DLVM: A modern compiler framework for neural network DSLs

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. 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. 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.

1 Introduction

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.

Moreover, some frameworks use operator-overloading algorithmic differentiation (AD) to compute gradients, leaving the gradient computation unoptimizable. The other approach to AD, source code transformation, 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

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

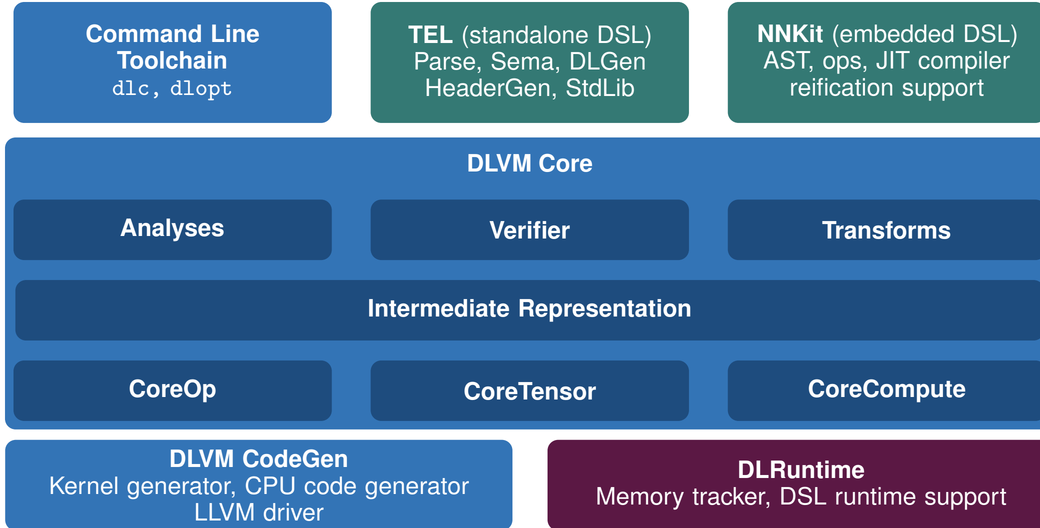

Figure 1: Software stack of the DLVM infrastructure. Blue components are the compiler framework.

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 framework for neural network DSLs that addresses shortcomings of existing deep learning compiler frameworks. Our solution includes (1) a domain-specific intermediate representation specifically designed for tensor computation, (2) 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, (3) code generation through a mature compiler infrastructure that allows for transparent targeting of various hardware, and (4) an embedded DSL that features type safety and natural expression of tensor computation and has a just-in-time (JIT) compiler targeting DLVM for AD, optimizations and code generation.

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 algorithmic differentiation 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 algorithmic differentiation, 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).

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

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let dg = gradient(of: g, withRespectTo: (1, 2), keeping: 0)
// dg has type: Rep<(Float2D, Float2D, Float2D) -> (Float2D, Float2D, Float2D)>

let (dg_dw, dg_db, result) = dg[x, w, b]
// 'dg' gets just-in-time compiled though DLVM,
// and computes ( dg/dw, dg/db, g(x, w, b) )
```

Figure 2: Example code in Swift using NNKit, a staged DSL targeting DLVM.

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 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 software developers by providing a safe environment for rapid prototyping that frees the developer from low-level tasks, while simultaneously generating 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.

NNKit is a staged DSL embedded in Swift, featuring natural expression of tensor computation alongside the host program without losing static analyses and optimizations on the tensor program. Inspired by Lightweight Modular Staging (Rompf and Odersky, 2010), NNKit leverages the static type system of the host language to guarantee type safety of the DSL. Tensor types are wrapped in `Rep<T>`, meaning the representation of some computation that produces data of type `T`. Tensor operators overloaded for `Rep` are essentially AST builders for delayed evaluation. Instead of generating computation nodes at runtime and performing operator-overloading AD like PyTorch or TensorFlow Eager (Google Brain Team, 2017), NNKit tensor computations are staged once during the lifetime of the host program. At invocation time of staged functions, NNKit emits shape-specialized DLVM IR and leverages DLVM to perform AD, optimizations, and code generation.

The NNKit just-in-time compiler has four important phases: The first phase, expression staging, produces an unshaped graph IR of the tensor computation. The second phase, shape specialization, prepares to generate statically shaped DLVM IR for staged functions when they are applied to shaped tensors. The third phase, lowering, generates DLVM IR and passes it through DLVM, producing a dynamic library containing a function symbol. The final phase, function reification, loads the binary and wraps the low-level function to a Swift function.

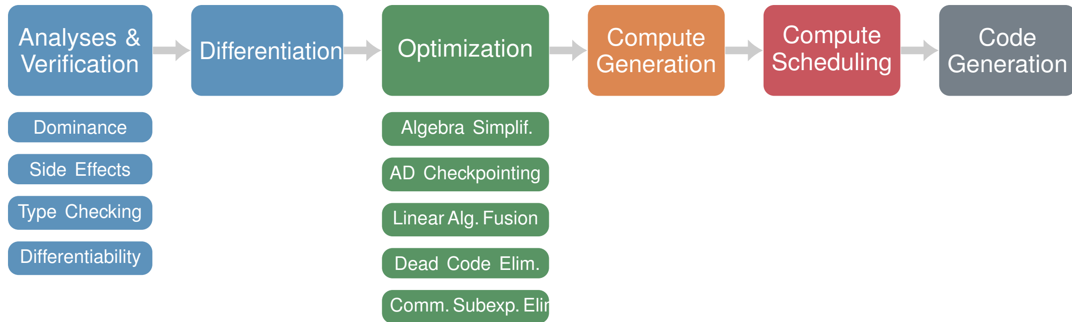

Figure 3: Compilation stages in the DLVM compilation pipeline.

4 DLVM

Deep Learning Virtual Machine (DLVM) is a compiler infrastructure designed for modern deep learning systems. 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 DSLs for neural networks. The complete DLVM software stack, including sample front-end neural network DSLs, is shown in Figure 1 on page 2.

Figure 3 illustrates the major stages in the DLVM compilation pipeline. The DVLM compilation stages address algorithmic differentiation, domain-specific optimizations, general-purpose optimizations, and static code generation targeting a variety of compute architectures. The *raw* and *optimizable* stages allow constructs for high level tensor operations and various high-level optimizations. The *compute* and *schedule* stages allow constructs for 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, control flow graphs, high-level types including a first-class tensor type, and a set of linear algebra operators 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, AD checkpointing, algebra simplification and linear algebra fusion.

4.1 DLVM Core

DLVM Core contains essential components for an optimizing compiler: IR, pass manager, and passes (see Figure 1 on page 2). 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 useful results as analyses of the program (analysis passes), or mutating the program for differentiation and optimizations (transform passes).

Inspired by the LLVM IR and the Swift Intermediate Language, 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 4 on the next page.

```
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 and seedable through function composition
[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: Example code in DLVM intermediate representation.

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 algorithmic differentiation algorithm to be applied to any differentiable program, with no need for special handling of composite cases.

On top of this architecture, gradients of differentiable functions can be created using **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 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 some of the existing deep learning compiler frameworks, AD is an essential phase of the compiler, not a front-end component. DSL developers simply need to generate gradient declarations for functions they would like to differentiate. AD as an IR transformation gives the compiler the freedom to apply optimizations on the gradient computation separately and enables higher order differentiation.

4.2 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 lowered programs. In order to attain these goals, DLVM code generation transforms DLVM IR into LLVM IR. LLVM is a robust and mature compiler infrastructure with multiple back-ends, including NVIDIA GPUs. Many high-level DLVM IR linear algebra instructions over tensors abstract lower-level operations. The DLVM compiler transforms the high-level DLVM IR into lower-level stages and ultimately into calls to BLAS and compute kernels in LLVM IR. Existing LLVM utilities are used to compile the generated LLVM IR to the final binary.

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 compiler extension to LLVM IR that allows for transparent targeting of diverse architectures from a data flow graph.

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	`concat %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 upto 5`
Random	`random 768 x 10 from 0.0: f32 upto 1.0: f32`
Select	`select %x: <8 x f64>, %y: <8 x f64> by %z: <8 x bool>`
Compare	`gt %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 f32>`
Branch	`branch 'block_name(%a: i32, %b: i32)`
Conditional branch	`conditional %cond: bool then 'bb0() else 'bb1()`
Shape cast	`shapeCast %a: <1 x 40 x f32> to 2 x 20`
Extract	`extract #x from %pt: $Point`
Insert	`insert 10: f32 to %pt: $Point at #x`

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

4.3 DLVM command line toolchain

The front-end software associated with each DSL (see Section 3) is responsible for generating a DLVM IR for a given source language program to be compiled. The DLVM compiler infrastructure itself is a compiler from DLVM IR to LLVM IR, therefore having a command line toolchain is necessary for verifying, transforming and compiling batches of DLVM IR files (`*.dl`). Unlike XLA and NNVM/TVM, which only provide a Python/C++ interface to their users, DLVM provides a command line interface like any industry-standard compiler.

The DLVM optimizer utility, `dlopt`, accepts `*.dl` IR files and applies user-specified optimization passes on them. The DLVM compiler driver, `dlc`, accepts `*.dl` IR files and performs user-specified tasks, such as verification, differentiation, optimization passes, stage lowering passes, and code generation; the driver invokes the DLVM core library to achieve these tasks. Because of having a textual format of the IR, the DLVM framework can easily make use of the LLVM Integrated Tester (lit) and FileCheck to perform robust unit testing. In future iterations, we plan to introduce a DLVM bitcode format for compact storage and high-throughput processing of DLVM code.

5 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 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 by utilizing HPVM as an additional back-end, while simultaneously supporting the use of DLVM by existing front-end DSLs 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. 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*.

Google Brain Team. 2017. Eager Execution: An imperative, define-by-run interface to TensorFlow. https://research.googleblog.com/2017/10/eager-execution-imperative-define-by.html.

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.

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.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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*. ACM, New York, NY, USA, PLDI '13, pages 519–530. https://doi.org/10.1145/2491956.2462176.

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*. ACM, New York, NY, USA, GPCE '10, pages 127–136. https://doi.org/10.1145/1868294.1868314.

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