and we won a multi-million dollar grant to pursue automatically mapping Relay programs to hardware designs. This work is being continued at UW and it is one exciting future direction we discuss in Chapter 7.

Future Work

This thesis demonstrates a principled approach to optimizing dynamic neural networks that generalizes the performance enjoyed by static neural networks to a greater number of networks. There are still open questions on how to optimize and compile generic tensor programs to arbitrary hardware targets. TVM is continuing to evolve rapidly with many changes in the past year not detailed in this thesis. In particular completing the IR unification, improved dynamic model support, training support, automatic scheduling for new hardware targets and much more. The remainder of this section touches on some of the challenges that lie in front of this work.

7.0.1 Unified IR

TVM has been working on unifying its multiple levels of IR into a family of dialects which can invoke functions at different level of abstractions. The start of this work is complete but using the new unified world to meaningfully improve optimizations is something that has not yet been demonstrated. For example one could lower an entire Relay program to a single GPU kernel using the new machinery, but no one has attempted such an optimization yet.

7.0.2 Going further with Dynamic Models

There are still problems which are unsolved in Relay's current iteration. There are analyses that can be further improved such as recover further static shape information for optimization using a more precise analysis. Or features not yet supported such as dynamically ranked tensors. There are still interesting challenges to solve here as models evolve and demand more from compilers.

7.0.3 Training Support

A team at OctoML and AMD have begun work on using TVM to accelerate training by building a framework that lowers all computation to Relay which can make use of all the techniques described in

this thesis. Although the work just began we were able to get models working on AMD GPUS with no custom code needed a feat most frameworks have failed to achieve. There significant work to make this a state of the art competitive approach with existing frameworks but an interesting area going forward as TVM allows training to be deployed to any supported device.

7.0.4 Automatic Scheduling

A final promising area being explored at OctoML is automatic scheduling of tensor programs, including tensorization. Originally in TVM a user must provide the network, operator definitions, and schedules. AutoTVM introduced the ability to define a template schedule where some parameters such as tiling factor, or loop split can be learned using ML. Ansor then introduced ability to learn both the template and the parameters using ML guided search. The finally vision of these systems is to allow a Relay program to be lowered to TVM's TIR and then scheduled automatically allowing users to interpolate between manually knowledge and fully automatically. This would enable users to leverage the expressivity of Relay, then lower to TIR, and finally rewrite the code to achieve near optimal performance using ML guided search.

7.0.5 Conclusion

These directions represent potentially years more work, and will be hopefully realized by collaborators PhD theses, future publications, the open source community and development teams at OctoML. The work done in this this presents a compelling foundation on which to build these future efforts.

Bibliography

Fortran recursion notes, 2020. URL http://www.ibiblio.org/pub/languages/fortran/ch1-12.html.

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. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, September 1993.

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

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

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.

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.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax, 2018.

Oliver Breuleux and Bart van Merriënboer. Automatic differentiation in myia. 2017.

Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K. Duvenaud. Neural ordinary differential equations. *CoRR*, abs/1806.07366, 2018a.

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.

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, 2018b. USENIX Association. ISBN 978-1-931971-47-8.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018c.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

Intel Corporation. Plaidml. `https://www.intel.ai/plaidml/`, 2017.

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.

Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 41–49. ACM, 2003.

GCC Developers. Gcc, the gnu compiler collection. https://gcc.gnu.org/, 2019.

PyTorch Developers. Pytorch. https://pytorch.org/, 2018.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Bill Dolan, Chris Brockett, and Chris Quirk. Microsoft research paraphrase corpus. *Retrieved March*, 29(2008):63, 2005.

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. ISBN 978-1-4503-5834-7. DOI: 10.1145/3211346.3211354.

Conal Elliott. Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*, 2009.

J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, June 2018. DOI: 10.1109/ISCA.2018.00012.

Josh Fromm, Bing Xu, Morgan Funtowicz, and Jason Knight. using-sparsity-in-apache-tvm-to-halve-your-cloud-bill-for-nl, 2020. URL https://medium.com/octoml/using-sparsity-in-apache-tvm-to-halve-your-cloud-bill-for-nlp-4964eb1ce4f2.

Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, page 31. ACM, 2018.

Gluon Team. Gluon. https://gluon.mxnet.io, 2018.

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

Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay D. Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *CoRR*, abs/1312.6082, 2013.

Google. Tensorflow lite supported datatypes. https://www.tensorflow.org/lite/guide/ops_compatibilitysupported_types, 2019.

Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

J.L. Gustafson. *The End of Error: Unum Computing*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2015. ISBN 9781482239867.

Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data, 2009. URL https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35179.pdf.

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

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag. ISBN 3-540-54262-0.

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.

Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018a.

Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018b. DOI: 10.21105/joss.00602.

Mike Innes, David Barber, Tim Besard, James Bradburyand Valentin Churavy, Simon Danisch, Alan Edelman, Stefan Karpinski, Jon Malmaud, Jarrett Revels, Viral Shah, Pontus Stenetorp, and Deniz Yuret. On machine learning and programming languages. https://julialang.org/blog/2017/12/ml&pl, 2017.

Intel. Intel math kernel library for deep neural networks (intel mkl-dnn). https://github.com/intel/mkl-dnn, 2018. [Online; accessed 13-May-2019].

Eunji Jeong, Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, and Byung-Gon Chun. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 453–468, 2019.

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.

Jeff Johnson. Rethinking floating point for deep learning.

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.

JuliaLang Team. The julia programming language. https://julialang.org, 2018.

Lei Kang, Wei Zhao, Bozhao Qi, and Suman Banerjee. Augmenting self-driving with remote control: Challenges and directions. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, page 19–24, 2018.

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

Edward Kmett, Barak Pearlmutter, and Jeffrey Mark Siskind. ad: Automatic differentiation. https://github.com/ekmett/ad, 2008.

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

Chris Lattner and Vikram 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*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

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

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. DOI: 10.1109/AICAS.2019.8771510.

Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable Architecture Search. *ArXiv e-prints*, June 2018.

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

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 2017a.

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

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. ISBN 978-1-4503-2326-0. DOI: 10.1145/2500365.2500601.

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

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.

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. ISSN 0272-1732. DOI: 10.1109/MM.2019.2928962.

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. ISSN 0163-5824. DOI: 10.1145/1093410.1093411.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dynamic neural network toolkit. https://arxiv.org/abs/1701.03980.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

OpenAI. Openai five. `https://blog.openai.com/openai-five/`, 2018.

Oracle. The java®virtual machine specification. `https://docs.oracle.com/javase/specs/jvms/se7/html/index.html/`, 2013.

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.

Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2):7:1–7:36, March 2008. ISSN 0164-0925. DOI: 10.1145/1330017.1330018.

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

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

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. ISBN 978-1-4503-2014-6. DOI: 10.1145/2491956.2462176.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *Lecture Notes in Computer Science*, page 525–542, 2016. ISSN 1611-3349.

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

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.

Erik Sandewall. A proposed solution to the funarg problem. *SIGSAM Bull.*, (17):29–42, January 1971. ISSN 0163-5824. DOI: 10.1145/1093420.1093422.

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.

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.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *In Scheme and Functional Programming Workshop*, pages 81–92, 2006.

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

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

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. ISBN 978-1-5090-4931-8.

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.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

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.

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.

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

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

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

ThoughtWorks Inc. Deepdarkfantasy - a programming language for deep learning. https://github.com/ThoughtWorksInc/DeepDarkFantasy, 2018a.

ThoughtWorks Inc. Deeplearning.scala. https://github.com/ThoughtWorksInc/DeepLearning.scala, 2018b.

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.

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

Fengbin Tu. Neural networks on silicon. https://github.com/fengbintu/Neural-Networks-on-Silicon, October 2018.

UCSB ArchLab. Opentpu. https://github.com/UCSBarchlab/OpenTPU.

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.

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. https://arxiv.org/abs/1802.04730, 2018.

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.

Mu Wang and Alex Pothen. An overview of high order reverse mode. 2017.

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

Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 937–950, 2018.

A. Yachir, K. Tari, Y. Amirat, A. Chibani, and N. Badache. Qos based framework for ubiquitous robotic services composition. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2019–2026, 2009.

Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, page 18. ACM, 2018.

Xianyi Zhang, Qian Wang, and Zaheer Chothia. Openblas. http://xianyi.github.io/OpenBLAS, 2014. [Online; accessed 13-May-2019].

Da Zheng. Optimize dynamic neural network models with control flow operators, 7 2018. URL https://cwiki.apache.org/confluence/display/MXNET/Optimize+dynamic+neural+network+models+with+control+flow+operators.

Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.