

# A Comprehensive Survey for Deep Learning Compilers

**Sinan Lin**

Sinan.lin@aalto.fi

**Tutor:** Hiroshi Doyu

## Abstract

*The deep learning compiler is a domain-specific compiler for parsing, optimizing, and generating binaries for a trained neural network model on a wide range of hardware platforms. This article comprehensively introduces the general compilation process of deep learning compilers and provides an in-depth comparison of three widely-used compilers, TVM, ONNC, and MLIR. These three compilers are widely used, but have significant difference in their designs. Specifically, this work summarizes the overall design, workflow, and optimization strategies of the three different compilers. Finally, the paper highlights what are similarities, differences and possible complementarity among the three compilers.*

**KEYWORDS:** Deep Learning Compiler, Model Inference, Deep Learning

## 1 Introduction

In recent years, the rising popularity of Artificial Intelligence applications has created the demand for deploying trained deep learning models for inference. Currently, there are two approaches for deploying a neural network model.

One approach is that hardware vendors develop and provide their deep learning inference frameworks aiming at their specific hardware design [17]. For example, Nvidia provides TensorRT for Nvidia Turing architecture [28], and Intel develops OpenVINO for its processors [6]. However, these frameworks highly depend on vendor-specific mathematical computing libraries, which results in losing interoperability for reusing optimization strategies from other inference frameworks [13]. Therefore, further optimizations have to be implemented in all inference frameworks from scratch, which leads to a great number of redundant efforts. Therefore, this solution is not sustainable in the long term [5].

The second approach to deploying a deep learning model is to use a deep learning compiler. Deep learning compilers are extensions to traditional compilers. They can take data with more complicated structure as input, lower the representations with the high-level abstraction (such as mathematical functions and operations), and generate optimized and executable code for diverse hardware, such as CPU, GPU, and accelerators. The deep learning compiler solution contains several benefits. First, it reduces the cost of deploying on heterogeneous hardware. Second, it provides graph-level optimization for neural network models, while conventional compilers cannot do such domain-specific optimizations. Third, the optimization strategies for high-level intermediate representation (IR) can be reused for deploying to different hardware. The strengths of deep learning compilers have grasped attention from companies and research institutions, increasing the number of deep learning compilers implemented. This paper provides a comprehensive review and comparison for three mainstream deep learning compilers: TVM [5], ONNC [14], and MLIR [11]. The paper analyzes the front-end design, optimization techniques, scheduler, and back-end solution.

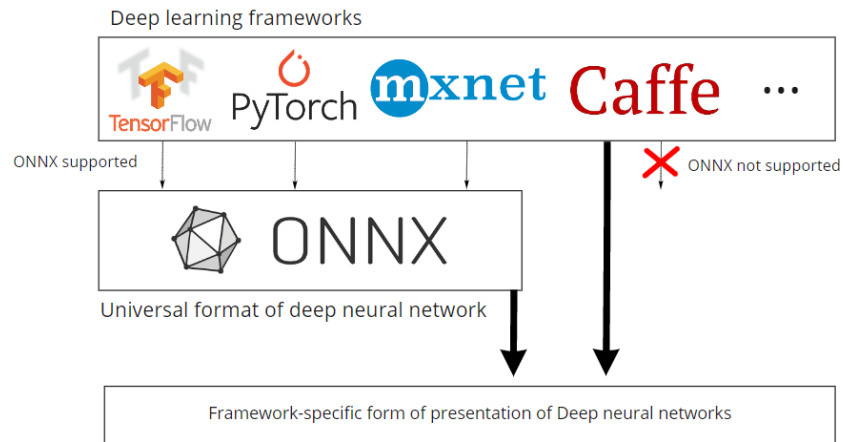
## **2 Background**

### **2.1 Deep Learning Frameworks**

Deep learning frameworks are used for training and inference of deep neural networks, and they can be categorized based on the way they create, represent, and run computations [19]. This section introduces two

typical frameworks, Tensorflow [1] and PyTorch [19] and one support framework ONNX [15] that is designed to be a universal open standard deep learning format.

**Figure 1.** An overview of deep learning frameworks



**Tensorflow** [1] is an end-to-end platform for deep learning. It has a comprehensive, flexible ecosystem of tools for engineers and researchers to build state-of-the-art models and employs static computational graphs with primitive operators to represent a neural network model.

**PyTorch** [19] is the next generation of Torch, a Lua-based deep learning framework. It constructs deep neural models with dynamic computational graphs, where the primitive functions in Python can use inside the models.

**Open Neural Network Exchange(ONNX)** [15] is an open standard format for representing deep learning models. Models from different deep learning frameworks can be converted into the ONNX format, which helps to achieve the model interoperability between different frameworks.

## 2.2 Deep Learning Hardware

The hardware for deep learning inference can be categorized into two types based on the purpose of their designs. 1) General-purpose hardware that is designed for multiple tasks, and it has been designed and created with extensive knowledge and experience, and 2) AI-customized hardware that is specialized for performing deep learning inference with customized circuit design.

### *General-purpose Hardware*

modern **CPUs** support vector instructions (SIMD instruction) for faster processing of vectorized data. For example, the AVX instructions in Intel x86 architecture enables the CPU to speed up the matrix multiplication task [8]. However, the CPU is designed for a wide range of applications and tailored to fit both data-intensive and compute-intensive programs. Thus, it results in a moderate number of arithmetic logic units (ALU) and cores and a relatively large cache.

The **GPU** offers fewer instruction sets, but instead, it has a larger number of ALUs and cores and a smaller storage space for running limited operations on a smaller data-set with massive parallelism. Also, GPUs have a shared memory design for reducing the overhead of context switching.

**AI-specific hardware** is designed to utilize the property of the deep neural network, and it significantly diverges in terms of architecture and computes primitives. For example, the hardware Tensor Processing Unit (TPU) [9] was developed by Google in 2015 for deep neural network training and inference. It contains matrix multiply units, which enables matrix multiplication to be a primitive operation of hardware, and speed up the inference to 15-30 times faster than contemporary general-purpose hardware [9].

## **3 Deep learning compilers**

### **3.1 TVM**

TVM is an end-to-end compiler designed for engineers who want to optimize and deploy their models to production. It is a mature toolchain for all compilation stages from parsing a model from deep learning frameworks, automated optimization on both high-level and low-level, to code generation, and it is well-encapsulated so that developers with no professional knowledge on compilers can manipulate it effortlessly.

### **3.2 ONNC**

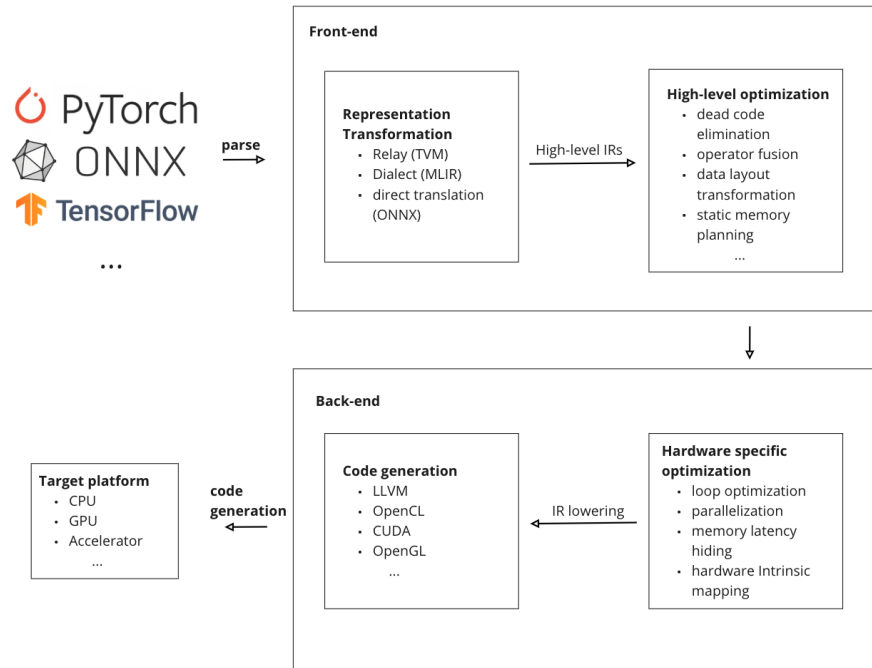
ONNC focuses on the compilation on NVDLA-based accelerators. ONNC has similar functionalities with TVM, but it only supports the ONNX format and does not support automated operator fusion and auto-tuning. In addition, ONNC is the first compiler that has primitive support for

NVDLA-based accelerators, and it provides a memory scheduling algorithm, which is useful for the edge device with limited memory.

### 3.3 MLIR

MLIR is an extension of LLVM compiler infrastructure [12]. MLIR has a 'dialect' component that provides extensions to represent complex data structure system and offers high-level abstraction of IRs. It also offers a standard dialect as a bridge for the conversion between different IRs. It offers high-level abstraction of data types and IRs to increase the reusability between different compilers. This allows the optimization passes from other compilers to be reusable. For example, the TensorFlow ecosystem has several compilers for different target hardware, such as XLA HLO to the general-purpose device, TensorFlow Lite to the mobile device, and CoreML to the neural engine of Apple, and the implementation of corresponding dialects and their mapping to the standard dialect enables the conversion among different compilers. Therefore, developers can avoid a large amount of redundant work, and the compiler on one domain can benefit from the optimization passes of other domains.

## 4 An Overview of the Architecture Design



**Figure 2.** The general workflow for the front-end of deep learning compilers

Deep learning compilers are used for deep learning tasks, especially deploying trained models from deep learning frameworks for inference. The workflow of the compiler can be categorized into two compilation processing stages, the front-end, and the back-end, as shown in Figure 2. The front-end stage is responsible for parsing the model to a representation that the compiler can recognize and perform graph-level optimization. In the front-end stage, the trained models in text form are translated into the specific intermediate representation (IR) of deep learning compilers in the form of the directed acyclic graph, control flow graph, or static single assignment form [2] that represent computation operation and data dependency between operations. Therefore, graph-level optimizations can be used on the IR to fuse operations and optimize data layouts [5]. The back-end stage aims to emit optimized machine code of models. In this stage, the optimized IR from the front-end can be lowered to loop-based tensor expression and then be further optimized for characteristics of target hardware. For example, in deep neural networks, the convolution and fully connection operation can be decomposed into matrix-matrix multiplication and matrix-vector multiplication [27] [25]. In the back-end stage, the optimized IRs can be further optimized for characteristics of the target hardware, and then they can be 1) translated into source code (such as CUDA [22], and OpenCL [23]) and compiled by using general-purpose compilers, or 2) mapped to the instruction of accelerators that has custom instruction set architectures.




## 5 Front-ends

The front-end of the deep learning compiler is mainly responsible for two tasks. First, it translates the input data that can be the trained models in Python and also can be the data-serialization format [15] of a computation graph, to its intermediate representation. Second, it performs graph-level optimizations, including hardware-independent optimization and data layout transformation.

### 5.1 Representation Translation

In order to represent the computation of neural network models, compilers should be able to translate the models from the text form or directly

**Figure 3.** Comparison of frontend among TVM, MLIR and ONNC

	Framework support	High-level IR/ Graph IR	Dynamic representation support	High-level Optimization	Quantization support
	<ul style="list-style-type: none"> <li>PyTorch</li> <li>Tensorflow</li> <li>Tensorflow lite</li> <li>ONNX</li> <li>Keras</li> <li>Mxnet</li> <li>darknet</li> </ul>	<ul style="list-style-type: none"> <li>let-binding-based IR</li> <li>CFG-based IR</li> </ul>	<ul style="list-style-type: none"> <li>control flow</li> <li>dynamic tensor shape</li> </ul>	<ul style="list-style-type: none"> <li>operator fusion</li> </ul>	<ul style="list-style-type: none"> <li>int8</li> <li>int16</li> <li>fp16</li> </ul>
	<ul style="list-style-type: none"> <li>ONNX</li> </ul>	<ul style="list-style-type: none"> <li>DAG-based IR</li> </ul>		<ul style="list-style-type: none"> <li>live variable analysis</li> <li>operator fusion</li> </ul>	<ul style="list-style-type: none"> <li>int8</li> <li>int16</li> <li>fp16</li> </ul>
		<ul style="list-style-type: none"> <li>hybrid IR <ul style="list-style-type: none"> <li>- structural</li> <li>- linear</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>dynamic tensor shape</li> <li>control flow</li> </ul>	<ul style="list-style-type: none"> <li>live variable analysis</li> </ul>	<ul style="list-style-type: none"> <li>int8</li> <li>int16</li> <li>fp16</li> </ul>
Common feature <ul style="list-style-type: none"> <li>dead node elimination</li> <li>data layout transformation</li> <li>customized pass</li> </ul>					

from the Python interface into intermediate representations of compilers. In this way, these compiler-specific IRs can be further recognized and built to be a computational graph [13] for profiling. The front-end of TVM is Relay [21], and it is responsible for translating work for TVM. It provides a flexible Python interface for the support of deep learning frameworks, including TensorFlow, PyTorch, MxNet, and ONNX, and translate models into Relay IR that is a purely-functional, statically-typed intermediate representation based on the design of OCaml programming language [18] [21]. However, ONNC only supports the models in ONNX format. The front-end of ONNC performs translation by using a one-to-one mapping from ONNX IR to ONNC IR. In terms of MLIR, it offers no built-in support for translating work, but Tensorflow implements a parser for MLIR to parse models of Tensorflow to MLIR IR [7] [11].

## 5.2 Graph-level Optimization

The intermediate representation of a deep neural network can be viewed as a computational graph where the nodes correspond to one or several arithmetic operations, and edges show the dependency between data flow. It provides a global view of operators and avoids specifying how each operator is implemented. The graph-level optimization includes semantics-preserving optimizations and data layout optimizations. Semantics-preserving optimizations rewrite computational graphs by removing redundant nodes and computation or by folding multiple nodes into a single node. These

graph-level optimization passes in the deep learning compiler are similar but different from the optimization strategies of traditional compilers.

(1) **Constant folding** is to statically compute parts of the graph that rely only on constant initializers, avoiding the need to compute them during runtime.

(2) **Redundant node elimination** is to remove all redundant nodes without changing the graph structure. Unlike dead code elimination [26], redundant node elimination works for nodes that contain one or more arithmetic operations in a computational graph.

(3) **Operator fusion** that folds several operators into a single kernel, so multiple operators can be performed without saving the intermediate result back to the memory, and thus reduce the execution time [5]. For example, a Conv-Relu fusion folds the Relu operator as the weight of the convolution operator.

The implementations of operator fusion are different for ONNC and TVM. TVM divides graph operators into four types based on the change in the shape of input and output dimensions. (1) injective operators (one-to-one mapping, such as Add operator), (2) reduction operators (several-to-one mapping, such as Sum operator), (3) complex-out-fusable operators (element-wise map, such as convolution), (4) infusable operators (such as Sort) and use specific fusion rules upon the combinations of these four types of operators. For example, multiple injective operators can be fused into a single one compound injective operator [5]. Differently, ONNC does not group these operators. Instead, it offers fusion rules for a range of specific combinations of operators.

### 5.3 Layout Transformation

This optimization changes the data layout to optimize access locality on the target hardware for performance improvements. For example, TVM targeting on CPUs optimizes the layout of convolution operations from NCHW into N[C/c]HWc , in which N, H, and W stand for the size of the batch, height, and weight respectively, and c means the split sub-dimension of channel C. Therefore, it is convenient to have smaller pieces of a channel as the innermost dimension as a channel is often larger than the width of SIMD instructions of x86 CPUs [16].



## 6 Back-ends

The back-end is responsible for hardware-dependent optimization and emitting binary code for target devices. In general, the optimized IRs from the front-end part can be lowered to LLVM IR and reuse the optimizer and code generator of LLVM infrastructure. The computation in neural network models is basically matrix multiplication, which benefits from aggressive loop optimization, including loop parallelization, tiling, and reordering [2]. However, general-purpose compilers usually generate poorly performing code when loop-based programs are directly passed to them [13]. In order to avoid this situation, deep learning compilers usually apply two approaches before delivering code to general-purpose compilers. First, the back-end maps a certain set of IR instructions to hardware intrinsics that are highly optimized for a specific combination of arithmetic operations. Second, the back-end performs target-specific loop optimization. As for the second scene, the optimal loop transformation depends on the characteristics of the hardware design, and each combination of choices of marking the implementations of loop tiling, reordering, vectorization is a scheduling option for the back-end. This leads to huge search space for the back-end optimizer.

### 6.1 Scheduling

Deep learning compilers apply different scheduling approaches to tuning the parameter to determine the loop parameters with the best performance for the target hardware. In the following, different scheduling techniques are discussed.

(1) **Black-box tuning** requires no machine-specific information and randomly try a different combination of loop transformation parameters, such as blocking size, loop unrolling factors, and loop order, on the target device. If a configuration achieves better performance than the previously-stored result, it is updated to be the current optimal configuration, and the scheduling process repeats these steps until all possible combinations are explored. This approach costs a significant amount of time and might lead to over-fitting, but the result is unbiased since it requires no prior information of hardware.

(2) **Pre-defined cost tuning** requires the information of the target hardware and pre-defines the weight of each loop transformation parameter for each supported hardware. It heuristically searches for the optimal

loop configuration based on the weights, instead of running all possibilities of loop configuration and testing on the target hardware [20]. TVM reuses the pre-defined cost scheduler from Halide and extends the primitives from CPU only to a wide range of hardware, including GPU and specialized accelerator [5] [20]. This scheduling method is efficient, and the result is optimal for the commonly used hardware. However, it is biased and not sustainable since the weights are added manually, and new weights should be provided to every new hardware.

(3) **Machine-learning-based tuning** predicts the performance after a set of loop transformation using a machine learning model. For each schedule configuration, the model takes the loop parameter as input and predicts its execution time on hardware. The model is updated by the measured runtime on the target device, and it does not require the detail of hardware. TVM offers a machine-learning-based auto-tuning module, AutoTVM, for tuning the result [4]. This approach is less biased than the pre-defined cost tuning method and more efficient than the black-box tuning method. Besides, the collected data can be stored for further use.

(4) **Polyhedral-based tuning** models loop-based source code into mathematical abstractions and uses linear programming to find the optimal affine transformations on the loop-based program without violating the program behaviors [3] [11]. This method provides a powerful mathematical framework to reason about loops in programs. MLIR uses techniques from polyhedral compilation to make dependence analysis and loop transformations efficient and reliable [11].

## 6.2 Code generation

As the target hardware has become increasingly diverse, the task of code emitting becomes more complicated than the traditional compilers. The back-end offers multiple code generators that work for different target devices.

### *General-purpose hardware*

After the optimization, deep learning compilers can generate machine code from the lowered representation for target hardware. The common approach for generating CPU machine code is to lower or map the IRs to LLVM IR and reuse the LLVM compiler infrastructure to generate machine code for target CPU [12]. As for GPUs, TVM and ONNC implement a code generator that emits code with PTX [10] instructions, and the emit-

ted code can be delivered to the CUDA compiler for generating executable code.

#### *AI-specific hardware*

There is not a generic way that generates code for AI-specific hardware because of the diversity in their design. However, if the target hardware has a manually optimized C/C++ library, such as Intel MKL to CPU and NVIDIA cuBLAS to GPU, the code generator of this hardware can also be customized. Hardware providers need to implement a code generator that generates C code (TVM) or ONNC IRs (ONNC) for subgraphs and integrates the code generator into the runtime module. Beside, TVM provides a graph representation generator that can generate optimized graph representation into other forms, and it can be used for the hardware that is built on a complete graph execution engine, such as TensorRT [24]. In addition, TVM and ONNC offer primitive support for TPU and NVDLA-based accelerator, respectively.

## **7 Discussion**

Although there is a discussion on the difference between these three deep learning compilers in sections 4 and 5, it is helpful to have an analysis on their unique feature and collaborative work.

### **7.1 TVM**

#### *Flexible interface for model parsing*

The front-end of TVM provides users with a wide range of interfaces, maximizing the flexibility for development. Models from deep learning compilers or the text form can be parsed without manually manipulating.

#### *Automated graph-level optimization*

Unlike ONNC, the automatic operator fusion for TVM is not only simple string matching, but also means the more complex graph matching. The front-end of TVM, Relay, groups operators into different groups, and apply different approaches on the combination of groups, instead of simply the names of operators.

## 7.2 ONNC

### *Specific optimization for some hardware*

ONNC primitively supports NVDLA-based accelerators [15], and ONNC provides optimized kernels for specific operations. Thus, these accelerators can be high-performing with a limited amount of time spent on auto-tuning [5].

## 7.3 MLIR

### *Multi-level IRs support*

With the dialect mechanism, MLIR supports multiple IRs for one single compilation. This enables compilers developers to bridge two different compilers and reuse the optimization passes existed. Therefore, MLIR compiler tends to be popular as a tool for compiler-related development, instead of direct deployment.

## 7.4 Collaboration among TVM, ONNC and MLIR

The design of TVM, ONNC and MLIR are modulized, and this allows the collaboration. For example, ONNC can intergrate TVM as its front-end by simply adding a TVM IR parser, which combines the flexibility of the front-end of TVM and specific optimization passes of NVDLA-based accelerators from the back-end of ONNC. Also, the dialect mechanism can allow this portability to be even more powerful and scalable, because of the dialect mechanism from MLIR.

## 8 Conclusion

In summary, these three compilers have different emphasis. TVM is a tool for general developers, it provides the best flexibility in terms of the forms of models and the number of target hardware it primitively supports. ONNC compiler has a specific support for a range of hardware, which allows ONNC to be the preference for a specific group of users. However, MLIR is the most distinct among these three compilers. It does not contain a front-end for deep learning models, but it can bridge the gaps of deep learning or other domain-specific compilers to reuse optimization passes from each other. Furthermore, these three compilers can collaborate to address problems efficiently.

## References

- [1] Martín Abadi and et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.
- [3] Riyadh Baghdadi and et al. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.
- [4] Tianqi Chen and et al. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.
- [5] Tianqi Chen and et al. Tvm: An automated end-to-end optimizing compiler for deep learning, Oct 2018.
- [6] A. Demidovskij, Y. Gorbachev, and et al. Openvino deep learning workbench: Comprehensive analysis and tuning of neural networks inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 783–787, 2019.
- [7] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.
- [8] Hassan and et al. Performance evaluation of matrix-matrix multiplications using intel’s advanced vector extensions (avx). *Microprocessors and Microsystems*, 47:369–374, 2016.
- [9] N. Jouppi, C. Young, N. Patil, and D. Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, 2018.
- [10] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 3–12. IEEE, 2009.
- [11] Chris Lattner and et al. Mlir: A compiler infrastructure for the end of moore’s law, 2020.
- [12] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.
- [13] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey, 2020.
- [14] W. Lin, D. Tsai, L. Tang, C. Hsieh, C. Chou, P. Chang, and L. Hsu. Onnc: A compilation framework connecting onnx to proprietary deep learning accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 214–218, 2019.

- [15] W. Lin, D. Tsai, L. Tang, C. Hsieh, C. Chou, P. Chang, and L. Hsu. Onnc: A compilation framework connecting onnx to proprietary deep learning accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 214–218, 2019.
- [16] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on cpus. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1025–1040, 2019.
- [17] S. m. Yoo and et al. Structure of deep learning inference engines for embedded systems. In *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 920–922, 2019.
- [18] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. " O'Reilly Media, Inc.", 2013.
- [19] Adam Paszke, Gross, and et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019.
- [20] Jonathan Ragan-Kelley and et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.
- [21] Jared Roesch and et al. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018.
- [22] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.
- [23] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.
- [24] Han Vanholder. Efficient inference with tensorrt, 2016.
- [25] J Welser, JW Pitera, and C Goldberg. Future computing hardware for ai. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 1–3. IEEE, 2018.
- [26] Hongwei Xi. Dead code elimination through dependent types. In *International Symposium on Practical Aspects of Declarative Languages*, pages 228–242. Springer, 1999.
- [27] Xing and et al. An in-depth comparison of compilers for deep neural networks on hardware. In *2019 IEEE International Conference on Embedded Software and Systems (ICESSE)*, pages 1–8. IEEE, 2019.
- [28] R. Xu, F. Han, and Q. Ta. Deep learning at scale on nvidia v100 accelerators. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 23–32, 2018.