DLVM: A MODERN COMPILER FRAMEWORK FOR NEURAL NETWORK DSLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Many current approaches to deep learning make use of high-level toolkits such as TensorFlow, Torch, or Caffe. Toolkits such as Caffe have a layer-based programming framework with hard-coded gradients specified for each layer type, making research using novel layer types problematic. Toolkits such as TensorFlow and Torch define a computation graph in a host language such as Python, then perform run-time interpretation over that graph, storing the results of forward calculations and reverse-accumulated gradients at each node. This approach is more flexible, but these toolkits take a very limited and ad-hoc approach to performing optimization. Also problematic are the facts that most toolkits lack type safety, and target only a single (usually GPU) architecture, limiting users' abilities to make use of heterogeneous and emerging hardware architectures. We introduce a novel framework for high-level programming that addresses all of the above shortcomings.

1 INTRODUCTION

Algorithmic differentiation (AD), also known as automatic differentiation, encompasses a family of a well-known techniques for algorithmically obtaining the derivatives of a function $F : \mathbf{x} \in \mathbb{R}^n \rightarrow \mathbf{y} \in \mathbb{R}^m$ (Naumann, 2011). The function F can be represented as a directed acyclic computation graph representing the composition of elementary computational operations for which the respective derivatives are well known. The partial derivative $\frac{\partial y_j}{\partial x_i}$ can be computed through recursive applications of the chain rule, either in the forward direction (corresponding to a bottom-up traversal of the computation graph) or in the backward direction (corresponding to a top-down traversal of the computation graph). The former approach, called forward-mode or tangent-mode AD, is used in some research communities, most commonly when $m \gg n$ (Goodfellow et al., 2016). The latter approach, which includes the back-propagation algorithm (Rumelhart et al., 1986) as a special case, is called reverse-mode or adjoint-mode AD, and encompasses the techniques most commonly used for training the weights in neural networks.

Within the deep learning community, most current approaches to neural networks make use of high-level toolkits such as TensorFlow (Abadi et al., 2016) or Torch (Collobert et al., 2011). In these toolkits, users define a computation graph in a host language such as Python, where each node in the graph typically represents a linear algebra operation parallelized as a compute kernel. In the reverse-mode AD approach taken by these toolkits, runtime interpretation is performed over the computation graph, first in a bottom-up pass to calculate the value of each composed function, and then in a top-down pass to calculate the reverse-accumulated gradient values. This approach is equivalent to the adjoint calculation through operator overloading used by some in the AD research community.

This approach of interpreting computation graphs at runtime fails to provide optimal performance. Users of such toolkits are required to precompile and preinstall all possible kernels that are specialized for every data type and every operator. When kernels are preinstalled, it is not possible to further optimize them based on actual data conditions (for example, when the product of grid size and block size equals the size of an array, boundary checks may be redundant). These toolkits embed control flow nodes and discrete functions in a data flow graph, making the AD data flow unnecessarily ad-hoc. Other issues include latency between kernel launches, failure to optimize constant expressions, and failure to perform kernel fusion. One recent project, the XLA compiler for TensorFlow (Leary & Wang, 2017), has begun to explore some of these optimization issues, and includes the ability to

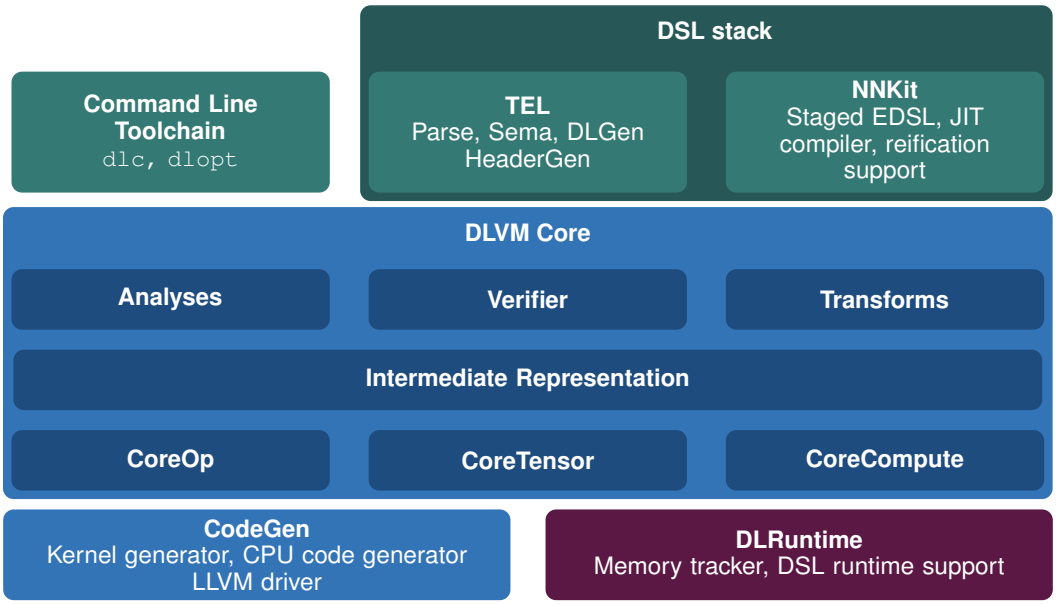

Figure 1: DLVM software stack, written entirely in Swift 4.

perform some types of kernel fusion. The interpreting approach to reverse-mode AD also makes the assumption that all forward pass calculations must be performed prior to any backward pass calculations; when computing gradients that do not require all values from the forward pass, some forward pass computations may never be used. In some cases where the result of a forward pass calculation requires large amounts of memory, an optimal solution may involve recomputing this forward pass result when actually needed by a backward pass calculation rather than holding the result in memory until the backward pass is complete; current deep learning toolkits, however, do not typically allow for such memory release and re-computation.

We introduce a novel framework for high-level programming that addresses shortcomings of such deep learning frameworks. The solution we present differs substantially from these, most notably by compiling the computation graph and its requisite computations into a directly executable binary instead of performing runtime interpretation over a computation graph. Our solution includes (1) a novel domain-specific intermediate representation specifically designed for tensor-intensive programming models such as deep neural networks, (2) principled use of modern compiler back-end optimization techniques to substantially simplify the generated computation graphs, including kernel fusion optimizations and code transformation for algorithmic differentiation, (3) code generation through a mature compiler-backend infrastructure (Lattner & Adve, 2004) that allows for transparent targeting of various hardware, and (4) a novel domain-specific neural network programming language with type safety. In addition, we provide C headers to enable use of this framework in any programming language (Python, C++, Swift, etc) that can interface with C.

2 Related Work

Most existing deep learning toolkits, including Torch (Collobert et al., 2011) and Caffe (Jia et al., 2014), rely on runtime interpretation of high level code, typically in the form of reverse-mode algorithmic differentiation over a runtime-generated computation graph, with some toolkits providing some level of limited optimizations. Moreover, the code representation in nearly all toolkits is a "sea of nodes" representation, embedding control flow nodes and discrete functions in a data flow graph, making the data flow of algorithmic differentiation unnecessarily ad-hoc. In contrast, our approach is designed from the start around the idea that the computation graph defined by a neural network is itself a program, which is best optimized through robust application of mature compiler techniques. We represent this program in SSA form with control flow graph, and perform algorithmic

```
func softmax <K*>(z: [K]) -> [K] {
    let e = exp(z)
    return e / reduce(+, e)
}

float net MNIST {
    W: [784 x 10]
    b: [1 x 10]

    inference(x: [? x 784]) -> [? x 10] {
        softmax(h * W + b)
    }
    loss(actual: [? x 10], reference: [? x 10]) -> [] {
        reduce(mean, (reference - actual)^2)
    }
    optimize<K*>(parameter: [K], gradient: [K],
                 iteration: [], learningRate: []) -> [K] {
        parameter - gradient .* learningRate
    }
}
```

Figure 2: Example code in TEL, a compiled DSL for neural networks.

differentiation (AD), domain-specific optimizations, general-purpose optimizations, compute kernel fusion, and low-level code generation.

XLA (Leary & Wang, 2017) takes a similar approach to ours, compiling TensorFlow graphs to a compiler IR and performing some optimizations. Our domain-specific intermediate representation is much more expressive than XLA's; this enables our approach to support more extensive low-level optimizations such as kernel fusion and loop tiling. Our approach also differs from XLA by representing functions such as `min` and `max` directly through primitive instructions and control flow, which enables us to perform standard AD. Unlike XLA/TensorFlow, we explicitly address the issue of derivative calculation as a compiler problem, generating code for adjoint functions directly as code generation through source code transformation. Importantly, our entire infrastructure was designed from the start around a robust compile-time framework for domain-specific neural network languages, whereas XLA has been adapted around the existing TensorFlow infrastructure with a particular focus on hardware support for Google's Tensor Processing Units (Jouppi et al., 2017).

The other most closely related work is NNVM/TVM (Chen et al., 2017). Our work differs from TVM in providing type safety in our DSLs. Our work differs from NNVM in the design and presence of a textual intermediate representation that supports custom data types and memory semantics. The textual intermediate representation enables robust unit testing via FileCheck. Where TVM and NNVM are built as a DSL and a graph library in Python with a C++ implementation, our work has similar architecture as LLVM and the Swift Intermediate Language (Groff & Lattner, 2015). Unlike TVM/NNVM, we provide command-line utilities for the respective components in our toolchain.

3 DLVM

Deep Learning Virtual Machine (DLVM) is a compiler infrastructure designed for modern deep learning software.[1] DLVM is designed to apply a multi-stage compiler optimization strategy to both high-level linear algebra and low-level parallelism, perform domain-specific transformations and relieve the overhead from front-end languages, and serve as the host for research and development of domain-specific languages (DSLs) for neural networks. The complete DLVM software stack, including two sample front-end neural network DSLs, is shown in Figure 1 on the preceding page.

Figure 3 on the next page illustrates the major stages in the DLVM compilation pipeline. The DVLM compilation stages address algorithmic differentiation, fusion of tensor linear algebra operations, other domain-specific optimizations, and static code generation targeting a variety of devices. The

[1] Code for DLVM and the TEL DSL is available under the Apache 2.0 license at http://anonymous.url

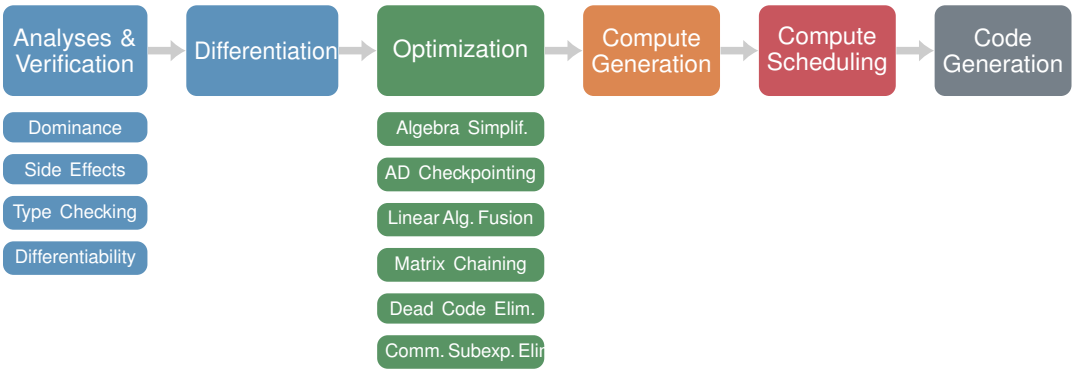

Figure 3: Compilation stages in the DLVM compilation pipeline.

raw and optimizable stages allow constructs for high level tensor operations and various high-level optimizations. The compute and schedule stages allow constructs for low-level array operations lowered from tensor operations in high-level stages, borrowing the design from Halide (Ragan-Kelley et al., 2013).

The DLVM Intermediate Representation (IR) is the core language of the system. It uses static single assignment (SSA) form, high-level types including a first-class tensor type (see Table 2 on page 7), and a set of linear algebra operations combined with a general-purpose instruction set (see Table 1). The system enables a wide variety of domain-specific analyses and transformations, such as reverse-mode algorithmic differentiation, algorithmic differentiation checkpointing, algebra simplification and linear algebra fusion.

To demonstrate how DLVM helps the development of domain-specific languages (DSLs), in Section 3.4 we present one prototypical compiled DSL: the Tensor Expression Language (TEL). TEL features type safety in linear algebra operations, modularity and uniformity, and targets DLVM for transformations, optimizations and low-level code generation.

3.1 DLVM Core

DLVM Core contains essential components for an optimizing compiler: intermediate representation and passes (see Figure 1 on page 2). The DLVM intermediate representation (IR) consists of a virtual instruction set, control flow graph and data flow representation. Passes are functions that traverse the intermediate representation of a program, either producing useful results as analyses of the program (analysis passes), or mutating the program for differentiation and optimizations (transform passes).

3.1.1 A Domain-specific Compiler Intermediate Representation for DLVM

Inspired by the LLVM IR (Lattner & Adve, 2004) and the Swift Intermediate Language (Groff & Lattner, 2015), DLVM IR is a graph-based, modular code representation, with both an in-memory format and a textual format. The code representation has a hierarchy of abstractions: **module**, **function**, **basic block**, and **instruction**. An instruction is the minimal unit of code that operates on values, which can be globals, function arguments or temporary virtual registers produced by instructions. Each module contains a collection of type definitions, global values and functions. Each function has a control flow graph (CFG) formed by basic blocks and control flow edges. Each basic block contains an ordered list of instructions with data dependencies forming a directed acyclic graph, which we call data flow graph (DFG).

The DLVM IR has a high-level type system based on the concept of the **tensor** as a first-class type. As a generalization of an n-dimensional array, a tensor is parameterized by an underlying data type and a shape. The underlying data type of a tensor represents the underlying type of its constituent scalar elements, and can take the value of LLVM IR numeric types, such as `f64` and `i32`. The shape of a tensor is an ordered list of zero or more positive integers. The total number of dimensions in a tensor's shape is the rank of the tensor. Examples of various tensor types are illustrated in Table 2. Most of the first class types present in LLVM IR are also supported by the DLVM IR.

Kind	Example
Element-wise unary	`tanh %a: <10 x f32>`
Element-wise binary	`power %a: <10 x f32>, %b: 2: f32`
Dot	`dot %a: <10 x 20 x f32>, %b: <20 x 2 x f32>`
Concatenate	`concatenate %a: <10 x f32>, %b: <20 x f32> along 0`
Reduce	`reduce %a: <10 x 30 x f32> by add along 1`
Transpose	`transpose %m: <2 x 3 x 4 x 5 x f32>`
Convolution	`convolve %a: <...> kernel %b: <...> stride %c: <...>`
Slice	`slice %a: <10 x 20 x i32> from 1 unto 5`
Random	`random 768 x 10 from 0.0: f32 unto 1.0: f32`
Compare	`greaterThan %a: <10 x 20 x bool>, %b: <1 x 20 x bool>`
Data type cast	`dataTypeCast %x: <10 x f32> to f64`
Function application	`apply %foo(%x: f32, %y: f32): (f32, f32) -> <10 x 10 x f32>`
Branch	`branch 'block_name(%a: i32, %b: i32)`
Conditional	`conditional %cond: bool then 'then_block() else 'else_block()`
Shape cast	`shapeCast %a: <1 x 40 x f32> to 2 x 20`
Allocate stack	`allocateStack $Point count 1`
Allocate heap	`allocateHeap $MNIST count 1`
Deallocate	`deallocate %x: *<10 x i32>`
Store	`store %x: <10 x i32> to %ptr: *<10 x i32>`

Table 1: This table illustrates a selection of the instructions in the DLVM virtual instruction set.

In addition to instructions for memory management, branching, and function application, the DLVM virtual instruction set includes mathematical instructions for element-wise and aggregate operations. These instructions include element-wise unary operations, such as `tanh` and `negate`, element-wise binary operations, such as `add` and `power`, and more complex operations such as `dot`, `transpose`, and `convolve`. A sample of DLVM IR code is shown in Figure 4 on page 7.

The DLVM instruction set does not include composite math functions such as `softmax` and `sigmoid`, or discrete functions based on control flow such as `min` and `max`. All of these functions can be composed of existing math instructions and control flow constructs. This design allows for the standard algorithmic differentiation algorithm to be applied to any differentiable program, with no need for special handling of composite or discrete cases.

3.1.2 Domain-specific Compiler Passes for DLVM

DLVM has a full-fledged pass infrastructure, performing two kinds of transformations: canonicalization and optimization. Canonicalization lowers certain high-level instructions to lower representations that enable more optimizations. Optimization is then performed on the canonical-stage IR, maximizing the code performance. Canonicalization currently includes automatic differentiation, which transforms each gradient instruction into a reference to a newly defined function produced by algorithmic differentiation (see Section 3.1.3 below). Optimizations include domains-specific optimizations, such as Algebra Simplification and Linear Algebra Fusion, and traditional compiler optimizations.

Since DLVM IR is aware of mathematical operators such as `tanh` and `power`, the Algebra Simplification pass can find and simplify certain mathematical operations that are expensive or redundant. For example, x^2 can be simplified to $x \odot x$ ($\odot$ stands for element-wise multiplication), and x^0 can be simplified to constant 1.

Matrix chain ordering is a classic optimization that minimizes the number of sub-operations in a chain of matrix multiplications with different dimensionality, based on matrix multiplication's associativity. Given a data flow with matrix multiplication instructions $\mathbf{M}_1\mathbf{M}_2\mathbf{M}_3 \dots \mathbf{M}_{n-1}\mathbf{M}_n$,

the Matrix Chain Ordering pass re-associates the order of matrix multiplication, for example: $\mathbf{M}_1(((\mathbf{M}_2\mathbf{M}_3)\ldots\mathbf{M}_{n-1})\mathbf{M}_n)$.

Since the DLVM optimizer is aware of linear algebra operations with static dimensionality, maximizing the performance by fusing verbose linear operations into a single matrix multiplication is extremely valuable to large scale networks. For example, it is very common to encounter expressions of the form $\mathbf{Wx} + \mathbf{b}$. When unoptimized, the matrix multiplication and the addition will be parallelized separately. Since launching compute kernels separately can be expensive, DLVM performs Linear Algebra Fusion, which transforms subexpressions involving both matrix multiplication and element-wise operations into a single matrix multiplication instruction on padded tensors. Besides the simple pattern like an addition of matrix multiplication and a vector, we can apply the same approach to a polynomial of multiple matrix multiplications, turning the polynomial into a single matrix multiplication. For example, in a simple recurrent neural network (RNN), each cell of the recurrence is a feed forward neural network that takes two inputs, $\mathbf{x}_t$ the input local to the current timestep, and $\mathbf{h}_t$ the hidden state carried along the recurrence. The Linear Algebra Fusion pass can simplify operations in $\mathbf{h}_t = f(\mathbf{Wx}_{t-1} + \mathbf{Uh}_{t-1} + \mathbf{b})$ from two multiplications and two additions into a single (concatenated) multiplication operation. A more aggressive, interprocedural version of Linear Algebra Fusion can optimize parameter passing and memory allocation, so that the entire concatenated matrix can be created and passed around in the first place without reallocation.

3.1.3 Algorithmic Differentiation through Adjoint Code Generation

The Differentiation pass is responsible for performing reverse-mode algorithmic differentiation. This pass is responsible for generating DLVM IR code that calculates the derivative of a differentiable function. A function is marked as being automatically differentiable via **gradient declarations**. A gradient declaration is a function in a module that is declared with its mathematical relation with another function in the module and no function body. The function `@foo_gradient` in Figure 4 is an example of such a function. The differentiation pass, when applied, canonicalizes every gradient declaration in the module to a normal function definition with a function body. Unlike most of the existing deep learning frameworks, algorithmic differentiation in DLVM is a source code transformation, not interpretation (operator overloading) over the same program. This makes the compiler able to perform optimizations on the gradient computation separately and enables higher order differentiation.

Given a differentiable function $\mathbf{f}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, this pass creates a new function that computes the Jacobian $\mathbf{J}_f$. This approach to AD has several advantages with respect to AD performed by operator overloading / graph interpretation. Unlike operator overloading, the gradient function produced by AD is a standalone function that sits uniformly alongside other functions in an IR module, representationally unrelated to the original function. The generated function takes original inputs and produces a tuple of partial derivatives with respect to the inputs. In AD, not all values in the forward evaluation will be used to compute derivatives. In DLVM, unused forward operations can be easily eliminated by the aggressive dead code elimination (ADCE) pass in the compilation pipeline (see Section 3.1.4). In addition, an AD-specific optimization technique called checkpointing can further reduce memory consumption during gradient computation.

Algorithmic differentiation in DLVM is configurable. Instead of blindly differentiating an output of a function with respect to all input arguments, differentiation can be configured by additional parameters to the gradient instruction. The front-end can choose to differentiate a function with respect to selected arguments, or even to keep some of the outputs of the original function. Our approach to AD is implemented as a transformation from one function to another function. This approach also makes higher-order differentiation possible; it can be accomplished by applying a gradient instruction on the result of another gradient instruction, and the Differentiation pass will canonicalize them by performing AD on the result of AD. In deep learning, it is sometimes desirable to take the second-order gradient of a loss function with respect to neural network parameters in order to observe the rate of gradient descent, eliminate superfluous weights, and estimate confidence intervals for both weights and network outputs (Buntine & Weigend, 1994).

Rank	Type notation	Description
0	`i64`	64-bit integer
1	`<100 x f32>`	float vector of size 100
2	`<100 x 300 x f64>`	double matrix size 100x300
n	`<100 x 300 x ... x bool>`	rank-n tensor

Table 2: Tensors are supported as first-class types in the DLVM intermediate representation.

```
func @foo: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) ->
                                                        <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %v0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %v1 = add %v0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %v1: <1 x 10 x f32>
}

[gradient @foo wrt 1, 2]
func @foo_gradient: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                   -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

Figure 4: Example code in DLVM intermediate representation.

3.1.4 General-purpose Compiler Passes for DLVM

General-purpose optimizations refer to traditional optimizations applied to DLVM IR. These optimizations are important at the DLVM stage in the compilation pipeline, since linear algebra computation can be highly optimized or eliminated before they get lowered to LLVM IR which contain parallel execution and low-level information that prevent LLVM optimizations from identifying high-level patterns. These optimizations include Aggressive Dead Code Elimination (ADCE), Common Subexpression Elimination (CSE), and Sparse Conditional Constant Propagation (SCCP).

Reducing expensive operations such as matrix multiplication and convolution are critical to performance. Such optimizations are not feasible in other approaches to algorithmic differentiation that use graph interpretation / operator overloading, because the forward pass and the backward pass are tied together in a single graph; mutating either evaluation pass will alter the semantics of the other.

3.2 Code generation

Two major design goals of DLVM are the ability to target multiple heterogenous parallel architectures from the same front-end DSL code (and corresponding DLVM IR), and the ability to perform aggressive optimizations on parallel data flow.

In order to attain these goals, DLVM code generation transforms DLVM IR into the lower-level intermediate representation of an existing compiler infrastructure. LLVM (Lattner & Adve, 2004) is a robust and mature compiler infrastructure with multiple hardware backends, including NVIDIA GPUs. The current implementation of DLVM targets LLVM IR. Many high-level DLVM IR linear algebra instructions over (potentially high-dimensional) tensors abstract lower-level operations; for example, the `dot` operation abstracts a loop operation, scalar multiplication, and scalar addition. If a DLVM program does not use any mathematical operation on high-dimensional tensors, then code generation can directly lower DLVM IR into LLVM IR with few changes. But, when a DLVM program does use high-dimensional operations that can be parallelized, our code generation module generates a data flow graph (DFG), which is then ultimately transformed into LLVM IR.

Finally, existing LLVM utilities are used to transform the generated LLVM IR into the final executable binary (or in some cases, binaries). In order to take full advantage of a variety of emerging heterogeneous parallel architectures, we plan for future versions of DLVM to target the IR of HPVM (Srivastava et al., 2016), a higher-level heterogeneous parallel-aware compiler architecture that allows for transparent targeting of diverse hardware.

3.3 DLVM COMMAND LINE TOOLCHAIN

The front-end software associated with each domain-specific programming language (see Section 3.4) is responsible for generating a DLVM IR for a given source language program to be compiled. The DLVM core compiler infrastructure itself is a compiler from DLVM IR to LLVM IR, therefore having a command line toolchain is necessary for optimizing DLVM IR and for transforming DLVM IR into LLVM IR. Unlike NNVM/TVM, which only provide a Python/C++ interface to their users, we provide traditional command line utilities for that implement the above functionality.

The DLVM optimizer is implemented as `dlopt`, which accepts `*.dl` IR files and applies transformation passes on them. The DLVM compiler is implemented as `dlc`, which accepts `*.dl` IR files and generates corresponding output files; users can specify what level of output they want to produce, and the driver invokes the core library to compile the input IR. Because the textual representation of the DLVM IR is available, the DLVM framework can easily make use of LLVM's `FileCheck` utility to perform robust unit testing.

3.4 NEURAL NETWORK DSLS

A major goal of DLVM is to simplify the development of neural network DSLs. A well-designed neural network DSL should support the needs of deep learning researchers by providing a safe environment for rapid prototyping that frees the developer from low-level tasks such as manual memory management and hand-coding gradients for novel activation functions and novel layer types, while simultaneously supporting the generation of highly efficient code for training and inference.

In our initial release of DLVM, we provide one such DSL, both as a proof-of-concept and as a reference implementation that showcases the capabilities of DLVM as a platform for neural network DSL development.

The Tensor Expression Language (TEL) is a compiled prototypical DSL for building feed-forward neural networks, designed to address type safety, modularity, and built-in neural network semantics. The TEL type system is based around tensors, with a generic type system used to specify tensor shapes. Composed operators and simple layers can be defined as library functions, enabling straightforward experimentation with novel operators and layer types. We provide a front-end command line utility to compile TEL, as well as a command line JIT utility. We anticipate other existing neural network frameworks, such as TensorFlow, could be adapted to use DLVM as a backend code generation framework.

4 CONCLUSION

The deep learning research community has a rich variety of available toolkits. While two existing projects have attempted a compilers approach to deep learning frameworks, and have respectively achieved good integration with existing systems (XLA + TensorFlow) and good performance (NNVM + TVM), their design philosophies have not entirely followed established best practices in optimizing compiler design. While well intentioned, the remaining vast majority of other toolkits have failed to observe that the problem of converting a neural network into efficient executable code is, at its core, a compilers problem. As a result, important issues of extensibility and optimization have been addressed in less than optimal fashion in such toolkits. Nevertheless, several such toolkits, including Torch and TensorFlow, have achieved wide adoption. We believe that the principled application of optimizing compiler techniques will lead to substantial improvements in the tools available to deep learning researchers. DLVM and its associated front-end DSLs have a major role to play in this future. Our existing implementation utilizes LLVM to target NVIDIA GPUs. In our ongoing work we plan to substantially increase the number of supported hardware architectures by utilizing HPVM as an additional backend, while simultaneously supporting the use of DLVM by existing frontend frameworks such as TensorFlow.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew

Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. URL `http://arxiv.org/abs/1603.04467`.

W. L. Buntine and A. S. Weigend. Computing second derivatives in feed-forward networks: a review. *IEEE Transactions on Neural Networks*, 5(3):480–488, May 1994. ISSN 1045-9227. doi: 10.1109/72.286919.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, and Haichen Shen. TVM: An end to end IR stack for deploying deep learning workloads on hardware platforms. `http://tvmlang.org/2017/08/17/tvm-release-announcement.html`, 2017.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A Matlab-like environment for machine learning. In *NIPS Big Learning Workshop: Algorithms, Systems, and Tools for Learning at Scale*, December 2011.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Joe Groff and Chris Lattner. Swift's High-Level IR: A Case Study of Complementing LLVM IR with Language-Specific Optimization. 2015 LLVM Developers' Meeting, 2015. URL `http://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf`.

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. URL `http://arxiv.org/abs/1408.5093`.

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. URL `http://arxiv.org/abs/1704.04760`.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

Chris Leary and Todd Wang. XLA: TensorFlow, compiled! TensorFlow Dev Summit 2017, February 2017.

Uwe Naumann. *The Art of Differentiating Computer Programs*. Society for Industrial and Applied Mathematics, 2011. doi: 10.1137/1.9781611972078. URL `http://epubs.siam.org/doi/abs/10.1137/1.9781611972078`.

NNVM. NNVM compiler: Open compiler for AI frameworks. `http://tvmlang.org/2017/10/06/nnvm-compiler-announcement.html`, 2017.

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, pp. 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176. URL http://doi.acm.org/10.1145/2491956.2462176.

David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group (eds.). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X.

Prakalp Srivastava, Maria Kotsifakou, and Vikram S. Adve. HPVM: A portable virtual instruction set for heterogeneous parallel systems. *CoRR*, abs/1611.00860, 2016. URL http://arxiv.org/abs/1611.00860.