A modern compiler infrastructure for deep learning systems with adjoint code generation in a domain-specific IR

Richard Wei
Departments of Computer Science & Linguistics
University of Illinois at Urbana-Champaign
Urbana, IL 61801
`xwei12@illinois.edu`

Lane Schwartz
Department of Linguistics
University of Illinois at Urbana-Champaign
Urbana, IL 61801
`lanes@illinois.edu`

Vikram Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
`vadve@illinois.edu`

Abstract

Deep learning software demands reliability and performance. However, many of the existing deep learning frameworks are software libraries that act as an unsafe DSL in Python and a computation graph interpreter, some with inefficient algorithmic differentiation by operator overloading. We present DLVM, a design and implementation of a compiler infrastructure with a linear algebra intermediate representation, algorithmic differentiation by adjoint code generation, domain-specific optimizations and a code generator targeting GPU via LLVM. Designed as a modern compiler framework inspired by LLVM, DLVM is more modular and more generic than existing deep learning compiler frameworks, and supports tensor DSLs with high expressivity. We argue that the DLVM system enables a form of modular, safe and performant frameworks for deep learning.

1 Motivation

Within the deep learning community, most current approaches to neural networks make use of high-level frameworks with a tensor domain-specific language (DSL) such as Torch (Collobert et al., 2011), TensorFlow (Abadi et al., 2016), PyTorch (PyTorch Development Team, 2016), and MXNet (Chen et al., 2015). Traditionally, developers would build a computation graph (or dynamically generate graph nodes) using a DSL and let the framework interpret the computation graph on heterogeneous parallel architectures such as GPU. While using hand-tuned GPU subroutines usually yields the best performance for complex operators, advanced compiler techniques can be applied to simplify computation, merge high-level operators based on shaping conditions, and fuse compatible element-wise operators to a single kernel to minimize the latency between kernel launches. Recent projects, the TensorFlow XLA compiler (Leary and Wang, 2017) and the NNVM compiler (NNVM, 2017) including TVM (Chen et al., 2017), have begun to apply compiler techniques to deep learning systems, targeting LLVM (Lattner and Adve, 2004) and various back-ends to achieve good performance. However, their design and implementation have not entirely followed established best practices in widely-used compiler frameworks in the industry.

Under review at the Autodiff Workshop (NIPS 2017), Long Beach, CA, USA.

Moreover, some frameworks use operator-overloading algorithmic differentiation (AD) to compute gradients, leaving the gradient computation unoptimizable. The other approach to AD, adjoint code generation, can produce more efficient code. While frameworks such as TensorFlow already perform AD as a graph transformation and apply various optimizations, their AD transformation is not designed as a transformation pass in the pipeline of their compiler framework, but as part of the DSL library. Making AD part of the compiler framework would greatly simplify the development of DSLs, achieving separation of concerns.

We introduce DLVM, a new compiler infrastructure for deep learning systems that addresses shortcomings of existing deep learning frameworks. Our solution includes (1) a domain-specific intermediate representation designed for tensor computation, (2) configurable reverse-mode algorithmic differentiation as an intermediate representation transformation via the gradient declaration and canonicalization mechanism (3) principled use of modern compiler optimization techniques to substantially optimize neural network computation, including algebra simplification, AD checkpointing, compute kernel fusion, and various traditional compiler optimizations, While we do not present details here, we have also developed two proof-of-concept domain-specific languages that make use of this compiler infrastructure.

2 Related Work

Two closely related projects are the TensorFlow XLA compiler and the NNVM compiler.

The code representation in these frameworks is a “sea of nodes” representation, embedding control flow nodes and composite nodes in a data flow graph. To apply AD on this IR requires non-standard processing. In contrast, our approach is designed from the start around the idea that the tensor computation defined by neural networks is itself a program, which is best optimized through robust application of mature techniques in a principled compilation pipeline. We represent tensor computation in static single assignment (SSA) form with control flow graph, and perform AD, domain-specific optimizations, general-purpose optimizations, low-level optimizations, and code generation.

XLA takes a similar approach to ours, transforming TensorFlow sub-graphs to XLA’s HLO graph and performing optimizations. Our intermediate representation is much more expressive than XLA’s by including modular IR components and general-purpose instructions; this enables our approach to support full-fledged DSLs including standalone compiled DSLs and perform more extensive optimizations such as inlining and interprocedual optimizations. Our approach also differs from XLA by representing composite functions such as `min` and `max` directly through primitive instructions such as `compare` and `select`, which enables us to apply generic AD, and by using SSA form with control flow graph, which allows for reusing battle-tested SSA optimization algorithms in the LLVM community. Importantly, our entire infrastructure was designed from the start around a robust compile-time framework for tensor DSLs, 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).

Where TVM and NNVM are built as a DSL and a graph library in Python with a C++ implementation, DLVM’s architecture is closer to LLVM and the Swift Intermediate Language (Groff and Lattner, 2015), having an IR file format and a full-fledged command line toolchain. More specifically, our work differs from NNVM and XLA in the design and presence of an IR that has a textual parsable format, a module - function - basic block hierarchy, custom type declarations and memory semantics. The textual IR enables robust unit testing via FileCheck, which is used extensively in LLVM and most LLVM-based compilers. Moreover, DLVM and its associated DSLs are implemented entirely in Swift, a safe systems programming language, and thus have an elegantly compact codebase and type-safe APIs.

3 DLVM Core: An Optimizing Compiler for Neural Network DSLs

DLVM Core contains essential components for an optimizing compiler: IR, pass manager, and passes. The DLVM 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

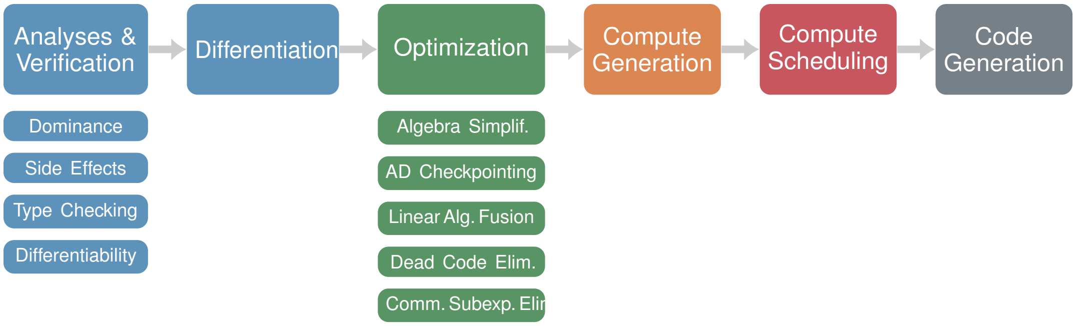

Figure 1: Compilation stages in the DLVM compilation pipeline.

```
module "my_module"
stage raw

// Representing function foo(x, w, b) = dot(x, w) + b
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 of @foo with respect to arguments 1 and 2
// Keeping original output 0 and seedable through function composition
[gradient @foo wrt 1, 2 keeping 0 seedable]
func @foo_grad:
    (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>, <1 x 10 x f32>)
   -> (<784 x 10 x f32>, <1 x 10 x f32>, <1 x 10 x f32>)
```

Figure 2: Example code in DLVM intermediate representation.

useful results as analyses of the program (analysis passes), or mutating the program for differentiation and optimizations (transform passes).

3.1 A Domain-specific Compiler Intermediate Representation for DLVM

Inspired by the LLVM IR (Lattner and Adve, 2004) and the Swift Intermediate Language (Groff and 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 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.

The DLVM IR has a high-level type system with tensor as a first-class type. The DLVM virtual instruction set includes domain-specific primitive math operators, as well as general-purpose instructions for memory management, control flow and function application. Domain-specific instructions include element-wise unary operators, such as `tanh` and `negate`, element-wise binary operators, such as `add` and `power`, and complex operators such as `dot`, `transpose`, and `convolve`. All element-wise binary operators support broadcasting. A sample of DLVM IR code is shown in Figure 2.

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

3.2 Algorithmic Differentiation through Adjoint Code Generation

Algorithmic differentiation (AD), also known as automatic differentiation, encompasses a family of a well-known techniques for algorithmically obtaining the derivatives of a function $f : \mathbf{x} \in \mathbb{R}^n \to \mathbf{y} \in \mathbb{R}^m$ (Naumann, 2011). 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 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.

In DLVM, the differentiation pass is responsible for performing reverse-mode AD. 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_grad` in Figure 2 is an example of such a function. Gradient declarations are configurable, e.g. specifying arguments to differentiate with respect to, keeping original outputs, and toggling seedability to accept back-propagated gradients. The differentiation pass, when applied, canonicalizes every gradient declaration in the module to a normal function definition with basic blocks and instructions. Unlike many of the existing deep learning frameworks, AD in DLVM is adjoint code generation, 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 $f(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, 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 necessarily be used to compute derivatives. In DLVM, unused operations can be easily eliminated by the aggressive dead code elimination pass in the compilation pipeline (see Figure 1 on the previous page). In addition, an AD-specific optimization technique called checkpointing can further reduce memory consumption during gradient computation.

AD in DLVM is configurable. The front-end can choose to differentiate a function with respect to selected arguments, to keep some of the outputs of the original function, to apply differentiation to a specific output when there are multiple return values, or to enable the function to accept back-propagated gradients (seeds) through function composition. Our approach to AD is implemented as a transformation from one function to another function. This approach also makes higher-order differentiation possible; this can be accomplished by declaring a higher-order gradient function that differentiates the original gradient function.

4 Conclusion

The deep learning research community has a rich variety of available frameworks. While two existing projects have attempted a compilers approach to deep learning frameworks, and have respectively achieved good integration with existing systems (TensorFlow XLA) 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 frameworks have failed to observe that the problem of front-end DSLs, algorithmic differentiation, and 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 frameworks. Nevertheless, several such frameworks 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 and explore more advanced AD techniques such as mixing forward and reverse modes.

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. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *CoRR* abs/1603.04467. http://arxiv.org/abs/1603.04467.

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

Tianqi Chen, Thierry Moreau, Ziheng Jiang, and Haichen Shen. 2017. 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`.

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

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

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. 2017. In-datacenter performance analysis of a tensor processing unit. *CoRR* abs/1704.04760. http://arxiv.org/abs/1704.04760.

Chris Lattner and Vikram Adve. 2004. 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.

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

Uwe Naumann. 2011. *The Art of Differentiating Computer Programs*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611972078.

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

PyTorch Development Team. 2016. Tensors and Dynamic neural networks in Python with strong GPU acceleration. http://pytorch.org.

David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, editors. 1986. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, Cambridge, MA, USA.