The Deep Learning Compiler: A Comprehensive Survey

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, *Member, IEEE*, Guangwen Yang, *Member, IEEE*, and Depei Qian

Abstract—The difficulty of deploying various deep learning (DL) models on diverse DL hardware has boosted the research and development of DL compilers in the community. Several DL compilers have been proposed from both industry and academia such as Tensorflow XLA and TVM. Similarly, the DL compilers take the DL models described in different DL frameworks as input, and then generate optimized codes for diverse DL hardware as output. However, none of the existing survey has analyzed the unique design architecture of the DL compilers comprehensively. In this article, we perform a comprehensive survey of existing DL compilers by dissecting the commonly adopted design in details, with emphasis on the DL oriented multi-level IRs, and frontend/backend optimizations. We present detailed analysis on the design of multi-level IRs and illustrate the commonly adopted optimization techniques. Finally, several insights are highlighted as the potential research directions of DL compiler. This is the first survey article focusing on the design architecture of DL compilers, which we hope can pave the road for future research towards DL compiler.

Index Terms—Neural networks, deep learning, compiler, intermediate representation, optimization

1 Introduction

The development of deep learning (DL) has generated a profound impact on various scientific fields. It has not only demonstrated remarkable value in artificial intelligence such as natural language processing (NLP) [1] and computer vision (CV) [2], but also proved great success in broader applications such as e-commerce [3], smart city [4] and drug discovery [5]. With the emergence of versatile deep learning models such as convolutional neural network (CNN) [6], recurrent neural network (RNN) [7], long short-term memory (LSTM) [8] and generative adversarial network (GAN) [9], it is critical to ease the programming of diverse DL models in order to realize their wide adoption.

With the continuous efforts from both industry and academia, several popular DL frameworks have been proposed such as TensorFlow [10], PyTorch [11], MXNet [12] and CNTK [13], in order to simplify the implementation of various DL models. Although there are strengths and weaknesses among the above DL frameworks depending on the tradeoffs in their designs, the interoperability becomes important to reduce the redundant engineering efforts when supporting emerging DL models across the existing DL models. To provide interoperability, ONNX [14] has been proposed, which defines a unified format for representing DL models to facilitate model conversion between different DL frameworks.

Meanwhile, the unique computing characteristics such as matrix multiplication have spurred the passion of chip architects to design customized DL accelerators for higher efficiency. Internet giants (e.g., Google TPU [15], Hisilicon NPU [16], Apple Bonic [17]), processor vendors (e.g., NVIDIA Turing [18], Intel NNP [19]), service providers (e.g., Amazon Inferentia [20], Alibaba Hanguang [21]), and even startups (e.g., Cambricon [22], Graphcore [23]) are investing tremendous workforce and capital in developing DL chips in order to boost the performance for DL models. Generally, the DL hardware can be divided into the following categories: *1)* general-purpose hardware with software-hardware co-design, *2)* dedicated hardware fully customized for DL models, and *3)* neuromorphic hardware inspired by biological brain science. For example, the general-purpose hardware (e.g., CPU, GPU) has added special hardware components such as AVX512 vector units and tensor cores to accelerate DL models. Whereas for dedicated hardware such as Google TPU, application-specific integrated circuits (e.g., matrix multiplication engine and high-bandwidth memory) have been designed to elevate the performance and energy efficiency to extreme. To the foreseeable future, the design of DL hardware would become even more diverse.

To embrace the hardware diversity, it is important to map the computation to DL hardware efficiently. On general-purpose hardware, the highly optimized linear algebra libraries such as Basic Linear Algebra Subprograms (BLAS) libraries (e.g., MKL and cuBLAS) serve as the basics for efficient computation of DL models. Take the convolution operation for example, the DL frameworks convert the convolution to matrix multiplication and then invoke the

• *Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian are with the School of Computer Science and Engineering, Beihang University, Beijing 100191, China. E-mail: {lmzhhh, yi.liu, liuxiaoyan, sunqingxiao, youxin2015, hailong.yang, zhongzhi.luan, depeiq}@buaa.edu.cn.*
• *Lin Gan and Guangwen Yang are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {lingan, ygw}@tsinghua.edu.cn.*

Manuscript received 28 Mar. 2020; revised 27 Aug. 2020; accepted 8 Oct. 2020. Date of publication 13 Oct. 2020; date of current version 28 Oct. 2020. (Corresponding author: Hailong Yang.) Recommended for acceptance by P. Balaji. Digital Object Identifier no. 10.1109/TPDS.2020.3030548

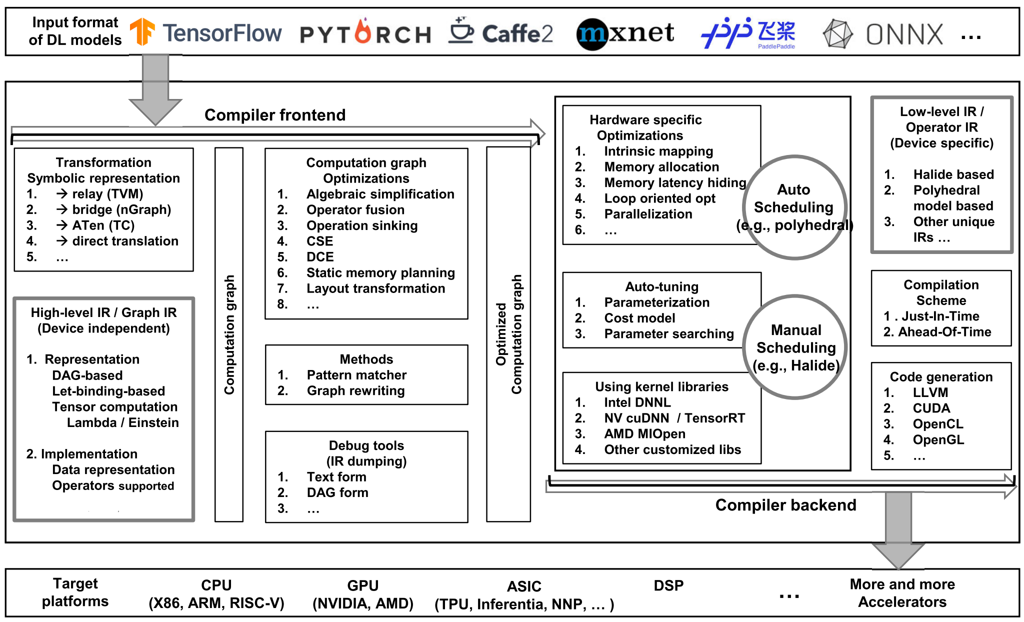


Fig. 1. The overview of commonly adopted design architecture of DL compilers.

GEMM function in the BLAS libraries. In addition, the hardware vendors have released specially optimized libraries tailored for DL computations (e.g., MKL-DNN and cuDNN), including forward and backward convolution, pooling, normalization, and activation. More advanced tools have also been developed to further speedup DL operations. For example, TensorRT [24] supports graph optimization (e.g., layer fusion) and low-bit quantization with large collection of highly optimized GPU kernels. On dedicated DL hardware, similar libraries are also provided [22], [23]. However, the drawback of relying on the libraries is that they usually fall behind the rapid development of DL models, and thus fail to utilize the DL chips efficiently.

To address the drawback of DL libraries and tools, as well as alleviate the burden of optimizing the DL models on each DL hardware manually, the DL community has resorted to the domain-specific compilers for rescue. Rapidly, several popular DL compilers have been proposed such as TVM [25], Tensor Comprehensions [26], Glow [27], nGraph [28] and XLA [29], from both industry and academia. The DL compilers take the model definitions described in the DL frameworks as inputs, and generate efficient code implementations on various DL hardware as outputs. The transformation between model definition and specific code implementation is highly optimized, targeting the model specification and hardware architecture. Specifically, they incorporate DL oriented optimizations such as layer and operator fusion, which enables highly efficient code generation. Moreover, existing DL compilers also leverage mature tool-chains from general-purpose compilers (e.g., LLVM [30]), which provides better portability across diverse hardware architectures. Similar to traditional compiler, DL compilers also adopt the layered design, including frontend, intermediate representation (IR), and backend. However, the uniqueness of the DL compiler lies in the design of multi-level IRs and DL specific optimizations.

In this paper, we provide a comprehensive survey of existing DL compilers by dissecting the compiler design into frontend, multi-level IRs and backend, with special emphasis on the IR design and optimization methods. To the best of our knowledge, this is the first paper that provides a comprehensive survey on the design of DL compiler. Specifically, this paper makes the following contributions:

- We dissect the commonly adopted design architecture of existing DL compilers, and provide detailed analysis of the key design components such as multi-level IRs, frontend optimizations (including node-level, block-level and dataflow-level optimizations) and backend optimizations (including hardware-specific optimization, auto-tuning and optimized kernel libraries).
- We provide a comprehensive taxonomy of existing DL compilers from various aspects, which corresponds to the key components described in this survey. The target of this taxonomy is to provide guidelines about the selection of DL compilers for the practitioners considering their requirements, as well as to give a thorough summary of the DL compilers for researchers.
- We have provided the quantitative performance comparison among DL compilers on CNN models, including full-fledged models and lightweight models. We have compared both end-to-end and per-layer (convolution layers since they dominate the inference time) performance to show the effectiveness of optimizations. The evaluation scripts and results are open sourced[1] for reference.
- We highlight several insights for the future development of DL compilers, including dynamic shape and pre-/post-processing, advanced auto-tuning, polyhedral model, subgraph partitioning, quantization, unified optimizations, differentiable programming

1. https://github.com/buaa-hipo/dlcompiler-comparison

and privacy protection, which we hope to boost the research in the DL compiler community.

The rest of this paper is organized as follows. Section 2 describes the common design architecture of DL compilers. Section 3 discusses the key components of DL compilers, including multi-level IRs, frontend optimizations and backend optimizations. Section 4 presents a comprehensive taxonomy. Section 5 provides the quantitative performance comparison. Section 6 highlights the future directions for DL compiler research.

2 Common Design Architecture of DL Compilers

The common design architecture of a DL compiler primarily contains two parts: the compiler frontend and the compiler backend, as shown in Fig. 1. The intermediate representation (IR) is spread across both the frontend and the backend. Generally, IR is an abstraction of the program and is used for program optimizations. Specifically, the DL models are translated into multi-level IRs in DL compilers, where the high-level IR resides in the frontend, and the low-level IR resides in the backend. Based on the high-level IR, the compiler frontend is responsible for hardware-independent transformations and optimizations. Based on the low-level IR, the compiler backend is responsible for hardware-specific optimizations, code generation, and compilation. Note that this survey focuses on the design principles of DL compilers. For functional and experimental comparisons of DL compilers, the readers can refer to [31], [32].

The high-level IR, also known as graph IR, represents the computation and the control flow and is hardware-independent. The design challenge of high-level IR is the ability of abstraction of the computation and the control flow, which can capture and express diverse DL models. The goal of the high-level IR is to establish the control flow and the dependency between the operators and the data, as well as provide an interface for graph-level optimizations. It also contains rich semantic information for compilation as well as offers extensibility for customized operators. The detailed discussion of high-level IR is presented in Section 3.1.

The low-level IR is designed for hardware-specific optimization and code generation on diverse hardware targets. Thus, the low-level IR should be fine-grained enough to reflect the hardware characteristics and represent the hardware-specific optimizations. It should also allow the use of mature third-party tool-chains in compiler backends such as Halide [33], polyhedral model [34], and LLVM [30]. The detailed discussion of low-level IR is presented in Section 3.2.

The frontend takes a DL model from existing DL frameworks as input, and then transforms the model into the computation graph representation (e.g., graph IR). The frontend needs to implement various format transformations To support the diverse formats in different frameworks. The computation graph optimizations incorporate the optimization techniques from both general-purpose compilers and the DL specific optimizations, which reduce the redundancy and improve the efficiency upon the graph IR. Such optimizations can be classified into node-level (e.g., nop elimination and zero-dim-tensor elimination), block-level (e.g., algebraic simplification, operator fusion, and operator sinking) and dataflow-level (e.g., CSE, DCE, static memory planning, and layout transformation). After the frontend, the optimized computation graph is generated and passed to the backend. The detailed discussion of the frontend is presented in Section 3.3.

The backend transforms the high-level IR into low-level IR and performs hardware-specific optimizations. On the one hand, it can directly transform the high-level IR to third-party tool-chains such as LLVM IR to utilize the existing infrastructures for general-purpose optimizations and code generation. On the other hand, it can take advantage of the prior knowledge of both DL models and hardware characteristics for more efficient code generation, with customized compilation passes. The commonly applied hardware-specific optimizations include hardware intrinsic mapping, memory allocation and fetching, memory latency hiding, parallelization as well as loop oriented optimizations. To determine the optimal parameter setting in the large optimization space, two approaches are widely adopted in existing DL compilers such as auto-scheduling (e.g., polyhedral model) and auto-tuning (e.g., AutoTVM). The optimized low-level IR is compiled using JIT or AOT to generate codes for different hardware targets. The detailed discussion of the backend is presented in Section 3.4.

3 Key Components of DL Compilers

3.1 High-Level IR

To overcome the limitation of IR adopted in traditional compilers that constrains the expression of complex computations used in DL models, existing DL compilers leverage high-level IR (as known as graph IR) with special designs for efficient code optimizations. To better understand the graph IR used in the DL compilers, we describe the representation and implementation of graph IR as follows.

3.1.1 Representation of Graph IR

The representation of graph IR influences the expressiveness of graph IR and also decides the way the DL compilers analyze the graph IR.

DAG-Based IR. DAG-based IR is one of the most traditional ways for the compilers to build a computation graph, with nodes and edges organized as a directed acyclic graph (DAG). In DL compilers [25], [26], [27], [28], [29], the nodes of a DAG represent the atomic DL operators (convolution, pooling, etc.), and the edges represent the tensors. And the graph is acyclic without loops, which differs from the data dependence graphs [35] (DDG) of generic compilers [30], [36]. And with the help of the DAG computation graph, DL compilers can analyze the relationship and dependencies between various operators and use them to guide the optimizations. There are already plenty of optimizations on DDG, such as common sub-expression elimination (CSE) and dead code elimination (DCE). By combining the domain knowledge of DL with these algorithms, further optimizations can be applied to the DAG computation graph, which will be elaborated in Section 3.3. DAG-based IR is convenient for programming and compiling due to its simplicity, but it has deficiencies such as semantic ambiguity caused by the missing definition of computation scope.

Let-Binding-Based IR. Let-binding is one method to solve the semantic ambiguity by offering *let* expression to certain functions with restricted scope used by many high-level programming languages such as Javascript [37], F# [38], and Scheme [39]. When using the *let* keyword to define an expression, a *let* node is generated, and then it points to the operator and variable in the expression instead of just building computational relation between variables as a DAG. In DAG-based compiler, when a process needs to get the return value of one expression, it first accesses the corresponding node and searches related nodes, also known as recursive descent technique. In contrast, the let-binding based compiler figures out all results of the variables in *let* expression and builds a variable map. When a particular result is needed, the compiler looks up this map to decide the result of the expression. Among the DL compilers, the Relay IR [40] of TVM adopts both DAG-based IR and let-binding-based IR to obtain the benefits of both.

Representing Tensor Computation. Different graph IRs have different ways to represent the computation on tensors. The operators of diverse DL frameworks are translated to graph IRs according to such specific representations. And the customized operators also need to be programmed in such representation. The representation of tensor computation can be divided into the following three categories.

1) *Function-based:* The function-based representation just provides encapsulated operators, which is adopted by Glow, nGraph and XLA. Take High Level Optimizer (HLO, the IR of XLA) for example, it consists of a set of functions in symbolic programming, and most of them have no side-effect. The instructions are organized into three levels, including HloModule (the whole program), HloComputaion (a function), and HloInstruction (the operation). XLA uses HLO IR to represent both graph IR and operation IR so that the operation of HLO ranges from the dataflow level to the operator level.
2) *Lambda expression:* The lambda expression, an index formula expression, describes calculation by variable binding and substitution. Using lambda expression, programmers can define a computation quickly without implementing a new function. TVM represents the tensor computation using the tensor expression, which is based on the lambda expression. In TVM, computational operators in tensor expression are defined by the shape of output tensor and the lambda expression of computing rules.
3) *Einstein notation:* The Einstein notation, also known as the summation convention, is a notation to express summation. Its programming simplicity is superior to lambda expression. Taking TC for example, the indexes for temporary variables do not need to be defined. The IR can figure out the actual expression by the occurrence of undefined variables based on Einstein notation. In Einstein notation, the operators need to be associative and commutative. This restriction guarantees the reduction operator can be executed by any order, making it possible for further parallelization.

3.1.2 Implementation of Graph IR

The implementation of graph IR in DL compilers fulfills the management of data and operation.

Data Representation. The data in DL compilers (e.g., inputs, weights, and intermediate data) are usually organized in the form of tensors, which are also known as multi-dimensional arrays. The DL compilers can represent tensor data directly by memory pointers, or in a more flexible way by placeholders. A placeholder contains the size for each dimension of a tensor. Alternatively, the dimension sizes of the tensor can be marked as unknown. For optimizations, the DL compilers require the data layout information. In addition, the bound of iterators should be inferred according to the placeholders.

1) *Placeholder:* Placeholder is widely used in symbolic programming (e.g., Lisp [41], Tensorflow [10]). A placeholder is simply a variable with explicit shape information (e.g., size in each dimension), and it will be populated with values at the later stage of the computation. It allows the programmers to describe the operations and build the computation graph without concerning the exact data elements, which helps separate the computation definition from the exact execution in DL compilers. Besides, it is convenient for the programmers to change the shape of input/output and other corresponding intermediate data by using placeholders without changing the computation definition.
2) *Unknown (Dynamic) shape representation:* The unknown dimension size is usually supported when declaring the placeholders. For instance, TVM uses *Any* to represent an unknown dimension (e.g., $Tensor\langle(Any, 3), fp32\rangle$); XLA uses *None* to achieve the same purpose (e.g., $tf.placeholder("float", [None, 3])$); nGraph uses its *PartialShape* class. The unknown shape representation is necessary to support the dynamic model. However, to fully support dynamic model, the bound inference and dimension checking should be relaxed. In addition, extra mechanism should be implemented to guarantee memory validity.
3) *Data layout:* The data layout describes how a tensor is organized in memory, and it is usually a mapping from logical indices to memory indices. The data layout usually includes the sequence of dimensions (e.g., NCHW and NHWC), tiling, padding, striding, etc. TVM and Glow represent data layout as operator parameters and require such information for computation and optimization. However, combining data layout information with operators rather than tensors enables intuitive implementation for certain operators and reduces the compilation overhead. XLA represents data layout as constraints related to its backend hardware. Relay and MLIR are going to add data layout information into their type systems for tensors.
4) *Bound inference:* The bound inference is applied to determine the bound of iterators when compiling DL models in DL compilers. Although the tensor representation in DL compilers is convenient to describe the inputs and outputs, it exposes special

challenges for inferring the iterator bound. The bound inference is usually performed recursively or iteratively, according to the computation graph and the known placeholders. For example, in TVM the iterators form a directed acyclic hyper-graph, where each node of the graph represents an iterator and each hyper-edge represents the relation (e.g., *split*, *fuse* or *rebase*) among two or more iterators. Once the bound of the root iterator is determined based on the shapes of placeholders, other iterators can be inferred according to the relations recursively.

Operators Supported. The operators supported by DL compilers are responsible for representing the DL workloads, and they are nodes of the computation graph. The operators usually include algebraic operators (e.g., $+$, $\times$, $\exp$ and topK), neural network operators (e.g., convolution and pooling), tensor operators (e.g., reshape, resize and copy), broadcast and reduction operators (e.g., min and argmin), as well as control flow operators (e.g., conditional and loop). Here, we choose three representative operators that are frequently used across different DL compilers for illustration. In addition, we discuss the case for customized operators.

1) *Broadcast:* The *broadcast* operators can replicate the data and generate new data with compatible shape. Without *broadcast* operators, the input tensor shapes are more constrained. For example, for an *add* operator, the input tensors are expected to be of the same shape. Some compilers such as XLA and Relay relax such restriction by offering the *broadcasting* operator. For example, XLA allows the element-wise addition on a matrix and a vector by replicating it until its shape matches the matrix.
2) *Control flow:* Control flow is needed when representing complex and flexible models. Models such as RNN and Reinforcement learning (RL) depend on recurrent relations and data-dependent conditional execution [42], which requires control flow. Without supporting control flow in graph IR of DL compilers, these models must rely on the control flow support of the host languages (e.g., *if* and *while* in Python) or static unrolling, which deteriorates the computation efficiency. Relay notices that arbitrary control flow can be implemented by recursion and pattern, which has been demonstrated by functional programming [40]. Therefore, it provides *if* operator and recursive function for implementing control flow. On the contrary, XLA represents control flow by special HLO operators such as *while* and *conditional*.
3) *Derivative:* The derivative operator of an operator *Op* takes the output gradients and the input data of *Op* as its inputs, and then calculates the gradient of *Op*. Although some DL compilers (e.g., TVM and TC) support automatic differentiation [43], they require the derivatives of all operators in high-level IR when the chain rule is applied. TVM is working towards providing the derivative operators of both algebraic operators and neural network operators. The programmers can use these derivative operators for building the derivatives of customized operators. On the contrary, PlaidML can generate derivative operators automatically, even for customized operators. Notably, DL compilers unable to support derivative operators fail to provide the capability of model training.
4) *Customized operators:* It allows programmers to define their operators for a particular purpose. Providing support for customized operators improves the extensibility of DL compilers. For example, when defining new operators in Glow, the programmers need to realize the logic and node encapsulation. In addition, extra efforts are needed, such as the lowering step, operation IR generation, and instruction generation, if necessary. Whereas, TVM and TC require less programming efforts except describing the computation implementation. Specifically, the users of TVM only need to describe the computation and the schedule and declare the shape of input/output tensors. Moreover, the customized operators integrate Python functions through hooks, which further reduces the programmers' burden.

3.1.3 Discussion

Nearly all DL compilers have their unique high-level IRs. However, they share similar design philosophies, such as using DAG and let-binding to build the computation graph. In addition, they usually provide convenient ways for programmers to represent tensor computation. The data and operators designed in high-level IRs are flexible and extensible enough to support diverse DL models. More importantly, the high-level IRs are hardware-independent and thus can be applied with different hardware backend.

3.2 Low-Level IR

3.2.1 Implementation of Low-Level IR

Low-level IR describes the computation of a DL model in a more fine-grained representation than that in high-level IR, which enables the target-dependent optimizations by providing interfaces to tune the computation and memory access. In this section, we classify the common implementations of low-level IRs into three categories: Halide-based IR, polyhedral-based IR, and other unique IR.

Halide-Based IR. Halide is first proposed to parallelize image processing, and it is proven to be extensible and efficient in DL compilers (e.g., TVM). The fundamental philosophy of Halide is the separation of *computation* and *schedule*. Rather than giving a specific scheme directly, the compilers adopting Halide try various possible *schedule* and choose the best one. The boundaries of memory reference and loop nests in Halide are restricted to bounded boxes aligned to the axes. Thus, Halide cannot express the computation with complicated patterns (e.g., non-rectangular). Fortunately, the computations in DL are quite regular to be expressed perfectly by Halide. Besides, Halide can easily parameterize these boundaries and expose them to the tuning mechanism. The original IR of the Halide needs to be modified when applied to backend of DL compilers. For example, the input shape of Halide is infinite, whereas the DL compilers need to know the exact shape of data in order to map the operator to hardware instructions. Some compilers, such as TC, require the fixed size of data, to ensure better temporal locality for tensor data.

TVM has improved Halide IR into an independent symbolic IR by following efforts. It removes the dependency on LLVM and refactors the structure of both the project module and the IR design of Halide, pursuing better organization as well as accessibility for graph IR and frontend language such as Python. The re-usability is also improved, with a runtime dispatching mechanism implemented to add customized operators conveniently. TVM simplifies the variable definition from string matching to pointer matching, guaranteeing that each variable has a single define location (static single-assignment, SSA) [44]).

Polyhedral-Based IR. The polyhedral model is an important technique adopted in DL compilers. It uses linear programming, affine transformations, and other mathematical methods to optimize loop-based codes with static control flow of bounds and branches. In contrast to Halide, the boundaries of memory reference and loop nests can be polyhedrons with any shapes in the polyhedral model. Such flexibility makes polyhedral models widely used in generic compilers. However, such flexibility also prevents the integration with the tuning mechanisms. Nevertheless, due to the ability to deal with deeply nested loops, many DL compilers, such as TC and PlaidML (as the backend of nGraph), have adopted the polyhedral model as their low-level IR. The polyhedral-based IR makes it easy to apply various polyhedral transformations (e.g., fusion, tiling, sinking, and mapping), including both device-dependent and device-independent optimizations. There are many toolchains that are borrowed by polyhedral-based compilers, such as isl [45], Omega [46], PIP [47], Polylib [48], and PPL [49].

TC has its unique design in low-level IR, which combines the Halide and polyhedral model. It uses Halide-based IR to represent the computation and adopts the polyhedral-based IR to represent the loop structures. TC presents detailed expressions through abstract instances and introduces specific node types. In brief, TC uses the *domain* node to specify the ranges of index variables and uses the *context* node to describe new iterative variables that are related to hardware. And it uses the *band* node to determine the order of iterations. A *filter* node represents an iterator combined with a statement instance. *Set* and *sequence* are keywords to specify the execution types (parallel and serial execution) for *filters*. Besides, TC uses *extension* nodes to describe other necessary instructions for code generation, such as the memory movement.

PlaidML uses polyhedral-based IR (called Stripe) to represent tensor operations. It creates a hierarchy of parallelizable code by extending the nesting of parallel polyhedral blocks to multiple levels. Besides, it allows nested polyhedrons to be allocated to nested memory units, providing a way to match the computation with the memory hierarchy. In Stripe, the hardware configuration is independent of the kernel code. The *tags* in Stripe (known as *passes* in other compilers) do not change the kernel structure, but provide additional information about the hardware target for the optimization passes. Stripe splits the DL operators into *tiles* that fit into local hardware resources.

Other Unique IR. There are DL compilers implementing customized low-level IRs without using Halide and polyhedral model. Upon the customized low-level IRs, they apply hardware-specific optimizations and lowers to LLVM IR.

The low-level IR in Glow is an instruction-based expression that operates on tensors referenced by addresses [27]. There are two kinds of instruction-based functions in Glow low-level IR: *declare* and *program*. The first one declares the number of constant memory regions that live throughout the lifetime of the program (e.g., input, weight, bias). The second one is a list of locally allocated regions, including functions (e.g., conv and pool) and temporary variables. Instructions can run on the global memory regions or locally allocated regions. Besides, each operand is annotated with one of the qualifiers: *@in* indicates the operand reads from the buffer; *@out* indicates that the operand writes to the buffer; *@inout* indicates that the operand reads and writes to the buffer. These instructions and operand qualifiers help Glow determine when certain memory optimizations can be performed.

MLIR is highly influenced by LLVM, and it is a purer compiler infrastructure than LLVM. MLIR reuses many ideas and interfaces in LLVM, and sits between the model representation and code generation. MLIR has a flexible type system and allows multiple abstraction levels, and it introduces *dialects* to represent these multiple levels of abstraction. Each *dialect* consists of a set of defined immutable operations. The current *dialects* of MLIR include TensorFlow IR, XLA HLO IR, experimental polyhedral IR, LLVM IR, and TensorFlow Lite. The flexible transformations between *dialects* are also supported. Furthermore, MLIR can create new *dialects* to connect to a new low-level compiler, which paves the way for hardware developers and compiler researchers.

The HLO IR of XLA can be considered as both high-level IR and low-level IR because HLO is fine-grained enough to represent the hardware-specific information. Besides, HLO supports hardware-specific optimizations and can be used to emit LLVM IR.

3.2.2 Code Generation Based on Low-Level IR

The low-level IR adopted by most DL compilers can be eventually lowered to LLVM IR, and benefits from LLVM's mature optimizer and code generator. Furthermore, LLVM can explicitly design custom instruction sets for specialized accelerators from scratch. However, traditional compilers may generate poor code when passed directly to LLVM IR. In order to avoid this situation, two approaches are applied by DL compilers to achieve hardware-dependent optimization: *1)* perform target-specific loop transformation in the upper IR of LLVM (e.g., Halide-based IR and polyhedral-based IR), and *2)* provide additional information about the hardware target for the optimization passes. Most DL compilers apply both approaches, but the emphasis is different. In general, the DL compilers that prefer frontend users (e.g., TC, TVM, XLA, and nGraph) might focus on *1)*, whereas the DL compilers that are more inclined to backend developers (e.g., Glow, PlaidML, and MLIR) might focus on *2)*.

The compilation scheme in DL compilers can be mainly classified into two categories: just-in-time (JIT) and ahead-of-time (AOT). For JIT compilers, it can generate executable codes on the fly, and they can optimize codes with better runtime knowledge. AOT compilers generate all executable binaries first and then execute them. Thus they have a larger

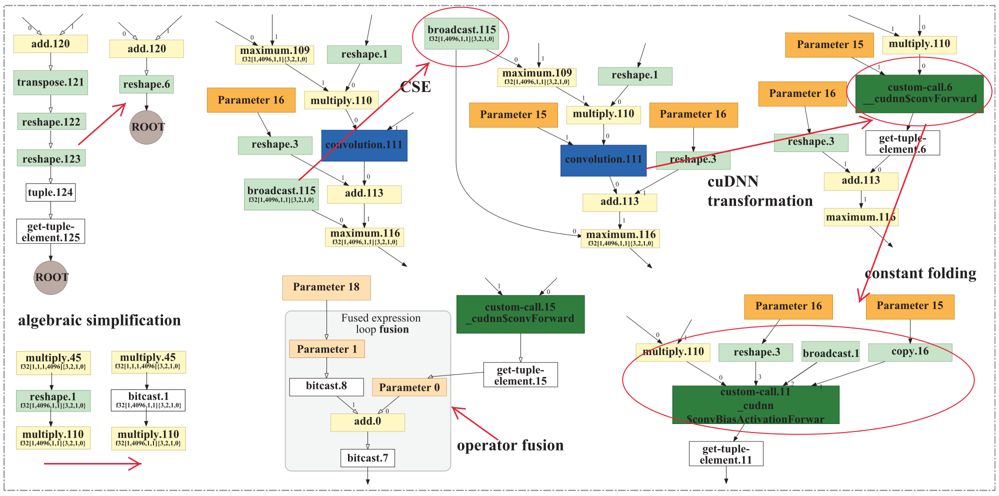


Fig. 2. Example of computation graph optimizations, taken from the HLO graph of Alexnet on Volta GPU using Tensorflow XLA.

scope in static analysis than JIT compilation. In addition, AOT approaches can be applied with cross-compilers of embedded platforms (e.g., C-GOOD [50]) as well as enable execution on remote machines (TVM RPC) and customized accelerators.

3.2.3 Discussion

In DL compilers, the low-level IR is a fine-grained representation of DL models, and it reflects detailed implantation of DL models on diverse hardware. The low-level IRs include Halide-based IRs, polyhedral-based IRs, and other unique IRs. Although they differ in designs, they leverage the mature compiler tool-chains and infrastructure, to provide tailored interfaces of hardware-specific optimizations and code generation. The design of low-level IRs can also impact the design of new DL accelerators (e.g., TVM HalideIR and Inferentia, as well as XLA HLO and TPU).

3.3 Frontend Optimizations

After constructing the computation graph, the frontend applies graph-level optimizations. Many optimizations are easier to be identified and performed at graph level because the graph provides a global view of the computation. These optimizations are only applied to the computation graph, rather than the implementations on backends. Thus they are hardware-independent and can be applied to various backend targets.

The frontend optimizations are usually defined by *passes*, and can be applied by traversing the nodes of the computation graph and performing the graph transformations. The frontend provides methods to *1)* capture the specific features from the computation graph and *2)* rewrite the graph for optimization. Besides the pre-defined *passes*, the developers can also define customized *passes* in the frontend. Most DL compilers can determine the shape of both input tensors and output tensors of every operation once a DL model is imported and transformed as a computation graph. This feature allows DL compilers to perform optimizations according to the shape information. Fig. 2 shows an example of computation graph optimizations with Tensorflow XLA.

In this section, we classify the frontend optimizations into three categories: *1)* node-level optimizations, *2)* block-level (peephole, local) optimizations, and *3)* dataflow-level (global) optimizations.

3.3.1 Node-Level Optimizations

The nodes of the computation graph are coarse enough to enable optimizations inside a single node. And the node-level optimizations include node elimination that eliminates unnecessary nodes and node replacement that replaces nodes with other lower-cost nodes.

In general-purpose compilers, Nop Elimination removes the no-op instructions which occupy a small amount of space but specify no operation. In DL compilers, Nop Elimination is responsible for eliminating the operations lacking adequate inputs. For example, the *sum* node with only one input tensor can be eliminated, the *padding* node with zero padding width can be eliminated.

Zero-dim-tensor elimination is responsible for removing the unnecessary operations whose inputs are zero-dimension tensors. Assume that A is a zero-dimension tensor, and B is a constant tensor, then the sum operation node of A and B can be replaced with the already existing constant node B without affecting the correctness. Assume that C is a 3-dimension tensor, but the shape of one dimension is zero, such as {0,2,3}, therefore, C has no element, and the argmin/argmax operation node can be eliminated.

3.3.2 Block-Level Optimizations

Algebraic simplification - The algebraic simplification optimizations consist of *1)* algebraic identification, *2)* strength reduction, with which we can replace more expensive operators by cheaper ones; *3)* constant folding, with which we can replace the constant expressions by their values. Such optimizations consider a sequence of nodes, then take advantage of commutativity, associativity, and distributivity of different kinds of nodes to simplify the computation.

In addition to the typical operators (+, ×, etc.), the algebraic simplification can also be applied to DL specific operators (e.g., *reshape*, *transpose*, and *pooling*). The operators can be reordered and sometimes eliminated, which reduces redundancy and improves the efficiency. Here we illustrate the common cases where algebraic simplification can be applied: *1) optimization of computation order,* in such case, the optimization finds and removes reshape/transpose operations according to specific characteristics. Taking the matrix multiplication (GEMM) for example, there are two matrices (e.g., A and B), both matrices are transposed (to produce A^T and B^T, respectively), then A^T and B^T are multiplied together. However, a more efficient way to implement GEMM is to switch the order of the arguments A and B, multiply them together, and then transpose the output of the GEMM, which reduces two *transpose* to just one; *2) optimization of node combination,* in such case, the optimization combines multiple consecutive *transpose* nodes into a single node, eliminates identity transpose nodes, and optimizes *transpose* nodes into *reshape* nodes when they actually move no data; *3) optimization of ReduceMean nodes,* in such case, the optimization performs substitutions of ReduceMean with AvgPool node (e.g., in Glow), if the input of the reduce operator is 4D with the last two dimensions to be reduced.

Operator Fusion. Operator fusion is indispensable optimization of DL compilers. It enables better sharing of computation, eliminates intermediate allocations, facilitates further optimization by combining loop nests [40], as well as reduces launch and synchronization overhead [26]. In TVM, the operators are classified into four categories: injective, reduction, complex-out-fusible, and opaque. When the operators are defined, their corresponding categories are determined. Targeting the above categories, TVM designs the fusion rules across operators. In TC, fusion is performed differently based on the automatic polyhedron transformations. However, how to identify and fuse more complicated graph patterns, such as blocks with multiple broadcast and reduce nodes, remains to be a problem. Recent works [51], [52] try to tackle this problem and propose a framework to explore and optimize aggressive fusion plans. It supports not only element-wise and reduction nodes, but also other computation/memory intensive nodes with complex dependencies.

Operator Sinking. This optimization sinks the operations such as transposes below operations such as batch normalization, ReLU, sigmoid, and channel shuffle. By this optimization, many similar operations are moved closer to each other, creating more opportunities for algebraic simplification.

3.3.3 Dataflow-Level Optimizations

Common Sub-Expression Elimination (CSE). An expression E is a common sub-expression if the value of E is previously computed, and the value of E has not to be changed since previous computation [53]. In this case, the value of E is computed once, and the already computed value of E can be used to avoid recomputing in other places. The DL compilers search for common sub-expressions through the whole computation graph and replace the following common sub-expressions with the previously computed results.

Dead Code Elimination (DCE). A set of code is dead if its computed results or side-effects are not used. And the DCE optimization removes the dead code. The dead code is usually not caused by programmers but is caused by other graph optimizations. Thus, the DCE, as well as CSE, are applied after other graph optimizations. Other optimizations, such as dead store elimination (DSE), which removes stores into tensors that are never going to be used, also belong to DCE.

Static Memory Planning. Static memory planning optimizations are performed to reuse the memory buffers as much as possible. Usually, there are two approaches: in-place memory sharing and standard memory sharing. The in-place memory sharing uses the same memory for input and output for an operation, and just allocates one copy of memory before computing. Standard memory sharing reuses the memory of previous operations without overlapping. The static memory planning is done offline, which allows more complicated planning algorithms to be applied. A recent work [54] first designs and performs memory-aware scheduling to minimize the peak activation memory footprint on edge devices, which presents new research directions of memory planning on memory-constrained devices.

Layout Transformation. Layout transformation tries to find the best data layouts to store tensors in the computation graph and then inserts the layout transformation nodes to the graph. Note that the actual transformation is not performed here, instead, it will be performed when evaluating the computation graph by the compiler backend.

In fact, the performance of the same operation in different data layouts is different, and the best layouts are also different on different hardware. For example, operations in the NCHW format on GPU usually run faster, so it is efficient to transform to NCHW format on GPU (e.g., TensorFlow). Some DL compilers rely on hardware-specific libraries to achieve higher performance, and the libraries may require certain layouts. Besides, some DL accelerators prefer more complicated layouts (e.g., tile). In addition, edge devices usually equip heterogenous computing units, and different units may require different data layouts for better utilization, thus layout transformation needs careful considerations. Therefore, the compilers need to provide a way to perform layout transformations across various hardware.

Not only the data layouts of tensors have a nontrivial influence on the final performance, but also the transformation operations have a significant overhead. Because they also consume the memory and computation resource.

A recent work [55] based on TVM targeting on CPUs alters the layout of all convolution operations to NCHW[x]c first in the computation graph, in which c means the split sub-dimension of channel C and x indicates the split size of the sub-dimension. Then all x parameters are globally explored by auto-tuning when providing hardware details, such as cache line size, vectorization unit size, and memory access pattern, during hardware-specific optimizations.

3.3.4 Discussion

The frontend is one of the most important components in DL compilers, which is responsible for transformation from DL models to high-level IR (e.g., computation graph) and hardware-independent optimizations based on high-level IR. Although the implementation of frontend may differ in the data representation and operator definition of high-level IR across DL compilers, the hardware-independent optimizations converge at three levels: node-level, block-level, and dataflow-level. The optimization methods at each level leverage the DL specific as well as general compilation optimization techniques, which reduce the computation redundancy

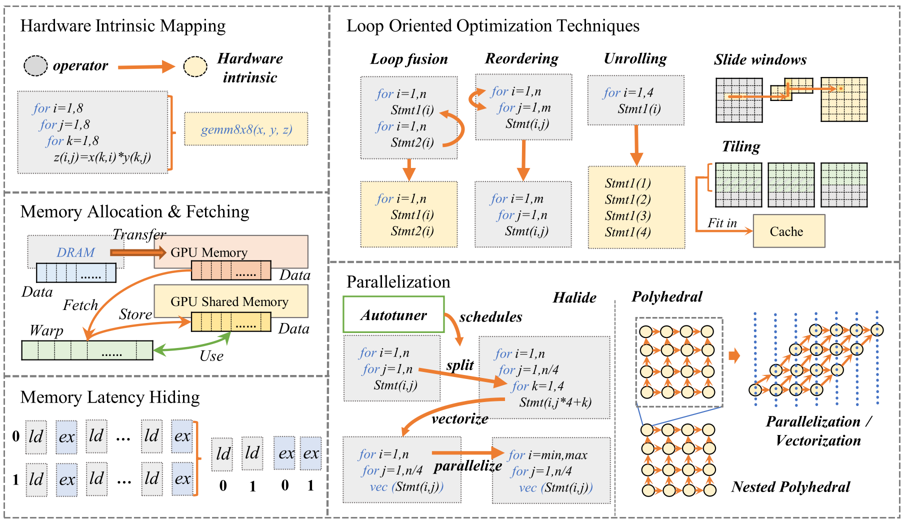


Fig. 3. Overview of hardware-specific optimizations applied in DL compilers.

as well as improve the performance of DL models at the computation graph level.

3.4 Backend Optimizations

The backends of DL compilers have commonly included various hardware-specific optimizations, auto-tuning techniques, and optimized kernel libraries. Hardware-specific optimizations enable efficient code generation for different hardware targets. Whereas, auto-tuning has been essential in the compiler backend to alleviate the manual efforts to derive the optimal parameter configurations. Besides, highly-optimized kernel libraries are also widely used on general-purpose processors and other customized DL accelerators.

3.4.1 Hardware-Specific Optimization

Hardware-specific optimizations, also known as target-dependent optimizations, are applied to obtain high-performance codes targeting specific hardware. One way to apply the backend optimizations is to transform the low-level IR into LLVM IR, to utilize the LLVM infrastructure to generate optimized CPU/GPU codes. The other way is to design customized optimizations with DL domain knowledge, leveraging the target hardware more efficiently. Since hardware-specific optimizations are tailored for particular hardware and cannot be included exhaustively in this paper, we present five widely adopted approaches in existing DL compilers. The overview of these hardware-specific optimizations is shown in Fig. 3, and the detailed descriptions are provided as follows.

Hardware Intrinsic Mapping. Hardware intrinsic mapping can transform a certain set of low-level IR instructions to kernels that have already been highly optimized on the hardware. In TVM, the hardware intrinsic mapping is realized in the method of *extensible tensorization*, which can declare the behavior of hardware intrinsic and the lowering rule for intrinsic mapping. This method enables the compiler backend to apply hardware implementations as well as highly optimized handcraft micro-kernels to a specific pattern of operations, which results in a significant performance gain. Whereas, Glow supports hardware intrinsic mapping such as *quantization*. It can estimate the possible numeric range for each stage of the neural network and support profile-guided optimization to perform quantization automatically. Besides, Halide/TVM maps specific IR patterns to SIMD opcodes on each architecture to avoid the inefficiency of LLVM IR mapping when encountering vector patterns.

Memory Allocation and Fetching. Memory allocation is another challenge in code generation, especially for GPUs and customized accelerators. For example, GPU contains primarily shared memory space (lower access latency with limited memory size) and local memory space (higher access latency with large capacity). Such memory hierarchy requires efficient memory allocation and fetching techniques for improving data locality. To realize this optimization, TVM introduces the scheduling concept of *memory scope*. Memory scope schedule primitives can tag a compute stage as *shared* or *thread-local*. For compute stages tagged as *shared*, TVM generates code with shared memory allocation as well as cooperative data fetching, which inserts memory barrier at the proper code position to guarantee correctness. Besides, TC also provides similar features (known as *memory promotion*) by extending PPCG [56] compiler. However, TC only supports limited pre-defined rules. Particularly, TVM enables special buffering in accelerators through *memory scope* schedule primitives.

Memory Latency Hiding. Memory latency hiding is also an important technique used in the backend by reordering the execution pipeline. As most DL compilers support parallelization on CPU and GPU, memory latency hiding can be naturally achieved by hardware (e.g., warp context switching on GPU). But for TPU-like accelerators with *decoupled access-execute* (DAE) architecture, the backend needs to perform scheduling and fine-grained synchronization to obtain correct and efficient codes. To achieve better performance as well as reduce programming burden, TVM introduces *virtual threading* schedule primitive, which enables users to specify the data parallelism on virtualized multi-thread architecture. Then TVM lowers these virtually parallelized threads by inserting necessary memory barriers and interleaves the operations from these threads into a single instruction stream, which forms a better execution pipeline of each thread to hide the memory access latency.

Loop Oriented Optimizations. Loop oriented optimizations are also applied in the backend to generate efficient codes for target hardware. Since Halide and LLVM [30] (integrated with the polyhedral method) have already incorporated such optimization techniques, some DL compilers leverage Halide and LLVM in their backends. The key techniques applied in loop oriented optimizations include loop fusion, sliding windows, tiling, loop reordering, and loop unrolling.

1) *Loop fusion:* Loop fusion is a loop optimization technique that can fuse loops with the same boundaries for better data reuse. For compilers such as PlaidML, TVM, TC, and XLA, such optimization is performed by the Halide schedule or polyhedral approach, while Glow applies loop fusion by its *operator stacking*.
2) *Sliding windows:* Sliding windows is a loop optimization technique adopted by Halide. Its central concept is to compute values when needed and store them on the fly for data reuse until they are no longer required. As sliding windows interleaves the computation of two loops and make them serial, it is a tradeoff between parallelism and data reuse.
3) *Tiling:* Tiling splits loops into several tiles, and thus loops are divided into outer loops iterating through tiles and inner loops iterating inside a tile. This transformation enables better data locality inside a tile by fitting a tile into hardware caches. As the size of a tile is hardware-specific, many DL compilers determine the tiling pattern and size by auto-tuning.
4) *Loop reordering:* Loop reordering (also known as loop permutation) changes the order of iterations in a nested loop, which can optimize the memory access and thus increase the spatial locality. It is specific to data layout and hardware features. However, it is not safe to perform loop reordering when there are dependencies along the iteration order.
5) *Loop unrolling:* Loop unrolling can unroll a specific loop to a fixed number of copies of loop bodies, which allows the compilers to apply aggressive instruction-level parallelism. Usually, loop unrolling is applied in combination with loop split, which first splits the loop into two nested loops and then unrolls the inner loop completely.

Parallelization. As modern processors generally support multi-threading and SIMD parallelism, the compiler backend needs to exploit parallelism to maximize hardware utilization for high performance. Halide uses a schedule primitive called *parallel* to specify the parallelized dimension of the loop for thread-level parallelization and supports GPU parallelization by mapping loop dimensions tagged as *parallel* with annotation of *block* and *thread*. And it replaces a loop of size n with a *n-wide* vector statement, which can be mapped to hardware-specific SIMD opcodes through hardware intrinsic mapping. Stripe develops a variant of the polyhedral model called *nested polyhedral model*, which introduces *parallel polyhedral block* as its basic execution element of iteration. After this extension, a nested polyhedral model can detect hierarchy parallelization among levels of tiling and striding. In addition, some DL compilers rely on handcraft libraries such as Glow or optimized math libraries provided by hardware vendors (discussed in Section 3.4.3). In the meanwhile, Glow offloads the vectorization to LLVM because the LLVM auto-vectorizer works well when the information of tensor dimension and loop trip count is provided. However, exploiting the parallelism entirely by compiler backend allows to apply more domain-specific knowledge of DL models, and thus leads to higher performance at the expense of more engineering efforts.

3.4.2 Auto-Tuning

Due to the enormous search space for parameter tuning in hardware-specific optimizations, it is necessary to leverage auto-tuning to determine the optimal parameter configurations. Among the studied DL compilers in this survey, TVM, TC, and XLA support the auto-tuning. Generally, the auto-tuning implementation includes four key components, such as parameterization, cost model, searching technique, and acceleration.

Parameterization. 1) Data and target: The data parameter describes the specification of the data, such as input shapes. The target parameter describes hardware-specific characteristics and constraints to be considered during optimization scheduling and code generation. For example, for the GPU target, the hardware parameters such as shared memory and register size need to be specified. *2) Optimization options*: The optimization options include the optimization scheduling and corresponding parameters, such as loop oriented optimizations and tile size. In TVM, both pre-defined and user-defined scheduling, as well as parameters, are taken into consideration. Whereas, TC and XLA prefer to parameterize the optimizations, which have a strong correlation with performance and can be changed later at a low cost. For example, the minibatch dimension is one of the parameters that is usually mapped to grid dimensions in CUDA and can be optimized during auto-tuning.

Cost Model. The comparison of different cost models applied in auto-tuning are as follows. *1) Black-box model*: This model only considers the final execution time rather than the characteristics of the compilation task. It is easy to build a black-box model, but easily ends up with higher overhead and less optimal solution without the guidance of task characteristics. TC adopts this model. *2) ML-based cost model*: ML-based cost model is a statistical approach to predict performance using a machine learning method. It enables the model to update as the new configuration is explored, which helps achieve higher prediction accuracy. TVM and XLA adopt this kind of model, for example, gradient tree boosting model (GBDT) and feedforward neural network [57] (FNN) respectively. *3) Pre-defined cost model*: An approach based on a pre-defined cost model expects a perfect model built on the characteristics of the compilation task and able to evaluate the overall performance of the task. Compared to the ML-based model, the pre-defined model generates less computation overhead when applied, but requires large engineering efforts for re-building the model on each new DL model and hardware.

Searching Technique. 1) Initialization and searching space determination: The initial option can either be set randomly or based on the known configurations, such as configurations given by users or historical optimal configurations. In terms of searching space, it should be specified before auto-tuning. TVM allows developers to specify the searching space with their domain-specific knowledge and provides automatic search space extraction for each hardware target based on the computational description. In contrast, TC relies on the compilation cache and the predefined rules. *2) Genetic algorithm (GA)* [58]: GA considers each tuning parameter as genes and each configuration as a candidate. The new candidate is iteratively generated by crossover, mutation, and selection according to the fitness value, which is a metaheuristic inspired by the process of natural selection. And finally, the optimal candidate is derived. The rate of crossover, mutation, and selection is used for controlling the tradeoff between exploration and exploitation. TC adopts GA in its auto-tuning technique. *3) Simulated annealing algorithm (SA)* [59]: SA is also a metaheuristic inspired by annealing. It allows us to accept worse solutions in a decreasing probability, which can find the approximate global optimum and avoid the precise local optimum in a fixed amount of iterations. TVM adopts SA in its auto-tuning technique. *4) Reinforcement learning (RL)*: RL performs with learning to maximize reward given an environment by the tradeoff between exploration and exploitation. Chameleon [60] (built upon TVM) adopts RLRL in its auto-tuning technique.

Acceleration. 1) Parallelization: One direction for accelerating auto-tuning is parallelization. TC proposes a multi-thread, multi-GPU strategy considering that the genetic algorithm needs to evaluate all candidates in each generation. First, it enqueues candidate configurations and compiles them on multiple CPU threads. The generated code is evaluated on GPUs in parallel, and each candidate owns its fitness used by the parent choosing step. After finishing the whole evaluation, the new candidate is generated, and the new compilation job is enqueued, waiting for compiling on CPU. Similarly, TVM supports cross-compilation and RPC, allowing users to compile on the local machine and run the programs with different auto-tuning configurations on multiple targets. *2) Configuration reuse*: Another direction for accelerating auto-tuning is to reuse the previous auto-tuning configurations. TC stores the fastest known generated code version corresponding to the given configuration by compilation cache. The cache is queried before each kernel optimization during the compilation, and the auto-tuning is triggered if cache miss. Similarly, TVM produces a log file that stores the optimal configurations for all scheduling operators and queries the log file for best configurations during compilation. It is worth mentioning that TVM performs auto-tuning for each operator in Halide IR (e.g., conv2d), and thus the optimal configurations are determined for each operator separately.

3.4.3 Optimized Kernel Libraries

There are several highly-optimized kernel libraries widely used to accelerate DL training and inference on various hardware. DNNL (previously MKL-DNN) from Intel, cuDNN from NVIDIA, and MIOpen from AMD are widely used libraries. Both computation-intensive primitives (e.g., convolution, GEMM, and RNN) and memory bandwidth limited primitives (e.g., batch normalization, pooling, and shuffle) are highly optimized according to the hardware features (e.g., AVX-512 ISA, tensor cores). And customizable data layouts are supported to make it easy to integrate into DL applications and avoid frequent data layout transformations. Besides, low-precision training and inference, including FP32, FP16, INT8, and non-IEEE floating-point format bfloat16 [61] are also supported. Other customized DL accelerators also maintain their specific kernel libraries [22], [23].

Existing DL compilers, such as TVM, nGraph, and TC, can generate the function calls to these libraries during code generation. However, if DL compilers need to leverage the existing optimized kernel libraries, they should first transform the data layouts and fusion styles into the types that are pre-defined in kernel libraries. Such transformation may break the optimal control flow. Moreover, the DL compilers treat the kernel libraries as a black box. Therefore they are unable to apply optimizations across operators (e.g., operator fusion) when invoking kernel libraries. In sum, using optimized kernel libraries achieves significant performance improvement when the computation can be satisfied by specific highly-optimized primitives, otherwise it may be constrained from further optimization and suffer from less optimal performance.

3.4.4 Discussion

The backend is responsible for bare-metal optimizations and code generation based on low-level IR. Although the design of backends may differ due to various low-level IRs, their optimizations can be classified into hardware-specific optimizations: auto-tuning techniques, and optimized kernel libraries. These optimizations can be performed separately or combined, to achieve better data locality and parallelization by exploiting the hardware/software characteristics. Eventually, the high-level IR of DL models is transformed into efficient code implementation on different hardware.

4 Taxonomy of DL Compilers

The DL compilers studied in this survey include TVM, nGraph, Tensor Comprehensions (TC), Glow, and XLA. We select these compilers since they are well-known, well maintained, and most importantly, widely used. Thus, we can find enough papers, documents, and discussions from both industry and academia in order to study their designs and implementations in-depth. Table 1 illustrates the taxonomy of the selected DL compilers from four perspectives, including frontend, backend, IR, and optimizations, which corresponds with the key components described in this survey.

Specifically, we provide more information about the compilers to the best of our knowledge. We not only provide whether a compiler supports a specific feature, but also describe how to use this feature through its programming interface. In addition, we also describe the developing status of specific features and the reasons why specific features are not supported in particular compilers. The target of this taxonomy is to provide guidelines about the selection of DL compilers for the practitioners considering their requirements, as well as to give a thorough summary of the DL compilers for researchers.

TABLE 1
The Comparison of DL Compilers, Including TVM, nGraph, TC, Glow, and XLA

		TVM	**nGraph**	**TC**	**Glow**	**XLA**
	Developer	Apache	Intel	Facebook	Facebook	Google
Frontend	Programming	Python/C++ Lambda expression	Python/C++ Tensor expression	Python/C++ Einstein notation	Python/C++ Layer programming	Python/C++ Tensorflow interface
	ONNX support	✓ tvm.relay.frontend .from_onnx (built-in)	✓ Use ngraph-onnx (Python package)	×	✓ ONNXModelLoader (built-in)	✓ Use tensorflow-onnx (Python package)
	Framework support	tvm.relay.frontend .from_* (built-in) tensorflow/tflite/keras pytorch/caffe2 mxnet/coreml/darknet	tensorflow paddlepaddle (Use *-bridge, act as the backend)	(Define and optimize a TC kernel, which is finally called by other frameworks.) pytorch/other DLPack supported frameworks	pytorch/caffe2 tensorflowlite (Use built-in ONNXIFI interface)	Use tensorflow interface
	Training support	× Under developing (Support derivative operators now)	✓ Only on NNP-T processor	✓ (Support auto differentiation)	✓ (Limited support)	✓ Use tensorflow interface
	Quantization support	✓ int8/fp16	✓ int8 (include training)	×	✓ int8	✓ int8/int16 (Use tensorflow interface)
IR	High-/low-level IR	Relay/Halide	nGraph IR/None	TC IR/Polyhedral	Its own high-/low-level IR	HLO (Both high- and low- level)
	Dynamic shape	✓ (*Any*)	✓ (*PartialShape*)	×	×	✓ (*None*)
Optimization	Frontend opt	Hardware independent optimizations (refer to Section 3.3) Hardware specific optimizations (refer to Section 3.4) And hybrid optimizations				
	Backend opt					
	Autotuning	✓ (To select the best schedule parameters)	× (Call optimized kernel libraries, no need)	✓ (To reduce JIT overhead)	× (Additional info is already provided in IR)	✓ (On default convolution and gemm)
	Kernel libraries	✓ mkl/cudnn/cublas	✓ eigen/mkldnn/cudnn/ Others	×	×	✓ eigen/mkl/ cudnn/tensorrt
Backend	Compilation methods	JIT AOT (experimental)	JIT	JIT	JIT AOT (Use built-in executable bundles)	JIT AOT (Generate executable libraries)
	Supported devices	CPU/GPU/ARM FPGA/Customized (Use VTA)	CPU/Intel GPU/NNP GPU/Customized (Use OpenCL support in PlaidML)	Nvidia GPU	CPU/GPU Customized (Official docs)	CPU/GPU/TPU Customized (Official docs)

In Table 1, we present the features of each DL compiler, including developer, programming language, ONNX/framework support, training support, and quantization support in the frontend category, and we present the compilation methods and supported devices in the backend category. These features are summarized because they strongly affect the usage of DL compilers in particular scenarios. Based on these features, practitioners or researchers can easily decide which DL compiler they would like to work upon.

Table 1, together with Fig. 1 can serve as a systematic summary of this survey. Through them, readers can identify the features each compiler supports as well as the key components of each compiler. More detailed information is presented in the following sections.

5 Evaluation

5.1 Experimental Setup

Our experiments are conducted on two GPU-equipped machines, and the hardware configuration is shown in Table 2. We evaluate the performance of TVM (v0.6.0), nGraph (0.29.0-rc.0), TC (commit fd01443), Glow (commit 7e68188) and XLA (TensorFlow 2.2.0) on CPU and GPU. We select 19 neural network models in ONNX format as our datasets, which are converted from the Torchvison[2] model zoo and the GluonCV[3] model zoo. These models include

2. https://pytorch.org/docs/stable/torchvision/models.html
3. https://gluon-cv.mxnet.io/model_zoo/index.html

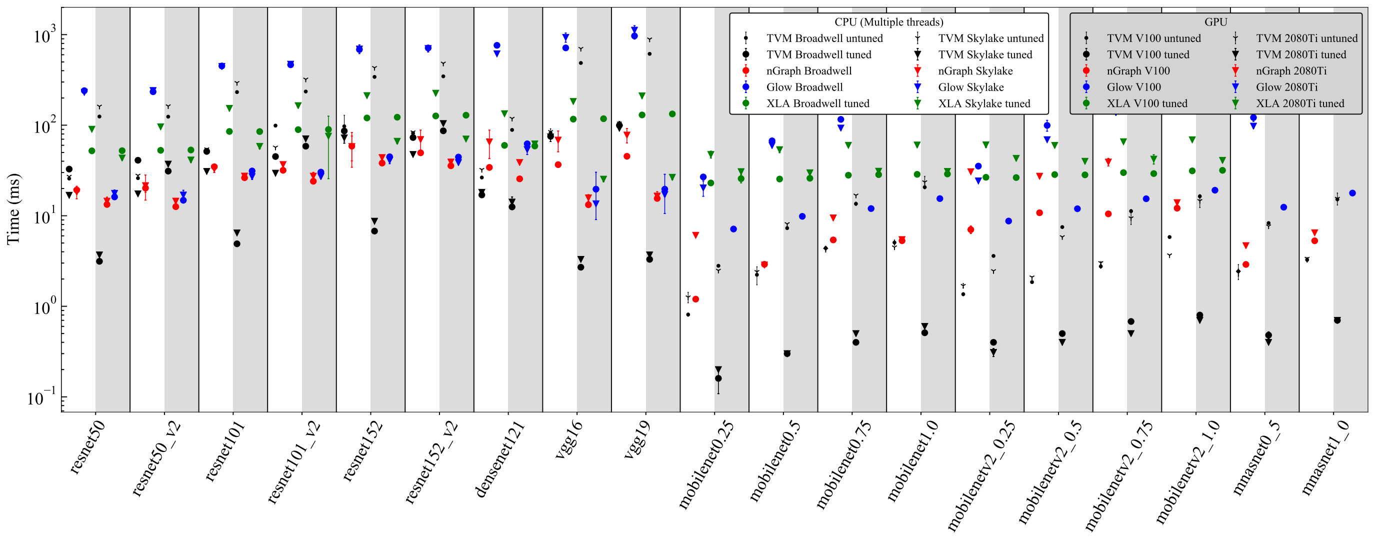


Fig. 4. The performance comparison of end-to-end inference across TVM, nGraph, Glow, and XLA on CPU and GPU.

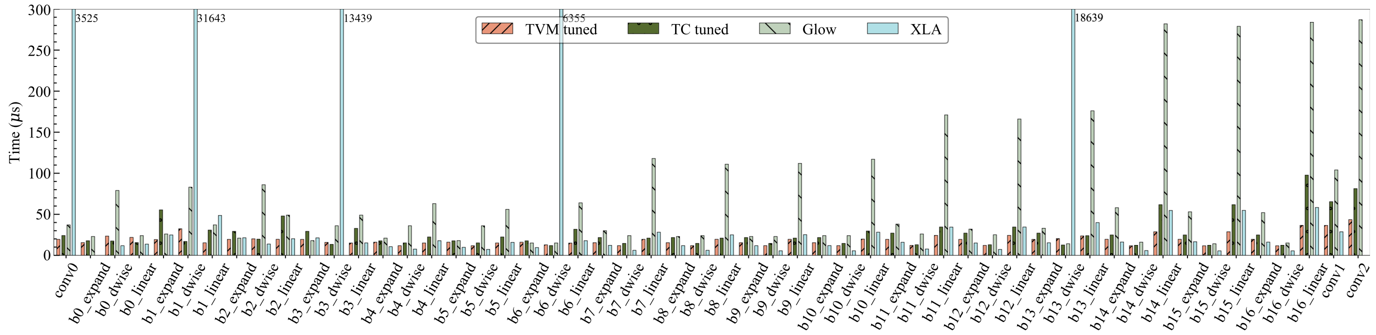


Fig. 5. The performance comparison of convolution layers in `MobileNetV2 _1.0` across TVM, TC, Glow, and XLA on V100 GPU.

full-fledged models: `ResNet`, `DenseNet` and `VGG` series, and lightweight models: `MobileNet` and `MNASNet` series. To import the ONNX models, as shown in Table 1, we use the built-in *tvm.relay.frontend.from_onnx* interface of TVM, the *ngraph-onnx* Python package of nGraph, the built-in *ONNXModelLoader* of Glow, and the *tensorflow-onnx* Python package of XLA. Notably, TC lacks the support of ONNX, so we only evaluate it in the following per-layer performance comparison. Each model is executed for 15 times, and we report the average execution time of the last 10 executions for each compiler, because we regard the first 5 executions as the warm-up to eliminate the overhead of JIT compilation.

5.2 End-to-End Performance Comparison

As shown in Fig. 4, we compare the performance of end-to-end inference across TVM, nGraph, Glow, and XLA. We evaluate these compilers on both CPUs (Broadwell and Skylake) and GPUs (V100 and 2080Ti). Note that, we omit the comparison of TC here. Because TC is more similar to a kernel library other than fully functional DL compiler, and it requires the users to implement all layers of a model with its Einstein notion manually, which leads to heavy engineering efforts for a fair comparison. Another reason is that TC only supports running on GPU, thus we cannot obtain its performance results on CPU. However, for detailed comparisons (Figs. 5 and 7), we still implement several `ResNet` and `MobileNetV2` models in TC. In sum, we compare and analyze the performance results from the following perspectives.

Compatibility. Although nGraph and XLA claims to support ONNX, there are still compatibility problems. *1)* nGraph fails to run the `DenseNet121`, `VGG16/19` and `MNASNet0_5/1_0` models due to tensors with dynamic shapes. Alternatively, we replace the `DenseNet121`, `VGG16/19` models with the corresponding models from the ONNX model zoo,[4] while `MNASNet0_5/1_0` models are not available. Besides, when we set PlaidML as the backend of nGraph on GPU, we fail to run all `MobileNet` models. Because PlaidML cannot handle the inconsistent definition of operators across different DL frameworks. *2)* XLA can run all selected models, however, the performance is quite

TABLE 2
The Hardware Configuration

	CPU	GPU
Platform a	Broadwell E5-2680v4 *2	Tesla V100 32GB
	(28 physical cores, 2.4GHz)	(15.7TFlops, FP32)
Platform b	Skylake Silver 4110 *2	Turing RTX2080Ti 11GB
	(16 physical cores, 2.1GHz)	(13.4TFlops, FP32)

4. https://github.com/onnx/models

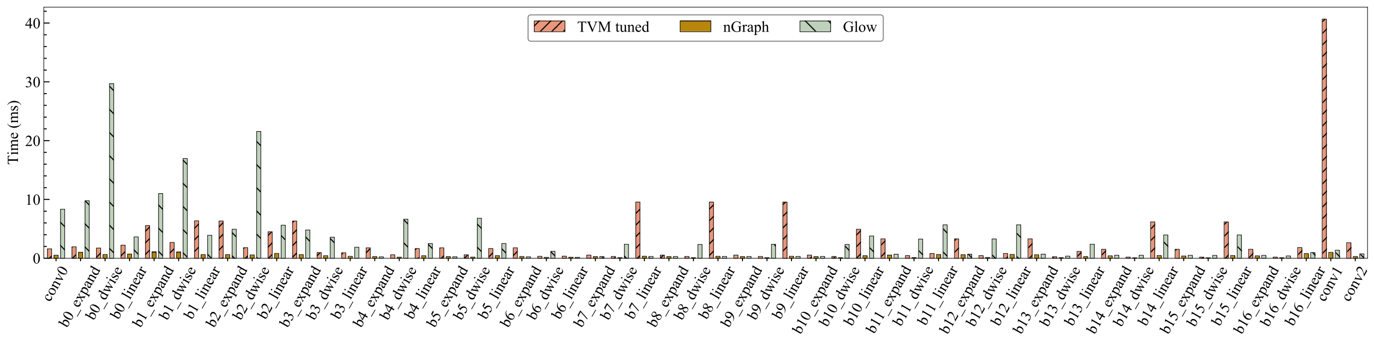


Fig. 6. The performance comparison of convolution layers in `MobileNetV2_1.0` across TVM, nGraph, and Glow on Broadwell CPU.

low. Thus, we replace the selected ONNX models with the *savedmodels* from the Tensorflow Hub,[5] while the `MNASNet0_5/1_0` models are not available. With models from Tensorflow Hub, XLA becomes two orders of magnitude faster, and the performance of XLA becomes competitive with other compilers.

Performance. From Fig. 4, we have several observations about the performance illustrated as follows.

1) On CPU, the performance of Glow is worse than other compilers. This is because Glow does not support thread parallelism. Thus it cannot fully utilize the multi-core CPU. Whereas TVM, nGraph, and XLA can leverage all CPU cores.

2) XLA has the similar end-to-end inference performance for both full-fledged models (`ResNet` *,* `DenseNet` *and* `VGG` *series) and lightweight models (* `MobileNet` *and* `MNASNet` *series). Besides, its inference performance on CPU and GPU is almost the same.* It is known that XLA is embedded in the Tensorflow framework. Tensorflow contains a complicated runtime compared to TVM, nGraph, and Glow, which introduces non-trivial overhead to XLA. In addition, if we increase the batch size (set to one by default in our evaluation) and focus on the throughput of DL compilers, then the overhead of XLA can be ignored with higher throughput.

3) In general, on CPU, TVM and nGraph achieve better performance across all models than other DL compilers, due to the limitations of Glow and XLA described above. TVM has comparable performance with nGraph on full-fledged models, while it is better than nGraph on lightweight models. nGraph relies on the DNNL (previously MKL-DNN) library for acceleration. Thus, nGraph can offload the optimized subgraphs to DNNL and benefit from DNNL's fine-grained instruction-level JIT optimizations tailored for Intel CPU.

4) The tuned TVM (tuned with 200 trials) *almost achieves the best performance on both CPU and GPU across all models, especially on lightweight models (*`MobileNet`*,* `MNASNet` *series).* Based on our investigation, this is because the schedules of classic operators inside these models have already been well designed by TVM developers, with the default parameters provided in TVM *tophub*. The default schedules and parameters can help TVM to achieve similar performance compared to other DL compilers. In addition, the performance difference between the tuned TVM and untuned TVM is negligible on CPU but quite significant on GPU (41.26× speedup on average). This is because the GPU has more complicated thread and memory hierarchy than CPU, thus to exploit the computation power, GPU requires more fine-grained scheduling (e.g., *tile*, *split*, and *reorder* in TVM). Therefore, it is crucial to determine the optimal scheduling parameters on GPU, where the autotuning exhibits its effectiveness.

5.3 Per-Layer Performance Comparison

To further compare the capability of backend optimizations of DL compilers, we evaluate the per-layer (convolution layers since they dominate the inference time) performance of the `ResNet50` and `MobileNetV2_1.0` on V100 GPU and Broadwell CPU (single-threaded since Glow lacks multi-threading support).

Methodology. To measure the execution time of individual layers, we adopt different methods considering the DL compilers, the hardware (CPU/GPU), and the CNN models. Specifically, *1)* On TVM, we re-use the logs of autotuning to extract the kernel shapes and the optimal schedule. Then we rebuild the individual convolution layers and use the *time_evaluator* for evaluation. *2)* We extract the execution time through the *tracing* files of Glow. *3)* And we measure the execution time of hand-written kernels on TC. *4)* As for nGraph, we make use of the *timeline* to measure the execution time on CPU. However, the *timeline* is not supported by its PlaidML backend (which provides GPU support through OpenCL). Besides, there are no available methods to profile the command queues within OpenCL. Therefore, we leave the profiling of the per-layer performance of nGraph on GPU for future work. *4)* As for XLA, we leverage the built-in *tf.profiler.experimental* method for CPU performance and the *DLProf* [62] toolkit from Nvidia for GPU performance.

Performance. From Figs. 5, 6, 7, and 8, we have several observations about the performance illustrated as follows.

1) nGraph achieves a better performance of the convolution layers on CPU, which benefits from the co-design of hardware (Intel CPU) and software (compiler, library, and runtime). Whereas, *TVM performs better on GPU across these compilers*. On `MobileNetV2_1.0`, the performance of TVM is not stable, especially on *conv1* layer. This is because the autotuning process is affected by other processes on the same machine, and thus it tends to derive the imprecise, even negative scheduling parameters.

2) TC allows users to define a tensor computation kernel (e.g., convolution) by the Einstein notion without specifying the shape of input/output tensors (e.g., kernel size). Then the kernel is autotuned and stored in its compilation cache

5. https://tfhub.dev/

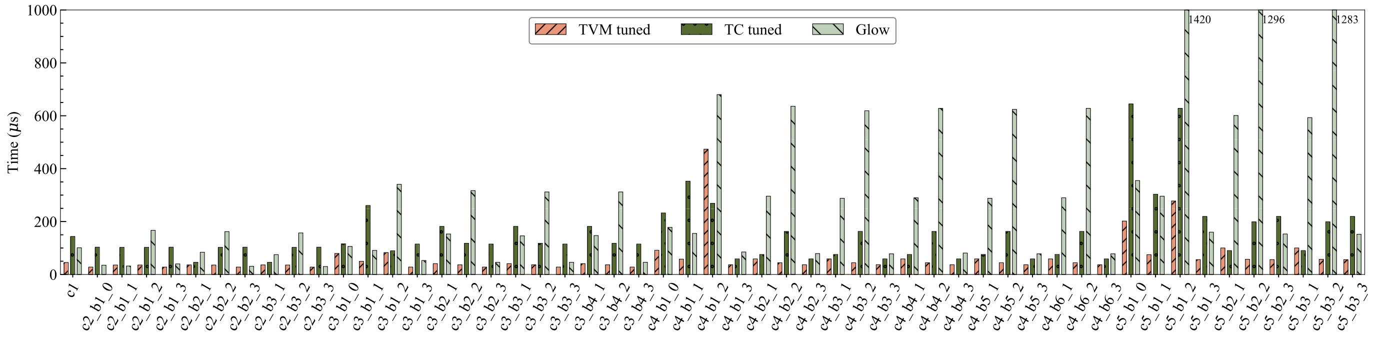


Fig. 7. The performance comparison of convolution layers in `ResNet50` across TVM, TC, and Glow on V100 GPU.

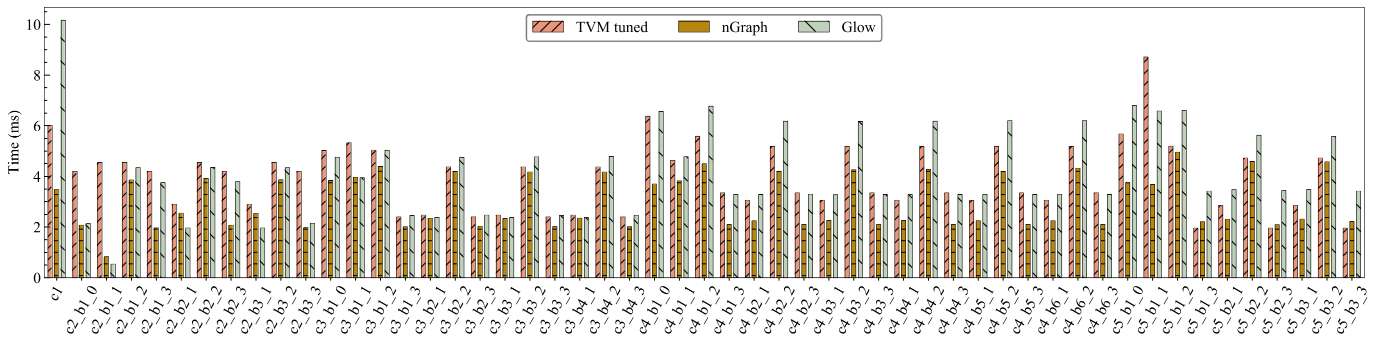


Fig. 8. The performance comparison of convolution layers in `ResNet50` across TVM, nGraph, and Glow on Broadwell CPU.

to accelerate further autotuning and compilation. *However, in our evaluation, we find the performance of TC heavily relies on the initially compiled kernels*. Take *MobileNetV2_1.0* for example, if we initialize the autotuning with layer *c1*, then *c1* can perform well. But the following `c*_b*_*` layers become much slower as the layers go deeper (far away from *c1* layer). To derive a consistent performance, we need to tune each kernel separately.

3) Glow falls behind other compilers to optimize the 1×1 *convolutions* (e.g., the b*_linear layers) of `MobileNetV2_1.0` *as well as the depth-wise separable convolutions* (e.g., c*_b*_2 layers) of `ResNet50`. It takes a longer time to compute these convolutions both on GPU and CPU. We notice the convolutions are usually fused with other layers (e.g., ReLU, BatchNorm) on Glow, which could be why the lower performance compared to other compilers. Moreover, on CPU, the convolutions at the end of `MobileNetV2_1.0` take a quite shorter time than convolutions at the beginning. According to the tracing log, we notice these convolutions are accelerated by the *CPUConvDKKC8* optimization [27], which applies tiling, layout transformation, and vectorization to convolutions with specific patterns.

4) As for XLA, it can automatically compile (*_XlaCompile*) the eligible subgraphs from Tensorflow and replace the subgraphs with the resultant binaries (*_XlaRun*). In addition, the convolution layers may be clustered with other kernels, and thus their performance is not easy to measure individually. Therefore, we have counted the clustered and the non-clustered convolutions, and the data is shown in Table 3. Note that the `MobileNetV2_1.0` model in Tensorflow is a little bit different from the ONNX model for the beginning and ending layers, however, the *linearbottleneck* layers are the same. Moreover, if a convolution is to be clustered, it could be measured at most twice till the finishing of *_XlaCompile*. Therefore, there are five extreme value in Fig. 5 (corresponding with 5 clustered convolutions in `MobileNetV2_1.0`). Actually, only the clustered kernels are optimized by XLA, while the non-clustered ones are optimized by Tensorflow. Therefore, it is impossible to measure the execution time of a standalone convolution layer optimized by XLA. Consequently, we decide not to include the performance of XLA in Figs. 6, 7, and 8.

5.4 Discussion

Through the above quantitative performance comparison across DL compilers, we can in-depth analyze the coarse-grained end-to-end performance with both frontend (graph-level) and backend (operator-level) optimizations, as well as the fine-grained per-layer performance about the convolutions with backend optimizations. However, there are still open challenges to accurately measure the effectiveness of the optimizations adopted by different DL compilers. One particular difficulty during our evaluation is that the frontend and backend optimizations are usually tightly coupled in existing DL compilers, because *1)* the frontend optimizations usually affect a series of operators. Thus the optimized operators as the inputs to the backend optimizations differ across different compilers; *2)* these

TABLE 3
The Number of the Clustered and Non-Clustered Convolutions of XLA on V100 GPU and Broadwell CPU

	MobileNetV2_1.0		ResNet50	
	Clustered	Non-clu-	Clustered	Non-clu-
V100	5	47	0	53
Broadwell	17	35	53	0

optimizations tend to be co-designed for further exploit the performance opportunities (e.g., clustering in XLA and more advanced optimizations [52], [55]). Therefore, it is difficult if not impossible to evaluate and compare specific optimizations across DL compilers individually.

To tackle this problem, we have been working on building a universal benchmarking framework for existing DL compilers to measure the per-layer performance. The fundamental idea is to extract the necessary structures and parameters of the target layers (we name them as *model fragments*), and rebuild the layers as acceptable inputs to a particular DL compiler, which allows the compiler to apply corresponding frontend and backend optimizations faithfully. We can then measure the performance of these optimized *model fragments* to understand the effectiveness of DL compilers at layers of interests. The benchmarking framework using *model fragments* is scalable to customized layers (e.g., fused layers) of interest. With such benchmarking framework available, we can derive both coarse-grained (e.g., end-to-end) and fine-grained (e.g., per-layer) performance metrics for each DL compiler, and thus compare the effectiveness of optimizations across different DL compilers at the level of interest. Currently, we have successfully experimented by extracting the target layers from the state-of-the-art CNN models, such as the *bottleneck* of `ResNet50` and the *linearbottleneck* of `MobileNetV2_1.0`. Our benchmarking framework is still under rapid development, and we hope to make it available to the community soon.

6 Conclusion and Future Directions

In this survey, we present a thorough analysis of the existing DL compilers targeting the design principles. First, we take a deep dive into the common architecture adopted in the existing DL compilers including the multi-level IR, the frontend and the backend. We present the design philosophies and reference implementations of each component in detail, with the emphasis on the unique IRs and optimizations specific to DL compilers. We provide a comprehensive taxonomy as well as the quantitative performance comparison among DL compilers. And we summarize the findings in this survey and highlight the future directions in DL compiler as follows:

Dynamic Shape and Pre/Post Processing. Dynamic model becomes more and more popular in the field of DL, whose input shape or even model itself may change during execution. Particularly, in the area of NLP, models may accept inputs of various shapes, which is challenging for DL compilers since the shape of data is unknown until runtime. Existing DL compilers require more research efforts to support dynamic shape efficiently for emerging dynamic models.

In addition, as future DL models become more complex, their entire *control flow* may inevitably include complicated pre/post-processing procedures. Currently, most DL compilers use Python as their programming language, the pre/post-processing could become a performance bottleneck when it is executed by the Python interpreter. Such potential performance bottleneck has not yet been considered by existing DL compilers. Supporting the entire *control flow* in DL compiler enables express and optimize the pre/post-processing along with DL models, which opens up new opportunities for performance acceleration in model deployment.

Advanced Auto-Tuning. Existing auto-tuning techniques focus on the optimization of individual operators. However, the combination of the local optimal does not lead to global optimal. For example, two adjacent operators that apply on different data layouts can be tuned together without introducing extra memory transformations in between. Besides, with the rise of edge computing, execution time is not only the optimization objective for DL compilers. New optimization targets should also be considered in the auto-tuning such as memory footprint and energy consumption.

Particularly, for the ML-based auto-tuning techniques, there are several directions worth further exploring. First, the ML techniques can be applied in other stages of auto-tuning, other than the cost model. For example, in the stage of selecting compiler options and optimization schedules, ML techniques can be used to predict the possibility directly and develop algorithms to determine the final configurations. Second, the ML-based auto-tuning techniques can be improved based on the domain knowledge. For example, incorporating the feature engineering (selecting features to represent program) [63] in auto-tuning techniques could be a potential direction for achieving better tuning results.

Polyhedral Model. It is a promising research direction to combine polyhedral model and auto-tuning techniques in the design of DL compilers for efficiency. On one hand, the auto-tuning can be applied to minimize the overhead of polyhedral JIT compilation by reusing the previous configurations. On the other hand, the polyhedral model can be used to perform auto-scheduling, which can reduce the search space of auto-tuning.

Another challenge of applying polyhedral model in DL compilers is to support the sparse tensor. In general, the format of a sparse tensor such as CSF [64] expresses the loop indices with index arrays (e.g., $a[b[i]]$) that is no longer linear. Such indirect index addressing leads to non-affine subscript expressions and loop bounds, which prohibits the loop optimization of the polyhedral model [65], [66]. Fortunately, the polyhedral community has made progress in supporting sparse tensor [67], [68], and integrating the latest advancement of the polyhedral model can increase the performance opportunities for DL compilers.

Subgraph Partitioning. DL compilers supporting subgraph partitioning can divide the computation graph into several subgraphs, and the subgraphs can be processed in different manners. The subgraph partitioning presents more research opportunities for DL compilers. First, it opens up the possibility to integrate graph libraries for optimization. Take nGraph and DNNL for example, DNNL is a DL library with graph optimizations leveraging vast collection of highly optimized kernels. The integration of DNNL with nGraph enables DNNL to speedup the execution of the subgraphs generated by nGraph. Second, it opens up the possibility of heterogeneous and parallel execution. Once the computation graph is partitioned into subgraphs, the execution of different subgraphs can be assigned to heterogeneous hardware targets at the same time. Take the edge device for example, its computation units may consist of ARM CPU, Mail GPU, DSP, and probably NPU. Generating subgraphs from the DL compilers that utilizes all computation units efficiently can deliver significant speedup of the DL tasks.

Quantization. Traditional quantization strategies applied in DL frameworks are based on a set of fixed schemes and datatypes with little customization for codes running on different hardware. Whereas, supporting quantization in DL compilers can leverage optimization opportunities during compilation to derive more efficient quantization strategies. For example, Relay [40] provides a quantization rewriting flow that can automatically generate quantized code for various schemes.

To support quantization, there are several challenges to be solved in DL compilers. The first challenge is how to implement new quantized operators without heavy engineering efforts. The attempt from AWS points out a possible direction that uses the concept of *dialect* to implement new operators upon basic operators, so that the optimizations at graph level and operator level can be reused. The second challenge is the interaction between quantization and other optimizations during compilation. For example, determining the appropriate stage for quantization and collaborating with optimizations such as operator fusion require future research investigations.

Unified Optimizations. Although existing DL compilers adopt similar designs in both computation graph optimizations and hardware-specific optimizations, each compiler has its own advantages in certain aspects. There is a missing way to share the state-of-the-art optimizations, as well as support of emerging hardware targets across existing compilers. We advocate unifying the optimizations from existing DL compilers so that the best practices adopted in each DL compiler can be reused. In addition, unifying the optimizations across DL compilers can accumulate a strong force to impact the design of general-purpose and dedicated DL accelerators, and provide an environment for efficient co-design of DL compiler and hardware.

Currently, Google MLIR is a promising initiative towards such direction. It provides the infrastructure of multi-level IRs, and contains IR specification and toolkit to perform transformations across IRs at each level. It also provides flexible *dialects*, so that each DL compiler can construct its customized *dialects* for both high-level and low-level IRs. Through transformation across *dialects*, optimizations of one DL compiler can be reused by another compiler. However, the transformation of *dialects* requires further research efforts to reduce the dependency on delicate design.

Differentiable Programming. Differentiable programming is a programming paradigm, where the programs are differentiable thoroughly. Algorithms written in differentiable programming paradigm can be automatically differentiated, which is attractive for DL community. Many compiler projects have adopted differentiable programming, such as Myia [69], Flux [70] and Julia [71]. Unfortunately, there is little support for differential programming in existing DL compilers.

To support differential programming is quite challenging for existing DL compilers. The difficulties come from not only data structure, but also language semantic. For example, to realize the transformation from Julia to XLA HLO IR, one of the challenges [72] is that the control flow is different between the imperative language used by Julia and the symbolic language used by XLA. In order to use HLO IR efficiently, the compiler also needs to provide operation abstraction for Julia in order to support the particular semantic of XLA, such as *MapReduce* and *broadcast*. Moreover, the semantic difference of differentiation between Julia and XLA, also requires significant changes of compiler designs.

Privacy Protection. In edge-cloud system, the DL models are usually split into two halves with each partial model running on the edge device and cloud service respectively, which can provide better response latency and consume less communication bandwidth. However, one of the drawbacks with the edge-cloud system is that the user privacy becomes vulnerable. The reason is that the attackers can intercept the intermediate results sent from the edge devices to cloud, and then use the intermediate results to train another model that can reveal the privacy information deviated from the original user task.

To protect privacy in edge-cloud system, existing approaches [73], [74], [75] propose to add noise with special statistic properties to the intermediate results that can reduce the accuracy of the attacker task without severely deteriorating the accuracy of the user task. However, the difficulty is to determine the layer where the noise should be inserted, which is quite labor intensive to identify the optimal layer. The above difficulty presents a great opportunity for DL compilers to support privacy protection, because the compilers maintain rich information of the DL model, which can guide the noise insertion across layers automatically.

Training Support. In general, the model training is far less supported in current DL compilers. As shown in Table 1, nGraph only supports training on the Intel NNP-T accelerator, TC only supports the auto differentiation of a single kernel, Glow has experimental training support for limited models, the training support of TVM is under development, while XLA relies on the training support of TensorFlow. In sum, current DL compilers mainly focus on bridging the gap of deploying DL models onto diverse hardware efficiently, and thus they choose inference as their primary optimization targets. However, expanding the capability of DL compilers to support model training would open up a large body of research opportunities such as optimization of gradient operators and high-order auto differentiation.

ACKNOWLEDGMENTS

The authors would like to thank Jun Yang from Alibaba and Yu Xing from Xilinx for their valuable comments and suggestions. They also would like to thank all anonymous reviewers for their insightful comments and suggestions. This work was supported by National Key R&D Program of China (Grant No. 2020YFB150001), National Natural Science Foundation of China (Grant No. 62072018, 61502019, and 61732002), and SenseTime Research Fund for Young Scholars.

REFERENCES

[1] C. D. Manning, C. D. Manning, and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.
[2] D. A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*. Englewood Cliffs, NJ, USA: Prentice Hall, 2002.

[3] J.-W. Ha, H. Pyo, and J. Kim, “Large-scale item categorization in e-commerce using multiple recurrent neural networks,” in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 107–115.

[4] M. Mohammadi, A. Al-Fuqaha, M. Guizani, and J.-S. Oh, “Semisupervised deep reinforcement learning in support of IoT and smart city services,” *IEEE Internet Things J.*, vol. 5, no. 2, pp. 624–635, Apr. 2018.

[5] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke, “The rise of deep learning in drug discovery,” *Drug Discov. Today*, vol. 23, no. 6, pp. 1241–1250, 2018.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[8] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[9] I. J. Goodfellow *et al.*, “Generative adversarial networks,” 2014, *arXiv:1406.2661*.

[10] M. Abadi *et al.*, “TensorFlow: A system for large-scale machine learning,” in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[11] A. Paszke *et al.*, “PyTorch: An imperative style, high-performance deep learning library,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.

[12] T. Chen *et al.*, “MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems,” 2015, *arXiv:1512.01274*.

[13] F. Seide and A. Agarwal, “CNTK: Microsoft’s open-source deep-learning toolkit,” in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 2135–2135.

[14] ONNX github repository. Accessed: Feb. 4, 2020. [Online]. Available: https://github.com/onnx/onnx

[15] N. P. Jouppi *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.

[16] H. Liao, J. Tu, J. Xia, and X. Zhou, “DaVinci: A scalable architecture for neural network computing,” in *Proc. IEEE Hot Chips 31 Symp.*, 2019, pp. 1–44.

[17] A. Kingsley-Hughes, “A11 bionic processor,” 2017.

[18] NVIDIA turing architecture. Accessed: Feb. 4, 2020. [Online]. Available: https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/

[19] Nervana neural network processor. Accessed: Feb. 4, 2020. [Online]. Available: https://www.intel.ai/nervana-nnp/

[20] AWS inferentia. Accessed: Feb. 4, 2020. [Online]. Available: https://aws.amazon.com/machine-learning/inferentia

[21] Announcing hanguang 800: Alibaba’s first AI-inference chip. Accessed: Feb. 4, 2020. [Online]. Available: https://www.alibabacloud.com/blog/announcing-hanguang-800-alibabas-first-ai-inference-chip_595482

[22] S. Liu *et al.*, “Cambricon: An instruction set architecture for neural networks,” in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 393–405.

[23] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, “Dissecting the graphcore IPU architecture via microbenchmarking,” 2019, *arXiv: 1912.03413*.

[24] TensorRT github repository. Accessed: Feb. 4, 2020. [Online]. Available: https://github.com/NVIDIA/TensorRT

[25] T. Chen *et al.*, “TVM: An automated end-to-end optimizing compiler for deep learning,” in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 578–594.

[26] N. Vasilache *et al.*, “Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions,” 2018, *arXiv: 1802.04730*.

[27] N. Rotem *et al.*, “Glow: Graph lowering compiler techniques for neural networks,” 2018, *arXiv: 1805.00907*.

[28] S. Cyphers *et al.*, “Intel nGraph: An intermediate representation, compiler, and executor for deep learning,” 2018, *arXiv: 1801.08058*.

[29] C. Leary and T. Wang, “XLA: Tensorflow, compiled,” *TensorFlow Dev Summit*, 2017.

[30] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proc. Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86.

[31] Y. Xing, J. Weng, Y. Wang, L. Sui, Y. Shan, and Y. Wang, “An in-depth comparison of compilers for deep neural networks on hardware,” in *Proc. IEEE Int. Conf. Embedded Softw. Syst.*, 2019, pp. 1–8.

[32] M. Li *et al.*, “The deep learning compiler: A comprehensive survey,” 2020, *arXiv: 2002.03794*.

[33] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 519–530.

[34] Polyhedral compilation. Accessed: Feb. 4, 2020. [Online]. Available: https://polyhedral.info

[35] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, “Dependence graphs and compiler optimizations,” in *Proc. 8th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 1981, pp. 207–218. [Online]. Available: https://doi.org/10.1145/567532.567555

[36] C. Lattner *et al.*, “MLIR: A compiler infrastructure for the end of Moore’s law,” 2020, *arXiv:2002.11054*.

[37] D. Goodman, *JavaScript Bible*. Hoboken, NJ, USA: Wiley, 2007.

[38] J. Harrop, “F# for scientists,” USA, Wiley-Interscience, 2008.

[39] H. Abelson *et al.*, “Revised 5 report on the algorithmic language scheme,” *Higher-Order Symbolic Comput.*, vol. 11, no. 1, pp. 7–105, 1998.

[40] J. Roesch *et al.*, “Relay: A high-level compiler for deep learning,” 2019, *arXiv:1904.08368*.

[41] J. McCarthy and M. I. Levin, *LISP 1.5 Programmer’s Manual*. Cambridge, MA, USA: MIT Press, 1965.

[42] Y. Yu *et al.*, “Dynamic control flow in large-scale machine learning,” in *Proc. 13th EuroSys Conf.*, 2018, Art. no. 18.

[43] B. van Merrienboer, O. Breuleux, A. Bergeron, and P. Lamblin, “Automatic differentiation in ML: Where we are and where we should be going,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 8757–8767.

[44] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[45] S. Verdoolaege, “isl: An integer set library for the polyhedral model,” in *Proc. Int. Congr. Math. Softw.*, 2010, pp. 299–302.

[46] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, “The omega library,” Univ. Maryland, Tech. Rep., 1996.

[47] P. Feautrier, “Parametric integer programming,” *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[48] V. Loechner, “PolyLib: A library for manipulating parameterized polyhedra,” 1999. [Online]. Available: https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz

[49] R. Bagnara, P. M. Hill, and E. Zaffanella, “The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems,” *Sci. Comput. Program.*, vol. 72, no. 1, pp. 3–21, 2008. [Online]. Available: https://doi.org/10.1016/j.scico.2007.08.001

[50] D. Kang, E. Kim, I. Bae, B. Egger, and S. Ha, “C-GOOD: C-code generation framework for optimized on-device deep learning,” in *Proc. Int. Conf. Comput.-Aided Des.*, 2018, Art. no. 105.

[51] G. Long, J. Yang, K. Zhu, and W. Lin, “FusionStitching: Deep fusion and code generation for tensorflow computations on GPUs,” 2018, *arXiv:1811.05213*.

[52] G. Long, J. Yang, and W. Lin, “FusionStitching: Boosting execution efficiency of memory intensive computations for DL workloads,” 2019, *arXiv:1911.11576*.

[53] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers: Principles, techniques, and tools,” Addison-Wesley, 1986. [Online]. Available: https://www.worldcat.org/oclc/12285707

[54] B. H. Ahn, J. Lee, J. M. Lin, H.-P. Cheng, J. Hou, and H. Esmaeilzadeh, “Ordering Chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices,” in *Proc. Conf. Mach. Learn. Syst.*, 2020, vol. 2, pp. 44–57. [Online]. Available: https://proceedings.mlsys.org/paper/2020/file/9bf31c7ff062936a96d3c8bd1f8f2ff3-Paper.pdf

[55] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, “Optimizing CNN model inference on CPUs,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 1025–1040.

[56] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, “Polyhedral parallel code generation for CUDA,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[57] S. Kaufman, P. M. Phothilimthana, and M. Burrows, “Learned TPU cost model for XLA tensor programs,” in *Proc. Workshop ML Syst. NeurIPS*, 2019, pp. 1–6.

[58] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Reading, MA, USA: Addison-Wesley, 1989.

[59] D. Bertsimas *et al.*, "Simulated annealing," *Statist. Sci.*, vol. 8, no. 1, pp. 10–15, 1993.

[60] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh, "Chameleon: Adaptive code optimization for expedited deep neural network compilation," in *Proc. Int. Conf. Learn. Representations*, 2020.

[61] S. Wang and P. Kanwar, "BFloat16 hardware numerics definition," 2017.

[62] DLProf user-guide. Accessed: Aug. 26, 2020. [Online]. Available: https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/

[63] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proc. IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov. 2018.

[64] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proc. 5th Workshop Irregular Appl.: Archit. Algorithms*, 2015, pp. 1–7.

[65] N. Vasilache, C. Bastoul, and A. Cohen, "Polyhedral code generation in the real world," in *Proc. Int. Conf. Compiler Construction*, 2006, pp. 185–201.

[66] C. Chen, "Polyhedra scanning revisited," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2012, pp. 499–508.

[67] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *Proc. Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2014, pp. 185–194.

[68] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2015, pp. 521–532.

[69] B. van Merriënboer, O. Breuleux, A. Bergeron, and P. Lamblin, "Automatic differentiation in ML: Where we are and where we should be going," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 8757–8767.

[70] M. Innes *et al.*, "Fashionable modelling with flux," 2018, *arXiv: 1811.01457*.

[71] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Rev.*, vol. 59, no. 1, pp. 65–98, 2017.

[72] K. Fischer and E. Saba, "Automatic full compilation of Julia programs and ML models to cloud TPUs," 2018, *arXiv: 1810.09868*.

[73] F. Mireshghallah, M. Taram, P. Ramrakhyani, A. Jalali, D. Tullsen, and H. Esmaeilzadeh, "Shredder: Learning noise distributions to protect inference privacy," in *Proc. 25th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2020, pp. 3–18.

[74] S. A. Osia, A. Taheri, A. S. Shamsabadi, K. Katevas, H. Haddadi, and H. R. Rabiee, "Deep private-feature extraction," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 1, pp. 54–66, Jan. 2020.

[75] R. Gao, M. Dun, H. Yang, Z. Luan, and D. Qian, "Privacy for rescue: A new testimony why privacy is vulnerable in deep models," 2019, *arXiv: 2001.00493*.

Mingzhen Li is currently working toward the PhD degree with the School of Computer Science and Engineering, Beihang University, Beijing, China. He is currently working on identifying performance opportunities for scientific applications and sparse matrix algorithms. His research interests include HPC, performance optimization, and code generation.

Yi Liu received the PhD degree from the Department of Computer Science, Xi'an Jiaotong University, Xi'an, China, in 2000. He is a professor with the School of Computer Science and Engineering, and director of the Sino-German Joint Software Institute (JSI), Beihang University, China. His research interests include computer architecture, HPC, and new generation of network technology.

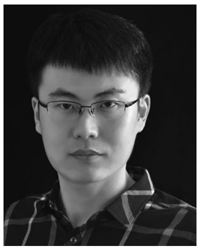
Xiaoyan Liu is currently working toward the PhD degree with the School of Computer Science and Engineering, Beihang University, Beijing, China. She is currently working on GPU hardware extension and approximation matrix algorithm. Her research interests include HPC, performance optimization, and deep learning.

Qingxiao Sun is currently working toward the PhD degree with the School of Computer Science and Engineering, Beihang University, Beijing, China. He is currently working on GPU hardware extension and performance optimization. His research interests include computer architecture, HPC, and deep learning.

Xin You is currently working toward the PhD degree with the School of Computer Science and Engineering, Beihang University, Beijing, China. He is currently working on GPU hardware extension and performance optimization. His research interests include computer architecture, HPC, and deep learning.

Hailong Yang received the PhD degree from the School of Computer Science and Engineering, Beihang University, Beijing, China, in 2014. He is an assistant professor with the School of Computer Science and Engineering, Beihang University. He has been involved in several scientific projects such as performance analysis for big data systems and performance optimization for large scale applications. His research interests include parallel and distributed computing, HPC, performance optimization, and energy efficiency. He is a member of the China Computer Federation (CCF).

Zhongzhi Luan received the PhD degree from the School of Computer Science, Xi'an Jiaotong University, Xi'an, China. He is an associate professor of computer science and engineering, and assistant director of the Sino-German Joint Software Institute (JSI) Laboratory, Beihang University, China. Since 2003, his research interests including distributed computing, parallel computing, grid computing, HPC, and the new generation of network technology.



Lin Gan (Member, IEEE) received the PhD degree in computer science from Tsinghua University, Beijing, China. He is an assistant researcher with the Department of Computer Science and Technology, Tsinghua University, and the assistant director of the National Supercomputing Center in Wuxi. His research interests include high performance computing solutions based on hybrid platforms such as GPUs, FPGAs, and Sunway CPUs. He is the recipient of the 2016 ACM Gordon Bell Prize, the 2017 ACM Gordon Bell Prize Finalist, the 2018 IEEE-CS TCHPC Early Career Researchers Award for Excellence in HPC, and the Most Significant Paper Award in 25 Years awarded by FPL 2015, etc.

Guangwen Yang (Member, IEEE) received the PhD degree in computer science from Tsinghua University, Beijing, China. He is a professor with the Department of Computer Science and Technology, Tsinghua University, and the director of the National Supercomputing Center in Wuxi. His research interests include parallel algorithms, cloud computing, and the earth system model. He has received the ACM Gordon Bell Prize in the year of 2016 and 2017, and the Most Significant Paper Award in 25 Years awarded by FPL 2015, etc.

Depei Qian received the master's degree from the University of North Texas, Denton, Texas, in 1984. He is a professor with the Department of Computer Science and Engineering, Beihang University, China. He is currently serving as the chief scientist of China National High Technology Program (863 Program) on high productivity computer and service environment. His research interests include innovative technologies in distributed computing, high performance computing, and computer architecture. He is also a fellow of the China Computer Federation (CCF).