DLVM: A modern compiler framework for deep learning

Richard Wei, Lane Schwartz, Vikram Adve
University of Illinois at Urbana-Champaign

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. 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 IR inspired by LLVM and the Swift Intermediate Language, DLVM IR is more modular and more generic than existing deep learning compiler IRs, and supports tensor DSLs with high expressivity. With our prototypical staged DSL embedded in Swift, we argue that the DLVM system enables a form of modular, safe and performant frameworks for deep learning.

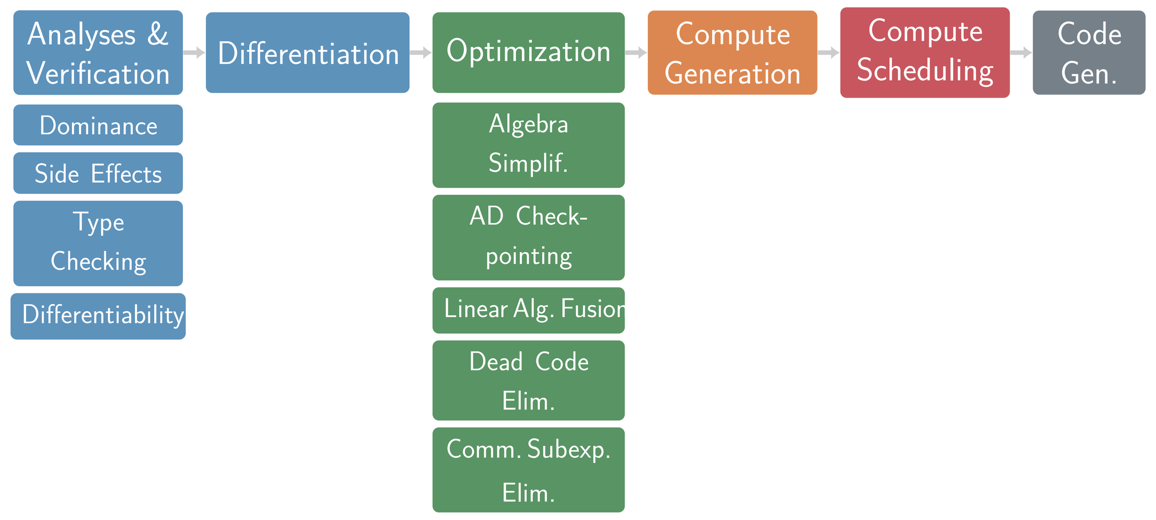

Figure 1: *Stages in the DLVM compilation pipeline.*

Overview

We introduce DLVM, a new compiler infrastructure for deep learning systems that addresses shortcomings of existing deep learning frameworks. Our solution includes:

- a domain-specific intermediate representation specifically designed for tensor computation,
- principled use of modern compiler optimization techniques to substantially simplify neural network computation, including algebra simplification, AD checkpointing, compute kernel fusion, and various traditional compiler optimizations,
- code generation through a mature compiler infrastructure,
- an embedded DSL that supports static analysis, type safety and natural expression of tensor computation and has a just-in-time (JIT) compiler targeting DLVM for AD, optimizations and code generation.

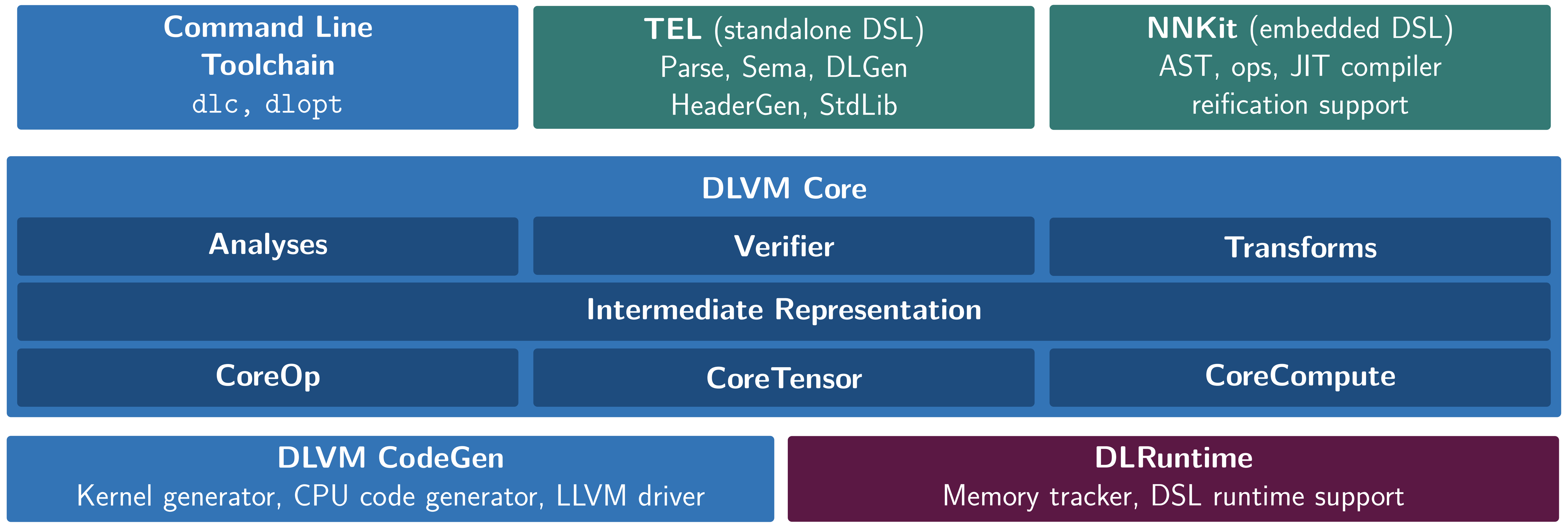

Figure 2: *Software stack of the DLVM infrastructure. Blue components are the compiler framework.*

Background

- 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, TensorFlow, PyTorch, and MXNet.
- Traditionally, developers would use one of these frameworks to define a computation graph (or dynamically generate graph nodes) that represents a neural network using a DSL, and the framework would interpret the computation graph at runtime, performing reverse-mode algorithmic differentiation to obtain gradients required for training neural network weights.

Novel Contributions

- The sequence of tensor computations defined by a neural network represents a computer program, which is best optimized through robust application of mature techniques in a principled compilation pipeline. We treat the task of building and training neural networks as a compilers problem.
- We define a compiler intermediate representation (see Figure 4) specifically designed for the data types and calculations required by neural networks, with first-class support for tensors and gradient calculations.
- We define principled compiler passes (see Figure 1) for analyzing, differentiating, and optimizing neural network code using current compilers best practices.
- We present neural network DSLs that utilize DLVM, our neural network compiler infrastructure (see Fig. 2 & 3).

Related Work

- The two most 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 algorithmic differentiation on this IR requires non-standard processing.
- 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, having an IR file format and a full-fledged command line toolchain.

DLVM

- We represent tensor computation in static single assignment (SSA) form with control flow graph, and perform algorithmic differentiation, domain-specific optimizations, general-purpose optimizations, low-level optimizations, and code generation.
- Our IR is much more expressive than XLA's, 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 interprocedural optimizations.
- We anticipate other existing deep learning frameworks, such as TensorFlow, could be adapted to use DLVM as a back-end.

```
// Staged function representing f(x, w, b) = dot(x, w) + b
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
            x • w + b
    }

// Staged function 'g', type-inferred from 'f'
let g = lambda { x, w, b in
    let linear = f[x, w, b] // staged function application
    return tanh(linear)
}

// Gradient of 'g' with respect to arguments 'w' and 'b'
let dg = gradient(of: g, withRespectTo: (1, 2), keeping: 0)
// 'dg' has type:
// Rep<(Float2D, Float2D, Float2D) -> (Float2D, Float2D, Float2D)>

// Call staged function on input data 'x', 'w' and 'b'
let (dg_dw, dg_db, result) = dg[x, w, b]
// At runtime, 'dg' gets just-in-time compiled though DLVM,
// and computes ( dg/dw, dg/db, g(x, w, b) )

// Second order derivative of 'g' with respect to 'w'
let d2g_dw2 = gradient(of: dg, from: 0, withRespectTo: (1))
// 'd2g_dw2' has type:
// Rep<(Float2D, Float2D, Float2D) -> Float2D>
```

Figure 3: *Code in Swift using NNKit, a staged DSL targeting DLVM.*

```
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 all arguments
[gradient @foo]
func @foo_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
          -> (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)

// Gradient of @foo with respect to arguments 1 and 2
[gradient @foo wrt 1, 2]
func @foo_grad_2: (<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>)

// Gradient of @foo with respect to arguments 1 and 2
// Keeping original output 0
// Seedable, able to take back-propagated gradient as a seed for AD
[gradient @foo wrt 1, 2 keeping 0 seedable]
func @foo_grad_3:
    (<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 4: *Code in DLVM intermediate representation.*

https://arxiv.org/abs/1711.03016

ICLR 2018 — Workshop Track — Vancouver, British Columbia — May 2018

CS + LING @ ILLINOIS