arXiv:2002.03794v4 [cs.DC] 28 Aug 2020

The Deep Learning Compiler: A Comprehensive Survey

MINGZHEN LI*, YI LIU*, XIAOYAN LIU*, QINGXIAO SUN*, XIN YOU*, HAILONG YANG*†, ZHONGZHI LUAN*, LIN GAN§, GUANGWEN YANG§, and DEPEI QIAN*, Beihang University* and Tsinghua University§

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 paper, 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 paper focusing on the design architecture of DL compilers, which we hope can pave the road for future research towards DL compiler.

Additional Key Words and Phrases: Neural Networks, Deep Learning, Compiler, Intermediate Representation, Optimization

1 INTRODUCTION

The development of deep learning (DL) has generated profound impact on various scientific fields. It has not only demonstrated remarkable value in artificial intelligence such as natural language processing (NLP) [64] and computer vision (CV) [26], but also proved great success in broader applications such as e-commerce [36], smart city [68] and drug discovery [15]. With the emergence of versatile deep learning models such as convolutional neural network (CNN) [54], recurrent neural network (RNN) [80], long short-term memory (LSTM) [38] and generative adversarial network (GAN) [29], it is critical to ease the programming of diverse DL models in order to realize their widely adoption.

With the continuous efforts from both industry and academia, several popular DL frameworks have been proposed such as TensorFlow [1], PyTorch [75], MXNet [16] and CNTK [81], 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 [66] has been proposed, that defines a unified format for representing DL models to facilitate model conversion between different DL frameworks.

In the 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 [44], Hisilicon NPU [56], Apple Bonic [49]), processor vendors (e.g., NVIDIA Turing [72], Intel NNP [41]), service providers (e.g., Amazon Inferentia [8], Alibaba Hanguang [7]), and even startups (e.g., Cambricon [57], Graphcore [43]) 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

†Corresponding author.
Authors' address: Mingzhen Li*; Yi Liu*; Xiaoyan Liu*; Qingxiao Sun*; Xin You*; Hailong Yang*†; Zhongzhi Luan*; Lin Gan§; Guangwen Yang§; Depei Qian*, Beihang University*, Tsinghua University§, {lmzhhh,yi.liu,liuxiaoyan,sunqingxiao, youxin2015,hailong.yang,zhongzhi.luan,depeiq}@buaa.edu.cn, {lingan,ygw}@tsinghua.edu.cn.

arXiv:2002.03794v4 [cs.DC] 28 Aug 2020

深度学习编译器：综合调查

李明珍*, 刘毅*, 刘晓燕*, 孙庆晓*, 尤欣*, 杨海龙*†, 梁中志*, 甘林§, 杨光文§, 和 钱德培*, 北京航空航天大学* 和清华大学 §

将各种深度学习（DL）模型部署到不同的深度学习（DL）硬件上的难度，推动了社区中深度学习（DL）编译器的研究与开发。工业界和学术界已提出了多个深度学习（DL）编译器，例如Tensorflow XLA和TVM。类似地，深度学习（DL）编译器以不同深度学习（DL）框架中描述的深度学习（DL）模型为输入，然后为不同的深度学习（DL）硬件生成优化代码作为输出。然而，现有的调查都没有全面分析深度学习（DL）编译器的独特设计架构。在本文中，我们通过详细剖析常用设计，对现有的深度学习（DL）编译器进行综合调查，重点在于面向深度学习的多级中间表示（IR）以及前端/后端优化。我们对多级中间表示（IR）的设计进行了详细分析，并说明了常用的优化技术。最后，我们强调了几个作为深度学习（DL）编译器潜在研究方向的观点。这是第一篇关注深度学习（DL）编译器设计架构的调查论文，我们希望它能为未来深度学习（DL）编译器的研究铺平道路。

其他关键词和短语：神经网络、深度学习、编译器、中间表示、优化

1 引言

深度学习（DL）的发展对各个科学领域产生了深远影响。它不仅在人机智能（如自然语言处理（NLP） [64] 和计算机视觉（CV） [26],方面展现了显著价值，还在更广泛的应用领域如电子商务 [36], 智慧城市 [68]和药物发现 [15]中取得了巨大成功。随着卷积神经网络（CNN） [54], 循环神经网络（RNN） [80], 长短期记忆网络（LSTM） [38] 和生成对抗网络（GAN） [29], 等多种深度学习模型的涌现，为了实现这些模型的广泛采用，简化不同深度学习模型的编程至关重要。

在产业界和学术界持续努力下，已经提出了几个流行的深度学习框架，如TensorFlow [1], PyTorch [75], MXNet [16] 和CNTK [81], ，以简化各种深度学习模型的实现。尽管这些深度学习框架在设计上的权衡存在优缺点，但互操作性对于减少在现有深度学习模型上支持新兴深度学习模型时的冗余工程工作变得重要。为了提供互操作性，ONNX [66] 已经被提出，它定义了一种统一的格式来表示深度学习模型，以促进不同深度学习框架之间的模型转换。

与此同时，矩阵乘法等独特的计算特性激发了芯片架构师设计定制深度学习加速器以提高效率的热情。互联网巨头（例如，Google TPU [44], 、华为 NPU [56],、苹果 Bonic [49]）、处理器供应商（例如，NVIDIA Turing [72], 、英特尔 NNP [41]）、服务提供商（例如，亚马逊 Inferentia [8],、阿里巴巴 Hanguang [7]），甚至初创公司（例如，寒武纪 [57],、Graphcore [43]）都在投入巨大的人力物力开发深度学习芯片，以提升深度学习模型的性能。通常，深度学习硬件可以分为以下几类：1）通用硬件、2）为深度学习模型定制的专用硬件，以及3）受生物脑科学启发的神经形态硬件。例如，通用硬件（例如，CPU、GPU）增加了AVX512 向量单元和张量核心等特殊硬件组件，以加速深度学习模型。而对于 Google TPU 等专用硬件，则设计了专用集成电路（例如，矩阵乘法引擎和高带宽内存），以将性能和能效提升到极致。展望未来，深度学习硬件的设计将变得更加多样化。

†通讯作者。作者地址：李明珍*; 刘毅*; 刘晓燕*; 孙庆晓*; 优信*; 杨海龙*†; 梁中志*; 甘林§; 杨光文§; 钱德培*, 北京航空航天大学*, 清华大学§, {lmzhhh,yi.liu,liuxiaoyan,sunqingxiao, youxin2015,hailong.yang,zhongzhi.luan,depeiq}@buaa.edu.cn, {lingan,ygw}@tsinghua.edu.cn.

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 core 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 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 the DL operations. For example, TensorRT [73] 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 [43, 57]. 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 [17], Tensor Comprehension [91], Glow [79], nGraph [21] and XLA [53], 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 are 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 [51]), 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 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.

例如，通用硬件（例如，CPU、GPU）增加了 AVX512 向量单元和张量核心等特殊硬件组件，以加速深度学习模型。而对于 Google TPU 等专用硬件，则设计了专用集成电路（例如，矩阵乘法引擎和高带宽内存），以将性能和能效提升到极致。展望未来，深度学习硬件的设计将变得更加多样化。

为了适应硬件多样性，高效地将计算映射到深度学习硬件非常重要。在通用硬件上，高度优化的线性代数库（如基本线性代数子程序库，例如MKL和cuBLAS）作为深度学习模型高效计算的基础。以卷积运算为例，深度学习框架将卷积转换为矩阵乘法，然后调用BLAS库中的GEMM函数。此外，硬件供应商已发布专门针对深度学习计算优化的库（例如MKL-DNN和cuDNN），包括前向和后向卷积、池化、归一化和激活。更多高级工具也已开发出来，以进一步加速深度学习运算。例如，TensorRT [73] 支持图优化（例如层融合）和低比特量化，并包含大量高度优化的GPU内核。在专用深度学习硬件上，也提供了类似的库 [43, 57]。然而，依赖库的缺点是它们通常落后于深度学习模型的快速发展，因此无法高效利用深度学习芯片。

为解决深度学习库和工具的不足，并减轻在每种深度学习硬件上手动优化深度学习模型的负担，深度学习社区已转向使用领域特定编译器来寻求解决方案。迅速地，来自工业界和学术界提出了几种流行的深度学习编译器，如TVM [17], 张量理解 [91], Glow [79], nGraph [21] 以及XLA [53], 。深度学习编译器以深度学习框架中描述的模型定义作为输入，并在各种深度学习硬件上生成高效的代码实现作为输出。模型定义和特定代码实现之间的转换针对模型规范和硬件架构进行了高度优化。具体来说，它们结合了面向深度学习的优化，如层和操作符融合，从而实现高效的代码生成。此外，现有的深度学习编译器还利用了通用编译器（例如，LLVM [51]）成熟的工具链，这提供了更好的跨不同硬件架构的移植性。类似于传统编译器，深度学习编译器也采用分层设计，包括前端、中间表示（IR）和后端。然而，深度学习编译器的独特之处在于多级中间表示的设计和深度学习特定优化。

在本文中，我们通过将编译器设计分解为前端、多级中间表示和后端，对现有的深度学习编译器进行了全面综述，并特别强调了中间表示设计和优化方法。据我们所知，这是第一篇全面综述深度学习编译器设计的论文。具体而言，本文做出了以下贡献：

- 我们剖析了现有深度学习编译器普遍采用的设计架构，并详细分析了关键设计组件，如多级中间表示、前端优化
  (包括节点级、块级和数据流级优化)以及后端优化(包括硬件特定优化、自动调优和优化内核库)。

- 我们从多个方面对现有深度学习编译器进行了全面的分类法，这与本调查中描述的关键组件相对应。该分类法的目的是为考虑其需求的从业者提供深度学习编译器选择的指南，并为研究人员提供深度学习编译器的全面总结。

- 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 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 presents the background of DL compilers, including the DL frameworks, DL hardware, as well as hardware (FPGA) specific DL code generators. Section 3 describes the common design architecture of DL compilers. Section 4 discusses the key components of DL compilers, including multi-level IRs, frontend optimizations and backend optimizations. Section 5 presents a comprehensive taxonomy. Section 6 provides the quantitative performance comparison. Section 7 highlights the future directions for DL compiler research.

2 BACKGROUND

2.1 Deep Learning Frameworks

In this section, we provide an overview of popular DL frameworks. The discussion might not be exhaustive but is meant to provide a guideline fo DL practitioners. Figure 1 presents the landscape of DL frameworks including currently popular frameworks, historical frameworks and ONNX supported frameworks.

TensorFlow - Among all the DL frameworks, TensorFlow has the most comprehensive support for language interfaces, including C ++, Python, Java, Go, R, and Haskell. TensorFlow employs a dataflow graph of primitive operators extended with restricted control edges to represent differentiable programs [78]. TensorFlow Lite is designed for mobile and embedded deep learning and provides an Android neural network API. To reduce the complexity of using TensorFlow, Google adopts Keras as a frontend to the TensorFlow core. Furthermore, The eager-mode in TensorFlow applies an approach similar to PyTorch to support dynamic computation graphs better.

Keras - Keras [19] is a high-level neural network library for quickly building DL models, written in pure Python. Though not a DL framework on its own, Keras provides a high-level API that integrates with TensorFlow, MXNet, Theano, and CNTK. With Keras, DL developers can build a neural network with just a few lines of code. Besides, Keras can integrate with other common DL packages, such as scikit-learn. However, Keras is not flexible enough due to over-encapsulation, which makes it too difficult to add operators or obtain low-level data information.

PyTorch - Facebook has rewritten the Lua-based DL framework Torch in Python and refactored all modules on *Tensor* level, which leads to the release of PyTorch. As the most popular dynamic framework, PyTorch embeds primitives for constructing dynamic dataflow graphs in Python, where the control flow is executed in the Python interpreter. PyTorch 1.0 integrated the codebases of PyTorch 0.4 and Caffe2 to create a unified framework. This allows PyTorch to absorb the benefits of Caffe2 to support efficient graph execution and mobile deployment. FastAI [39] is an advanced API layer based on PyTorch's upper-layer encapsulation. It fully borrows Keras to ease the use of PyTorch.

Caffe/Caffe2 - Caffe [42] was designed for deep learning and image classification by UC Berkeley. Caffe has the command line, Python, and MATLAB APIs. Caffe's simplicity makes the source codes easy to extend, which is suitable for developers to analyze in-depth. Therefore, Caffe is mainly

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

- 我们提供了在CNN模型上对深度学习编译器的定量性能比较，包括完整模型和轻量级模型。我们比较了端到端和逐层（由于卷积层主导推理时间）的性能，以展示优化的有效性。评估脚本和结果已开源[1]供参考。

- 我们强调了深度学习编译器未来发展的几个见解，包括动态形状和预处理/后处理、高级自动调优、多面体模型、子图划分-
  ing、量化、统一优化、可微分编程和隐私保护，我们希望这些能推动深度学习编译器社区的研究。

本文其余部分的结构安排如下。第2节介绍了深度学习编译器的背景，包括深度学习框架、深度学习硬件以及硬件（FPGA）特定的深度学习代码生成器。第3节描述了深度学习编译器的通用设计架构。第4节讨论了深度学习编译器的主要组件，包括多级中间表示、前端优化和后端优化。第5节提出了一个全面的分类法。第6节提供了定量的性能比较。第7节强调了深度学习编译器研究的未来方向。

2 背景

2.1 深度学习框架

在本节中，我们概述了流行的深度学习框架。讨论可能并不详尽，但旨在为深度学习从业者提供指南。图1展示了深度学习框架的格局，包括当前流行的框架、历史框架和ONNX支持的框架。

TensorFlow - 在所有深度学习框架中，TensorFlow对语言接口的支持最为全面，包括C ++、Python、Java、Go、R和Haskell。TensorFlow采用由原始操作符扩展的受限控制边的数据流图来表示可微程序 [78]。TensorFlow Lite专为移动和嵌入式深度学习设计，并提供了一个Android神经网络API。为了降低使用TensorFlow的复杂性，Google将Keras作为前端集成到TensorFlow核心中。此外，TensorFlow中的即时模式采用类似PyTorch的方法来更好地支持动态计算图。

Keras - Keras [19] 是一个用纯Python编写的高级神经网络库，用于快速构建深度学习模型。虽然它本身不是一个深度学习框架，但Keras提供了一个高级API，可以与TensorFlow、MXNet、Theano和CNTK集成。通过Keras，深度学习开发者只需几行代码即可构建一个神经网络。此外，Keras可以与其他常见的深度学习包集成，例如scikit-learn。然而，由于过度封装，Keras不够灵活，这使得添加操作符或获取低级数据信息变得过于困难。

PyTorch - Facebook 将基于 Lua 的 DL 框架 Torch 改写成 Python 并在 Tensor 层重构所有模块，由此发布了 PyTorch。作为最受欢迎的动态框架，PyTorch 嵌入了用于在 Python 中构建动态数据流图的原始操作，其中控制流在 Python 解释器中执行。PyTorch 1.0 集成了 PyTorch 0.4 和 Caffe2 的代码库，创建了一个统一的框架。这使 PyTorch 能够吸收 Caffe2 的优势，以支持高效的图执行和移动部署。FastAI [39] 是基于 PyTorch 的高层封装的先进 API 层。它完全借鉴了 Keras，以简化 PyTorch 的使用。

Caffe/Caffe2- Caffe [42] 由加州大学伯克利分校为深度学习和图像分类而设计。Caffe具有命令行、Python和MATLAB API。Caffe的简洁性使得源代码易于扩展，适合开发者深入分析。因此，Caffe主要

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

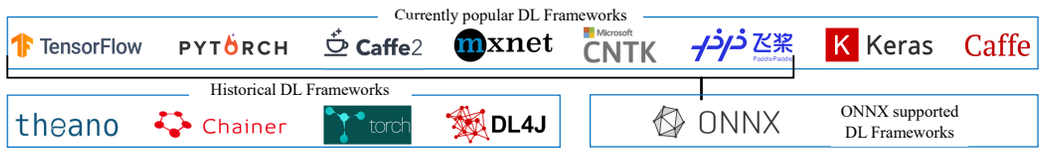

Fig. 1. DL framework landscape: *1)* Currently popular DL frameworks; *2)* Historical DL frameworks; *3)* ONNX supported frameworks.

positioned in research, which has made it popular from the beginning to the present. Caffe2 is built upon the original Caffe project. Caffe2 is similar to TensorFlow in code structure, albeit with a lighter API and easier access to the intermediate results in the computation graph.

MXNet - MXNet supports multiple language APIs including Python, C++, R, Scala, Julia, Matlab, and JavaScript. It was intended to be scalable and was designed from the perspective to reduce data loading and I/O complexity [16]. MXNet offers different paradigms: declarative programming like Caffe and Tensorflow as well as imperative like PyTorch. In December 2017, Amazon and Microsoft jointly released Gluon [69] based on MXNet, which is an advanced interface similar to Keras and FastAI. Gluon supports both flexible, dynamic graphs and efficient, static graphs.

CNTK - CNTK can be used through Python, C++ and C# APIs, or its own scripting language (i.e., BrainScript). CNTK is designed to be easy-to-use and production-ready for large-scale data in production [37]. However, CNTK does not yet support the ARM architecture, which limits its usage on mobile devices. It uses the static computation graph similar to TensorFlow and Caffe, in which a DL model is treated as a series of computational steps through a directed graph.

PaddlePaddle - The original design of PaddlePaddle [11] is similar to Caffe, where each model can be represented as a set of layers. However, PaddlePaddle v2 has adopted the concept of operators with reference to TensorFlow, which breaks layers into finer-grained operators, thereby supporting more complex DL models. And PaddlePaddle Fluid is similar to PyTorch because it provides own interpreter so as to avoid the limited performance of Python interpreter.

ONNX - The Open Neural Network Exchange (ONNX) [66] defines a scalable computation graph model, and thus computation graphs built by different DL frameworks can be easily transformed into ONNX. With ONNX, it becomes easier to convert models between DL frameworks. For example, it allows developers to build an MXNet model and then run the model using PyTorch for inference. As shown in Figure 1, ONNX has been integrated into PyTorch, MXNet, PaddlePaddle, and so on. For several DL frameworks (e.g., TensorFlow and Keras) that are not directly supported yet, and ONNX adds converters to them.

Historical Frameworks - Due to the rapid evolvement in DL community, many historical DL frameworks are no longer active. For example, PyTorch has replaced Torch [20]. As one of the oldest DL frameworks, Theano [86] is no longer under maintenance. Deeplearning4J [85] a distributed DL framework based on Java and Scala, however becomes inactive due to the lack of large developer community. Chainer [87] was once the preferred framework for dynamic computation graphs, however replaced by MXNet, PyTorch and TensorFlow with similar features.

Previous works [10, 25, 35, 70, 82, 100] have compared the performance of DL frameworks on different applications (e.g., computer vision and image classification) and different hardware (e.g., CPU, GPU, and TPU). For detailed information about each DL framework, the readers can refer to [37]. Different from them, this survey focuses on the research efforts on DL compilers which provide more general approach to execute various DL models on diverse hardware efficiently.

图1. 深度学习框架格局：1) 当前流行的深度学习框架；2) 历史深度学习框架；3) 支持ONNX的框架。

在研究领域，这使得它从一开始就流行至今。Caffe2 是基于原始 Caffe 项目构建的。Caffe2 在代码结构上与 TensorFlow 类似，尽管其 API 更轻量级，并且更容易访问计算图中的中间结果。

MXNet- MXNet支持多种语言API，包括Python、C++、R、Scala、Julia、Matlab和JavaScript。它旨在可扩展，并从减少数据加载和I/O复杂性的角度进行设计 [16]。MXNet提供不同的范式：声明式编程如Caffe和TensorFlow，以及命令式如PyTorch。2017年12月，亚马逊和微软联合发布了基于MXNet的Gluon [69]，这是一个类似于Keras和FastAI的高级接口。Gluon支持灵活的动态图和高效的静态图。

CNTK - CNTK 可以通过 Python、C++ 和 C# API 使用，或使用其自身的脚本语言（即 BrainScript）。CNTK 设计为易于使用，并可用于生产中的大规模数据 [37]。然而，CNTK 目前不支持 ARM 架构，这限制了其在移动设备上的使用。它使用与 TensorFlow 和 Caffe 类似的静态计算图，其中深度学习模型被视为通过有向图的一系列计算步骤。

PaddlePaddle - PaddlePaddle [11] 的原始设计类似于 Caffe，其中每个模型可以表示为一组层。然而，PaddlePaddle v2 采用了与 TensorFlow 参考的算子概念，将层细分为更细粒度的算子，从而支持更复杂的深度学习模型。PaddlePaddle Fluid 类似于 PyTorch，因为它提供自己的解释器，以避免 Python 解释器的性能限制。

ONNX - 开放神经网络交换（ONNX）[66] 定义了一种可扩展的计算图模型，因此由不同深度学习框架构建的计算图可以轻松转换为 ONNX。通过 ONNX，在深度学习框架之间转换模型变得更加容易。例如，它允许开发者构建一个 MXNet 模型，然后使用 PyTorch 进行推理。如图 1 所示，ONNX 已集成到 PyTorch、MXNet、PaddlePaddle 等框架中。对于尚未直接支持的几个深度学习框架（例如 TensorFlow 和 Keras），ONNX 为它们添加了转换器。

历史框架 - 由于深度学习领域的快速发展，许多历史深度学习框架已不再活跃。例如，PyTorch 已取代 Torch [20]。作为最古老的深度学习框架之一，Theano [86] 已停止维护。Deeplearning4J [85] 是一个基于 Java 和 Scala 的分布式深度学习框架，但由于缺乏大型开发者社区而变得不活跃。Chainer [87] 曾是动态计算图的首选框架，但后来被具有类似功能的 MXNet、PyTorch 和 TensorFlow 取代。

以往工作 [10, 25, 35, 70, 82, 100] 已比较了深度学习框架在不同应用（例如计算机视觉和图像分类）以及不同硬件（例如 CPU、GPU 和 TPU）上的性能。关于每个深度学习框架的详细信息，读者可以参考 [37]。与它们不同，本调查重点关注深度学习编译器的研究工作，后者提供更通用的方法，能够高效地在各种硬件上执行各种深度学习模型。

2.2 Deep Learning Hardware

The DL hardware can be divided into three categories based on the generality: *1)* general-purpose hardware that can support DL workloads through hardware and software optimization; *2)* dedicated hardware that focus on accelerating DL workloads with fully customized circuit design; *3)* neuromorphic hardware that function by mimicking the human brain.

General-purpose Hardware - The most representative general-purpose hardware for DL models is Graphic Processing Unit (GPU), which achieves high parallelism with many-core architecture. For example, Nvidia GPUs have introduced tensor cores since the Volta architecture. Tensor cores can accelerate mixed-precision matrix multiply-and-accumulate calculations in parallel, which are widely used in DL models during both training and inference. Co-optimized with the hardware, NVIDIA also launches highly optimized DL libraries and tools such as cuDNN [18] and TensorRT [73] to further accelerate the computation of DL models.

Dedicated Hardware - Dedicated hardware is fully customized for DL computation to improve performance and energy efficiency to extreme. The rapid expansion of DL applications and algorithms has spurred many startups developing dedicated DL hardware (e.g., Graphcore GC2, Cambricon MLU270). Besides, traditional hardware companies (e.g., Intel NNP, Qualcomm Cloud AI 100) and cloud service providers (e.g., Google TPU, Amazon Inferentia, and Alibaba Hanguang) have also invested in this field. The most well known dedicated DL hardware is Google's TPU series. A TPU includes Matrix Multiplier Unit (MXU), Unified Buffer (UB), and Activation Unit (AU), which is driven with CISC instructions by the host processor. The MXU is mainly composed of a systolic array, which is optimized for power and area efficiency in performing matrix multiplications. Compared to CPU and GPU, TPU is still programmable but uses a matrix as a primitive instead of a vector or scalar. The Amazon Inferentia has also attracts the attention recently. This chip has four NeuroCores that are designed for tensor-level operations, and it has large on-chip cache to avoid the frequent main memory access.

Neuromorphic Hardware - Neuromorphic chips use electronic technology to simulate the biological brain. Representative products of the this kind are IBM's TrueNorth and Intel's Loihi. Neuromorphic chips (e.g., TrueNorth) have very high connectivity between their artificial neurons. Neuromorphic chips also replicate a structure similar to the brain tissue: neurons can simultaneously store and process the data. Traditional chips distribute processors and memory in different locations, but neuromorphic chips usually have many microprocessors, each of which has a small amount of local memory. Compared to TrueNorth, Loihi has a learning ability more similar to the brain. Loihi introduces the pulse-time-dependent synaptic plasticity model (STDP), a mechanism that regulates synaptic strength by the relative time of pre-synaptic and post-synaptic pulses. However, neuromorphic chips are far away from Large-scale commercial production. Despite that, in computer science domain, neuromorphic chips can help to capture the process of rapid, life-long learning which is ignored by regular DL models, and in neurology domain, they are helpful to figure out how the various parts of the brain work together to create thoughts, feelings, and even consciousness.

2.3 Hardware-specific DL Code Generator

Field Programmable Gate Arrays (FPGAs) are reprogrammable integrated circuits that contain an array of programmable logic blocks. Programmers can configure them after manufacturing. Besides the reprogrammable nature, the low-power and high-performance nature of the FPGA make it widely used in so many domains, such as communication, medical, image processing, and ASIC prototyping. As for the domain of deep learning, the high-performance CPUs and GPUs are highly-reprogrammable but power-hungry, while the power-efficient ASICs are specialized for

2.2 深度学习硬件

深度学习硬件可以根据通用性分为三类：1）通用硬件，通过软硬件优化支持深度学习工作负载；2）专用硬件，通过完全定制的电路设计专注于加速深度学习工作负载；3）神经形态硬件，通过模拟人脑来运行。

通用硬件 - DL模型最具有代表性的通用硬件是图形处理单元（GPU），它通过多核架构实现高并行性。例如，Nvidia GPU从Volta架构开始引入了张量核心。张量核心可以并行加速混合精度矩阵乘累加计算，这些计算在DL模型训练和推理过程中被广泛使用。与硬件协同优化，NVIDIA还推出了高度优化的深度学习库和工具，如cuDNN [18] 和TensorRT [73]，以进一步加速DL模型的计算。

专用硬件 - 专用硬件为DL计算完全定制，以极致提升性能和能效。DL应用和算法的快速扩展推动了众多初创公司开发专用DL硬件（例如，Graphcore GC2、Cambricon MLU270）。此外，传统硬件公司（例如，Intel NNP、Qualcomm Cloud AI 100）和云服务提供商（例如，Google TPU、Amazon Inferentia和阿里巴巴汉云）也投资了这一领域。最著名的专用DL硬件是Google的TPU系列。一个TPU包括矩阵乘法单元（MXU）、统一缓存（UB）和激活单元（AU），由主机处理器使用CISC指令驱动。MXU主要由收缩阵列组成，该阵列针对矩阵乘法中的功耗和面积效率进行了优化。与CPU和GPU相比，TPU仍然可编程，但使用矩阵作为基本单元，而不是向量或标量。亚马逊的Inferentia芯片最近也引起了关注。这款芯片拥有四个为张量级操作设计的NeuroCore，并且具有大容量片上缓存，以避免频繁的主存访问。

神经形态硬件 - 神经形态芯片利用电子技术模拟生物大脑。这类代表性产品有IBM的TrueNorth和英特尔的Loihi。神经形态芯片（例如TrueNorth）的人工神经元之间具有非常高的连接性。神经形态芯片还复制了类似脑组织的结构：神经元可以同时存储和处理数据。传统芯片将处理器和内存分布在不同的位置，但神经形态芯片通常包含许多微处理器，每个微处理器都配有少量本地内存。与TrueNorth相比，Loihi的学习能力更接近大脑。Loihi引入了脉冲时间依赖性突触可塑性模型（STDP），这是一种通过突触前和突触后脉冲的相对时间来调节突触强度的机制。然而，神经形态芯片距离大规模商业化生产还很遥远。尽管如此，在计算机科学领域，神经形态芯片有助于捕捉常规深度学习模型所忽略的快速、终身学习过程；在神经病学领域，它们有助于揭示大脑不同部分如何协同工作以产生思想、情感，甚至意识。

2.3 硬件专用DL代码生成器

现场可编程门阵列（FPGA）是可重新编程的集成电路，其中包含一个可编程逻辑块阵列。程序员可以在制造后对其进行配置。除了可重新编程的特性外，FPGA的低功耗和高性能特性使其在通信、医疗、图像处理和ASIC原型设计等众多领域得到广泛应用。在深度学习领域，高性能的CPU和GPU虽然可重新编程性高但功耗大，而功耗高效的ASIC则专门用于固定应用。然而，FPGA可以弥合CPU/GPU和ASIC之间的差距，这使得FPGA成为深度学习的有吸引力平台。

fixed applications. However, the FPGA can bridge the gap between CPUs/GPUs and ASICs, which causes the FPGA to be an attractive platform for deep learning.

The High-Level Synthesis (HLS) programming model enables the FPGA programmers to generate effective hardware designs conveniently using high-level languages such as C and C++. It avoids writing lots of Verilog or VHDL descriptions, which lowers the programming threshold and reduces the long design circle. Xilinx Vivado HLS and Intel FPGA SDK for OpenCL are two of the popular HLS tools targeting their own FPGAs. However, mapping DL models to FPGAs remains a complicated work even with HLS, because that 1) DL models are usually described by the languages of DL frameworks rather than bare mental C/C++ code, and 2) DL-specific information and optimizations are hard to be leveraged.

The hardware-specific code generator targeting FPGA take the DL models or their domain-specific languages (DSLs) as the input, conduct the domain-specific (about FPGA and DL) optimizations and mappings, then generate the HLS or Verilog/VHDL and finally generate the bitstream. They can be classified into two categories according to the generated architectures of FPGA-based accelerators: the processor architecture and the streaming architecture [93].

The processor architecture has similarities with general-purpose processors. An FPGA accelerator of this architecture usually comprises several Processing Units (PUs), which are comprised of on-chip buffers and multiple smaller Processing Engines (PEs). It usually has a virtual instruction set (ISA), and the control of hardware and the scheduling of the execution should be determined by software. What's more, the static scheduling method avoids the overheads of von Neumann execution (including instruction fetching and decoding). A hardware template is a generic and fundamental implementation with configurable parameters. The DL code generator targeting this architecture adopt the hardware templates to generate the accelerator designs automatically. With the configurable parameters of templates, the code generator achieve the scalability and flexibility [104]. The scalability means that the code generator can generate designs for FPGAs ranging from high-performance to power-efficient, and the flexibility means that the code generator can generate designs for various DL models with different layer types and parameters. The number of PUs and the number of PEs per PU are template parameters of importance. Besides, the tilling size and batch size are also essential scheduling parameters about mapping the DL models to PUs and PEs. All these parameters are usually determined by the design space exploration using various strategies, such as combining the performance model and auto-tuning. DNN Weaver [83], Angel-Eye [33], ALAMO [63], FP-DNN [32], SysArrayAccel [101] are typical FPGA DL code generator targeting the processor architecture. What's more, the PUs and PEs are usually responsible for coarse-grained basic operations such as matrix-vector multiplication, matrix-matrix multiplication, pooling, and some element-wise operations. The optimizations of these basic operations are mainly guided by the tradeoff between the parallelism and data reuse, which is similar to general optimizations.

The streaming architecture has similarities with pipelines. An FPGA accelerator of this architecture consists of multiple different hardware blocks, and it nearly has one hardware block for each layer of an input DL model. With the input data of a DL model, this kind of accelerators process the data through the different hardware blocks in the same sequence with layers. Additionally, with the streaming input data, all hardware blocks can be fully utilized in a pipeline manner. However, the streaming architecture usually follows an initial assumption that the on-chip memory the computation resources on target FPGA are sufficient to accommodate the DL models, which bring barriers to deploy deep models with complicated layers. The DL code generator targeting this architecture can solve this problem by leveraging the reconfigurability of FPGA or adopting dynamic control flow. And the further optimization of a single block resembles that of basic operations of the processor

固定应用。然而，FPGA可以弥合CPU/GPUs和ASICs之间的差距，这导致FPGA成为深度学习的有吸引力平台。

高级综合（HLS）编程模型使 FPGA 程序员能够方便地使用 C 和 C++ 等高级语言生成有效的硬件设计。它避免了编写大量的 Verilog 或 VHDL 描述，从而降低了编程门槛并缩短了漫长的设计周期。Xilinx Vivado HLS 和 Intel FPGA SDK for OpenCL 是两种针对各自 FPGA 的流行 HLS 工具。然而，即使使用 HLS，将深度学习模型映射到 FPGA 仍然是一项复杂的工作，因为 1) 深度学习模型通常由深度学习框架的语言描述，而不是裸机 C/C++ 代码，以及 2) 深度学习特定的信息和优化难以利用。

针对 FPGA 的硬件特定代码生成器以深度学习模型或其领域特定语言（DSL）作为输入，执行领域特定（关于 FPGA 和深度学习）的优化和映射，然后生成 HLS 或 Verilog/VHDL，并最终生成比特流。它们可以根据 FPGA 加速器的生成架构分为两类：处理器架构和流式架构 [93]。

处理器架构与通用处理器有相似之处。这种架构的 FPGA 加速器通常包含多个处理单元（PU），这些 PU 由片上缓冲区和多个较小的处理引擎（PE）组成。它通常具有虚拟指令集（ISA），硬件控制和执行调度应由软件决定。此外，静态调度方法避免了冯·诺依曼执行的开销（包括指令获取和解码）。硬件模板是一种具有可配置参数的通用和基础实现。面向该架构的 DL 代码生成器采用硬件模板自动生成加速器设计。通过模板的可配置参数，代码生成器实现了可扩展性和灵活性 [104]。可扩展性意味着代码生成器可以为从高性能到节能型的高性能 FPGA 生成设计，灵活性意味着代码生成器可以为具有不同层类型和参数的各种 DL 模型生成设计。PU 的数量和每个 PU 的 PE 数量是重要的模板参数。此外，tiling 大小和批大小也是关于将 DL 模型映射到 PU 和 PE 的关键调度参数。所有这些参数通常通过结合性能模型和自动调优等策略进行设计空间探索来确定。DNN Weaver [83], Angel- Eye [33], ALAMO [63], FP-DNN [32], SysArrayAccel [101] 是针对处理器架构的典型 FPGA DL 代码生成器。此外，PU 和 PE 通常负责矩阵-向量乘法、矩阵-矩阵乘法、池化和一些逐元素操作等粗粒度基本操作。这些基本操作优化的主要指导原则是并行性和数据重用的权衡，这与通用优化类似。

流式架构与管道有相似之处。这种架构的FPGA加速器由多个不同的硬件模块组成，几乎每个输入深度学习模型的层都对应一个硬件模块。使用深度学习模型的输入数据时，这类加速器会按照层相同的顺序通过不同的硬件模块处理数据。此外，通过流式输入数据，所有硬件模块可以以管道方式被充分利用。然而，流式架构通常基于一个初始假设，即目标FPGA上的片上内存和计算资源足以容纳深度学习模型，这给部署具有复杂层的深度模型带来了障碍。针对这种架构的深度学习代码生成器可以通过利用FPGA的可重构性或采用动态控制流来解决这个问题。而单个模块的进一步优化则类似于处理器的基本操作

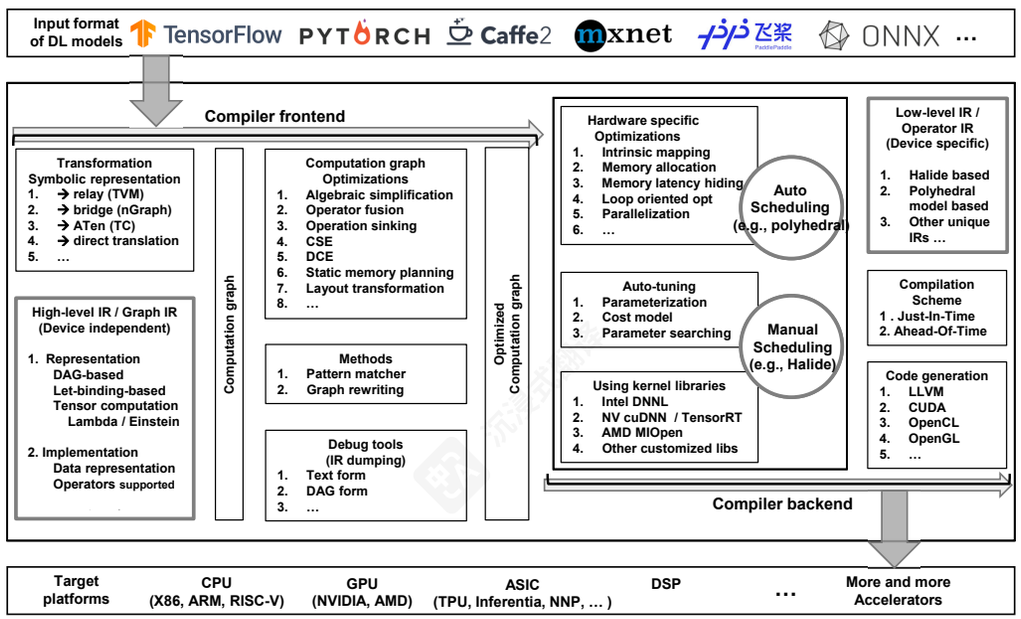

Fig. 2. The overview of commonly adopted design architecture of DL compilers.

architecture. fpgaConvNet [92], DeepBurning [98], Haddoc2 [2], and AutoCodeGen [59] are typical corresponding DL code generator.

For the detailed survey of specific compilation techniques that map DL models to FPGAs, the readers can refer to [34, 93, 104]. Different from [34, 93, 104], this survey focuses on general DL compilation techniques that can be applied to broader DL hardware other than bounding to FPGA.

3 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 Figure 2. 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 [55, 102].

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 4.1.

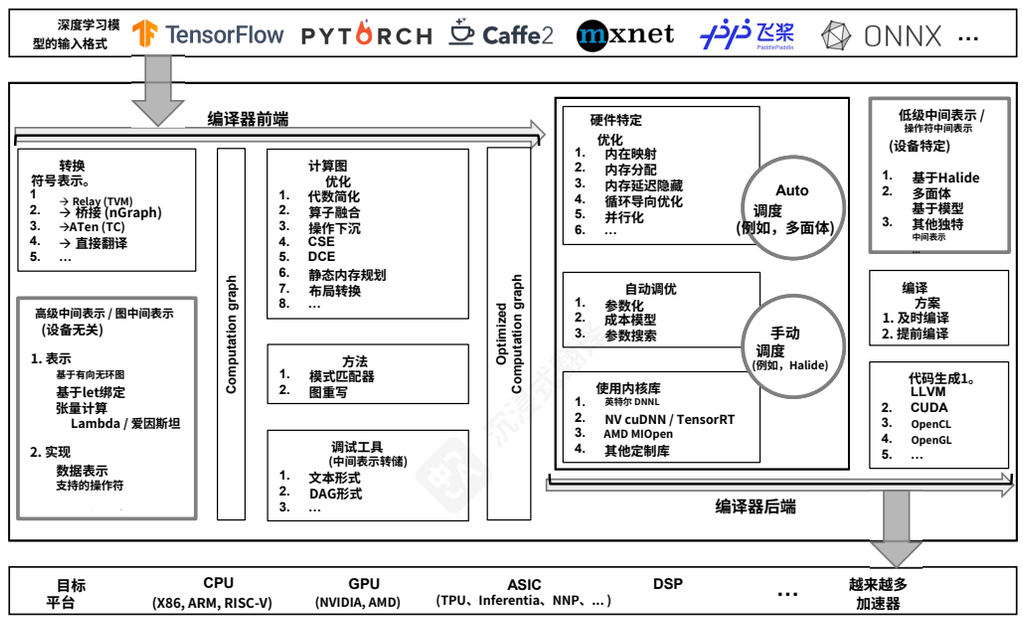

图2。DL编译器常用设计架构概述。

architecture.fpgaConvNet [92], DeepBurning [98], Haddoc2 [2], 和 AutoCodeGen [59] 是典型的相应深度学习代码生成器

要了解将深度学习模型映射到FPGA的具体编译技术，读者可以参考 [34, 93, 104]。与 [34, 93, 104], 不同，本调查重点关注可应用于更广泛深度学习硬件的一般深度学习编译技术，而非局限于FPGA。

3 DLCOMPILERS的常见设计架构

深度学习编译器的常见设计架构主要包含两部分：编译器前端和编译器后端，如图2所示。中间表示（IR）贯穿于前端和后端。通常，IR是程序的抽象表示，用于程序优化。具体而言，深度学习模型在深度学习编译器中被翻译成多级中间表示，其中高级IR位于前端，低级IR位于后端。基于高级IR，编译器前端负责与硬件无关的转换和优化。基于低级IR，编译器后端负责硬件特定优化、代码生成和编译。请注意，本调查重点关注深度学习编译器的设计原则。关于深度学习编译器的功能和实验比较，读者可以参考 [55, 102]。

高级中间表示（也称为图中间表示）表示计算和控制流，并且与硬件无关。高级中间表示的设计挑战在于其计算和控制流的抽象能力，能够捕获和表达多样的深度学习模型。高级中间表示的目标是建立控制流以及操作符与数据之间的依赖关系，并提供一个用于图级优化的接口。它还包含丰富的语义信息以供编译使用，并提供了对定制操作符的扩展性。高级中间表示的详细讨论在4.1节中呈现。

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 [77], polyhedral model [31], and LLVM [51]. The detailed discussion of low-level IR is presented in Section 4.2.

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 4.3.

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 4.4.

4 KEY COMPONENTS OF DL COMPILERS

4.1 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.

4.1.1 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 [17, 21, 53, 79, 91], 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 [50] (DDG) of generic compilers [51, 52]. 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 4.3. DAG-based

低级中间表示是为硬件特定优化和在多种硬件目标上进行代码生成而设计的。因此，低级中间表示应该足够细粒度以反映硬件特性并表示硬件特定优化。它还应该允许在编译器后端（如Halide [77], 多面体模型 [31],和LLVM [51]）中使用成熟的第三方工具链。低级中间表示的详细讨论在4.2节中呈现。

前端从现有的深度学习框架中获取一个深度学习模型作为输入，然后将模型转换为计算图表示（例如，图中间表示）。前端需要实现各种格式转换以支持不同框架中的多种格式。计算图优化结合了通用编译器和深度学习特定优化的优化技术，这些技术减少了冗余并提高了图中间表示的效率。此类优化可分为节点级（例如，无操作符消除和零维张量消除）、块级（例如，代数简化、算子融合和算子下沉）以及数据流级（例如，常量传播、死代码消除和静态内存规划）。前端生成优化后的计算图并将其传递给后端。前端的具体讨论在4.3节中呈现。

后端将高级中间表示转换为低级中间表示并执行硬件特定优化。一方面，它可以直接将高级中间表示转换为LLVM中间表示等第三方工具链，以利用现有的一般优化和代码生成基础设施。另一方面，它可以利用深度学习模型和硬件特性的先验知识，通过自定义编译 passes 进行更高效的代码生成。常用的硬件特定优化包括硬件内建映射、内存分配和获取、内存延迟隐藏、并行化以及循环导向优化。为了在大优化空间中确定最佳参数设置，现有深度学习编译器中广泛采用两种方法：自动调度（例如，多面体模型）和自动调优（例如，AutoTVM）。优化的低级中间表示使用JIT或AOT进行编译，以生成不同硬件目标的代码。后端的具体讨论在4.4节中呈现。

深度学习编译器的4个关键组成部分

4.1 高级中间表示

为了克服传统编译器所采用的中间表示在表达深度学习模型中使用的复杂计算方面的局限性，现有的深度学习编译器利用了具有特殊设计的高级中间表示（也称为图中间表示）来进行高效的代码优化。为了更好地理解深度学习编译器中使用的图中间表示，我们描述了图中间表示的表示和实现方式如下。

4.1.1 图中间表示的表示。图中间表示的表示方式影响着图中间表示的表达能力，同时也决定了深度学习编译器分析图中间表示的方式。

基于有向无环图的中间表示 - 基于有向无环图的中间表示是编译器构建计算图的最传统方法之一，其节点和边组织成有向无环图（DAG）。在深度学习编译器 [17, 21, 53, 79, 91]，中，DAG 的节点表示原子深度学习操作符（卷积、池化等），边表示张量。并且该图是无环的，没有环路，这与通用编译器 [51, 52] 的数据依赖图 [50]（DDG）不同。借助 DAG 计算图，深度学习编译器可以分析各种操作符之间的关系和依赖，并利用它们来指导优化。DDG 上已经有很多优化，例如公共子表达式消除（CSE）和死码消除（DCE）。通过结合深度学习的领域知识与这些算法，可以进一步对 DAG 计算图应用优化，这将在第 4.3 节中详细阐述。基于有向无环图的

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 [30], F# [76], and Scheme [3]. 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 [78] 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.

4.1.2 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.

中间表示因其简单性便于编程和编译，但它存在诸如计算范围定义缺失导致的语义模糊等缺陷。

基于let绑定的中间表示 - let绑定是解决语义歧义的一种方法，它通过为某些具有受限作用域的函数提供let表达式，被许多高级编程语言（如Javascript [30], F# [76], 和Scheme [3]）使用。当使用let关键字定义表达式时，会生成一个let节点，然后它指向表达式中的操作符和变量，而不是像DAG那样仅构建变量之间的计算关系。在基于有向无环图的编译器中，当一个过程需要获取某个表达式的返回值时，它首先访问相应的节点并搜索相关节点，也称为递归下降技术。相比之下，基于let绑定的编译器会计算出let表达式中的所有变量结果，并构建一个变量映射。当需要特定结果时，编译器会查询这个映射以确定表达式的结果。在深度学习编译器中，TVM的Relay IR [78]同时采用了基于有向无环图的中间表示和基于let绑定的中间表示，以获得两者的优势。

表示张量计算 - 不同的图中间表示（IR）有不同的方式来表示张量上的计算。各种深度学习框架（DL frameworks）的操作符会根据这些特定的表示翻译成图中间表示。自定义操作符也需要用这种表示来编程。张量计算表示可以分为以下三类。

1) 基于函数的：基于函数的表示只提供封装好的操作符，Glow、nGraph 和 XLA 都采用了这种方式。以高级优化器（HLO，XLA 的中间表示）为例，它由符号编程中的一组函数组成，其中大部分没有副作用。指令组织成三个层级，包括 HloModule（整个程序）、HloComputation（一个函数）和 HloInstruction（一个操作）。XLA 使用 HLO 中间表示来表示图中间表示和操作中间表示，因此 HLO 的操作范围从数据流级别到操作符级别。

2) Lambda 表达式：Lambda 表达式是一种索引公式表达式，通过变量绑定和替换来描述计算。使用 Lambda 表达式，程序员可以快速定义计算，而无需实现新函数。TVM 使用张量表达式来表示张量计算，张量表达式基于 Lambda 表达式。在 TVM 中，张量表达式中的计算操作符由输出张量的形状和计算规则的 Lambda 表达式定义。

3) 爱因斯坦记号：爱因斯坦记号，也称为求和约定，是一种用于表示求和的记号。它在编程上的简洁性优于lambda表达式。以TC为例，临时变量的索引无需定义。IR可以根据未定义变量的出现情况，基于爱因斯坦记号推导出实际表达式。在爱因斯坦记号中，操作符需要满足结合律和交换律。这一限制保证了归约操作符可以按任意顺序执行，从而实现进一步的并行化。

4.1.2 图中间表示的实现。深度学习编译器中图中间表示的实现满足了数据与操作符的管理。

数据表示 - 深度学习编译器中的数据（例如输入、权重和中间数据）通常以张量的形式组织，张量也称为多维数组。深度学习编译器可以通过内存指针直接表示张量数据，或者通过占位符以更灵活的方式表示。占位符包含张量每个维度的尺寸。或者，张量的维度尺寸可以标记为未知。为了优化，深度学习编译器需要数据布局信息。此外，迭代器的边界应根据占位符推断。

1) Placeholder: Placeholder is widely used in symbolic programming (e.g., Lisp [65], Tensorflow [1]). 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., +, ×, 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 [103], which requires control flow. Without supporting control flow in graph

1) 占位符：占位符在符号编程中广泛使用（例如，Lisp [65],Tensor-Flow [1]）。占位符本质上是一个具有显式形状信息的变量（例如，每个维度的大小），它将在计算后期被填充值。它允许程序员描述操作并构建计算图，而无需关心确切的数据元素，这有助于将计算定义与深度学习编译器中的确切执行分离。此外，通过使用占位符，程序员可以方便地更改输入/输出的形状以及其他相应的中间数据，而无需更改计算定义。

2) 未知（动态）形状表示：在声明占位符时通常支持未知维度大小。例如，TVM 使用 Any 表示未知维度（例如，Tensor (Any, 3), f p32）；XLA 使用 None 实现相同的目的（例如，t f .placeholder (“float” , [None, 3]) ）；nGraph 使用其 PartialShape 类。未知形状表示对于支持动态模型是必要的。然而，为了完全支持动态模型，应放宽边界推理和维度检查。此外，应实现额外的机制来保证内存有效性。

3) 数据布局：数据布局描述了张量在内存中的组织方式，通常是从逻辑索引到内存索引的映射。数据布局通常包括维度的顺序（例如，NCHW 和 NHWC）、分块、填充、步长等。TVM 和 Glow 将数据布局表示为操作符参数，并在计算和优化时需要这些信息。然而，将数据布局信息与操作符结合而不是张量，可以实现对某些操作符的直观实现，并减少编译开销。XLA 将数据布局表示为其后端硬件相关的约束。Relay 和 MLIR 将要将其类型系统中的数据布局信息添加到张量中。

4) 边界推理：边界推理应用于编译深度学习模型时确定迭代器的边界。尽管深度学习编译器中的张量表示方便描述输入和输出，但它为推理迭代器边界带来了特殊挑战。边界推理通常根据计算图和已知的占位符递归或迭代地执行。例如，在 TVM 中，迭代器形成一个有向无环超图，其中图的每个节点表示一个迭代器，每个超边表示两个或多个迭代器之间的关系（例如，拆分、融合或重基）。一旦根据占位符的形状确定了根迭代器的边界，就可以根据关系递归地推断其他迭代器。

支持的算子 - 深度学习编译器支持的算子负责表示深度学习工作负载，它们是计算图中的节点。算子通常包括代数算子（例如， +、 ×、exp 和 topK）、神经网络算子（例如，卷积和池化）、张量算子（例如，reshape、resize 和 copy）、广播和归约算子（例如，min 和 argmin），以及控制流算子（例如，条件判断和循环）。这里，我们选择了三种在不同深度学习编译器中经常使用的代表性算子进行说明。此外，我们还讨论了自定义算子的案例。

1) 广播：广播算子可以复制数据并生成形状兼容的新数据。没有广播算子，输入张量的形状约束更为严格。例如，对于加法算子，输入张量需要具有相同的形状。一些编译器如 XLA 和 Relay 通过提供广播算子来放宽这种限制。例如，XLA 允许通过复制矩阵和向量直到其形状匹配，从而在矩阵和向量上进行逐元素加法。

2) 控制流：在表示复杂和灵活的模型时需要控制流。RNN 和强化学习（RL）等模型依赖于循环关系和数据相关的条件执行 [103], ，这需要控制流。如果没有在图中支持控制流

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 [78]. 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 [88], 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.

4.1.3 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.

4.2 Low-level IR

4.2.1 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 firstly 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

深度学习编译器的中间表示，这些模型必须依赖宿主语言（例如Python中的if和while）的控制流支持或静态展开，这会降低计算效率。Relay注意到任意控制流可以通过递归和模式实现，这一点已经由函数式编程 [78]证明。因此，它提供了if操作符和递归函数来实现控制流。相反，XLA通过特殊的HLO操作符（如while和conditional）来表示控制流。

3) 求导：操作符Op的求导操作符（operatorOp）以Op的输出梯度和输入数据为输入，然后计算Op的梯度。尽管一些深度学习编译器（例如TVM和TC）支持自动求导 [88]，但在应用链式法则时，它们需要高级中间表示中所有操作符的导数。TVM正在努力提供代数操作符和神经网络操作符的求导操作符。程序员可以使用这些求导操作符来构建自定义操作符的导数。相反，PlaidML可以自动生成求导操作符，即使对于自定义操作符也是如此。值得注意的是，无法支持求导操作符的深度学习编译器无法提供模型训练的能力。

4) 自定义操作符：它允许程序员为其特定目的定义自己的操作符。提供对自定义操作符的支持提高了深度学习编译器的可扩展性。例如，在Glow中定义新操作符时，程序员需要实现逻辑和节点封装。此外，如果需要，还需要额外的工作，例如降低步骤、操作IR生成和指令生成。而TVM和TC除了描述计算实现外，编程工作量较少。具体来说，TVM的用户只需要描述计算、调度并声明输入/输出张量的形状。此外，自定义操作符通过钩子将Python函数集成，这进一步减轻了程序员的负担。

4.1.3 讨论。几乎所有深度学习编译器都有自己独特的高级中间表示。然而，它们的设计理念相似，例如使用DAG和let-binding来构建计算图。此外，它们通常为程序员提供方便的方式来表示张量计算。高级中间表示中设计的数据和操作符足够灵活和可扩展，以支持多种深度学习模型。更重要的是，高级中间表示与硬件无关，因此可以应用于不同的硬件后端。

4.2 低级中间表示

4.2.1 低级中间表示的实现。低级中间表示以比高级中间表示更细粒度的形式描述了DL模型中的计算，这通过提供用于调整计算和内存访问的接口来实现针对目标代码的优化。在本节中，

我们将低级中间表示的常见实现分为三类：基于Halide的中间表示、基于多面体的中间表示和其他独特的中间表示。

基于Halide的中间表示 - Halide最初被提出用于并行化图像处理，并在深度学习编译器（例如TVM）中被证明是可扩展且高效的。Halide的基本理念是计算与调度的分离。采用Halide的编译器不会直接给出特定方案，而是尝试各种可能的调度并选择最佳方案。Halide中内存引用和循环嵌套的边界被限制为与轴对齐的有界框。因此，Halide无法表达具有复杂模式（例如非矩形）的计算。幸运的是，深度学习中的计算模式非常规整，可以完美地用Halide表达。此外，Halide可以轻松地参数化这些边界并将其暴露给调优机制。当Halide应用于深度学习编译器的后端时，其原始中间表示需要进行修改。例如，Halide的输入形状是无限的，而深度学习编译器需要知道数据的精确形状以便将操作符映射到硬件指令。一些编译器（如TC）要求数据具有固定大小，以确保张量数据具有更好的时间局部性。

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) [22]).

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 [96], Omega [48], PIP [23], Polylib [60], and PPL [9].

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 [79]. 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

一些编译器（如TC）要求数据具有固定大小，以确保张量数据具有更好的时间局部性。

TVM 通过持续努力改进了 Halide IR，使其成为独立的符号 IR。它消除了对 LLVM 的依赖，并重构了项目模块和 Halide IR 的结构设计，以实现更好的组织以及图 IR 和前端语言（如 Python）的易访问性。可重用性也得到了提升，通过实现运行时调度机制，可以方便地添加自定义操作符。TVM 将变量定义从字符串匹配简化为指针匹配，保证了每个变量只有一个定义位置（静态单赋值，SSA) [22])。

基于多面体的 IR - 多面体模型是深度学习编译器采用的一项重要技术。它使用线性规划、仿射变换等数学方法来优化具有静态控制流（边界和分支）的基于循环的代码。与 Halide 不同，在多面体模型中，内存引用和循环嵌套的边界可以是任意形状的多面体。这种灵活性使得多面体模型在通用编译器中得到了广泛应用。然而，这种灵活性也阻碍了其与调优机制的结合。尽管如此，由于能够处理深度嵌套的循环，许多深度学习编译器（如 TC 和 PlaidML（作为 nGraph 的后端））都采用了多面体模型作为其低级 IR。基于多面体的 IR 使得应用各种多面体变换（例如融合、瓦片、下沉和映射）变得容易，包括设备相关和设备无关的优化。许多基于多面体的编译器借鉴了工具链，如 isl [96], Omega [48], PIP [23], Polylib [60], 和 PPL [9]。

TC 在低级中间表示（IR）中具有其独特的设计，它结合了 Halide 和多面体模型。它使用基于 Halide 的 IR 来表示计算，并采用基于多面体的 IR 来表示循环结构。TC 通过抽象实例呈现详细表达式，并引入了特定的节点类型。简而言之，TC 使用域节点来指定索引变量的范围，并使用上下文节点来描述与硬件相关的新的迭代变量。此外，它使用带节点来确定迭代顺序。过滤器节点表示与语句实例结合的迭代器。集和序列是关键字，用于指定过滤器（并行和串行执行）的执行类型。此外，TC 使用扩展节点来描述代码生成所需的其它指令，例如内存移动。

PlaidML 使用基于多面体的 IR（称为 Stripe）来表示张量操作。它通过将并行多面体块的嵌套扩展到多个级别来创建可并行化代码的层次结构。此外，它允许嵌套多面体分配到嵌套的内存单元，为计算与内存层次结构匹配提供了一种方式。在 Stripe 中，硬件配置与内核代码无关。Stripe 中的标签（在其他编译器中称为 passes）不会改变内核结构，但为优化 passes 提供了有关硬件目标的额外信息。Stripe 将 DL 操作符拆分为适配本地硬件资源的瓦片。

其他独特的中间表示—有些深度学习编译器在不使用Halide和多面体模型的情况下实现了定制的低级中间表示。在定制的低级中间表示上，它们应用硬件特定优化并降低到LLVM中间表示。

Glow 中的低级中间表示是一种基于指令的表达式，它操作由地址 [79] 引用的张量。在 Glow 低级中间表示中，基于指令的函数有两种：声明和程序。前者声明程序生命周期内存在的常量内存区域的数量（例如输入、权重、偏差），后者是本地分配区域的列表，包括函数（例如卷积和池化）和临时变量。指令可以在全局内存区域或本地分配区域上运行。此外，每个操作数都带有一种限定符注释：@in 表示操作数从缓冲区读取；@out 表示操作数写入缓冲区；@inout 表示操作数既读取又写入缓冲区。这些

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.

4.2.2 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 scope in static analysis than JIT compilation. In addition, AOT approaches can be applied with cross-compilers of embedded platforms (e.g., C-GOOD [46]) as well as enable execution on remote machines (TVM RPC) and customized accelerators.

4.2.3 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).

4.3 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

指令和操作数限定符帮助 Glow 确定何时可以执行某些内存优化。

MLIR 受 LLVM 深受影响，并且它比 LLVM 是一个更纯粹的编译器基础设施。MLIR 重用了 LLVM 中的许多思想和接口，并且位于模型表示和代码生成之间。MLIR 具有灵活的类型系统，并允许多个抽象级别，并且它引入了方言来表示这些多个抽象级别。每个方言都由一组定义的不可变操作组成。MLIR 当前的方言包括 TensorFlow IR、XLA HLO IR、实验性多面体 IR、LLVM IR 和 TensorFlow Lite。方言之间的灵活转换也得到了支持。此外，MLIR 可以创建新的方言以连接到新的低级编译器，为硬件开发人员和编译器研究人员铺平了道路。

XLA 的 HLO IR 可以被认为是既是高级中间表示又是低级中间表示，因为 HLO 足够细粒度以表示硬件特定信息。此外，HLO 支持硬件特定优化，并且可以被用来生成 LLVM 中间表示。

4.2.2 基于低级IR的代码生成。大多数深度学习编译器采用的低级IR最终可以降级为LLVM IR，并受益于LLVM成熟的优化器和代码生成器。此外，LLVM可以从头开始为专用加速器显式设计自定义指令集。然而，传统编译器直接传递给LLVM IR时可能会生成较差的代码。为了避免这种情况，深度学习编译器采用两种方法来实现与硬件相关的优化：1) 在LLVM的上层IR（例如基于Halide的IR和基于多面体的IR）中执行针对特定目标的循环转换，2) 为优化传递关于硬件目标的其他信息。大多数深度学习编译器应用这两种方法，但侧重点不同。一般来说，倾向于前端用户（例如TC、TVM、XLA和nGraph）的深度学习编译器可能更关注1)，而更倾向于后端开发者的深度学习编译器（例如Glow、PlaidML和MLIR）可能更关注2)。

深度学习编译器中的编译方案主要可分为两大类：即时编译（JIT）和提前编译（AOT）。对于JIT编译器，它可以在运行时动态生成可执行代码，并能利用更好的运行时知识优化代码。AOT编译器则先生成所有可执行二进制文件，再执行它们。因此，AOT编译在静态分析方面比JIT编译具有更广的适用范围。此外，AOT方法还可以与嵌入式平台的交叉编译器（例如C-GOOD [46]）结合使用，同时支持在远程机器（TVM RPC）和定制加速器上执行。

4.2.3 讨论。在深度学习编译器中，低级中间表示（IR）是深度学习模型的细粒度表示，它反映了深度学习模型在不同硬件上的详细实现。低级IR包括基于Halide的IR、基于多边形模型的IR以及其他独特的IR。尽管它们在设计上有所不同，但都利用成熟的编译器工具链和基础设施，提供针对特定硬件优化的定制接口和代码生成。低级IR的设计也会影响新型深度学习加速器（例如TVM HalideIR和Inferentia，以及XLA HLO和TPU）的设计。

4.3 前端优化

构建计算图后，前端会应用图级优化。许多优化更容易在图级被识别和执行，因为图提供了计算的全局视图。这些优化仅应用于计算图，而不是后端的实现。因此它们与硬件无关，可以应用于各种后端目标。

前端优化通常由 passes 定义，并且可以通过遍历计算图的节点来执行图转换来应用。前端提供

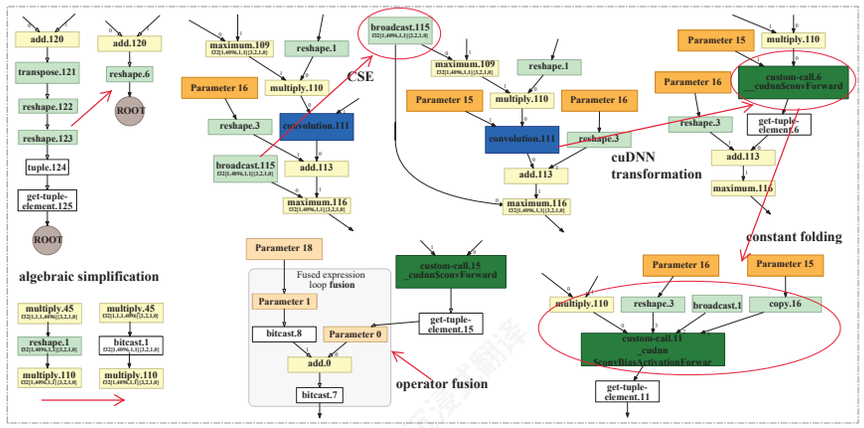

Fig. 3. Example of computation graph optimizations, taken from the HLO graph of Alexnet on Volta GPU using Tensorflow XLA.

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. Figure 3 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.

4.3.1 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.

4.3.2 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.

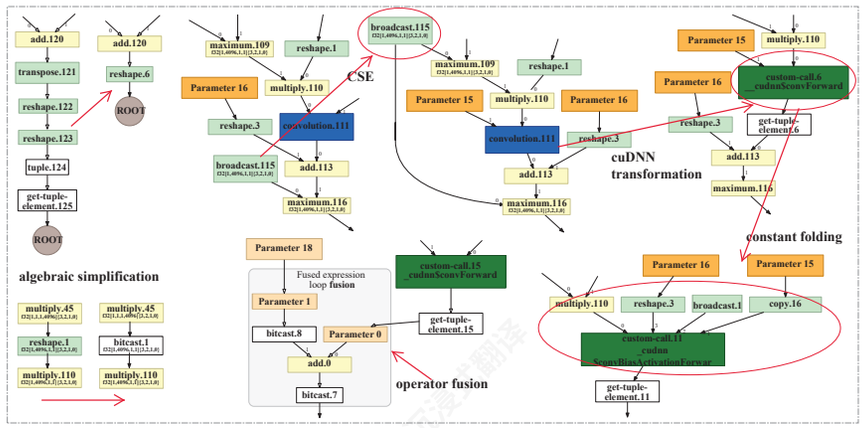

图3。计算图优化的示例，来自在Volta GPU上使用TensorFlow XLA的Alexnet的HLO图。

方法来 1) 从计算图中捕获特定功能，以及 2) 为优化重写图。除了预定义的 passes，开发者还可以在前端定义自定义 passes。大多数深度学习编译器在深度学习模型导入并转换为计算图后，可以确定每个操作的输入张量和输出张量的形状。此功能允许深度学习编译器根据形状信息执行优化。图 3 展示了使用 TensorFlow XLA 进行计算图优化的示例。

在本节中，我们将前端优化分为三类：1）节点级优化，2）块级（窥孔、局部）优化，和3）数据流级（全局）优化。

4.3.1 节点级优化。计算图的节点足够粗粒度，可以在单个节点内进行优化。节点级优化包括节点消除（消除不必要的节点）和节点替换（用其他低成本的节点替换节点）。

在通用编译器中，Nop消除会移除占用少量空间但指定无操作的no-op指令。在DL编译器中，Nop消除负责消除缺少足够输入的操作。例如，只有一个输入张量的求和节点可以被消除，填充宽度为零的填充节点可以被消除。

零维张量消除负责移除输入为零维张量的不必要操作。假设 A 是一个零维张量，B 是一个常量张量，那么 A 和 B 的求和操作节点可以用已存在的常量节点 B 替换，而不会影响正确性。假设 C 是一个三维张量，但其中一个维度的形状为零，例如 {0,2,3}，因此 C 没有元素，argmin/argmax 操作节点可以被消除。

4.3.2 块级优化。代数简化 - 代数简化优化包括 1) 代数识别，2) 强度降低，通过它可以用更便宜的替换更昂贵的操作符；3) 常量传播，通过它可以替换常量表达式为其值。此类优化考虑一系列节点，然后利用不同类型节点的交换律、结合律和分配律来简化计算。

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 [78], as well as reduces launch and synchronization overhead [91]. 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 [61, 62] 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.

4.3.3 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 [6]. 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 [4] firstly designs and performs memory-aware scheduling to minimize the peak

除了典型的操作符(+, ×等)之外，代数简化还可以应用于深度学习特定操作符（例如，reshape、transpose和pooling）。这些操作符可以重新排序，有时还可以被消除，从而减少冗余并提高效率。在这里，我们说明了代数简化可以应用的常见情况：1）计算顺序优化，在这种情况下，优化会根据特定特征查找并删除reshape/transpose操作。以矩阵乘法（GEMM）为例，有两个矩阵（例如，A和B），这两个矩阵都被转置（分别产生A^T和B^T），然后A^T和B^T相乘。然而，实现GEMM更有效的方法是交换参数A和B的顺序，将它们相乘，然后转置GEMM的输出，这样就将两个转置减少为一次；2）节点组合优化，在这种情况下，优化将多个连续的transpose节点组合成一个节点，消除身份transpose节点，并在实际上不移动数据时将transpose节点优化为reshape节点；3）ReduceMean节点优化，在这种情况下，优化将ReduceMean替换为AvgPool节点（例如，在Glow中），如果reduce操作符的输入是4D的，并且要减少最后两个维度。

算子融合 - 算子融合是深度学习编译器不可或缺的优化。它能够实现更好的计算共享、消除中间分配、通过组合循环嵌套 [78]，以及减少启动和同步开销 [91] 来促进进一步优化。在 TVM 中，算子被分为四类：单射型、归约型、复杂输出可融合型以及不透明型。当算子定义后，其对应的类别即被确定。针对上述类别，TVM 设计了跨算子的融合规则。在 TC 中，根据自动多边形变换，融合操作有所不同。然而，如何识别和融合更复杂的图模式，例如包含多个广播和归约节点的块，仍然是一个问题。近期工作 [61, 62] 试图解决这个问题，并提出一个框架来探索和优化激进的融合计划。它不仅支持逐元素和归约节点，还支持其他具有复杂依赖关系的计算/内存密集型节点。

算子下沉 - 这种优化将转置等操作下沉到批归一化、ReLU、sigmoid 和通道洗牌等操作下方。通过这种优化，许多相似的操作被移动得更近，从而创造了更多代数简化的机会。

4.3.3 数据流级优化。公共子表达式消除（CSE） - 表达式E是公共子表达式，如果E的值先前已计算，并且自上次计算以来E的值无需更改 [6]。在这种情况下，E的值计算一次，先前计算好的E值可以用于避免其他地方的重复计算。深度学习编译器在整个计算图中搜索公共子表达式，并用先前计算的结果替换后续的公共子表达式。

死代码消除（DCE） - 一段代码如果其计算结果或副作用未被使用，则该代码为死代码。DCE优化会移除死代码。死代码通常不是由程序员引起的，而是由其他图优化引起的。因此，与CSE一样，DCE也是在其他图优化之后应用的。其他优化，如死存储消除（DSE），它会移除那些永远不会被使用的张量存储，也属于DCE。

静态内存规划 - 静态内存规划优化会尽可能重用内存缓冲区。通常有两种方法：就地内存共享和标准内存共享。就地内存共享使用相同的内存作为操作的输入和输出，并且在计算前只分配一份内存。标准内存共享重用先前操作的内存而不重叠。静态内存规划是在离线完成的，这使得可以应用更复杂的规划算法。最近一项工作 [4] 首先设计和执行了内存感知调度，以最小化峰值

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 [58] 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.

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

4.4 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.

4.4.1 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

激活内存占用在边缘设备上，这为内存受限设备上的内存规划提出了新的研究方向。

布局转换 - 布局转换试图找到最佳的数据布局来存储计算图中的张量，然后将布局转换节点插入到图中
.
请注意，实际的转换在此处不会执行，而是在编译器后端评估计算图时执行

实际上，相同操作在不同数据布局中的性能不同，最佳布局也不同。例如，GPU上的NCHW格式操作通常运行更快，因此将GPU上的数据转换为NCHW格式（例如TensorFlow）是高效的。一些深度学习编译器依赖硬件特定库来实现更高的性能，而库可能需要特定的布局。此外，一些深度学习加速器更喜欢更复杂的布局（例如tile）。此外，边缘设备通常配备异构计算单元，不同的单元可能需要不同的数据布局以实现更好的利用率，因此布局转换需要仔细考虑。因此，编译器需要提供一种跨各种硬件执行布局转换的方法

不仅张量的数据布局对最终性能有非平凡的影响，而且转换操作也有显著的开销。因为它们也会消耗内存和计算资源。

最近一项基于TVM、面向CPU的工作 [58] 改变了计算图中所有卷积操作的布局，首先将其设置为NCHW[x]c，其中c表示通道C的拆分子维度，x表示子维度的拆分大小。然后在硬件特定优化期间，在提供硬件详细信息（如缓存行大小、向量化单元大小和内存访问模式）时，通过自动调优全局探索所有x参数。

4.3.4 讨论。前端是深度学习编译器中最重要的组件之一，负责将深度学习模型转换为高级中间表示（例如计算图），并基于高级中间表示进行硬件无关优化。尽管不同深度学习编译器在高级中间表示的数据表示和操作符定义上可能有所不同，但硬件无关优化在三个级别上收敛：节点级、块级和数据流级。每个级别的优化方法利用深度学习特定以及通用编译优化技术，从而减少计算冗余，并提高深度学习模型在计算图级别的性能。

4.4 后端优化

深度学习编译器的后端通常包含各种硬件特定优化、自动调优技术和优化内核库。硬件特定优化能够为不同的硬件目标生成高效代码。而自动调优在编译器后端中至关重要，它有助于减轻手动推导最佳参数配置的工作量。此外，高度优化的内核库也广泛应用于通用处理器和其他定制化的深度学习加速器。

4.4.1 硬件特定优化。硬件特定优化，也称为与目标相关的优化，应用于生成针对特定硬件的高性能代码。应用后端优化的方式之一是将底层IR转换为LLVM IR，利用LLVM基础设施生成优化的CPU/GPU代码。另一种方式是利用深度学习领域知识设计定制优化，更高效地利用目标硬件。由于硬件特定优化是为特定硬件量身定制的，且无法在本论文中穷尽，我们介绍了现有深度学习编译器中五种广泛采用的硬件特定优化方法。这些硬件特定优化的概述如图4所示，详细描述如下。

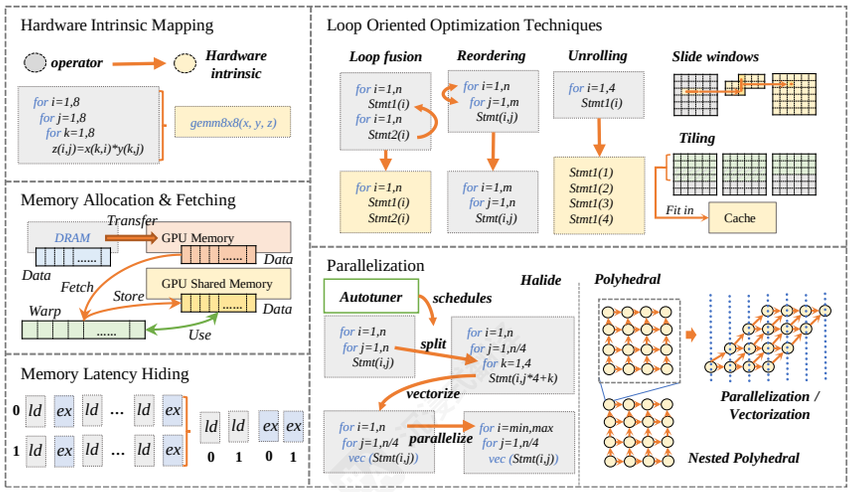

Fig. 4. Overview of hardware-specific optimizations applied in DL compilers.

compilers. The overview of these hardware-specific optimizations is shown in Figure 4, 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 [97] 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,

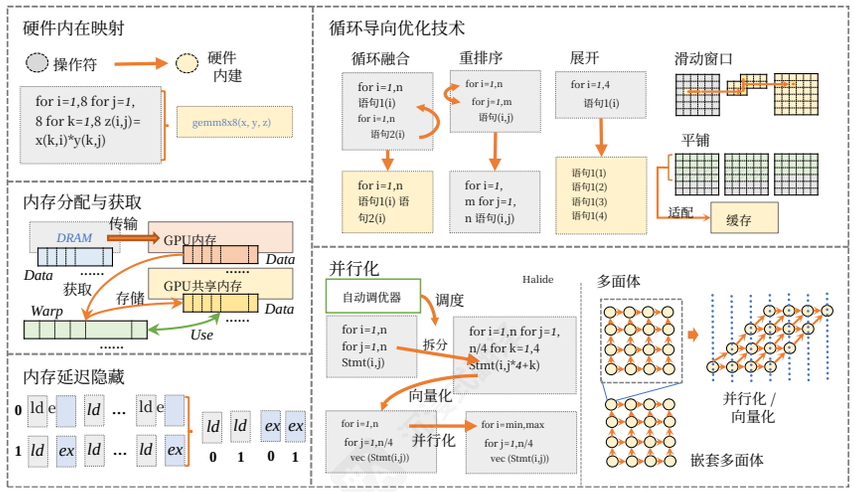

图4。DL编译器中应用的硬件特定优化概述。

这些硬件特定优化概述如图4所示，详细描述如下。

硬件内建映射 - 硬件内建映射可以将一组特定的低级IR指令转换为在硬件上已高度优化的内核。在TVM中，硬件内建映射通过可扩展张量化方法实现，该方法可以声明硬件内建的行为以及内建映射的降低规则。这种方法使编译器后端能够将硬件实现以及高度优化的手工微内核应用于特定的操作模式，从而显著提升性能。而Glow支持硬件内建映射，如量化。它可以估计神经网络每个阶段的可能数值范围，并支持基于分析的优化来自动执行量化。此外，Halide/TVM将特定的IR模式映射到每个架构上的SIMD指令码，以避免在遇到向量模式时LLVM IR映射的低效性。

内存分配与获取 - 内存分配是代码生成中的另一个挑战，特别是在GPU和定制加速器方面。例如，GPU主要包含共享内存空间（访问延迟低，内存容量有限）和本地内存空间（访问延迟高，容量大）。这种内存层次结构需要高效的内存分配和获取技术来提高数据局部性。为了实现这种优化，TVM引入了内存作用域的调度概念。内存作用域调度原语可以将计算阶段标记为共享或线程本地。对于标记为共享的计算阶段，TVM生成使用共享内存分配以及协作数据获取的代码，并在适当的位置插入内存屏障以确保正确性。此外，TC也通过扩展PPCG [97] 编译器提供了类似的功能（称为内存提升）。然而，TC仅支持有限的自定义规则。特别是，TVM通过内存作用域调度原语在加速器中启用特殊缓冲区。

内存延迟隐藏 - 内存延迟隐藏也是后端使用的一种重要技术，通过重新排序执行流水线来实现。由于大多数深度学习编译器支持 CPU 和 GPU 上的并行化，内存延迟隐藏可以自然地由硬件实现（例如 GPU 上的扭曲上下文切换）。但对于具有解耦访问-执行（DAE）架构的 TPU 类加速器，

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 [51] (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 4.4.3). 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.

后端需要执行调度和细粒度同步，以获得正确高效的代码。为了实现更好的性能并减少编程负担，TVM 引入了虚拟线程调度原语，该原语允许用户在虚拟化的多线程架构上指定数据并行性。然后 TVM 通过插入必要的内存屏障来降低这些虚拟并行化线程，并将这些线程的操作交错到一个单一的指令流中，这形成了一个更好的线程执行流水线，以隐藏内存访问延迟。

循环导向优化 - 循环导向优化也会在后端应用，以为目标硬件生成高效的代码。由于 Halide 和 LLVM [51] (集成多边形方法) 已经包含了这种优化技术，一些 DL 编译器

的后端利用了 Halide 和 LLVM。循环导向优化应用的关键技术包括循环融合、滑动窗口、分块、循环重排序和循环展开。

优化包括循环融合、滑动窗口、分块、循环重排和循环展开。

1) 循环融合：循环融合是一种优化技术，可以将边界相同的循环融合以更好地重用数据。对于 PlaidML、TVM、TC 和 XLA 等编译器，这种优化由 Halide 调度或多边形方法执行，而 Glow 通过其操作栈应用循环融合。

循环融合：循环融合是一种优化技术，可以将边界相同的循环融合以更好地重用数据。对于 PlaidML、TVM、TC 和 XLA 等编译器，这种优化由 Halide 调度或多边形方法执行，而 Glow 通过其操作栈应用循环融合。

2) 滑动窗口：滑动窗口是Halide采用的一种循环优化技术。其核心概念是在需要时计算值，并动态存储以实现数据重用，直到这些值不再需要为止

它们不再需要时。由于滑动窗口交错计算两个循环并使它们变为串行，因此这是并行性和数据重用之间的权衡。

3) 分块：分块将循环分割成多个块，因此循环被分为外循环遍历块和内循环在块内迭代。这种转换通过将块适配到硬件缓存中，在块内实现了更好的数据局部性

由于块的大小是硬件特定的，许多深度学习编译器通过自动调优来确定分块模式和大小。

4) 循环重排：循环重排（也称为循环置换）会改变嵌套循环中的迭代顺序，这可以优化内存访问并从而增加空间局部性。它特定于数据布局和硬件特性。然而，当存在迭代顺序上的依赖关系时，执行循环重排是不安全的。

但是，当存在迭代顺序上的依赖关系时，执行循环重排是不安全的。

5) 循环展开：循环展开可以将特定循环展开为固定数量的循环体副本，这允许编译器应用激进的指令级并行化。通常，循环展开

是与其他技术（如循环分裂）结合使用的，循环分裂首先将循环分裂为两个嵌套循环，然后完全展开内层循环。

并行化 - 由于现代处理器通常支持多线程和SIMD并行性，

编译器后端需要利用并行性来最大化硬件利用率以实现高性能。Halide 使用一种称为 parallel 的调度原语来指定循环的并行化维度，以实现线程级并行化，并通过将标记为 parallel 的循环维度与块和线程的注解映射来支持 GPU 并行化。它还用一个 n 宽的向量语句替换一个大小为 n 的循环，该向量语句可以通过硬件内建映射映射到硬件特定的 SIMD 指令。Stripe 开发了一种称为嵌套多面体模型的 polyhedral 模型的变体，它引入并行多面体块作为其迭代的基本执行元素。在此扩展之后，嵌套多面体模型可以在瓦片和步进的级别之间检测层次并行化。此外，一些深度学习编译器依赖于手工制作的库，如 Glow 或硬件供应商提供的优化数学库（讨论在第 4.4.3 节）。同时，Glow 将向量化卸载到 LLVM，因为当提供张量维度和循环迭代次数的信息时，LLVM 的自动向量化器工作得很好。然而，完全通过编译器后端利用并行性允许应用更多深度学习模型的领域特定知识，从而以更多工程工作为代价带来更高的性能。

4.4.2 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 [47] (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 pre-defined rules. *2) Genetic algorithm (GA)* [28]: 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)* [12]: 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 [5] (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

4.4.2 自动调优。由于硬件特定优化中参数调优的搜索空间巨大，因此有必要利用自动调优来确定最佳参数配置。在本调查中研究的深度学习编译器中，TVM、TC 和 XLA 支持自动调优。通常，自动调优实现包括四个关键组件，如参数化、成本模型、搜索技术和加速。

参数化 - 1) 数据和目标：数据参数描述数据的规范，例如输入形状。目标参数描述在优化调度和代码生成过程中需要考虑的特定硬件特性与约束。例如，对于GPU目标，需要指定共享内存和寄存器大小等硬件参数。2) 优化选项：优化选项包括优化调度及其对应参数，例如循环导向优化和瓦片大小。在TVM中，会考虑预定义和用户定义的调度以及参数。而TC和XLA更倾向于参数化优化，这些优化与性能密切相关，且可以低成本地稍后更改。例如，批处理维度是通常映射到CUDA网格维度的一个参数，可以在自动调优期间进行优化。

成本模型 - 自动调优中应用的不同成本模型的比较如下。1) 黑盒模型：该模型仅考虑最终执行时间，而不考虑编译任务的特征。黑盒模型易于构建，但在缺乏任务特征指导的情况下，容易导致更高的开销和次优解。TC采用此模型。2) 基于机器学习的成本模型：基于机器学习的成本模型是一种使用机器学习方法预测性能的统计方法。它使模型能够在探索新配置时更新，从而帮助实现更高的预测精度。TVM和XLA采用此类模型，例如梯度树提升模型（GBDT）和前馈神经网络 [47] (FNN)分别。3) 预定义成本模型：基于预定义成本模型的方法期望构建一个基于编译任务特征的完美模型，并能够评估任务的整体性能。与基于机器学习的模型相比，预定义模型在应用时产生的计算开销较少，但需要大量工程工作来在每次新的DL模型和硬件上重建模型。

搜索技术 - 1) 初始化和搜索空间确定：初始选项可以随机设置，也可以基于已知配置，例如用户提供的配置或历史最优配置。在搜索空间方面，应在自动调优之前指定。TVM允许开发者利用其领域特定知识指定搜索空间，并根据计算描述为每个硬件目标提供自动搜索空间提取。相比之下，TC依赖于编译缓存和预定义规则。2) 遗传算法（GA） [28]: GA将每个调优参数视为基因，将每个配置视为候选。根据适应度值，通过交叉、变异和选择迭代生成新候选，适应度值是一种受自然选择过程启发的元启发式算法。最终，得到最优候选。交叉、变异和选择的速率用于控制探索与利用之间的权衡。TC在其自动调优技术中采用GA。3) 模拟退火算法（SA） [12]: SA也是一种受退火过程启发的元启发式算法。它允许我们在递减的概率下接受更差的解，这可以在固定的迭代次数内找到近似全局最优解并避免精确局部最优解。TVM在其自动调优技术中采用SA。4) 强化学习(RL)：RL通过探索与利用之间的权衡来学习，以在给定环境中最大化奖励。Chameleon [5] (基于TVM)在其自动调优技术中采用RLRL。

加速 - 1) 并行化：加速自动调优的一个方向是并行化。TC 考虑到遗传算法需要评估每一代中的所有候选方案，提出了多线程、多 GPU 的策略。

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.

4.4.3 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 [45] are also supported. Other customized DL accelerators also maintain their specific kernel libraries [43, 57].

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.

4.4.4 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.

5 TAXONOMY OF DL COMPILERS

The DL compilers studied in this survey include TVM, nGraph, Tensor Comprehension (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,

2) 配置重用：加速自动调优的另一个方向是重用之前的自动调优配置。TC 通过编译缓存存储与给定配置对应的已知最快生成代码版本。在编译期间每次内核优化前都会查询缓存，若缓存未命中则会触发自动调优。类似地，TVM 生成一个日志文件，其中存储所有调度操作符的最佳配置，并在编译期间查询最佳配置。值得一提的是，TVM 对 Halide IR 中的每个操作符（例如 conv2d）进行自动调优，因此最佳配置是针对每个操作符单独确定的。

4.4.3 优化内核库。存在多个高度优化的内核库被广泛用于加速在各种硬件上的深度学习训练和推理。来自英特尔的 DNNL（前身是 MKL-DNN）、来自英伟达的 cuDNN 以及来自 AMD 的 MIOpen 是广泛使用的库。计算密集型原语（例如卷积、 GEMM 和 RNN ）和内存带宽受限原语（例如批归一化、池化和 shuffle）都根据硬件特性（例如 AVX-512 指令集、张量核心）进行了高度优化。此外，还支持可自定义的数据布局，以便轻松集成到深度学习应用中并避免频繁的数据布局转换。另外，还支持低精度训练和推理，包括 FP32、FP16、INT8 和非 IEEE 浮点格式的 bfloat16 [45] 。其他定制深度学习加速器也维护其特定的内核库 [43, 57]。

现有的深度学习编译器，如 TVM、nGraph 和 TC，可以在代码生成期间生成调用这些库的函数。然而，如果深度学习编译器需要利用现有的优化内核库，它们应首先将数据布局和融合风格转换为内核库中预定义的类型。此类转换可能会破坏最优控制流。此外，深度学习编译器将内核库视为黑盒。因此，在调用内核库时，它们无法在操作符之间应用优化（例如操作符融合）。总之，当计算可以由特定的高度优化原语满足时，使用优化内核库可以实现显著的性能提升，否则它可能会受到进一步优化的限制并遭受性能欠佳。

4.4.4 讨论。后端负责裸机优化和基于低级中间表示的代码生成。尽管由于各种低级中间表示，后端的设计可能有所不同，但它们的优化可以归类为硬件特定优化：自动调优技术和优化内核库。这些优化可以单独执行或组合，通过利用硬件/软件特性来提高数据局部性和并行化。最终，深度学习模型的的高级中间表示被转换为在不同硬件上的高效代码实现。

5 深度学习编译器分类法

本次调查研究的深度学习编译器包括TVM、nGraph、张量理解（TC）、Glow和XLA。我们选择这些编译器，因为它们知名度高、维护良好，最重要的是应用广泛。因此，我们可以找到足够的论文、文档和行业与学术界的讨论，以便深入研究它们的设计和实现。表1从前端、后端、中间表示和优化四个角度，展示了所选深度学习编译器的分类法，这与本调查描述的关键组件相对应。

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.

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 Figure 2 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.

6 EVALUATION

6.1 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 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.

6.2 End-to-end Performance Comparison

As shown in Figure 5, 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 (Figure 6 and 8), we still implement several `ResNet` and `MobileNetV2` models in TC. In sum, we compare and analyze the performance results from the following perspectives.

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

后端、中间表示和优化，这与本调查描述的关键组件相对应。

具体而言，我们根据现有知识提供了更多关于编译器的信息。我们不仅提供编译器是否支持特定功能的说明，还描述了如何通过其编程接口使用该功能。此外，我们还描述了特定功能的发展状态以及特定功能在特定编译器中不受支持的原因。本分类法的目的是为考虑其需求的从业者提供关于深度学习编译器选择的指南，并为研究人员提供深度学习编译器的全面总结。

在表1中，我们展示了每个深度学习编译器的功能，包括开发者、编程语言、ONNX/框架支持、训练支持以及前端分类中的量化支持，并在后端分类中展示了编译方法和支持设备。这些功能被总结起来，因为它们强烈影响深度学习编译器在特定场景中的使用。基于这些功能，从业者或研究人员可以轻松决定他们希望工作的深度学习编译器。

表1，连同图2可以作为本次调查的系统总结。通过它们，读者可以识别每个编译器支持的功能以及每个编译器的主要组件。更详细的信息将在以下各节中介绍。

6 评估

6.1 实验设置

我们的实验在两台配备GPU的机器上进行，硬件配置如表2所示。我们在CPU和GPU上评估了TVM（v0.6.0）、nGraph（0.29.0-rc.0）、TC（commit fd01443）、Glow（commit 7e68188）和XLA（TensorFlow2.2.0）的性能。我们选择了19个ONNX格式的神经网络模型作为我们的数据集，它们是从Torchvison[2] 模型库和GluonCV[3] 模型库转换而来的。这些模型包括完整模型：ResNet、DenseNet和VGG系列，以及轻量级模型：MobileNet和MNASNet系列。为了导入ONNX模型，如表1所示，我们使用了TVM内置的tvm.relay.frontend.from_onnx接口、nGraph的ngraph-onnx Python包、Glow内置的ONNXModelLoader以及XLA的tensorflow-onnx Python包。值得注意的是，TC缺乏对ONNX的支持，因此我们仅在以下逐层性能比较中评估它。每个模型执行15次，我们报告每个编译器最后10次执行的平均执行时间，因为我们认为前5次执行是预热，以消除JIT编译的开销。

6.2 端到端性能比较

如图5所示，我们比较了TVM、nGraph、Glow和XLA的端到端推理性能。我们在CPU（Broadwell和Skylake）和GPU（V100和2080Ti）上评估了这些编译器。请注意，我们在此省略了TC的比较。因为TC更像是内核库，而不是功能完整的深度学习编译器，并且它要求用户使用其Einstein概念手动实现模型的所有层，这导致公平比较需要大量的工程工作。另一个原因是TC仅支持在GPU上运行，因此我们无法获得其在CPU上的性能结果。然而，对于详细的比较（图6和图8），我们仍然在TC中实现了几个ResNet和MobileNetV2模型。总之，我们从以下几个方面比较和分析性能结果。

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

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 Backend opt	Hardware independent optimizations (refer to Section 4.3) Hardware specific optimizations (refer to Section 4.4) And hybrid optimizations				
	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)

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

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

表1。深度学习编译器的比较，包括TVM、nGraph、TC、Glow和XLA。

		TVM	nGraph	TC	Glow	XLA
	开发者	Apache	英特尔	Facebook	Facebook	Google
Frontend	编程-ing	Python/C++ Lambda表达式	Python/C++ 张量表达式	Python/C++ 爱因斯坦记号	Python/C++ 层编程	Python/C++ TensorFlow接口
	ONNX 支持	✓ tvm.relay前端 .from_onnx (内置)	✓ 使用ngraph-onnx (Python包)	×	✓ ONNXModelLoader (内置)	✓ 使用tensorflow-onnx (Python包)
	框架 支持	tvm.relay前端 .from_* (内置) tensorflow/tflite/keras pytorch/caffe2 mxnet/coreml/darknet	TensorFlow PaddlePaddle (使用*-桥接, 充当后端)	(定义和优化 一个TC内核，该内核 最终被调用 其他框架。) PyTorch/其他DLPack 支持的框架	PyTorch/Caffe2 TensorFlow Lite (使用内置 ONNXIFI 接口)	使用 TensorFlow 接口
	训练 支持	× 正在开发中 (支持导数 操作符)	✓ 仅在 NNP-T 处理器上 处理器	✓ (支持自动 微分)	✓ (有限支持)	✓ 使用TensorFlow 接口
	量化 支持	✓ int8/fp16	✓ int8（包含训练）	×	✓ int8	✓ int8/int16（使用 TensorFlow 接口）
IR	高-/低-级别中间表示	Relay/Halide	nGraph IR/None	TC IR/Polyhedral	其自身的 高-/低-级别 IR	HLO (两者 高和低)
	动态 形状	✓ *(任何)*	✓ *(部分形状)*	×	×	✓ *(无)*
Optimization	前端 opt 后端 opt	硬件无关优化（参见第4.3节） 硬件特定优化（参见第4.4节） 以及混合优化				
	自动调优	✓ （要选择最佳） 计划参数)	× (调用优化后的内核) 库，不需要)	✓ （为了减少JIT） 顶部)	× (附加信息是) 已在中间表示中提供)	✓ (在默认 卷积和GEMM)
	核 库	✓ mkl/cudnn/cublas	✓ eigen/mkldnn/cudnn/ 其他	×	×	✓ eigen/mkl/ cudnn/tensorrt
Backend	编译 方法	JIT AOT（实验性）	JIT	JIT	JIT AOT（使用内置 可执行包）	JIT AOT（生成 可执行库）
	支持 设备	CPU/GPU/ARM FPGA/定制（ 使用VTA）	CPU/英特尔GPU/NNP GPU/定制（ 使用OpenCL支持 在PlaidML中）	Nvidia GPU	CPU/GPU 定制（ 官方文档)	CPU/GPU/TPU 定制化 (官 方文档)

兼容性 - 虽然nGraph和XLA声称支持ONNX，但仍然存在兼容性问题。1) 由于动态形状张量，nGraph无法运行DenseNet121、VGG16/19和`MNASNet0_5/1_0` 模型。我们用ONNX模型库[4], 中的相应模型替换了DenseNet121、VGG16/19模型，而MNASNet0_5/1_0 模型不可用。此外，当我们将PlaidML设置为nGraph的GPU后端时，我们无法运行

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

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)

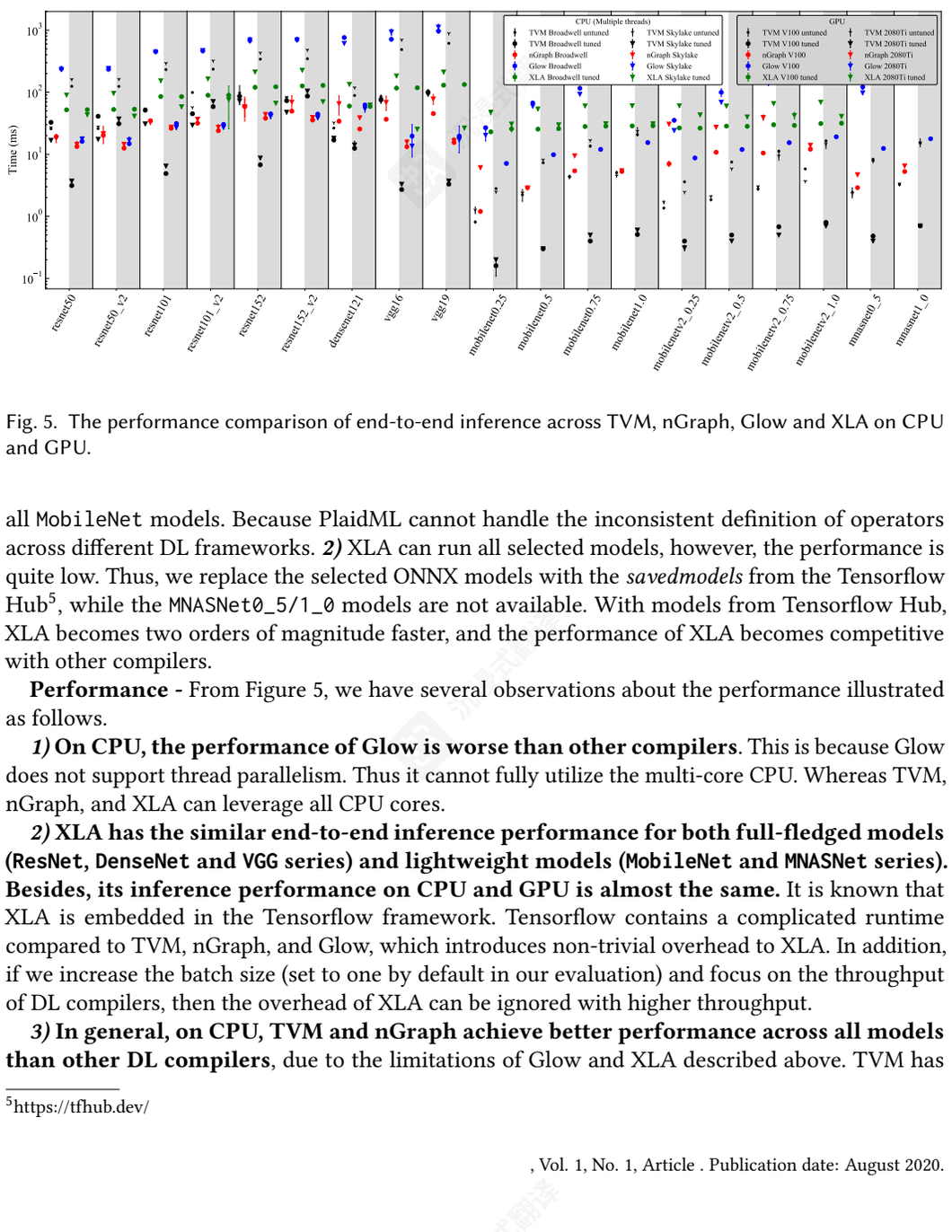

Fig. 5. The performance comparison of end-to-end inference across TVM, nGraph, Glow and XLA on CPU and GPU.

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 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 Figure 5, 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

[5]https://tfhub.dev/

表2。硬件配置。

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

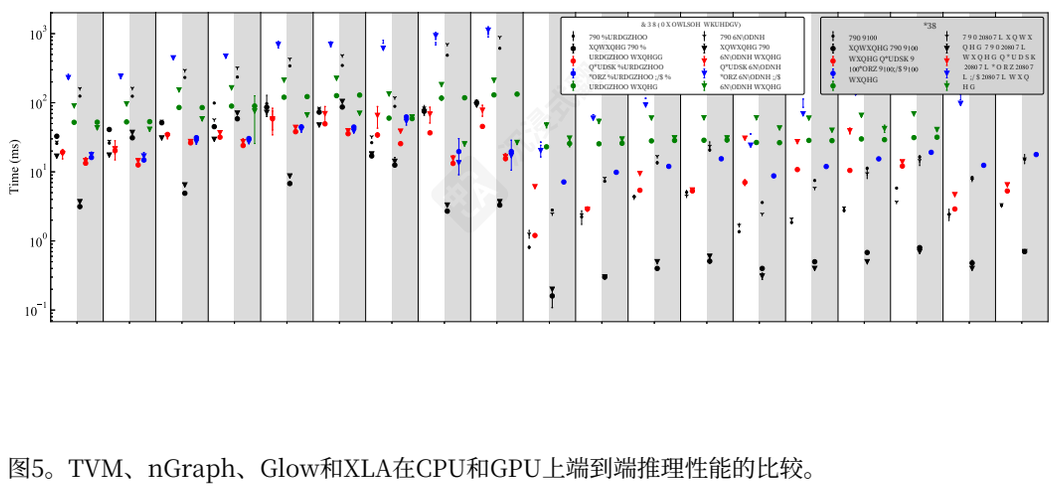

图5。TVM、nGraph、Glow和XLA在CPU和GPU上端到端推理性能的比较。

所有MobileNet模型。因为PlaidML无法处理不同深度学习框架中操作符定义的不一致性。2) XLA可以运行所有选定的模型，但是性能相当低。因此，我们将选定的ONNX模型替换为Tensorflow Hub[5], 保存的模型，而MNASNet`0_5/1_0` 模型不可用。使用Tensorflow Hub的模型后，XLA的速度提高了两个数量级，其性能与其他编译器变得具有竞争力。

性能 - 从图5中，我们可以观察到以下性能表现：

1) 在CPU上，Glow的性能比其他编译器差。这是因为Glow不支持线程并行。因此它无法充分利用多核CPU。而TVM、nGraph和XLA可以利用所有CPU核心。

2) XLA在完整模型（ResNet、DenseNet和VGG系列）和轻量级模型（MobileNet和MNASNet系列）上具有相似的端到端推理性能。此外，其在CPU和GPU上的推理性能几乎相同。众所周知，XLA嵌入在Tensorflow框架中。与TVM、nGraph和Glow相比，Tensorflow包含一个复杂的运行时，这给XLA带来了非平凡的额外开销。此外，如果我们增加批大小（在我们的评估中默认设置为1）并关注深度学习编译器的吞吐量，那么随着吞吐量的提高，XLA的额外开销可以忽略不计。

3) 一般来说，在CPU上，TVM和nGraph在所有模型上的性能都优于其他深度学习编译器，这是由于上文所述的Glow和XLA的局限性。TVM有

[5]https://tfhub.dev/

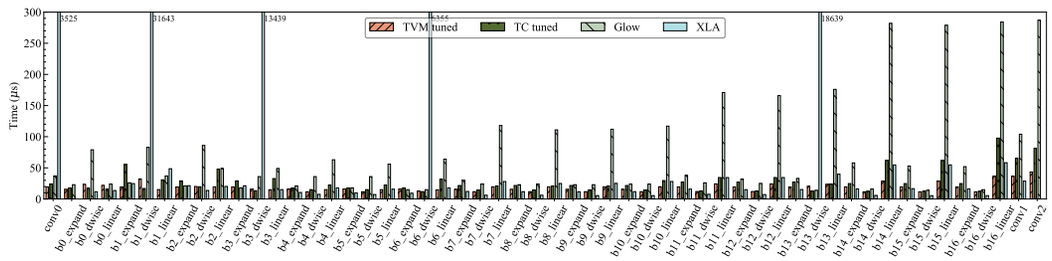

Fig. 6. The performance comparison of convolution layers in `MobileNetV2_1.0` across TVM, TC, Glow and XLA on V100 GPU.

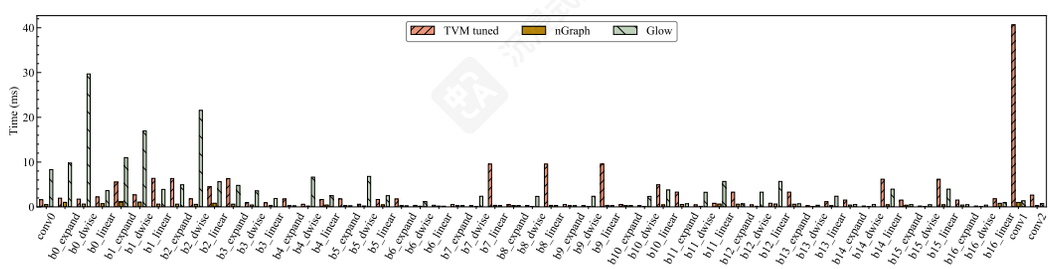

Fig. 7. The performance comparison of convolution layers in `MobileNetV2_1.0` across TVM, nGraph and Glow on Broadwell CPU.

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.

6.3 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)*

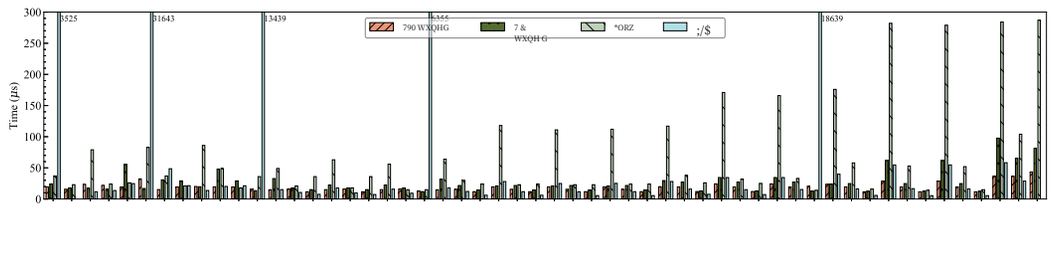

图6。MobileNetV2_1.0中卷积层的性能比较，在TVM、TC、Glow和XLA上使用V100 GPU进行测试。

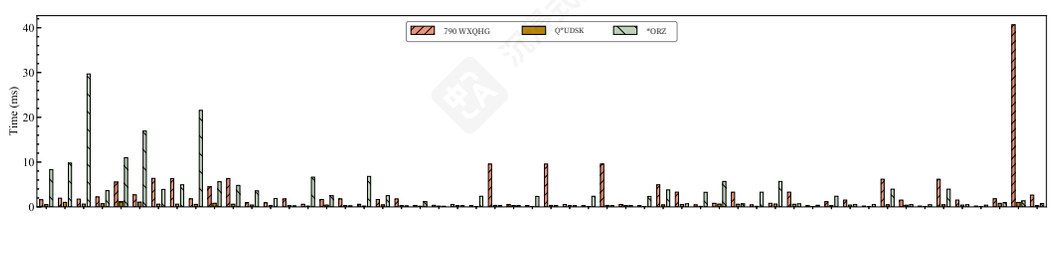

图7。MobileNetV2_1.0中卷积层的性能比较，在TVM、nGraph和Glow上使用Broadwell CPU进行测试。

在完整模型上与nGraph性能相当，但在轻量级模型上优于nGraph。nGraph依赖于DNNL（之前称为MKL-DNN）库进行加速
.
因此，nGraph可以将优化后的子图卸载到DNNL，并受益于DNNL为Intel CPU量身定制的细粒度指令级JIT优化。

4) 经过调优的TVM（使用200次试验调优）在所有模型中几乎都实现了最佳性能，尤其是在轻量级模型（MobileNet、MNASNet系列）上。根据我们的调查，这是因为这些模型内部的经典操作符的调度已经被TVM开发者很好地设计，并且TVMtophub中提供了默认参数。默认的调度和参数可以帮助TVM实现与其他深度学习编译器相似的性能。此外，在CPU上，调优的TVM与未调优的TVM之间的性能差异可以忽略不计，但在GPU上则相当显著（平均加速41.26× 倍）。这是因为GPU比CPU具有更复杂的线程和内存层次结构，因此为了发挥计算能力，GPU需要更细粒度的调度（例如TVM中的tiling、splitting和reordering）。因此，确定GPU上的最佳调度参数至关重要，
在自动调优中展现出其有效性之处。

6.3 按层性能比较

为进一步比较深度学习编译器后端优化的能力，我们评估了ResNet50和MobileNetV2_1.0在V100 GPU和Broadwell CPU（单线程，因为Glow缺乏多线程支持）上的按层（由于卷积层主导推理时间）性能。

方法论 - 为了测量各个层的执行时间，我们根据深度学习编译器、硬件（CPU/GPU）和CNN模型采用不同的方法。具体来说，1)

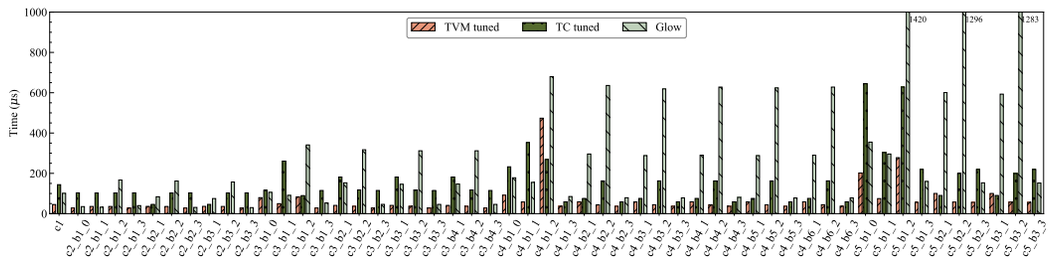

Fig. 8. The performance comparison of convolution layers in ResNet50 across TVM, TC and Glow on V100 GPU.

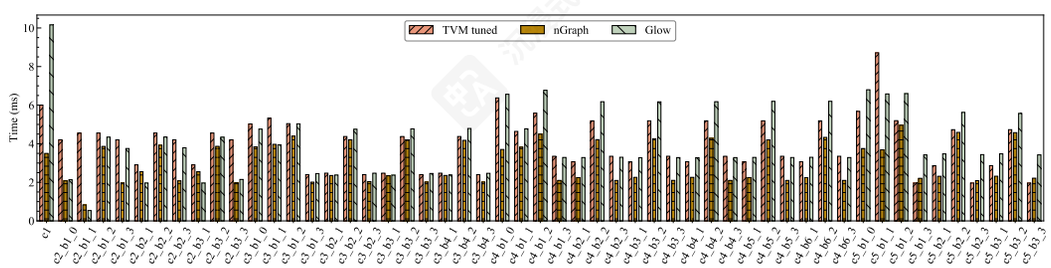

Fig. 9. The performance comparison of convolution layers in ResNet50 across TVM, nGraph and Glow on Broadwell CPU.

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* [71] toolkit from Nvidia for GPU performance.

Performance - From Figure 6, 7 8, 9, 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 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

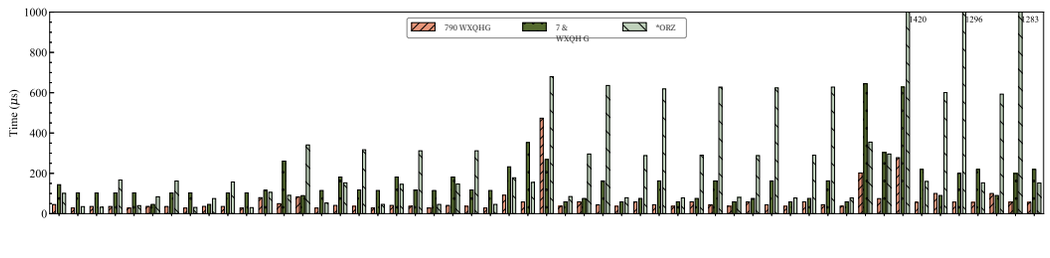

图8. ResNet50中卷积层的性能比较，跨TVM、TC和Glow在V100 GPU上。

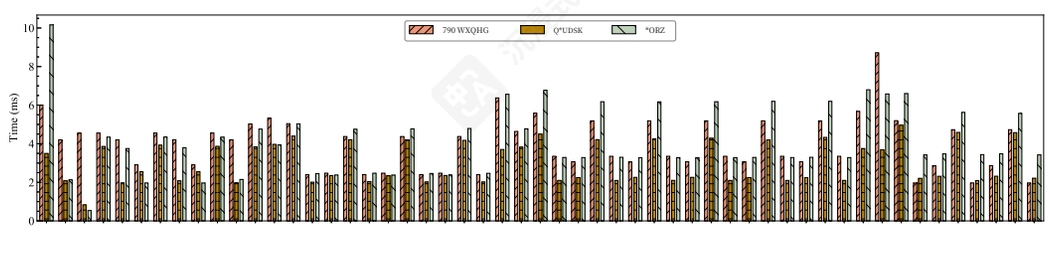

图9. ResNet50中卷积层的性能比较，跨TVM、nGraph和Glow在Broadwell CPU上。

在TVM上，我们复用自动调优的日志来提取内核形状和最优调度。然后我们重新构建各个卷积层，并使用时间_评估器进行评估。2) 我们通过Glow的跟踪文件提取执行时间。3) 我们在TC上测量手写内核的执行时间。4) 对于nGraph，我们利用时间线来测量CPU上的执行时间。不过，它的时间线功能不被其PlaidML后端（通过OpenCL提供GPU支持）支持。此外，目前没有可用的方法来分析OpenCL中的命令队列。因此，我们将nGraph在GPU上每层性能的分析工作留待未来研究。4) 对于XLA，我们利用内置的tf.profiler.experimental方法来测量CPU性能，以及Nvidia提供的DLProf [71] 工具包来测量GPU性能。

性能 - 从图6、7、8、9中，我们可以观察到关于所展示性能的几个要点，具体如下：

1) nGraph在CPU上的卷积层性能更好，这得益于硬件（英特尔CPU）和软件（编译器、库和运行时）的协同设计。而TVM在这些编译器中在GPU上表现更好。在MobileNetV2_1.0上，TVM的性能不稳定，尤其是在conv1层。这是因为在同一台机器上的其他进程会影响自动调优过程，因此它倾向于推导出不精确甚至负的调度参数。

2) TC允许用户通过Einstein概念定义张量计算内核（例如，卷积），而无需指定输入/输出张量的形状（例如，内核大小）。然后内核被自动调优并存储在其编译缓存中，以加速进一步的自动调优和编译。然而，在我们的评估中，我们发现TC的性能严重依赖于最初编译的内核。以MobileNetV2_1.0为例，如果我们用层c1初始化自动调优，那么c1可以表现良好。但随着层数的增加，接下来的c∗_b∗_∗ 层会变得非常慢。

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

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 [79], 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 Figure 6 (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 Figure 7 - 9.

6.4 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 optimizations tend to be co-designed for further exploit the performance opportunities (e.g., clustering in XLA and more advanced optimizations [58, 61]). 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*),

表3。XLA在V100 GPU和Broadwell CPU上的聚类和非聚类卷积数量。

	MobileNetV2 1.0_		ResNet50	
	聚类	非-聚类	集群化	非集群-
V100	5	47	0	53
Broadwell	17	35	53	0

更深（远离c1层）。为了获得一致的性能，我们需要单独调整每个内核。

3) Glow 落后于其他编译器，以优化 MobileNetV2_1.0 的 1×1 卷积（例如 b∗_ 线性层）以及 ResNet50 的深度可分离卷积（例如 c∗_b∗_2 层）。在 GPU 和 CPU 上计算这些卷积都需要更长时间。我们注意到 Glow 上的卷积通常与其他层（例如 ReLU、BatchNorm）融合，这可能就是性能低于其他编译器的原因。此外，在 CPU 上，MobileNetV2_1.0 结尾的卷积比开头的卷积快得多。根据跟踪日志，我们发现这些卷积是通过 CPUConvDKKC8 优化 [79], 加速的，该优化针对具有特定模式的卷积应用了分块、布局转换和向量化技术。

4) 至于XLA，它可以自动编译(_XlaCompile)来自TensorFlow的符合条件的子图，并用生成的二进制程序(_XlaRun)替换这些子图。此外，卷积层可能会与其他内核进行聚合，因此它们的性能不容易单独测量。因此，我们已经统计了聚合和非聚合的卷积层，数据如表3所示。请注意，TensorFlow中的MobileNetV2_1.0模型在起始和结束层与ONNX模型略有不同，但线性瓶颈层是相同的。此外，如果一个卷积层需要聚合，那么在完成_XlaCompile之前最多可以测量两次。因此，图6中有五个极值（对应MobileNetV2_1.0中的5个聚合卷积层）。实际上，只有聚合的内核被XLA优化，而非聚合的内核则由TensorFlow优化。因此，不可能测量由XLA优化的独立卷积层的执行时间。因此，我们决定不在图7-9中包含XLA的性能。

6.4 讨论

通过上述跨深度学习编译器的定量性能比较，我们可以深入分析具有前端（图级）和后端（操作符级）优化的粗粒度端到端性能，以及关于具有后端优化的卷积的细粒度每层性能。然而，仍然存在挑战，难以准确衡量不同深度学习编译器所采用的优化的有效性。在我们的评估过程中，一个特别困难之处在于，现有深度学习编译器中的前端和后端优化通常是紧密耦合的，因为1）前端优化通常会影响一系列操作符。因此，作为后端优化输入的优化操作符在不同编译器之间有所不同；2）这些优化往往协同设计，以进一步利用性能机会（例如XLA中的聚类和更高级的优化 [58, 61]）。因此，如果要在深度学习编译器之间单独评估和比较特定优化，则很难甚至不可能。

为解决此问题，我们一直在开发一个通用的基准测试框架，用于测量现有深度学习编译器的每层性能。基本思路是提取目标层（我们称其为模型片段）所需的必要结构和参数，

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.

7 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 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) [99] 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

并将这些层重新构建为特定深度学习编译器的可接受输入，这允许编译器忠实地应用相应的前端和后端优化。然后，我们可以测量这些优化模型片段的性能，以了解深度学习编译器在感兴趣层级的有效性。使用模型片段的基准测试框架可扩展到感兴趣的定制层（例如，融合层）。有了这样的基准测试框架，我们可以为每个深度学习编译器推导出粗粒度（例如，端到端）和细粒度（例如，每层）的性能指标，从而比较不同深度学习编译器在感兴趣层级上的优化效果。目前，我们已经成功从最先进的CNN模型中提取了目标层，例如ResNet50的瓶颈和MobileNetV2_1.0的线性瓶颈。我们的基准测试框架仍在快速开发中，希望很快能向社区开放。

7 结论与未来方向

在本调查中，我们对现有的深度学习编译器的设计原则进行了深入分析。首先，我们深入探讨了现有深度学习编译器中采用的通用架构，包括多级中间表示、前端和后端。我们详细介绍了每个组件的设计理念和参考实现，重点强调深度学习编译器特有的中间表示和优化。我们总结了本调查的发现，并重点指出深度学习编译器的未来方向如下：

动态形状和预/后处理 - 动态模型在深度学习领域越来越受欢迎，其输入形状甚至在执行过程中都可能发生变化。特别是，在自然语言处理领域，模型可能接受各种形状的输入，这对深度学习编译器来说是一个挑战，因为数据的形状直到运行时才未知。现有的深度学习编译器需要更多研究工作来高效支持动态形状，以应对新兴的动态模型。

此外，随着未来的深度学习模型变得更加复杂，其整个控制流可能不可避免地包含复杂的预/后处理步骤。目前，大多数深度学习编译器使用Python作为其编程语言，当预/后处理由Python解释器执行时，可能会成为性能瓶颈。这种潜在的性能瓶颈尚未被现有的深度学习编译器考虑。支持深度学习编译器中的整个控制流，能够表达和优化与深度学习模型一起的预/后处理，为模型部署中的性能加速开辟了新的机遇。

高级自动调优 - 现有的自动调优技术主要关注单个操作符的优化。然而，局部最优的组合并不一定能达到全局最优。例如，两个应用于不同数据布局的相邻操作符可以一起调优，而无需在它们之间引入额外的内存转换。此外，随着边缘计算的兴起，执行时间不仅是对深度学习编译器的优化目标。自动调优中还应考虑新的优化目标，如内存占用和能耗。

特别是，对于基于机器学习的自动调优技术，有几种值得进一步探索的方向。首先，机器学习技术可以应用于自动调优的其他阶段，而不仅仅是成本模型。例如，在选择编译器选项和优化计划阶段，机器学习技术可以直接预测可能性，并开发算法来确定最终配置。其次，基于机器学习的自动调优技术可以根据领域知识进行改进。例如，将特征工程（选择用于表示程序的特性）[99] 融入自动调优技术中，可能是实现更好调优结果的一个潜在方向。

多面体模型 - 将多面体模型与自动调优技术相结合，在设计深度学习编译器以提高效率方面，是一个有前景的研究方向。一方面，自动调优可以应用于通过重用先前的配置来最小化多面体JIT编译的开销。另一方面，多面体模型可用于执行自动调度，这可以减少自动调优的搜索空间。

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 [84] 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 [14, 90]. Fortunately, the polyhedral community has made progress in supporting sparse tensor [94, 95], 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. Secondly, 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 [78] 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.

配置。另一方面，多面体模型可用于执行自动调度，这可以减少自动调优的搜索空间。

将多面体模型应用于深度学习编译器所面临的另一个挑战是支持稀疏张量。通常，稀疏张量（如CSF [84] ）的格式使用索引数组（例如a[b[i]]）表示循环索引，这不再是线性的。这种间接索引寻址会导致非仿射下标表达式和循环边界，从而禁止多面体模型的循环优化 [14, 90]。幸运的是，多面体社区在支持稀疏张量 [94, 95], 方面取得了进展，并且整合最新的多面体模型进展可以增加深度学习编译器的性能机会。

子图划分 - 支持子图划分的深度学习编译器可以将计算图划分为多个子图，并且子图可以以不同的方式处理。子图划分给深度学习编译器带来了更多研究机会。首先，它为优化集成了图库提供了可能性。以nGraph和DNNL为例，DNNL是一个具有图优化的深度学习库，它利用了大量高度优化的内核集合。DNNL与nGraph的集成使DNNL能够加速nGraph生成的子图的执行。其次，它为异构并行执行提供了可能性。一旦计算图被划分为子图，不同子图的执行可以同时分配给异构硬件目标。以边缘设备为例，其计算单元可能由ARM CPU、Mail GPU、DSP和可能NPU组成。从深度学习编译器生成利用所有计算单元高效运行的子图可以显著加速深度学习任务。

量化- 传统深度学习框架中应用的量化策略基于一套固定的方案和数据类型，对运行在不同硬件上的代码几乎没有定制。而，在深度学习编译器中支持量化可以利用编译过程中的优化机会，以推导出更高效的量化策略。例如，Relay [78] 提供了一个量化重写流程，可以自动为各种方案生成量化代码。

为了支持量化，深度学习编译器需要解决几个挑战。第一个挑战是如何在不投入大量工程工作的情况下实现新的量化操作符。AWS 的尝试指出了一个可能的方案，即使用方言的概念在基本操作符上实现新的操作符，以便在图级别和操作符级别的优化可以复用。第二个挑战是编译过程中量化和其他优化之间的交互。例如，确定量化的适当阶段以及与算子融合等优化协同工作，需要未来的研究探索。

统一优化- 虽然现有的深度学习编译器在计算图优化和硬件特定优化方面采用了相似的设计，但每个编译器在某些方面都有自己的优势。目前缺少一种方式来共享最先进的优化，以及支持现有编译器中的新兴硬件目标。我们主张统一现有深度学习编译器的优化，以便每个深度学习编译器中采用的最佳实践可以被复用。此外，统一深度学习编译器之间的优化可以积累强大的力量来影响通用和专用深度学习加速器的设计，并为深度学习编译器和硬件的高效协同设计提供环境。

目前，Google MLIR 正在朝着这样的方向发展，是一个有前景的举措。它提供了多级中间表示的基础设施，并包含用于在每一级中间表示之间进行转换的中间表示规范和工具包。它还提供了灵活的方言，以便每个深度学习编译器都能为其高级和低级中间表示构建定制的方言。通过方言之间的转换，一个深度学习编译器的优化可以被另一个编译器重用。然而，方言的转换需要进一步的研究工作，以减少对精细设计的依赖。

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 [89], Flux [40] and Julia [13]. 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 [24] 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 [27, 67, 74] 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.

ACKNOWLEDGEMENTS

The authors would like to thank Jun Yang from Alibaba, Yu Xing from Xilinx, and Byung Hoon Ahn from UCSD for their valuable comments and suggestions.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. USENIX Association, Savannah, GA, USA, 265–283.

[2] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, Cedric Bourrasset, and François Berry. 2017. Tactics to directly map CNN graphs on embedded FPGAs. *IEEE Embedded Systems Letters* 9, 4 (2017), 113–116.

可微分编程 - 可微分编程是一种编程范式，其中程序是完全可微分的。在可微分编程范式中编写的算法可以自动微分，这对深度学习社区很有吸引力。许多编译器项目已经采用了可微分编程，例如 Myia [89],Flux [40] 和 Julia [13]。不幸的是，现有的深度学习编译器对微分编程的支持很少。

要支持微分编程对现有的深度学习编译器来说相当具有挑战性。这些困难不仅来自数据结构，还来自语言语义。例如，为了实现从 Julia 到 XLA HLO IR 的转换，其中一个挑战 [24] 是 Julia 使用的命令式语言与 XLA 使用的符号语言在控制流上存在差异。为了高效地使用 HLO IR，编译器还需要为 Julia 提供操作抽象，以支持 XLA 的特定语义，例如 MapReduce 和广播。此外，Julia 和 XLA 在微分语义上的差异，也要求编译器设计进行重大改变。

隐私保护 - 在边缘云系统中，深度学习模型通常被分成两半，每个部分模型分别运行在边缘设备和云服务上，这可以提供更好的响应延迟并减少通信带宽消耗。然而，边缘云系统的一个缺点是用户隐私变得容易受到攻击。原因是攻击者可以拦截从边缘设备发送到云端的中间结果，然后利用这些中间结果训练另一个模型，该模型可以揭示偏离原始用户任务的隐私信息。

为在边缘云系统中保护隐私，现有方法 [27, 67, 74]建议向中间结果添加具有特殊统计特性的噪声，以在不严重降低用户任务准确性的情况下降低攻击者任务的准确性。然而，难点在于确定噪声应插入的层，这需要大量工作才能识别最佳层。上述难点为深度学习编译器支持隐私保护提供了巨大机遇，因为编译器维护着丰富的深度学习模型信息，可以指导跨层自动插入噪声。

训练支持 - 一般来说，当前深度学习编译器对模型训练的支持远不够。如表1所示，nGraph仅支持在Intel NNP-T加速器上训练，TC仅支持单个内核的自动微分，Glow对有限模型提供实验性的训练支持，TVM的训练支持正在开发中，而XLA则依赖TensorFlow的训练支持。总之，当前深度学习编译器主要专注于高效地将深度学习模型部署到各种硬件上，因此它们选择推理作为主要优化目标。然而，扩展深度学习编译器支持模型训练的能力将开辟大量研究机会，例如梯度算子优化和高阶自动微分。

致谢

作者感谢阿里巴巴的杨军、Xilinx的徐星以及UCSD的安炳勋提供的宝贵评论和建议。

参考文献

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. USENIX Association, Savannah, GA, USA, 265–283.

[2] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, Cedric Bourrasset, and François Berry. 2017. Tactics to directly map CNN graphs on embedded FPGAs. *IEEE Embedded Systems Letters* 9, 4 (2017), 113–116.

[3] Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, NI Adams, Daniel P. Friedman, E Kohlbecker, GL Steele, David H Bartley, Robert Halstead, et al. 1998. Revised 5 report on the algorithmic language Scheme. *Higher-order and symbolic computation* 11, 1 (1998), 7–105.
[4] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. 2020. Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices. arXiv:cs.DC/2003.02369
[5] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation. arXiv:cs.LG/2001.08743
[6] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.
[7] Alibaba. 2019. Announcing Hanguang 800: Alibaba's First AI-Inference Chip. https://www.alibabacloud.com/blog/announcing-hanguang-800-alibabas-first-ai-inference-chip_595482. Accessed February 4, 2020.
[8] Amazon. 2018. AWS Inferentia. https://aws.amazon.com/machine-learning/inferentia. Accessed February 4, 2020.
[9] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. 2006. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems.
[10] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. 2015. Comparative Study of Deep Learning Software Frameworks. arXiv:cs.LG/1511.06435
[11] Baidu. 2016. PaddlePaddle Github repository. https://github.com/PaddlePaddle/Paddle. Accessed February 4, 2020.
[12] Dimitris Bertsimas, John Tsitsiklis, et al. 1993. Simulated annealing. *Statistical science* 8, 1 (1993), 10–15.
[13] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.
[14] Chun Chen. 2012. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, Beijing, China, 499–508.
[15] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. 2018. The rise of deep learning in drug discovery. *Drug discovery today* 23, 6 (2018), 1241–1250.
[16] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems.
[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. USENIX Association, Carlsbad, CA, USA, 578–594.
[18] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. arXiv:cs.NE/1410.0759
[19] François Chollet et al. 2015. Keras. https://keras.io.
[20] R. Collobert, K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*. Curran Associates, Granada, Spain, 6.
[21] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. 2018. Intel ngraph: An intermediate representation, compiler, and executor for deep learning.
[22] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.
[23] P. Feautrier. 1988. Parametric integer programming. *RAIRO Recherche Opérationnelle* 22, 3 (1988), 243–268.
[24] Keno Fischer and Elliot Saba. 2018. Automatic full compilation of julia programs and ML models to cloud TPUs.
[25] Rubén D Fonnegra, Bryan Blair, and Gloria M Díaz. 2017. Performance comparison of deep learning frameworks in image classification problems using convolutional and recurrent networks. In *2017 IEEE Colombian Conference on Communications and Computing (COLCOM)*. IEEE, Cartagena, Colombia, 1–6.
[26] David A Forsyth and Jean Ponce. 2002. *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, USA.
[27] Ruiyuan Gao, Ming Dun, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2019. Privacy for Rescue: A New Testimony Why Privacy is Vulnerable In Deep Models.
[28] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.
[29] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Networks. arXiv:stat.ML/1406.2661
[30] Danny Goodman. 2007. *JavaScript bible*. John Wiley & Sons, Hoboken, NJ, USA.
[31] Tobias Grosser. 2000. Polyhedral Compilation. https://polyhedral.info. Accessed February 4, 2020.
[32] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid

[3] Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, NI Adams, Daniel P. Friedman, E Kohlbecker, GL Steele, David H Bartley, Robert Halstead, et al. 1998. Revised 5 report on the algorithmic language Scheme. *Higher-order and symbolic computation* 11, 1 (1998), 7–105.
[4] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. 2020. Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices. arXiv:cs.DC/2003.02369
[5] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation. arXiv:cs.LG/2001.08743
[6] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.
[7] Alibaba. 2019. Announcing Hanguang 800: Alibaba's First AI-Inference Chip. https://www.alibabacloud.com/blog/announcing-hanguang-800-alibabas-first-ai-inference-chip_595482. Accessed February 4, 2020.
[8] Amazon. 2018. AWS Inferentia. https://aws.amazon.com/machine-learning/inferentia. Accessed February 4, 2020.
[9] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. 2006. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems.
[10] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. 2015. Comparative Study of Deep Learning Software Frameworks. arXiv:cs.LG/1511.06435
[11] Baidu. 2016. PaddlePaddle Github repository. https://github.com/PaddlePaddle/Paddle. Accessed February 4, 2020.
[12] Dimitris Bertsimas, John Tsitsiklis, et al. 1993. Simulated annealing. *Statistical science* 8, 1 (1993), 10–15.
[13] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.
[14] Chun Chen. 2012. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, Beijing, China, 499–508.
[15] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. 2018. The rise of deep learning in drug discovery. *Drug discovery today* 23, 6 (2018), 1241–1250.
[16] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems.
[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. USENIX Association, Carlsbad, CA, USA, 578–594.
[18] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. arXiv:cs.NE/1410.0759
[19] François Chollet et al. 2015. Keras. https://keras.io.
[20] R. Collobert, K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*. Curran Associates, Granada, Spain, 6.
[21] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. 2018. Intel ngraph: An intermediate representation, compiler, and executor for deep learning.
[22] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.
[23] P. Feautrier. 1988. Parametric integer programming. *RAIRO Recherche Opérationnelle* 22, 3 (1988), 243–268.
[24] Keno Fischer and Elliot Saba. 2018. Automatic full compilation of julia programs and ML models to cloud TPUs.
[25] Rubén D Fonnegra, Bryan Blair, and Gloria M Díaz. 2017. Performance comparison of deep learning frameworks in image classification problems using convolutional and recurrent networks. In *2017 IEEE Colombian Conference on Communications and Computing (COLCOM)*. IEEE, Cartagena, Colombia, 1–6.
[26] David A Forsyth and Jean Ponce. 2002. *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, USA.
[27] Ruiyuan Gao, Ming Dun, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2019. Privacy for Rescue: A New Testimony Why Privacy is Vulnerable In Deep Models.
[28] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.
[29] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Networks. arXiv:stat.ML/1406.2661
[30] Danny Goodman. 2007. *JavaScript bible*. John Wiley & Sons, Hoboken, NJ, USA.
[31] Tobias Grosser. 2000. Polyhedral Compilation. https://polyhedral.info. Accessed February 4, 2020.
[32] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid

templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, IEEE Computer Society, Napa, CA, USA, 152–159.

[33] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 1 (2017), 35–47.

[34] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2017. A Survey of FPGA-Based Neural Network Accelerator. arXiv:cs.AR/1712.08934

[35] Qianyu Guo, Xiaofei Xie, Lei Ma, Qiang Hu, Ruitao Feng, Li Li, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2018. An Orchestrated Empirical Study on Deep Learning Frameworks and Platforms.

[36] Jung-Woo Ha, Hyuna Pyo, and Jeonghee Kim. 2016. Large-scale item categorization in e-commerce using multiple recurrent neural networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, San Francisco, CA, USA, 107–115.

[37] William Grant Hatcher and Wei Yu. 2018. A survey of deep learning: platforms, applications and emerging research trends. *IEEE Access* 6 (2018), 24411–24432.

[38] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[39] Jeremy Howard et al. 2018. fastai. https://github.com/fastai/fastai.

[40] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal Singh, and Viral Shah. 2018. Fashionable Modelling with Flux.

[41] Intel. 2019. Nervana Neural Network Processor. https://www.intel.ai/nervana-nnp/. Accessed February 4, 2020.

[42] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, ACM, Orlando, FL, USA, 675–678.

[43] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU Architecture via Microbenchmarking.

[44] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, Toronto, ON, Canada, 1–12.

[45] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. 2019. A Study of BFLOAT16 for Deep Learning Training. arXiv:cs.LG/1905.12322

[46] Duseok Kang, Euiseok Kim, Inpyo Bae, Bernhard Egger, and Soonhoi Ha. 2018. C-GOOD: C-code generation framework for optimized on-device deep learning. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, ACM, San Diego, CA, USA, 105.

[47] Samuel Kaufman, Phitchaya Mangpo Phothilimthana, and Mike Burrows. 2019. Learned TPU Cost Model for XLA Tensor Programs. In *Proceedings of the Workshop on ML for Systems at NeurIPS 2019*. Curran Associates, Vancouver, Canada, 1–6.

[48] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The Omega calculator and library, version 1.1. 0. *College Park, MD* 20742 (1996), 18.

[49] Adrian Kingsley-Hughes. 2017. Inside AppleâĂŹs new A11 Bionic processor.

[50] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. Association for Computing Machinery, New York, NY, USA, 207âĂŞ218. https://doi.org/10.1145/567532.567555

[51] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, IEEE Computer Society, San Jose, CA, USA, 75.

[52] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:cs.PL/2002.11054

[53] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled.

[54] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[55] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2020. The Deep Learning Compiler: A Comprehensive Survey.

[56] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE, IEEE, Cupertino, CA, USA, 1–44.

templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, IEEE Computer Society, Napa, CA, USA, 152–159.

[33] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 1 (2017), 35–47.

[34] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2017. A Survey of FPGA-Based Neural Network Accelerator. arXiv:cs.AR/1712.08934

[35] Qianyu Guo, Xiaofei Xie, Lei Ma, Qiang Hu, Ruitao Feng, Li Li, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2018. An Orchestrated Empirical Study on Deep Learning Frameworks and Platforms.

[36] Jung-Woo Ha, Hyuna Pyo, and Jeonghee Kim. 2016. Large-scale item categorization in e-commerce using multiple recurrent neural networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, San Francisco, CA, USA, 107–115.

[37] William Grant Hatcher and Wei Yu. 2018. A survey of deep learning: platforms, applications and emerging research trends. *IEEE Access* 6 (2018), 24411–24432.

[38] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[39] Jeremy Howard et al. 2018. fastai. https://github.com/fastai/fastai.

[40] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal Singh, and Viral Shah. 2018. Fashionable Modelling with Flux.

[41] Intel. 2019. Nervana Neural Network Processor. https://www.intel.ai/nervana-nnp/. Accessed February 4, 2020.

[42] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, ACM, Orlando, FL, USA, 675–678.

[43] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU Architecture via Microbenchmarking.

[44] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, Toronto, ON, Canada, 1–12.

[45] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. 2019. A Study of BFLOAT16 for Deep Learning Training. arXiv:cs.LG/1905.12322

[46] Duseok Kang, Euiseok Kim, Inpyo Bae, Bernhard Egger, and Soonhoi Ha. 2018. C-GOOD: C-code generation framework for optimized on-device deep learning. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, ACM, San Diego, CA, USA, 105.

[47] Samuel Kaufman, Phitchaya Mangpo Phothilimthana, and Mike Burrows. 2019. Learned TPU Cost Model for XLA Tensor Programs. In *Proceedings of the Workshop on ML for Systems at NeurIPS 2019*. Curran Associates, Vancouver, Canada, 1–6.

[48] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The Omega calculator and library, version 1.1. 0. *College Park, MD* 20742 (1996), 18.

[49] Adrian Kingsley-Hughes. 2017. Inside AppleâĂŹs new A11 Bionic processor.

[50] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. Association for Computing Machinery, New York, NY, USA, 207âĂŞ218. https://doi.org/10.1145/567532.567555

[51] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, IEEE Computer Society, San Jose, CA, USA, 75.

[52] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:cs.PL/2002.11054

[53] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled.

[54] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[55] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2020. The Deep Learning Compiler: A Comprehensive Survey.

[56] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE, IEEE, Cupertino, CA, USA, 1–44.

[57] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Seoul, South Korea, 393–405. https://doi.org/10.1109/ISCA.2016.42
[58] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing {CNN} Model Inference on CPUs. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. USENIX Association, Renton, WA, USA, 1025–1040.
[59] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. 2016. Automatic code generation of convolutional neural networks in FPGA implementation. In *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, IEEE, Xi'an, China, 61–68.
[60] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra. https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz
[61] Guoping Long, Jun Yang, and Wei Lin. 2019. FusionStitching: Boosting Execution Efficiency of Memory Intensive Computations for DL Workloads. arXiv:cs.DC/1911.11576
[62] Guoping Long, Jun Yang, Kai Zhu, and Wei Lin. 2018. FusionStitching: Deep Fusion and Code Generation for Tensorflow Computations on GPUs. arXiv:cs.DC/1811.05213
[63] Yufei Ma, Naveen Suda, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2018. ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler. *Integration* 62 (2018), 14–23.
[64] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. MIT press, Cambridge, MA, USA.
[65] John McCarthy and Michael I Levin. 1965. LISP 1.5 programmer's manual.
[66] Microsoft. 2017. ONNX Github repository. https://github.com/onnx/onnx. Accessed February 4, 2020.
[67] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhyani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh. 2020. Shredder: Learning noise distributions to protect inference privacy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne, Switzerland, 3–18.
[68] Mehdi Mohammadi, Ala Al-Fuqaha, Mohsen Guizani, and Jun-Seok Oh. 2017. Semisupervised deep reinforcement learning in support of IoT and smart city services. *IEEE Internet of Things Journal* 5, 2 (2017), 624–635.
[69] MXNet. 2017. Gluon. https://gluon.mxnet.io. Accessed February 4, 2020.
[70] Madhumitha Nara, BR Mukesh, Preethi Padala, and Bharath Kinnal. 2019. Performance Evaluation of Deep Learning frameworks on Computer Vision problems. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. IEEE, Tirunelveli, India, India, 670–674.
[71] NVIDIA. 2019. DLProf User-guide. https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/. Accessed August 26, 2020.
[72] NVIDIA. 2019. Nvidia Turing Architecture. https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/. Accessed February 4, 2020.
[73] NVIDIA. 2019. TensorRT Github repository. https://github.com/NVIDIA/TensorRT. Accessed February 4, 2020.
[74] Seyed Ali Osia, Ali Taheri, Ali Shahin Shamsabadi, Kleomenis Katevas, Hamed Haddadi, and Hamid R Rabiee. 2018. Deep private-feature extraction. *IEEE Transactions on Knowledge and Data Engineering* 32, 1 (2018), 54–66.
[75] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. Curran Associates, Vancouver, BC, Canada, 8024–8035.
[76] Tomas Petricek and Don Syme. 2012. Syntax Matters: Writing abstract computations in F#.
[77] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI âĂŹ13)*. Association for Computing Machinery, New York, NY, USA, 519âĂŞ530.
[78] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. 2019. Relay: A High-Level Compiler for Deep Learning. arXiv:cs.LG/1904.08368
[79] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. arXiv:cs.PL/1805.00907
[80] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.
[81] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, ACM, San Francisco, CA, USA, 2135–2135.

[57] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Seoul, South Korea, 393–405. https://doi.org/10.1109/ISCA.2016.42
[58] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing {CNN} Model Inference on CPUs. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. USENIX Association, Renton, WA, USA, 1025–1040.
[59] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. 2016. Automatic code generation of convolutional neural networks in FPGA implementation. In *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, IEEE, Xi'an, China, 61–68.
[60] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra. https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz
[61] Guoping Long, Jun Yang, and Wei Lin. 2019. FusionStitching: Boosting Execution Efficiency of Memory Intensive Computations for DL Workloads. arXiv:cs.DC/1911.11576
[62] Guoping Long, Jun Yang, Kai Zhu, and Wei Lin. 2018. FusionStitching: Deep Fusion and Code Generation for Tensorflow Computations on GPUs. arXiv:cs.DC/1811.05213
[63] Yufei Ma, Naveen Suda, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2018. ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler. *Integration* 62 (2018), 14–23.
[64] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. MIT press, Cambridge, MA, USA.
[65] John McCarthy and Michael I Levin. 1965. LISP 1.5 programmer's manual.
[66] Microsoft. 2017. ONNX Github repository. https://github.com/onnx/onnx. Accessed February 4, 2020.
[67] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhyani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh. 2020. Shredder: Learning noise distributions to protect inference privacy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne, Switzerland, 3–18.
[68] Mehdi Mohammadi, Ala Al-Fuqaha, Mohsen Guizani, and Jun-Seok Oh. 2017. Semisupervised deep reinforcement learning in support of IoT and smart city services. *IEEE Internet of Things Journal* 5, 2 (2017), 624–635.
[69] MXNet. 2017. Gluon. https://gluon.mxnet.io. Accessed February 4, 2020.
[70] Madhumitha Nara, BR Mukesh, Preethi Padala, and Bharath Kinnal. 2019. Performance Evaluation of Deep Learning frameworks on Computer Vision problems. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. IEEE, Tirunelveli, India, India, 670–674.
[71] NVIDIA. 2019. DLProf User-guide. https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/. Accessed August 26, 2020.
[72] NVIDIA. 2019. Nvidia Turing Architecture. https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/. Accessed February 4, 2020.
[73] NVIDIA. 2019. TensorRT Github repository. https://github.com/NVIDIA/TensorRT. Accessed February 4, 2020.
[74] Seyed Ali Osia, Ali Taheri, Ali Shahin Shamsabadi, Kleomenis Katevas, Hamed Haddadi, and Hamid R Rabiee. 2018. Deep private-feature extraction. *IEEE Transactions on Knowledge and Data Engineering* 32, 1 (2018), 54–66.
[75] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. Curran Associates, Vancouver, BC, Canada, 8024–8035.
[76] Tomas Petricek and Don Syme. 2012. Syntax Matters: Writing abstract computations in F#.
[77] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI âĂŹ13)*. Association for Computing Machinery, New York, NY, USA, 519âĂŞ530.
[78] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. 2019. Relay: A High-Level Compiler for Deep Learning. arXiv:cs.LG/1904.08368
[79] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. arXiv:cs.PL/1805.00907
[80] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.
[81] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, ACM, San Francisco, CA, USA, 2135–2135.

[82] Shayan Shams, Richard Platania, Kisung Lee, and Seung-Jong Park. 2017. Evaluation of deep learning frameworks over different HPC architectures. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, IEEE Computer Society, Atlanta, GA, USA, 1389–1396.
[83] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, IEEE Computer Society, Taipei, Taiwan, China, 17.
[84] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, Austin, Texas, USA, 1–7.
[85] D Team et al. 2016. Deeplearning4j: Open-source distributed deep learning for the jvm.
[86] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions.
[87] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. 2019. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, Anchorage, AK, USA, 2002–2011.
[88] Bart Van Merriënboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. Automatic differentiation in ML: Where we are and where we should be going. In *Advances in neural information processing systems*. Curran Associates, Montréal, Canada, 8757–8767.
[89] Bart van MerriÃńnboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. Automatic differentiation in ML: Where we are and where we should be going. arXiv:cs.LG/1810.11530
[90] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. 2006. Polyhedral code generation in the real world. In *International Conference on Compiler Construction*. Springer, Springer, Vienna, Austria, 185–201.
[91] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions.
[92] Stylianos I Venieris and Christos-Savvas Bouganis. 2016. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, IEEE Computer Society, Washington, DC, USA, 40–47.
[93] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. 2018. Toolflows for Mapping Convolutional Neural Networks on FPGAs. *Comput. Surveys* 51, 3 (Jun 2018), 1âĂŞ39. https://doi.org/10.1145/3186332
[94] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI âĂŹ15)*. ACM, Portland, OR, USA, 521âĂŞ532.
[95] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, Orlando, FL, USA, 185–194.
[96] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, Springer, Kobe, Japan, 299–302.
[97] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713
[98] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, ACM, Austin, TX, USA, 110.
[99] Zheng Wang and Michael OâĂŹBoyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.
[100] Gu-Yeon Wei, David Brooks, et al. 2019. Benchmarking tpu, gpu, and cpu platforms for deep learning.
[101] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, ACM, Austin, TX, USA, 29.
[102] Yu Xing, Jian Weng, Yushun Wang, Lingzhi Sui, Yi Shan, and Yu Wang. 2019. An In-depth Comparison of Compilers for Deep Neural Networks on Hardware. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE, IEEE, Las Vegas, NV, USA, 1–8.
[103] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. 2018. Dynamic Control Flow in Large-Scale Machine Learning. In *Proceedings of the Thirteenth EuroSys Conference*

[82] Shayan Shams, Richard Platania, Kisung Lee, and Seung-Jong Park. 2017. Evaluation of deep learning frameworks over different HPC architectures. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, IEEE Computer Society, Atlanta, GA, USA, 1389–1396.
[83] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, IEEE Computer Society, Taipei, Taiwan, China, 17.
[84] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, Austin, Texas, USA, 1–7.
[85] D Team et al. 2016. Deeplearning4j: Open-source distributed deep learning for the jvm.
[86] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions.
[87] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. 2019. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, Anchorage, AK, USA, 2002–2011.
[88] Bart Van Merriënboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. Automatic differentiation in ML: Where we are and where we should be going. In *Advances in neural information processing systems*. Curran Associates, Montréal, Canada, 8757–8767.
[89] Bart van MerriÃńnboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. Automatic differentiation in ML: Where we are and where we should be going. arXiv:cs.LG/1810.11530
[90] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. 2006. Polyhedral code generation in the real world. In *International Conference on Compiler Construction*. Springer, Springer, Vienna, Austria, 185–201.
[91] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions.
[92] Stylianos I Venieris and Christos-Savvas Bouganis. 2016. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, IEEE Computer Society, Washington, DC, USA, 40–47.
[93] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. 2018. Toolflows for Mapping Convolutional Neural Networks on FPGAs. *Comput. Surveys* 51, 3 (Jun 2018), 1âĂŞ39. https://doi.org/10.1145/3186332
[94] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI âĂŹ15)*. ACM, Portland, OR, USA, 521âĂŞ532.
[95] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, Orlando, FL, USA, 185–194.
[96] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, Springer, Kobe, Japan, 299–302.
[97] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713
[98] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, ACM, Austin, TX, USA, 110.
[99] Zheng Wang and Michael OâĂŹBoyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.
[100] Gu-Yeon Wei, David Brooks, et al. 2019. Benchmarking tpu, gpu, and cpu platforms for deep learning.
[101] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, ACM, Austin, TX, USA, 29.
[102] Yu Xing, Jian Weng, Yushun Wang, Lingzhi Sui, Yi Shan, and Yu Wang. 2019. An In-depth Comparison of Compilers for Deep Neural Networks on Hardware. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE, IEEE, Las Vegas, NV, USA, 1–8.
[103] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. 2018. Dynamic Control Flow in Large-Scale Machine Learning. In *Proceedings of the Thirteenth EuroSys Conference*

(EuroSys âĂŹ18). Association for Computing Machinery, New York, NY, USA, Article Article 18, 15 pages. https://doi.org/10.1145/3190508.3190551

[104] R. Zhao, S. Liu, H. Ng, E. Wang, J. J. Davis, X. Niu, X. Wang, H. Shi, G. A. Constantinides, P. Y. K. Cheung, and W. Luk. 2018. Hardware Compilation of Deep Neural Networks: An Overview. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE Computer Society, Milano, Italy, 1–8. https://doi.org/10.1109/ASAP.2018.8445088

(EuroSys âĂŹ18). Association for Computing Machinery, New York, NY, USA, Article Article 18, 15 pages. https://doi.org/10.1145/3190508.3190551

[104] R. Zhao, S. Liu, H. Ng, E. Wang, J. J. Davis, X. Niu, X. Wang, H. Shi, G. A. Constantinides, P. Y. K. Cheung, and W. Luk. 2018. Hardware Compilation of Deep Neural Networks: An Overview. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE Computer Society, Milano, Italy, 1–8. https://doi.org/10.1109/ASAP.2018.8445088