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

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

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

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 Torch and TensorFlow define a computation graph in a host language such as Python, where each node represents a linear algebra operation parallelized as a compute kernel on GPU and stores the result of evaluation; some of these toolkits subsequently perform runtime 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. We introduce a novel framework for high-level programming that addresses all of the above shortcomings.

1 Motivation

Algorithmic differentiation (AD), also known as automatic differentiation, encompasses a family of well-known techniques for algorithmically obtaining the derivatives of a function \( f : x \in \mathbb{R}^n \rightarrow y \in \mathbb{R}^m \) [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 \( \frac{dy}{dx} \) 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

Under review at the Autodiff Workshop (NIPS 2017), Long Beach, CA, USA.
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 most 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 adjoint calculation through operator overloading.

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 spe-
cialized 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
group 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 and Wang 2017), has begun to explore some of these optimization issues, and
includes the ability to perform limited 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 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 existing
deep learning frameworks. The solution we present differs substantially from most existing toolkits;
instead of performing reverse-mode AD through interpretation / operator overloading, we make use
of source code transformation to automatically generate adjoint code to calculate the derivative of
each differentiable function. We optimize and compile the resulting program into a binary (or in
some cases binaries) instead of performing run-time 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 automatic differentiation,
and (3) code generation through a mature compiler-backend infrastructure (Lattner and Adve 2004)
that allows for transparent targeting of various hardware. 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

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

XLA (Leary and Wang 2017) compiles TensorFlow graphs to a compiler IR and performs 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 including 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, 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 and Lattner, 2015). Unlike TVM/NNVM, we provide command-line utilities for the respective components in our toolchain.

3 DLVM: An Optimizing Compiler for Neural Network DSLs

Deep Learning Virtual Machine (DLVM) is a compiler infrastructure designed for modern deep learning software. 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 DLVM compilation stages address algorithmic differentiation, fusion of tensor linear algebra operations, other domain-specific optimizations, and static code generation targeting a variety of devices.

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 Intermediate Representation (IR) is a graph-based, modular code representation, with both an in-memory format and a textual format. It uses static single assignment (SSA) form, high-level types including a first-class tensor type, and a set of linear algebra operations combined with a general-purpose instruction set. 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.

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. 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.

\[\text{Code for DLVM and our example DSLs is available under the Apache 2.0 license at http://anonymous.url}\]
3.2 Algorithmic Differentiation through Adjoint Code Generation

DLVM has a full-fledged pass infrastructure, performing two kinds of transformations: differentiation and optimization. Optimizations include Algebra Simplification and Linear Algebra Fusion, Matrix Chaining, and Automatic Differentiation Checkpointing. Since DLVM IR is aware of mathematical operators such as \( \text{tanh} \) and \( \text{power} \), the Algebra Simplification pass can find and simplify certain mathematical operations that are expensive or redundant. For example, the Linear Algebra Fusion pass can simplify operations in \( h_t = f(Wx_{t-1} + Uh_{t-1} + b) \) 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. Other general-purpose optimizations include Aggressive Dead Code Elimination, Common Subexpression Elimination, and Sparse Conditional Constant Propagation.

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 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 \( f(x_1, x_2, \ldots, x_n) \), this pass creates a new function that computes the Jacobian \( J_f \). 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 pass in the compilation pipeline. In addition, 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 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 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 in conjunction with a custom intermediate representation designed specifically for deep learning 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 initial release of DLVM includes two example front-end domain-specific languages for neural networks. We believe that existing toolkits such as TensorFlow could be adapted to use DLVM as a back-end optimizer and code generator.
References


