Empirical Software Engineering (2022) 27:172
https://doi.org/10.1007/s10664-022-10221-7

On the usage and development of deep learning compilers: an empirical study on TVM

Xiongfei Wu[1] · Jinqiu Yang[2] · Lei Ma[1] · Yinxing Xue[3] · Jianjun Zhao[1]

Accepted: 29 July 2022 / Published online: 20 September 2022

Abstract

Recent advances in deploying deep learning (DL) models have inspired the innovation of DL compilers from both industry and academia such as Facebook Glow and TVM. Given the importance of DL compilers, we seek for answering the important question to ease the adoption and development of TVM: *What challenges do users face when using DL compilers and what are common challenges for developers when developing DL compilers.* This paper presents the first empirical study on identifying the challenges in both usage and development of a DL compiler. We choose TVM as the representative DL compiler and manually inspect 347 sampled posts from its official discuss forum. We identify a taxonomy of challenges in usage of TVM consisting of 15 categories and seven types of common topics about developing TVM. Furthermore, we characterize TVM bugs in total of four impacts to obtain an initial understanding on defects of TVM through manual inspection of 44 bug reports and propose five implications for both developers and researchers in order to improve the development practices and build more robust DL compilers.

Communicated by: Shaowei Wang, Tse-Hsun (Peter) Chen, Sebastian Baltes, Ivano Malavolta, Christoph Treude, Alexander Serebrenik

This article belongs to the Topical Collection: *Collective Knowledge in Software Engineering*

✉ Xiongfei Wu
xiongfei.wu.a94@s.kyushu-u.ac.jp

Jinqiu Yang
jinqiu.yang@concordia.ca

Lei Ma
malei@ait.kyushu-u.ac.jp

Yinxing Xue
yxxue@ustc.edu.cn

Jianjun Zhao
zhao@ait.kyushu-u.ac.jp

[1] Kyushu University, Fukuoka, Japan

[2] Concordia University, Montreal, Canada

[3] University of Science and Technology of China, Hefei, China

Keywords Deep learning · Compiler · Optimization · Intermediate representation

1 Introduction

Deep learning (DL) has been widely applied to many cutting-edge areas, e.g., machine translation (Wang et al. 2020; Hoang et al. 2018), natural language processing (Deng and Liu 2018), image processing (Hemanth and Estrela 2017), cancer diagnosis (Fakoor et al. 2013), self-driving cars (Badue et al. 2021). To meet the requirement of these wide applications, various DL models such as convolutions neural network (CNN) (Lecun et al. 1998), recurrent neural network (RNN) (Rumelhart et al. 1986), long-short term memory (LSTM) (Schmidhuber and Hochreiter 1997) have been proposed. With the rapidly growing complexity of DL models, it is decisive to alleviate the efforts on programming such DL models. Up to now, many high-performance DL frameworks such as TensorFlow (Abadi et al. 2016), PyTorch (Paszke et al. 2019) have been proposed, allowing researchers to quickly implement and experiment with various DL models.

Nevertheless, due to the specific data-driven programming paradigm, DL applications often come with high computation complexity. Generally, most of the current DL workloads are running on general-purpose platforms, i.e., GPU, CPU. To further push the limit of performance on DL workloads and energy efficiency, enormous effort has been put into designing DL-specific hardware from both industrial and academia, e.g., Google TPU (Jouppi et al. 2017), dedicated DL accelerator based on field programmable gate array (FPGA) (Lacey et al. 2016), Apple Bionc (Kingsley-Hughes 2017), Cambricon (Liu et al. 2016).

The diversity of DL hardware and DL frameworks has witnessed the prosperity of DL community. However, it can be tedious for developers when it comes to actually deploying DL applications built upon different frameworks for various hardware, especially considering that DL models and operations need to be optimized for each hardware to get the optimal performance. Besides, deployment issues are even more difficult to solve compared to other aspects of a DL application (Chen et al. 2020). To migrate this problem and alleviate the burden of optimizing DL models for various hardware, several DL compilers have been proposed such as nGraph (Cyphers et al. 2018), TVM (Chen et al. 2018a), Tensor Comprehensions (Vasilache et al. 2018), XLA (Leary and Wang 2017), Glow (Rotem et al. 2018). Given a DL model by one of the DL frameworks, a DL compiler parses the model definitions and generates the optimized implementation for a target hardware. So DL compilers seem to be a promising solution, however, there is a fundamental question remain unclear: *What challenges do users face when using DL compilers and what are common challenges for developers when developing DL compilers.*

To answer this question, this paper presents the first empirical study on identifying the challenges in both usage and development of a DL compiler. Regarding the popularity of DL compilers, this study can help DL compiler users to avoid common pitfalls in using DL compilers and developers to be more clear about how to better help users in a more specific way. Several potential directions have been discussed for researchers in order to build more robust DL compilers. Building on these considerations, we select TVM as the representative DL compiler since TVM has a outstanding performance compared to other DL compilers (Li et al. 2021) and it has sufficient documents and discussions. We analyze relevant posts from the TVM Discuss Forum, which is the main communication channel for both TVM users and developers. We manually analyze 347 randomly sampled posts from

the discussion forum including both usage and development topics. Based on these posts, we focus on the following research questions (RQs).

RQ1: What are the challenges users may have when using TVM? To figure out what challenges user may face when using TVM, we randomly sample and analyze 279 posts on the usage of TVM. We finally build a taxonomy of challenges consisting of 15 categories.

RQ2: What are common topics that TVM developers discuss? To have a better understanding of the challenges and inspire future research or tool support, we carefully analyze 22 posts on the development of TVM. We find TVM developers generally have seven types of posts.

RQ3: What are the impacts of user- and self-reported TVM bugs? We take an initial step to understand the impacts of the TVM bugs by analyzing 44 bug reports identified from the discussion forum and 297 bug-relevant commits crawled from the official repository of TVM on GitHub. We finally summarized four types of impacts of TVM bugs.

To the best of our knowledge, this is the first paper to analyze challenges in using/developing DL compilers through mining the collective knowledge. Besides, We make all the materials we used in this study public. The crawled Apache TVM Community posts and the manual inspection results are made publicly available.[1] Researchers interested in conducting analysis on DL compilers may utilize this dataset. The rest of this paper is organized as follows. Section 2 provides background knowledge about DL frameworks, DL hardware and DL compilers. Section 3 describes methodology used to collect the posts, to build the taxonomy. Section 4 presents the taxonomy of challenges in using TVM along with the description of these categories. Section 5 describes the taxonomy of challenges and common topics about developing TVM. Section 6 describes the characterization of TVM bugs. Section 7 contains a discussion of our findings and describes several implications. Section 8 reviews our threats of validity. Section 9 discusses related work and Section 10 finally concludes this paper.

2 Background

In this section, we describe background knowledge about deep learning (DL) frameworks, DL hardware and DL compilers, especially TVM, i.e., the study subject of this work. All of these three are essential to developing DL software, i.e., DL frameworks provide training and executing DL models, DL hardware provide better hardware support to enable more efficient computation of DL models, and DL compilers support the deployment and optimization of DL models generated by DL frameworks on DL hardware.

2.1 Deep Learning Frameworks

Deep learning frameworks provides building blocks for designing, training and executing various DL models. In this section, we briefly introduce some popular DL frameworks to provide an overview of DL frameworks. As shown in Fig. 1, DL frameworks are

[1] Dataset:https://bit.ly/3I9xohu

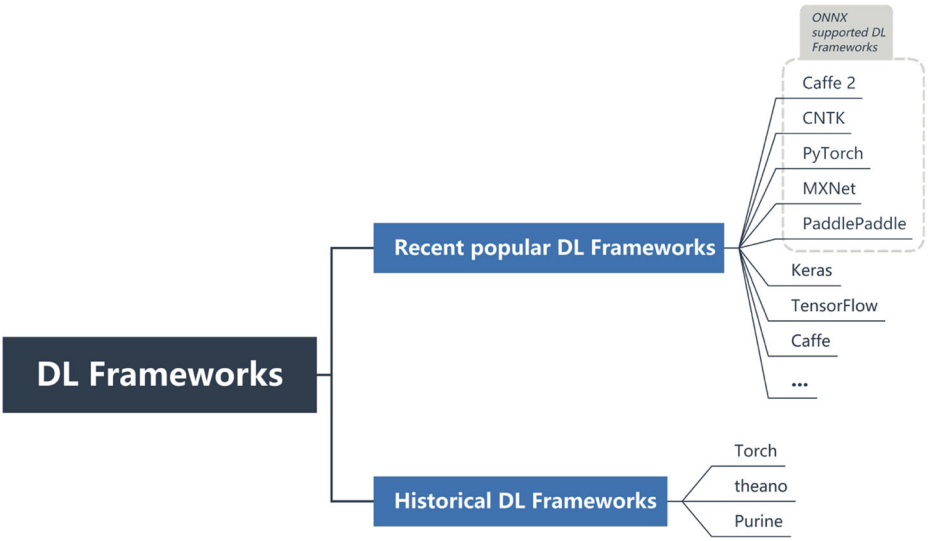

Fig. 1 Deep learning frameworks

divided into recent popular frameworks, ONNX supported DL frameworks and historical DL frameworks.

Recent Popular DL Frameworks Due to the prosperity of DL community, various DL frameworks have been proposed from both industry and academia. TensorFlow (Abadi et al. 2016) and PyTorch (Paszke et al. 2019) are two representative DL frameworks. TensorFlow is famous for its static computation graphs while PyTorch adopts dynamic computation graphs and defines a neural network on-the-fly (Zhang et al. 2019).

ONNX Supported DL Frameworks Open Neural Network Exchange (ONNX) is an open format for representing deep learning models, which allows developers to train DL model in one framework and then export and deploy the model into other frameworks for inference (ONNX 2020). ONNX provides a definition of an extensible computation graph model so that DL models from different DL frameworks can be transformed into ONNX format. Most of the current DL compilers support ONNX interchange format in frontend, allowing them to parse models from different DL frameworks. As shown in Fig. 1, ONNX is supported in frameworks such as Caffe 2, MXNet and so forth. Note that frameworks like Keras and TensorFlow can be converted into ONNX using the converter provided by ONNX. However, such conversion is not yet officially supported by the DL frameworks (e.g., Keras and TensorFlow).

2.2 Deep Learning Hardware

DL hardware, which can enhance the performance for DL models, has drawn attention from Internet giants to newly startups. As deep learning requires extensive matrix-based computations, it requires specialized hardware support. Generally, the DL hardware can be categorized into two types: 1) *general-purpose hardware*, which can support DL workloads through adding specially designed components and providing highly optimized libraries.

2) *DL-specific hardware*, which is specially customized to have better performance on DL workloads (LeCun 2019).

General-Purpose Hardware The current mainstream solution to accelerating DL workloads is to use Graphics Processing Unit (GPU). The massive parallelism of GPUs allows them to speed up computations that involve matrix-based operations, which are the heart of many DL implementations (Nguyen et al. 2019). CPU is an alternative to GPUs that can be used as general-purpose hardware due to its flexibility. Besides, manufacturers may offer accelerated libraries to further improve the performance of their products on DL workloads. For example, NVIDIA provides CUDA Deep Neural Network library (cuDNN) that includes highly optimized primitives for deep neural networks, which can leverage specialized hardware components of NVIDIA GPUs and thus improves the performance (NVIDIA 2020). Intel offers oneAPI Math Kernel Library (oneMKL) to increase application performance on Intel-based systems (Intel 2020). Except libraries from hardware manufacturers, Open Computing Language (OpenCL) is a platform that can provide heterogeneous parallel computing ability on cross-vender and cross-platform hardware (Tompson and Schlachter 2012).

DL-Specific Hardware DL-specific hardware is fully customized for DL workloads to further push the limit of performance and energy efficiency. Popular DL-specific hardware includes dedicated hardware based on Field Programmable Gate Array (FPGA) (Lacey et al. 2016) and Google Tensor Processing Unit (TPU) (Jouppi et al. 2017).

2.3 Deep Learning Compilers

DL compilers are proposed to alleviate the engineering efforts of developers when deploying or optimizing DL models on different hardware. Given a DL model by one of the DL frameworks, a DL compiler parses the model definitions and generates the optimized code implementation (i.e., deployable module) for a target DL hardware.

2.3.1 The Architecture of DL Compiler

In general, the compilation process of DL compilers that transforms a model definition to the highly optimized code implementation can be divided into four layers: 1) *frontend*, 2) *intermediate representation (IR)*, 3) *optimization*, 4) *backend*, as shown in Fig. 2.

Frontend The frontend takes a DL model from one DL framework as input and transforms it into a computation graph representation (high-level IR). Then computation graph optimization techniques will be applied to the computation graph. Finally, the optimized computation graph will be passed to the backend for further hardware-specific optimizations. TVM uses a frontend named Relay (Roesch et al. 2018), which supports to parse DL model from almost all the popular DL frameworks and could perform various hardware-independent optimizations.

IR There are two kinds of IR involved in the DL compilation process, namely the *high-level IR (graph IR)* and *low-level IR*. TVM uses Relay IR (Roesch et al. 2018) which is a functional IR that adopts both directed-acyclic graph (DAG)-based IR and let-binding-based IR as its high-level IR. The low-level IR of TVM is based on well-known IR called Halide

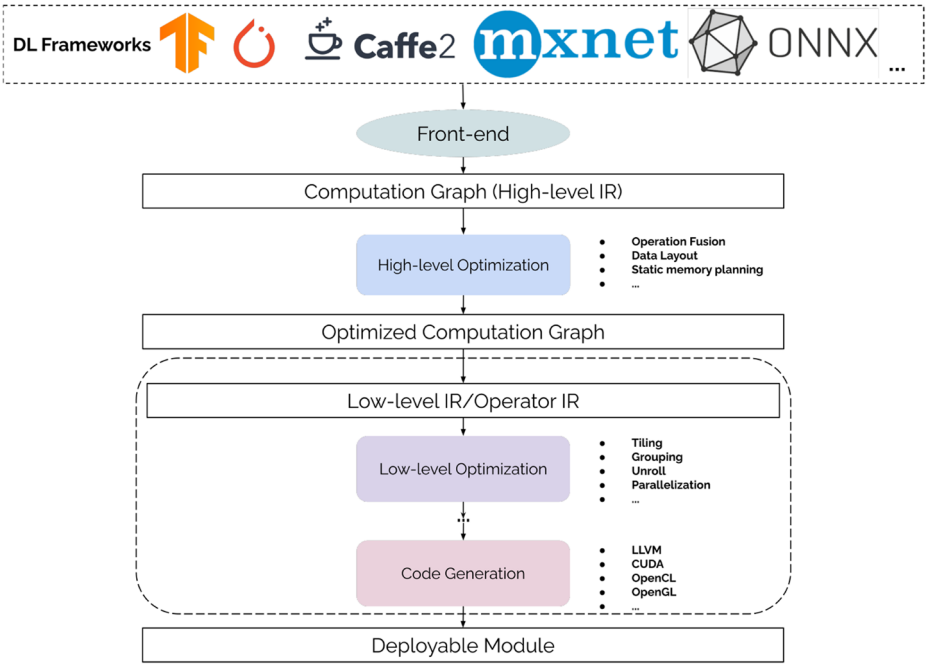

Fig. 2 Generic architecture of DL compiler

IR (Ragan-Kelley et al. 2013) and TVM has further improved Halide IR into an independent symbolic IR.

- High-level IR is a high-level abstraction of DL models, expressed as a computation graph and is hardware independent (Xing et al. 2019). High-level IR enables DL compilers to perform graph-level optimizations.
- Low-level IR resides in backend of the DL compiler and can represents the computation of the DL model in a more fine-grained view. It enables the DL compilers to utilize hardware-specific optimizations and optimized libraries regarding specific target platform.

Optimization Since optimization is associated with IR, there are also two kinds of optimization involved in the compilation process. For high-level optimization, TVM supports standard optimizations such as fusion and constant propagation. TVM also supports traditional hardware-specific optimizations such as hardware intrinsic mapping, memory allocations and fetching in backend for low-level optimization. Furthermore, TVM utilizes an auto-tuning optimization based on machine learning for further optimization.

- High-level optimizations are involved in the frontend of the DL compiler and are applied to the computation graph.
- Low-level optimizations are performed in backend of the DL compiler by using hardware-specific optimizations, auto-tuning methods and optimized libraries.

Backend The backend transforms the optimized computation graph into low-level IR, then performs optimization regarding the target hardware, and finally packs the generated code

into a deployable module (Chen et al. 2018a). TVM defines the compiled object as module, which can be deployed on the target device (TvmDeveloper 2020c).

Except the aforementioned four common components, different DL compilers may have their specific components in order to enhance its functionality. For example, TVM provides Versatile Tensor Accelerator (VTA), which is an open and customizable deep learning accelerator with TVM-based compiler stack. TVM regards it as an extension of the TVM framework in order to advance deep learning and hardware innovation (TVM 2020a; Moreau et al. 2018). Furthermore, TVM provides remote procedure call (RPC) that is useful for cross-compilation and can alleviate users from remote testing.

2.3.2 An Example of Using TVM to Deploy DL Models

The whole TVM stack can be divided into two components, namely the TVM compiler and TVM runtime. The TVM compiler is to perform all the compilation and optimizations while the TVM runtime runs on the target devices. Users do not need to build the whole TVM stack on target device, especially when target device only has limited computing resources. TVM allows users to cross-compile a DL model on a desktop or server and then deploy the compiled module on target device installed with TVM runtime that is very minimal (TvmDeveloper 2020b).

Figure 3 shows an example of how to use TVM to compile a pre-trained ResNet18 model from MXNet (Foundation 2020) and deploy the compiled runtime module on Raspberry Pi 3b+ that only has TVM runtime installed. The code snippet in the upper-left corner will download and compile the ResNet18 model (i.e., through `relay.build`).

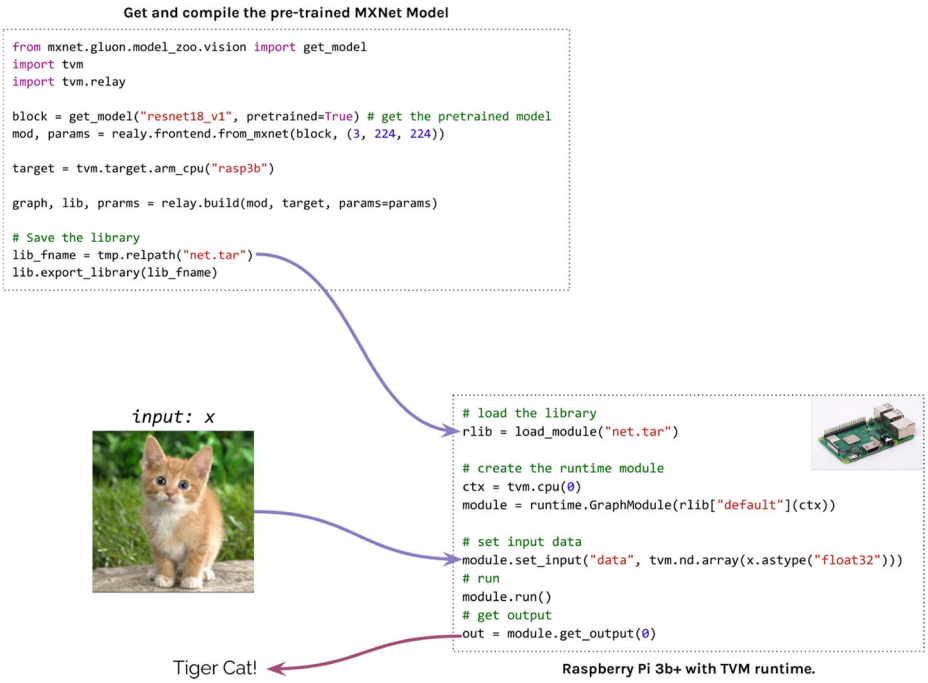

Fig. 3 An example of how a TVM user can use TVM to deploy the ResNet-18 model on Raspberry Pi 3b+

Note that the variable `target` that contains the target description and the codegen module is already simplified by TVM, i.e., TVM has built-in the target parameters for Raspberry Pi 3b+ (`tvm.target.arm_cpu("rasp3b")`). The complete form of this description is `tvm.target.Target("llvm -device=arm_cpu -model=bcm2837 -mtriple=armv7l-linux-gnueabihf -mattr=+neon")`. Users will need to manually specify these target parameters if their device parameters are not built-in with TVM.

The compiled module contains the optimized computation graph (`graph`), module containing necessary libraries (`lib`) and parameters of the final graph (`param`), and the library is saved to "*net.tar*" for further deploying (i.e., through `export_library`). When deploying the compiled module on Raspberry Pi, the first thing to do is to load the compiled module ("*net.tar*") and then to create a runtime module (`module`). Finally, the compiled module could be run on Raspberry Pi. For developers, there are various areas of TVM stack they could contribute to. For example, developers could add new operators or a compiler pass to relay, which is related to the frontend of TVM. As for the backend, developers can implement new backend for new hardware platform regarding their demand (e.g., Hexagon,[2] TI DSP[3]).

3 Methodology

To better understand the challenges in using/developing TVM, we analyze relevant questions and answers posted on Apache TVM Community,[4] which is the official discussion forum of TVM where users/developers seek for technical advices on unsolved issues. In this section, we provide descriptions on how we selected the study subject (i.e., TVM) and data source (i.e., the TVM discuss forum), how we collected the data for this study, and how we performed the study.

3.1 Data Collection

Among the well-known and widely used DL compilers, i.e., TVM, nGraph, Tensor Comprehensions (TC), Glow and XLA (Li et al. 2021), we focus our study on using and developing one DL compiler, i.e., TVM, for two main reasons. First, TVM supports more DL frameworks than other DL compilers with outstanding performance (Li et al. 2021). Second, we do not find sufficient data to study the use and development of these DL compilers except for TVM. XLA, which is a DL compiler supported by Google as part of TensorFlow, does not have a discussion forum. Both Glow and Tensor Comprehensions are developed by Facebook, while Glow is part of PyTorch and shares the discussion forum with PyTorch.[5] Tensor Comprehensions does not has a discussion forum and mostly uses a slack channel[6] for discussion and most of the messages are posted 2 years ago. nGraph is supported by Intel and it is part of the Intel OPENVINO project. nGraph uses the OPENVINO discussion forum[7] and there are only 78

[2]Introducing Hexagon backend: https://discuss.tvm.apache.org/t/introducing-hexagon-backend/2421
[3]Use TVM for TI DSP: https://discuss.tvm.apache.org/t/use-tvm-for-ti-dsp/1200
[4]Apache TVM Community: https://discuss.tvm.apache.org/
[5]Glow discussion forum: https://discuss.pytorch.org/c/glow/10
[6]Tensor Comprehensions slack channel: https://tensorcomprehensions.slack.com/
[7]OPENVINO forum: https://community.intel.com/t5/Intel-Distribution-of-OpenVINO/bd-p/distribution-openvino-toolkit

posts found to be related to nGraph after manual inspection. Our search of Stack Overflow for questions on any of the abovementioned DL compilers yields very few posts. Thus, we choose TVM to be the representative DL compiler in this study.

To study TVM, we identified a list of resources we could utilize for understanding the challenges in developing and using TVM, namely Stack Overflow, TVM slack channel, TVM discuss forum and TVM GitHub repository. Upon further investigation, we found that there are few discussions about TVM on Stack Overflow. As shown in Table 1, we used these search terms to cover all the questions related to TVM and the important components of TVM (e.g., relay, autotvm). The *"answers:1"* parameter is used to ensure that the returned results will have at least one answer. Then we manually inspected all the returned results and only found 11 questions related to TVM. This is not surprising, as 1) TVM is an emerging but relatively new topic (i.e., TVM released the first version in 25 October, 2017; 2) Answering TVM questions requires non-trivial expertise; and 3) the TVM community encourages discussions on the official forum.

In addition, for TVM slack channel, we find that developers are advised to only use Slack as a non-archival place for quick sync and the discussions should still happen in discussion forum or GitHub.[8] In the end, we decide to choose the Apache TVM Community and TVM GitHub repository as the only data sources of this study. Specifically, we utilize the Apache TVM Discuss Forum for studying the challenges TVM developers and users may face and TVM GitHub repository for studying TVM bugs. The Apache TVM Community (also referred to as the TVM Discuss Forum) is launched in April 2018 (around the same time that TVM releases version v0.4) and has been the main communication channel for TVM users and developers.

Step 1: Crawling the TVM discussion forum. We collected the TVM dataset by crawling the official Apache TVM Community (i.e., the TVM Discuss Forum) on November 23, 2020. We collected a dataset of 3,727 posts from 4 April, 2018 to 23 November, 2020. The crawled data of each post contains all the metadata of the post, including the title of the post, all the replies, the link to the post, number of replies, number of views and the time of the latest activity.

Step 2: Identifying relevant topics. We performed an initial screening on the collected posts as some of the topics that the posts cover are not of interests to this study. In particular, we leverage the official categories associated with each post to identify relevant topics. Table 2 shows the 10 official categories for classifying topics (second column) and the number of posts under each category in the collected dataset. The official categories are recommended by the TVM discussion forum and are manually identified by the forum users when posting the questions, i.e., one post can only have one associated category.

However, after examining 50 randomly sampled posts, we found that the users loosely follow the intention of the official categories, i.e., the standard of applying the categories is inconsistent across the posts, especially on usage-related categories such as troubleshooting. However, we noticed that one official category *development* is consistently used by the posts that TVM developers publish for communicating on TVM development issues. Hence, to facilitate the subsequent stratified sampling process, we merge the five usage-related (i.e., non-development) official categories and produce two high-level categories, namely, the categories *Usage* and *Development*. We manually examined

[8] https://discuss.tvm.apache.org/t/request-for-invite-to-the-slack-channel/2888/17

Table 1 Questions about TVM on stack overflow

Search Term	# Returned results	# Results related to TVM
"tvm answers:1"	190	11
"relay tvm answers:1"	0	0
"autotvm answers:1"	0	0
"tir answers:1"	136	0

all the posts under the categories *Announcement*, *Meetup*, *d2l*, *uTVM*, and *Site Feedback* and concluded that these posts are indeed irrelevant of usage and development of TVM. Hence, we excluded these categories from the remaining of the study.

Step 3: Crawling bug-fixing pull requests (PRs). We collected bug-fixing PRs of the official repository of TVM from July 30, 2019 to November 23, 2020 on GitHub using the GitHub search API (git 2021). Upon examining the sampled posts from TVM discuss forum, we notice that there is only a small portion of posts about TVM bugs, which leads to the small number of TVM bug posts for study in the sample. Hence we include bug-fixing PRs to enhance the dataset of TVM bugs. We followed the previous work (Garcia et al. 2020) to collect the bug-fixing PRs that contain at least one bug-related keyword (i.e., fix, defect, error, bug, issue, mistake, incorrect, fault and flaw). Then the first two authors manually inspected and classified these bug-relevant PRs independently. As a result, 297 TVM bugs are identified.

3.2 Manual Investigation

To categorize the challenges of using and developing TVM, we follow the open coding method in Berg et al. (2004).

3.2.1 Construction of Taxonomy of Challenges in Using/Developing TVM

Step 1: Initial category distillation. In this step, the first two authors jointly inspected 50 randomly sampled posts from the crawled dataset and constructed an initial set of categories. The detailed procedure is described as follows.

Table 2 Statistics of the TVM Discuss Forum

Topics	Official TVM Category	# Posts
Usage	Questions	1,884
	Uncategorized	721
	Troubleshooting	537
	Application	109
Development	Development	355
Excluded	Announcement	103
	Meetup	7
	d2l	5
	uTVM	4
	Site Feedback	2

The first two authors thoroughly read all the posts to get familiar with them. All elements in the posts including title, main body, replies, code snippets will be carefully examined. URLs mentioned in the posts will also be tracked in order to get a precise understanding of the question.

Once the two coders have been familiar with the samples, they start assigning short phrases as initial tags to describe the challenges behind these posts. To determine the category of a post, we follow the method adopted in Chen et al. (2020). Specifically, for those posts raised without deep investigation (usually in the form of "how", e.g., "How to schedule fused ops?") or detailed information, the two coders often can summarize the challenges based on the post descriptions; for those posts with detailed descriptions of faults or unexpected results, the coders identify the challenges based on their causes. For instance, if a developer files a post and seeks for help on an error he/she encountered when exporting DL models and the coders can identify that the cause is the incorrect build configuration of TVM from the descriptions, comments and replies from other users, the coders consider build configuration as the challenge behind this post. Then the two coders will start clustering similar tags into categories and create a hierarchical taxonomy of challenges. If there are conflicts between the proposed categories, an arbitrator will be involved into the discussion and the conflicts are marked as resolved when all the participants have reached a consensus. Finally, an initial set of categories are distilled.

Step 2: Independent labeling and constructing extended categorization. Upon constructing the initial set of categories, the two coders continued to analyze a statistically sample of posts independently. We adopted the stratified sampling strategy to randomly sample a total of 347 posts to ensure a 95% confidence level and 5% confidence interval in the population of 3606 posts (i.e., the categories of usage or development of TVM). The sample contains 312 posts under the category of *usage* and 35 posts under the category of *development*.

For questions not related to usage/development of TVM, we mark these posts as *False Positives* and are excluded from the dataset in this study. *False Positives* are posts that are not related to both *Usage* and *Development* of TVM. As shown in Table 3, upon manual inspection, we excluded 46 posts (two false positives and 44 bug reports) that do not belong to neither *development* nor *usage* and corrected the corresponding category of some posts: there are four posts about *development*, however marked with *usage* in the sample posts, one post that is irrelevant to both usage and development of TVM, and 11 posts marked as *development* are actually about *usage*. The posts with incorrect categories are corrected by the authors, and are utilized to conduct RQ 1 or RQ 2 according to its corrected category. Note that the 2 *False Positives* are excluded from all the RQs in this paper. Hence we end up with 279 posts in the *usage* category and 22 posts in the *development* category, which were used to conduct RQ 1–2. 44 bug reports were excluded from RQ 1–2 and were used to conduct RQ 3.

During the labeling process, the two coders evolve the initial categories into the final taxonomy in an iterative manner, in which the two coders continuously look at the existing categories and the post being inspected to refine the taxonomy. There are two kinds of changes that may be applied to the initial categories: 1) if any coder can not fit a post into one of the initial categories, this post will be jointly inspected by the aforementioned two coders with and arbitrator to determine whether a new category should be added; 2) if any coder find a category is not representative, all the authors will meet up and discuss about revising the corresponding category. If agreement has been reached to change a category, the

Table 3 Distribution of posts before/after manual inspection

Topics (# posts)	Corrections	# Final posts
Usage (312 posts)	Marked as Development (−4)	279
	Marked as bug reports (−39)	
	False positive (−1)	
	Usage posts identified from Development (+11)	
Development (35 posts)	Marked as Usage (−11)	22
	Marked as bug reports (−5)	
	False positive (−1)	
	Development posts identified from Usage (+4)	

corresponding category will be modified and all the posts in this category will be inspected and labeled again to avoid misclassification.

Furthermore, the two coders kept a note of the resolution status of the examined posts. There are multiple ways that TVM users can mark the resolution status of the posts: 1. Users may add *"SOLVED"* to the title after getting a correct answer; 2. There could an explicit reply of the post such as *"Thanks, it really works."* even if the user does not add *"SOLVED"* to the title. The two coders took consideration of the aforementioned cases when marking the resolution status of the posts.

The questions on using and developing TVM are actively viewed and discussed by TVM community. Figures 4 and 5 show the boxplots of the number of view counts and replies for the posts under the *Usage* (279 posts) and *Development* (22 posts) respectively. As the figures show, the posts under both of the categories have been receiving active discussions and attention, i.e., the median view count is 269 for *Usage* posts and 244 for *Development* and the median of number of replies received is 4 and 4.5 respectively.

In summary, 347 posts are inspected during the manual inspection, and 39.7% of the inspected posts have an accepted answer (either marked as resolved or with an explicit acknowledge in the replies). The inter-rater measurement of independent labeling results is

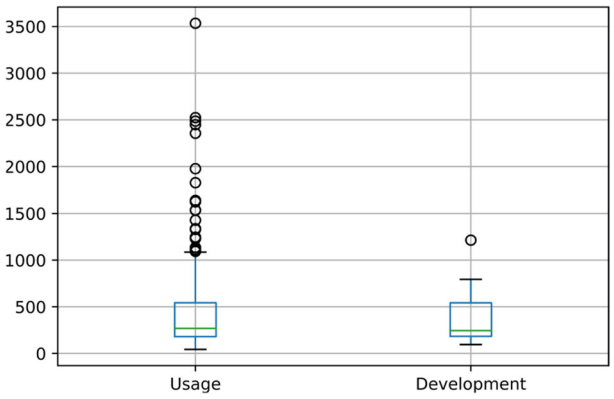

Fig. 4 Boxplot of the view counts of Usage and Development

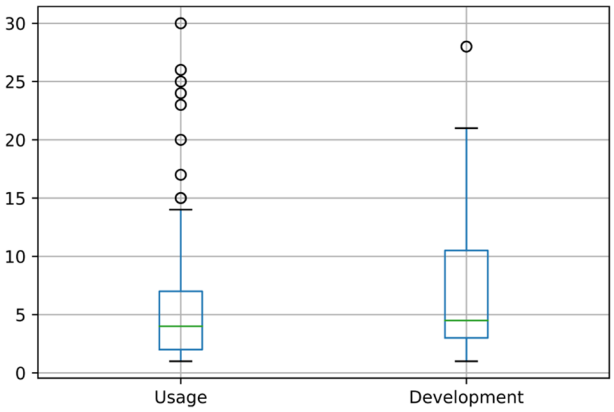

Fig. 5 Boxplot of the number of replies of Usage and Development

0.72 using Cohen's Kappa (Cohen 1960), which implies substantial agreement and demonstrates the reliability of our coding procedure. The manual inspection procedure takes about 400 man-hours. The final taxonomy is shown in Fig. 6

4 RQ1: What are the Challenges Users May Have When Using TVM?

Motivation As discovered by previous studies (Zhang et al. 2019; Chen et al. 2020), developing and deploying machine-learning backed software pose unique challenges for data scientists and software engineers. The extensive use of frameworks (e.g., DL frameworks, DL inference engines) makes the development convenient, fast-evolving, but also introduces overhead to practitioners. Deep learning compiler represents an emerging line of techniques that aims to provide end-to-end model optimization and deployment. It is important and timely to study the challenges and problems that the users of DL compilers (i.e., developers of machine-learning software) may commonly encounter. Our findings will identify future research ideas and tool support to better facilitate the adoption of DL compilers in the development of machine-learning software.

Method We followed the steps described in Section 3.2 and identified a total of 279 posts (including both the questions and all the replies) on the usage of TVM. Note that we do find some posts (39 under the category of *Usage*) by TVM users appear to be about troubleshooting in the beginning, but later turn out to be caused by bugs in the current TVM implementation. Such bug reports require further analysis and are beyond the scope of using TVM. Hence we classify 39 bug reports into an individual category (i.e., not included in the category of *usage*) and discuss the bug reports in RQ3. We categorized the posts based on the challenges and problems described.

Results We describe the categories derived for the posts on the usage of TVM, present the distribution and discuss the main challenges we identify. At the high level, we identify three categories: troubleshooting (150/279, 53.8%), general questions (119/279, 42.6%), and feature request (10/279, 3.6%). Figure 7 shows the distribution of the three high-level categories and the distributions of the sub-categories under each of the high-level category.

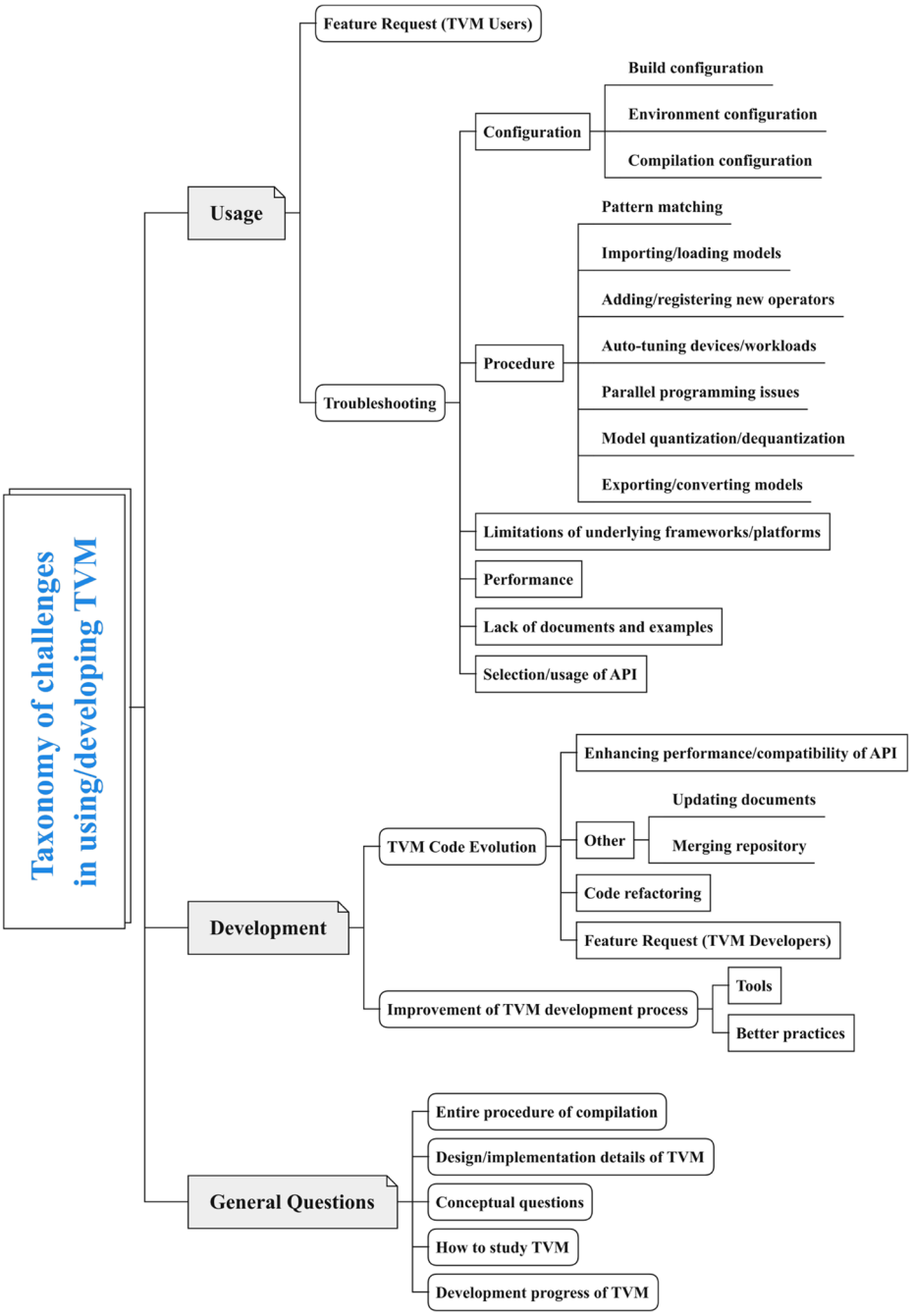

Fig. 6 The Final Taxonomy of the TVM challenges

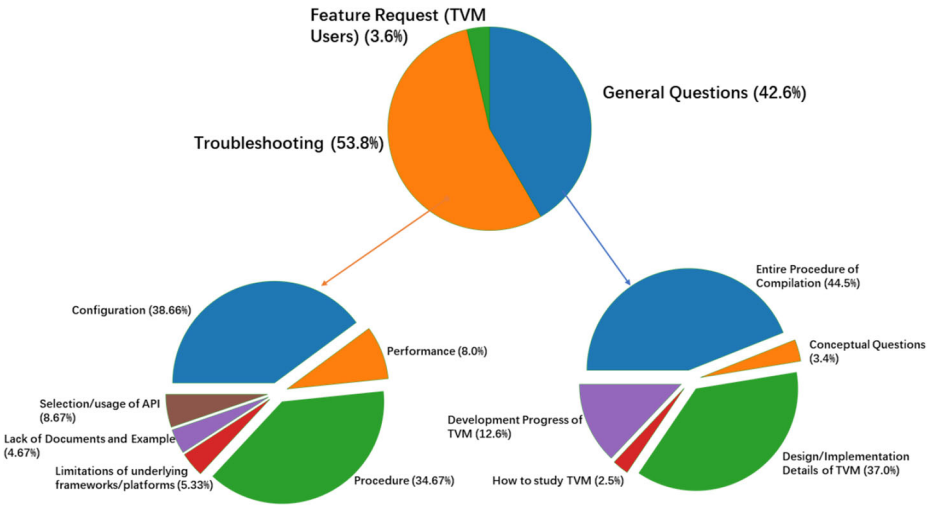

Fig. 7 Pie chart of the distribution of categories of usage

Category 1: Troubleshooting Troubleshooting covers the largest number of posts on the usage of TVM. TVM users encounter errors and issues (e.g., compilation errors, runtime errors) when using TVM and thus seeking for help in the discussion forum. We further derive the following six sub-categories, namely configuration, procedure, performance, selection/usage of API, limitation of frameworks/platforms, and lack of documentation/examples.

Configuration (58/279, 20.8%). Improper configurations of TVM may cause severe reliability issues in the client code (i.e., machine learning software), such as crashes and low performance. There are three types of configurations that TVM users need to set when using TVM.

- *Build configuration* (24/279, 8.6%).This category is about the challenges about building TVM from source and issues caused by building TVM incorrectly. To use TVM, developers are required to build TVM from source. To build TVM, developers need to edit configuration files (i.e., `config.cmake`) in order to enable external libraries (e.g., BLAS, cuBLAS) or backends (e.g., LLVM, CUDA). Wrong configuration can result in build failure or application crash. For example, users may forget to build TVM with LLVM enabled. This may not have any effect during the building process, but may raise errors when using TVM to export the deployable module. However, the official guide of building TVM from source[9] only says that it is recommended to build TVM with LLVM to enable all the features without further explanation.
- *Environment configuration* (27/279, 9.7%). Using TVM requires to correctly setting up the environment which includes configuring complex software and hardware dependencies. Similar to the findings by Zhang et al. (2019) regarding using DL frameworks, we notice configuring environment for TVM is prone to various types of challenges such as version incompatibilities issues. For example, when running remote procedure call (RPC) tutorial on remote device (i.e., Jetson TX2), one of the configurations a TVM

[9]Install from Source: https://tvm.apache.org/docs/install/from_source.html

user need to configure is the version of CUDA on both the host and target devices. Due to a misconfiguration, a TVM user experienced runtime errors that are difficult to resolve (i.e., costing 12 days to resolve).[10] The misconfiguration is a simple CUDA version mismatch between the host and target device. To prevent similar incorrect configurations in the future, TVM developers write a complete RPC deployment tutorial on how to configure common boards and targets.

– *Compilation configuration* (7/279, 2.5%). A deep learning compiler such as TVM provides highly configurable compilation process. In fact, among the steps in the complicated compilation process (as shown in Fig. 2), TVM users can opt to configure many of the steps, i.e., quantization size, shape of the GEMM tensor, optimization level. Incorrect or conflicted compilation options could result in crash, compilation failure, undesired behaviors and low-grade performance. For example, if the batch size is set too large or the optimization level is too high ($opt\_level \geq 2$), TVM may cost too much device memory and raise `CUDA_ERROR_LAUNCH_OUT_OF_THE_RESOURCE` error (TvmUser 2019b).

We also find that TVM users complain about lack of sufficient examples or tutorials, which makes the configuration problem even harder to solve than it is already.

Procedure (52/279, 18.6%). We find that users often ask for help about how to perform a very specific task typically when users have difficulties debugging the errors caused by their code. Despite the efforts of providing learning resources of TVM by TVM developers (e.g., tutorials (TVM 2020d), language reference (TVM 2020c), and a book under construction (TVM 2020b)), it indicates a gap between the available learning resources of TVM and TVM users' needs.

This type of questions is different from more general how-to questions (later explained in "Category 2—general questions"), e.g., "Entire procedure of compilation", and shows that the users have certain level of knowledge of using TVM. Due to TVM's highly diverse functionalities, there are seven sub-categories under *Procedure*.

– *Adding/registering new operators/targets* (5/279, 1.8%). Due to the quick involving DL community, the operators/targets provided by TVM are insufficient for developers. Developers may need to implement the operators (targets) and adding the customized operator to run-time library (or a certain backend), which can be challenging for those who are unfamiliar with TVM.
– *Auto-tuning devices/workloads* (16/279, 5.7%). TVM provides auto-tuning for developers to get the best performance for a specific device (workload), i.e., ARM CPU, x86 CPU. However, the auto-tuning process is highly sophisticated and may consist of many steps, i.e., install additional dependencies, define workload, configure tuning setting and create tasks. Using AutoTVM (the auto-tuning module of TVM) requires developers to write tuning templates regarding their workloads (devices). Incorrectly written templates or configurations can result in tuning failure (TvmUser 2019c) or performance degradation (TvmUser 2020a).
– *Parallel programming issues* (2/279, 0.7%). Developers may have difficulty dealing with parallel programming, i.e., synchronization, thread scope. For example, a TVM developer sought for help in order to do global synchronization in IR builder on GPU (TvmUser 2018c).

[10] https://discuss.tvm.apache.org/t/cant-run-rpc-gpu-tutorial-on-my-own-device/564/9

- *Model quantization/dequantization* (3/279, 1.1%). TVM utilizes quantization to enable high-performance inference on edge devices (TvmDeveloper 2018b) and reduce power and compute requirements. For this technique, developers have difficulty in quantizing a model which has a operator that is not quantized in TVM (TvmUser 2019f) or dequantizing the weights back when needed (TvmUser 2020h).
- *Pattern matching* (5/279, 1.8%). TVM developers often need to identify pure data-flow sub-graphs of the Relay (frontend of TVM) program and transform these into example passes, i.e., fusion, external code generation and device specific optimizations, which requires a lots of tedious boilerplate code (TvmDocs 2020). To alleviate users, TVM provides a pattern language and APIs to enable pattern matching and pattern processing. TVM users often have troubles writing the correct pattern to match a specific operator (TvmUser, 2020b g) or finding existing patterns regarding their purposes (TvmUser 2018f).
- *Exporting/converting models* (13/279, 4.7%). These posts cover the challenges in exporting/converting models into the formats for a specific target platform to deploy the models. Similar to the findings by Chen et al. (2020), we notice exporting/converting models using TVM is prone to various types of challenges such as confusion about a specific step in the exporting (converting) process. One example is *"[SOLVED] How to export model library to so file instead of tar for armv7 on x86 box"* (TvmUser 2018g). While how to export a model library as tar file is known to the user, the user had issues debugging the compilation failures and therefore sought for help. As TVM offers highly configurable functionalities and much flexibility, learning resources (i.e., tutorials, documentation) may not cover every aspect of such flexibility although the code snippet that demonstrates a similar task may be covered in the tutorials.
- *Importing/loading models* (8/279, 1.8%). To deploy compiled DL models, developers also need to tackle with importing and loading models. For example, a user had issues loading the exported parameters in big endian system (TvmUser 2018d). In addition, users may also suffer from calling unsupported operator issues during the importing/loading process (TvmUser 2019d).

Limitations of the underlying frameworks/platforms (8/279, 2.9%). TVM is expected to support various hardware, DL frameworks with respect to different platforms (e.g., Linux, Windows). However, due to the inherent differences of such variations, TVM users often have troubles finding the solutions to resolve the problems, e.g., one TVM code works fine at one platform but not at another. For example, a TVM user sought for help regarding the difficulties to run tutorial code in Windows (TvmUser 2018a) due to the failure of `autoTVM.LocalRunner` of TVM. TVM developers suggested a workaround solution. However the workaround does not provide a seamless solution, i.e., it requires extra steps to set up in Windows compared to Linux. Another type of limitations is caused by defects, either in the underlying frameworks/libraries or caused by the incompatibility issues between the TVM version and the frameworks/libraries. For example, a TVM user was suffering from stuck during tuning problem and the follow-up investigation shows the problem is caused by an incompatible version of XGBoost (i.e., a widely-used gradient boosting library, which used as the cost model when auto-tuning with TVM) suggested the user downgrade to XGBoost 0.9.0 in order to avoid unexpected errors. (TvmUser 2020e).

Performance (12/279, 4.3%). TVM users have performance concerns about the time spent on auto-tuning, resource usage of TVM and runtime performance of the compiled model by TVM. Many posts complain why the performance of the compiled model is slower than the original one. For example, a TVM user sought for help about how to limit the CPU

usage of the TVM when performing model inference. The developer suggested the user to directly change the number of CPU threads by using a `config_threadpool` function, which was not documented in the tutorials of TVM (TvmUser 2019e). Other performance issues contain: 1) resource usage of TVM, which is usually about the usage of GPU memory and CPU usage. Some of these issues are caused by the external dependencies (e.g. TensorFlow).[11] 2) The performance of the compiled model is slower than the original one. For example, A TVM user complained about the inference time of the compiled model is very slow. Further exploration indicated that the selection of API is the root cause for this issue (TvmUser 2019a).

Lack of documents and examples (7/279, 2.5%). TVM users are sometimes clueless about how to achieve a specific step when using TVM. Different from the *Procedure* category where TVM users may have some clues (i.e., indicated by the code provided in the question), TVM users may have no ideas about how to perform certain tasks due to the lack of documents and tutorials/examples.

Selection/usage of API (13/279, 4.7%). TVM provides a large number of APIs (more than 630 C++ APIs[12]) as it needs to take models of different frameworks and output an executable target for various hardware/platform. We find that users may be confused about which APIs to use to fulfill their demands. For example, a TVM developer was confused about the `relay.build_module.build()` and `relay.build_module.create_executor()` API. These two APIs can both be used to generate code and were used in two separate tutorials. The results turned out that there was some subtle difference between these two APIs (TvmUser 2019i). Note that the user is actually the lead maintainer of XGBoost and has contributed to solve a TVM issue.[13] However, we found other posts in which a user complained about the performance regression after getting the DL model compiled. That user finally found that there was significant performance difference between these two APIs (TvmUser 2019a).

Category 2: General Questions This category covers relatively high-level questions that are not about a specific step in using TVM. We conclude the following three sub-categories.

Entire procedure of compilation (53/279, 19.0%). This category refers to general questions about the whole procedure of compiling DL models using TVM, usually raised without any in-depth investigation.

Although TVM provides tutorials that may cover similar tasks that TVM users ask, the users' use cases are usually more complicated than what the tutorials provide. Furthermore, some of the tutorial may lack of background knowledge and thus hard to comprehend by TVM users. For example, a TVM user wanted to know how to populate a tensor after reading the C++ deployment example.[14] It seems like that the tutorials and examples are designed to give users a "feeling" and make them understand the basic concepts, but this will become a problem when users actually start using/modifying TVM since some background knowledge is lost and the example is too simple.

Design/Implementation Details of TVM (44/279, 15.8%). As the TVM community continues to draw attention, more and more developers are joining the TVM devel-

[11] https://discuss.tvm.apache.org/t/solved-qeustions-about-gpu-memory-usage/5309

[12] C++ doyxgen API: https://tvm.apache.org/docs/api/doxygen/index.html

[13] https://bit.ly/3nbNNH5

[14] https://discuss.tvm.apache.org/t/how-should-you-index-dltensors-in-c-for-the-multi-dimensional-case/6257

opment. This category represents a strong desire of TVM users/developers to have an in-depth understanding of TVM. Questions are mainly about the implementation details or design philosophy of TVM and may vary from naive ones like *"Where is DLDataType defined?"* (TvmUser 2020i) to very profound questions like *"TensorArray GlobalVar and GlobalTypeVar Confusion"* (TvmUser 2019h) that needs several proficient TVM developers to work out a solution.

Conceptual Questions (4/279, 1.4%). Questions in this category are raised to understand fundamental concepts or background knowledge about DL compilers, such as *"What's the Model bias in TVM paper"* (TvmUser 2019j). This category of questions are also spotted in previous studies developing machine learning software (Bagherzadeh and Khatchadourian 2019; Chen et al. 2020).

How to study TVM. (3/279, 1.1%). TVM users ask questions for the purpose of learning TVM better, i.e., steps to follow for a better understanding. For example, "What should I do to understand tvm source code?", "How can I understand IR?" and "Any material of Relay for beginners?". These questions are not related to any specific step in using TVM and it's not as specific as the posts in "design/implementation details".

Development Progress of TVM. (15/279, 5.4%). TVM users may ask about the recent development progress of TVM, especially if they are waiting for new features to be released or bugs fixed. For example, a TVM user may file a post to ask whether dynamic shaped tensors has been supported in TVM or are there anyone is working on this feature.[15] These posts are different from the category *Lack of Doc/Example*, because these users only want to check whether TVM support the functionality they want yet.

Category 3: Feature Request (TVM Users) (10/279, 3.6%) We find that TVM users may request for new features, e.g., support for new operations or specific hardware. TVM developers may follow up with the feature requests, e.g., some of the feature requests are later implemented and some may be divided into several follow-up posts (i.e., multiple feature requests) in the *development* category. For example, the post with largest view count (3534) is in this category, namely the "INT8 quantization proposal" (TvmUser 2018e). This post is further divided into two posts in *Development* category.

Discussions Among all the usage categories, *Configuration*, *Procedure* and *Entire procedure of compilation* are the top three most frequently asked questions. Except *How to study TVM*, *Feature Request* questions seems to be most difficult questions to solve, 10% of the *Feature Request* questions have been resolved, compared to 41.35% of all other questions. *Limitations of underlying Frameworks/Platforms* requires longest time to get an answer. Figures 8 and 9 show the boxplot of the number of view counts and replies for inner categories under the *Usage*. As the figure show, the median of number of replies of these categories is near and the median view count is also close except *How to study TVM* and *Lack of Documents and Examples*, which is 691 and 749.5 respectively. However, the boxplot of the response time in Fig. 10 shows that the median of the response time varies broadly (from 314 to 10814).

[15] https://discuss.tvm.apache.org/t/can-tvm-now-support-dynamic-shaped-tensors/5094

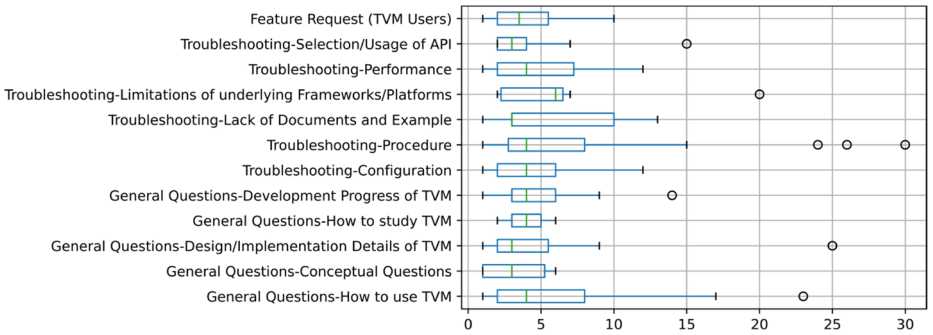

Fig. 8 Boxplot of the replies of Usage Categories

5 RQ2: What are the Common Topics that TVM Developers Discuss?

Motivation As deep learning compilers draw more and more attention, the TVM community has grown significantly since 2018, i.e., the contributors of TVM has experienced a 70% growth to 295 contributors from both academia and industry (University of Washington, UCB, Cornell, Amazon, Huawei, etc.) according the 2019 TVM conference keynote (TVMConf 2019). In fact, the current number of contributors of TVM on GitHub is 466.[16] In the process of onboarding newcomers and facilitating the fast development of new features of TVM, TVM developers are making great use of the TVM discussion forum for an open communication. The development and quality of TVM is utmost important in the current development ecosystem of machine learning software. Analyzing such communication posts closely allows us to have a better understanding of the challenges or recurring topics during discussion to inspire future research or tool support can help.

Method Similar to RQ1, we followed the steps described in Section 3.1 and examined a total of 22 posts on the development of TVM. Compared to GitHub issues and pull requests, the TVM discussion forum contains more detailed information. In fact, we notice that some PRs will even not be merged before the RFC (short for request for comment) on discussion forum has been discussed.[17] Note that when we manually examined the *development* posts, we also checked the referred pull requests and GitHub issues for a better understanding if they are referred to in the replies.

Results We categorize two main categories for the posts under *Development*, namely TVM code evolution, development process improvement. Below we describe the findings of each category in detail. Figure 11 shows the distribution of the two high-level categories and the distributions of the sub-categories under each of the high-level category.

Category 1: TVM Code Evolution This is a large category that includes the posts that are related to changes TVM developers made on TVM code, and it is further divided into 4 sub-categories.

[16] https://github.com/apache/tvm/graphs/contributors
[17] https://github.com/apache/tvm/pull/2116#issuecomment-444726352

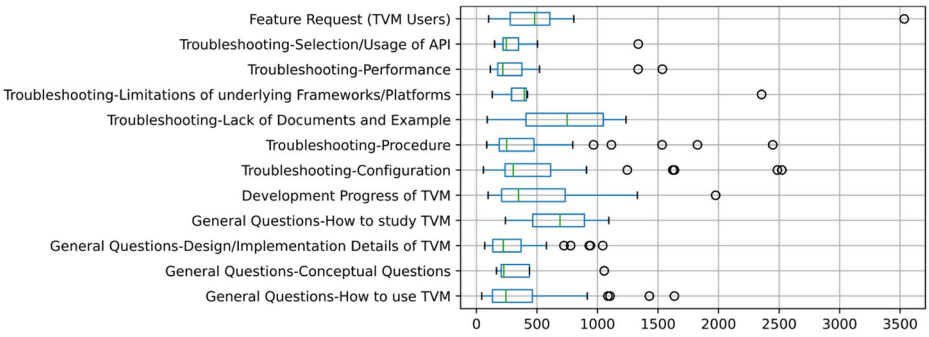

Fig. 9 Boxplot of the view counts of Usage Categories

Enhancing Performance/compatibility of API (12/22, 54.5%). We find that TVM Developers often seek for advice or discussion before they may need to change the implementation or design of an API in TVM. The motivation of such changes could be unsatisfactory performance or improving compatibility. For example, a TVM developer wants to discuss how to improve quantization accuracy in the process of developing a feature namely calibration upon observing a larger accuracy loss for one model (TVMDeveloper 2019). In the replies, fellow developers shared their relevant experiences and suggested adding extra models for evaluation.

Feature Request (TVM Developers) (3/22, 13.6%). Similar to TVM users, TVM developers may also request for new features in the TVM discussion forum. Compared to the ones proposed by TVM users (in *Usage* category in RQ1), we find that developers tend to provide more details when requesting new features. For example, when requesting to introduce a formal `DataLayout` into the node system of TVM (TvmDeveloper 2018a), a developer clearly documented the abstract interface of the layout and provided two options for concrete data types while TVM users often only express the high-level idea and ask whether there are any interest on their proposal.

Code Refactoring (1/22, 4.6%). TVM developers constantly improve the quality and readability of TVM implementation and use the posts to seek for comments and discussion. For example, developers often file a post to ask other developers to determine the naming conventions in TVM code (TvmDeveloper 2020a).

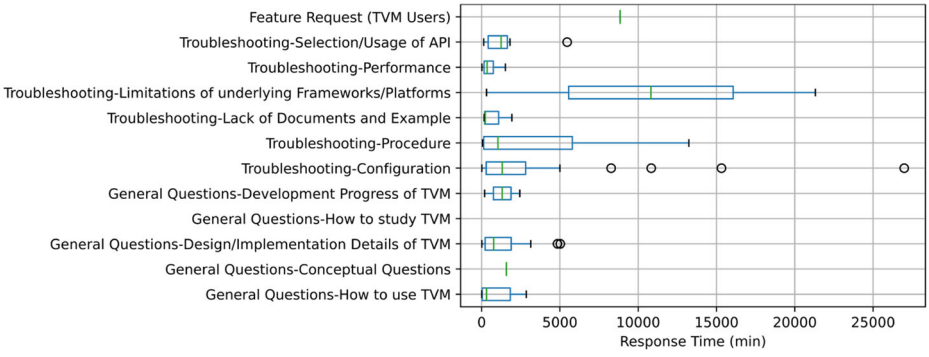

Fig. 10 Boxplot of the resolution time of Usage Categories

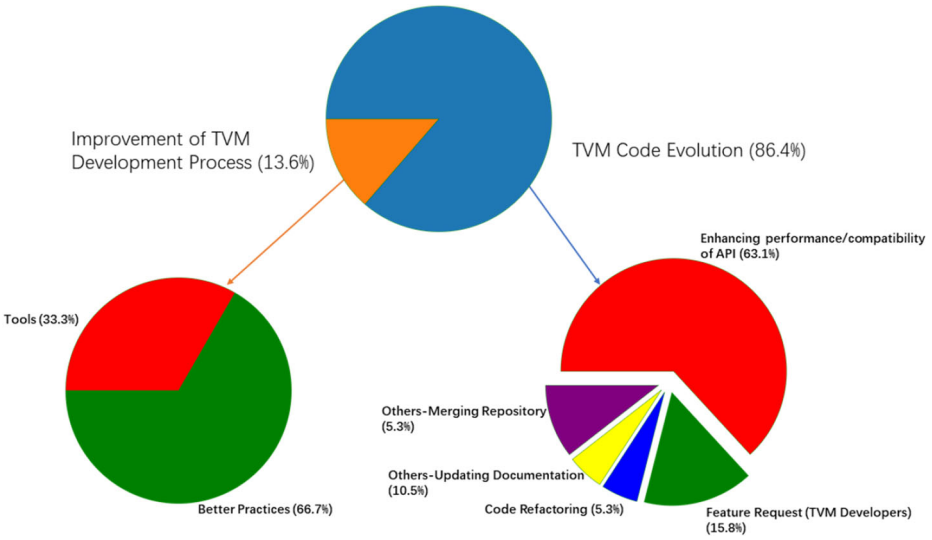

Fig. 11 Pie chart of the common topics discussed by TVM developers

Others (3/22, 13.6%). There are some other posts about maintaining and improving the TVM code repository.

- *Merging Repository.* (1/22, 4.6%). These posts are about merging other repositories into the TVM codebase. For example, a TVM developer suggested to incorporate an existing Ahead-of-Time (AoT) compiler into the TVM codebase since the TVM community was targeting to bring an AoT compiler to TVM. Another developer provided some comments to this topic and advised that the existing Relay AoT is different from the solution they discussed before and it will need extra efforts if they decided to incorporating the current implementation.[18]
- *Updating Documentation.* (2/22, 9%) The posts are about updating the documentation of TVM. For example, a TVM developer noticed that there was little documentation for the `InferBound` pass, and thus wrote a tutorial for it.[19] The documentation was finally merged into the TVM documentation after discussing with other developers.

Category 2: Improvement of TVM Development Process To improve the development efficiency and the quality of TVM development in general, TVM developers constantly discuss and adopt better software engineering practices. This category of questions differs from "*Category 1: TVM Code Evolution*" as it does not related to TVM implementation. We find that the effort to improve TVM development process comes from two aspects: tools and practices.

Tools. (1/22, 4.6%) TVM developers propose new development tools (e.g., as IDE plugins) to help the development of TVM to accommodate its peculiar characteristics. For example, a developer proposed to develop a language server tool that can better navigating across different programming languages (Chen 2020).

[18] https://discuss.tvm.apache.org/t/rfc-incorporate-existing-relay-aot-compiler-in-mainline/7393
[19] https://discuss.tvm.apache.org/t/discuss-contributing-new-docs-for-inferbound/2151

Better Practices. (2/22, 9%) This category includes the posts that are raised to discuss how to improve the development process in general, e.g., how developers can collaborate in a more efficient way. For example, a TVM developer proposed to have a better formatted pull requests for bug fixes (TvmUser 2020d) , e.g., adding tags *[Bugfix]* in the pull request.

Discussion Based on our observations, developers mainly use TVM forum as a place to discuss about development of TVM. Among the 355 posts of the category *development*, 60.2% of the posts are tagged as RFC (i.e., "[RFC]" in the title). Developers often file a post before diving into implementation details for hearing from other developers' opinion on their proposal. Besides, due to the complexity of TVM, TVM developers want to do their best to avoid affecting other parts of TVM implementations. Furthermore, according to TVM contribution guide, when major changes are proposed, an RFC should be sent to allow discussion by the community.[20] Discussion forum is one of the advised choices to open an RFC. This can also explain why 60.2% of the development posts are tagged as RFC. Figures 12 and 13 show the boxplot of the number of view counts and replies of for posts under *Improvement of Development Process* and *TVM Code Evolution*. As the figure show, the posts in *Improvement of Development Process* are more active than *TVM Code Evolution*, i.e., the median view counts is 304 for *Improvement of Development Process* and 255 for *TVM Code Evolution*, and the median number of replies received is 9 and 4 respectively. As Fig. 14 shows, posts in *Improvement of Development Process* need longer time than those in *TVM Code Evolution*, the median of response time is 4170 and 3768 minutes respectively.

6 RQ3: What are the Impacts of Self- or User-Reported TVM Bugs?

Motivation The quality of DL compilers has a significant impacts on the correctness and efficiency of the deployable modules (i.e., deployed DL models). In addition, developing and testing DL compilers have unique challenges compared to traditional software (e.g., language compilers). Hence, in this RQ, we set off to obtain an initial understanding on the defects reported by TVM users and developers, particularly their severe impacts to TVM users and developers.

Method We include two data sources for studying TVM bugs. The first source is bug-report posts in TVM discuss forum. The second source is bug-fixing pull requests from the TVM GitHub repository. Both users and developers of TVM may file a post in TVM discuss forum or initiate a bug-fixing pull request to report or fix the bugs they find in TVM. For the bug-report posts in TVM discuss forum, the main body of these posts usually contains preliminary analysis of potential causes of the bug, which differs these posts from the *Troubleshooting* category in Section 4. We notice that some posts may appear to be about troubleshooting in the beginning, but through rounds of investigations and discussions, TVM developers identified that the erroneous behaviours are caused by TVM bugs instead of incorrect usage of TVM. For analyzing bug-fixing PRs, we include all the information we can find, including their corresponding TVM posts and GitHub issues.

Similar to RQ1 and RQ2, we also mark the resolution status of each bug report post, i.e., whether or not there is an explicit *[Resolved]* in the title or any indications in the replies,

[20] https://tvm.apache.org/docs/contribute/community.html#general-development-process

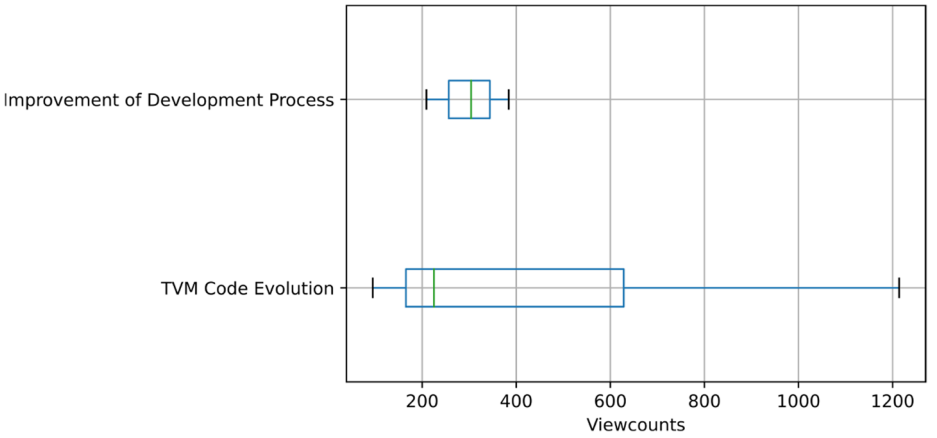

Fig. 12 Boxplot of the view counts of Development Categories

such as a link to a pull request in the replies or users acknowledging the bug was resolved after checking out the latest commit of TVM. For bug-fixing PRs, as we only include the PRs that are already merged, such bugs are considered resolved. Furthermore, we analyzed the impact of the bugs regarding the usage and development of TVM. In addition, we analyzed the component of TVM that is the potential root cause of the bugs based on the fix commits and discussion.

Results There are a total of 341 bugs in RQ3, including 44 bug reports in the sampled posts from TVM discuss forum and 297 bug-fixing PRs from TVM GitHub repository. 22 (50.0%) out of the 44 bug-report posts are resolved and the remaining unresolved bug posts are under the investigation. All the bug-fixing PRs are merged into the repository.

Regarding the reproducibility of the studied bugs, in general, we notice that there exists a high level engagement of TVM developers on the 44 bug reports (i.e., at least one reply confirming the reproduction of the reported bugs and kicking off follow-up investigations)

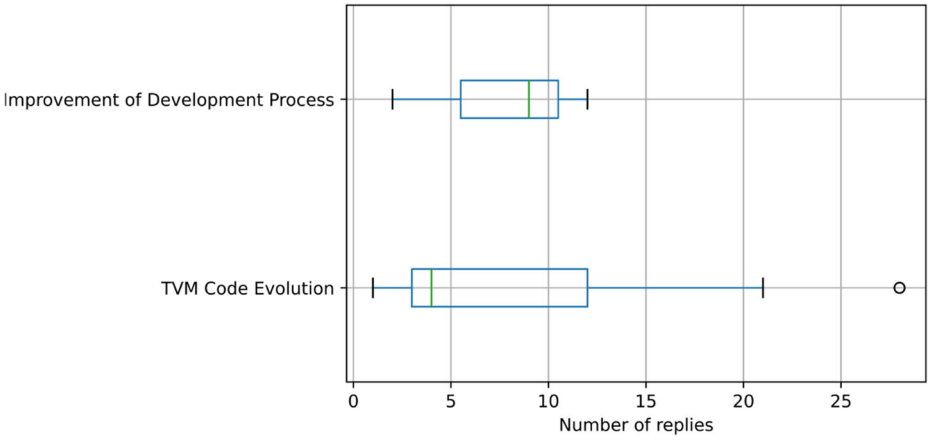

Fig. 13 Boxplot of the replies of Development Categories

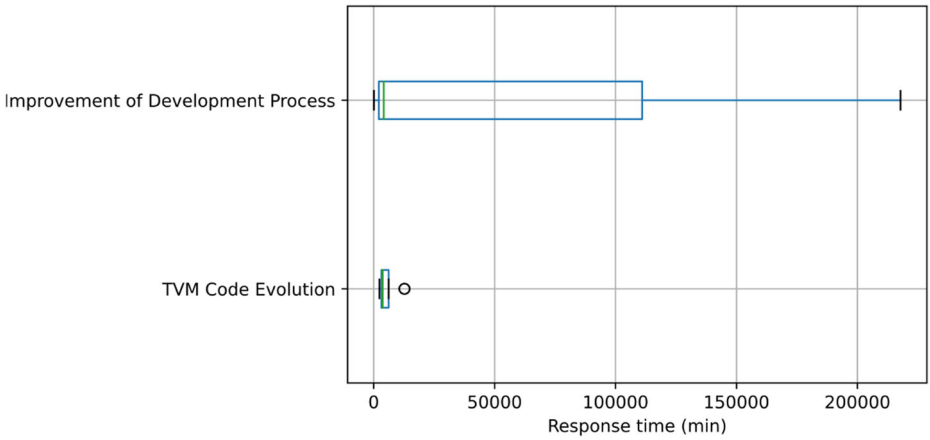

Fig. 14 Boxplot of the resolution time of Development Categories

and therefore we consider all the 44 bug reports are reproducible. We consider all bugs fixed by the included bug-fixing PRs reproducible.

We identified a total of four types of impacts: compilation failure, non-compilation runtime error (including crashes and exceptions, similar to Islam et al. (2019)), poor efficiency and low effectiveness.

Impact 1: Compilation Failure. The TVM compiler may fail to compile the input DL model due to various reasons such as unsupported operators and incompatibility issues. For example, a TVM user (TvmUser 2018b) complained that TVM failed to compile an operator for CUDA target due to unsupported operator (i.e., `argmax` and `argmin` from MXNet). Later a TVM developer solved the issue and submitted a pull request.

Impact 2: Non-compilation Runtime Error. In addition to compilation failures, TVM also suffers from runtime errors that are not related to compilation, i.e., crashes or exceptions. Such errors prevent TVM users and developers continuing working on TVM. For example, a TVM user encountered a runtime error that caused the connections get reset when creating large arrays using RPC (TvmUser 2020f). In particular, an out of memory error was thrown from the RPC server and reset the connection. This issue was fixed by developers by preventing a temporary buffer from being too large in order to avoid out of memory errors.

Impact 3: Poor Efficiency. TVM aims to optimize the performance of DL models. However, bugs in TVM lead to poor performance of the compiled DL models from TVM, e.g., unexpected long inference time, consuming a large amount of resources. For example, a TVM user TvmUser (2020c) found that after merging the latest pull request, there had been a significant performance regression. Later a TVM developer confirmed this performance regression and was able to fix it. The performance regression was caused by a refactoring commit, which incorrectly modified an attribute and lead to this performance regression.

Impact 4: Low Effectiveness. The compiled DL models by TVM may show poor accuracy and abnormal behaviours after compilation. For example, a TVM user reported that the compiled GluonCV SSD model kept outputting incorrect inference results on ARM Mali due to that one of the supported operators by TVM that are required by the model (i.e., operator `vision.get_valid_counts`) did not work correctly on Mali GPU (TvmUser 2019g). This issue was solved by two subsequent pull requests.

Discussion Based on the analysis, *Compilation Failure* (101/341, 29.6%) is the most common bugs in TVM. Table 4 presents the distribution of the 341 bugs in different TVM components (i.e., frontend, IR, backend, VTA and RPC) and the impacts. Note that the component will be marked as "Unclear" if there is no indication or discussion about the component of the bug. As explained in Section 2.3, frontend, IR and backend are the common components in DL compilers and not unique to TVM. Both VTA and RPC are unique components in TVM. VTA is a customizable deep learning accelerator with a TVM-based compiler stack. RPC is provided by TVM in order to perform cross-compilation and alleviate users from remote testing. All of these components are important to the correct use of TVM and any bugs in these components will prevent the compiled DL models from functioning properly. Among all the components, frontend and backend contributes the largest number of bugs. This is not surprising since the frontend needs to deal with many DL frameworks and the backend needs to generate codes for various hardware, which makes them subject to problems such as compatibility issues, unsupported operators.

7 Discussion and Implication of Our Findings

Generality of Our Findings Although our study is conducted on TVM, we believe our taxonomy of challenges in using/developing TVM is prevalent and most of our findings can be generalized to other DL compilers. The main reason is that most DL compilers share a common process flow (e.g., frontend, IR, high-level optimization, low-level optimization, backend) (Xing et al. 2019). As an example, *Environment configuration* under *Usage-Troubleshooting-Configuration* reveals environment-related challenges. It is well-known that DL software or DL library compilation may often suffer from challenges in setting up the environments. We believe these findings can apply to other DL compilers since most DL compilers require users to prepare an environment with sophisticated dependencies (e.g., compatible CUDA toolkit, cuDNN library) in order to generate optimized modules for a specific target platform. For instance, the installation guide of Tensor Comprehension asks the users to be careful about the CUDA toolkit dependency problem.[21] Furthermore, we collected 15 posts (all posts with "ngraph" in its title) from the official forum of nGraph, 30 randomly sampled posts from the official forum of Glow, and 30 bug-related PRs from their GitHub repositories. Then we use the taxonomy we summarized in Fig. 6 and four impacts of TVM bugs to classify these posts/PRs. The results show that all these posts/PRs can be classified using the findings from our study, which further proves the generality of our findings. The labeling results are also made publicly available.[22]

Lack of Benchmark for Performance Evaluation During the labeling process, we notice that there are a non-trivial number (13/279, 4.7%) of the sampled posts discussing about the performance of the compiled models by TVM. Some users may have no idea about whether their model has been tuned to achieve the optimal performance. There has been a study on the comparison of DL compilers (Xing et al. 2019), which provides a benchmark for evaluating the performance of seven DL compilers including TVM on six deep neural networks (DNNs). TVM community has also released an official benchmark (TvmDeveloper 2018c). However, such efforts are far from covering the need of users and preventing

[21] https://facebookresearch.github.io/TensorComprehensions/installation.html#advanced-development-mode-installation
[22] https://bit.ly/3I9xohu

Table 4 The distribution of TVM bugs based on impacts and TVM components. Note that the results are based on 44 bug-report posts on TVM discuss forum (i.e., the first number in brackets) and 297 bug-fixing PRs (i.e., the second number in brackets)

Symptoms \ Component	Frontend	IR	Backend	VTA	RPC	Unclear
Non-compilation Runtime Error	29 (4|25)	46 (0|46)	12 (0|12)	2 (2|0)	2 (2|0)	10 (4|6)
Compilation Failure	32 (7|25)	18 (0|18)	33 (8|25)	0 (0|0)	2 (1|1)	16 (3|13)
Poor Efficiency	1 (1|0)	10 (1|9)	11 (6|5)	0 (0|0)	1 (1|0)	0 (0|0) 3
Low Effectiveness	28 (3|25)	39 (0|39)	26 (0|26)	2 (1|1)	0 (0|0)	9 (4|5)
Total	90	113	82	4	5	35

performance regressions as TVM is rapid evolving. For example, the official benchmark has not been updated since October 2018. Although it is be impractical to cover all the deep neural networks and hardware, the developers and researchers may take a survey and provide a benchmark of popular DNNs and hardware in a larger scale as needed, which will be helpful for both users and developers as a reference.

Needs for Continuously Improving and Updating Documentation and Tutorials Our study shows that many users and developers ask basic questions due to lack of understanding of the basic concepts in TVM. We believe that providing better documentation especially more comprehensive tutorial examples will ease the onboarding process for newcomers and flatten the learning curve. In summary, we have the following advice for TVM developers as well as research community. Future research work can utilize crowd sourcing to automatically collect more examples and update existing examples and documentation as TVM evolves.

- *Improve the quality of documents.* We find that, *How to use TVM*, *Conceptual Questions* and *Selection/usage of APIs* occupies 19.2% of the sampled posts. This suggests that many users and developers lack basic understanding of TVM and there is insufficient resource to cover such an aspect. Furthermore, we notice that users often complain about not being able to find the examples in need or outdated tutorials. A continuous effort to improving the quality of such learning resources will alleviate these issues.
- *Enrich the contents of update announcements.* TVM is under rapid evolution, which has a large number of undergoing and future API changes and the corresponding compatibility changes. However, such changes often are not included in the current TVM change.[23] This may lead to API misuse and confuse the TVM users. Thus, TVM developers should provide more detailed change logs (e.g., compatible library version) to better help users become aware of these changes and avoid potential compatibility issues.
- *Adding documentation/tutorials for more diverse use cases.* We notice that sometimes, TVM users or developers ask about how to perform certain tasks with the guidance from existing learning resources, which describe how to perform similar but not identical tasks. However, due the differences between the desired task and the task (e.g., different devices, different compilation options, and different DL frameworks) described by learning resources, TVM users and developers are still having a difficult time to figure

[23] TVM Change LOG: https://github.com/apache/tvm/blob/main/NEWS.md

out how to complete the tasks. Providing more comprehensive learning resources by considering more diverse use cases will help resolve such challenges.

Needs for Automated Test Generation Support From the previous discussion in RQ3, we have noticed that the TVM not only suffers from various bugs like traditional software but also have some unique bugs due to its functionality. Consider its importance as the foundation of deploying DL application, it is critical and urgent to have more efficient techniques that can provide automated test generation support in order to make DL compilers to be more robust. On one hand, TVM could benefit from some of the existing automated test generation techniques, such as fuzzing the input (i.e., DL models) to create more test cases. On the other hand, TVM's unique characteristics could be exploited to improve its quality assurance practice. For example, TVM supports various DL frameworks and different devices, which could be utilized by differential testing techniques to test the frontend and backend of TVM.

Needs for Better Logging Practice and Debugging Support After inspecting bug reports and troubleshooting posts in RQ1 and RQ3, we notice that the logging information of TVM is not always very informative. While some TVM developers may not have difficulties in identifying the corresponding component to inspect benefiting from their familiarity of TVM codebase, TVM users are often confused about the error messages and have no idea about how the solve the problem. To better assist users in performing troubleshooting practice, we suggest TVM developers to improve the logging format of TVM and add more informative logs in critical code locations. Also, traditional debugging techniques may not work well for DL compilers due to the peculiarity of the input, i.e., DL models. Future research efforts can be paid to propose specialized debugging techniques for debugging DL compilers, such as BigDebug (Gulzar et al. 2016) for Spark.

Needs for More Descriptive Pull-Request Description During the inspection of bug fix pull-requests in RQ3, we notice that the description body of a TVM pull-request is not always informative, 24.8% of the bug fix commits do not have a description body. While TVM community requires a commit must be reviewed by at least one active TVM contributor,[24] a bug fix commit may add new bug to the codebase considering the complexity of TVM. To prevent technical debt for long-term in fixing the involved bugs, we suggest TVM developers to maintain a more descriptive pull-request description, i.e., the root cause of the bug, how this bug is fixed, which part of TVM may be affected.

Needs for Integrated Bug Knowledge Database with Improved Traceability and Management In the process of manually analyzing discussion forum posts, including reading relevant sources such as links to GitHub issues and pull requests, we notice several inefficiencies of current management of bug knowledge. First, often a bug needs to be reported for several time before eventually being resolved. This shows the lack of trackability and traceability within the same database, i.e., the problem of duplicate bug reports. Second, with multiple venues involved (i.e., GitHub issues, pull requests, and discussion forum), the materials of one bug are often scattered across different knowledge databases. For example, developers often forget to post the solution to an issue even if they have fixed it on GitHub.

[24] TVM Community Guideline: https://tvm.apache.org/docs/contribute/community.html#reviewers

Based on these considerations, we think there should be an integrated bug knowledge base for better traceability.

8 Threats to Validity

In this section, we will discuss about the threats to validity of our work.

Selection of Data Source In this study, we utilize the official discussion forum of TVM as the only data source to investigate the challenges that users/developers encounter when using/developing TVM. As a consequence, we may neglect constructive insights from other sources. In the future, we plan to further validate our results by conducting in-depth interviews with researchers and developers. However, since 1) the official discussion forum may contain both experts' and novices' posts; 2) the fact that most modern DL compilers share a common architecture (Xing et al. 2019); and 3) our experimental results of generalizing our findings on nGraph and Glow, we believe our findings are still valid.

Selection Strategy of the Posts for Initial Taxonomy Distillation When constructing the taxonomy, we chose to start with 50 randomly selected posts in order to distill an initial set of taxonomy and then iteratively refine the initial taxonomy, which may introduce threats to the findings. First, we agree that starting with N posts (N is greater than 50) may yield a slightly different initial taxonomy, it may not have much impact on the final taxonomy, due to having rounds of refinement. Second, there exists no golden standard regarding how many instances should be selected for an initial taxonomy. Third, the differences between the initial taxonomy and the refined taxonomy are expected to some extent. For example, in Humbatova et al. (2020), the authors added 13 new leaf categories in their final round of refinement. Moreover, we would like to clarify that the differences in our study are not that significant: most of the differences are about replacing the old category with a more representative name instead of changing the definition of one category, for example, *Non-development* to *Usage*, *Configuration-Installing/Building TVM framework* to *Configuration-Build Configuration*. Last, we would like to emphasize that during the labeling process, if we decided to change a category, all the posts previously under that category will be checked again to avoid possible mis-classification.

Subjectivity of Inspection Our study involves manual inspection on posts from the discussion forum. These subjective steps may introduce bias and present threats to the validity of our taxonomy. Furthermore, some of the sampled posts may change their topics upon further investigation. For example, a TVM user complained about the performance of the compiled module (subject to *Performance*) and then the user were confused about two APIs that both can generate code and there were a significant performance difference between these two APIs (*Selection/usage of API*). To reduce this threat, two authors are employed to separately inspect the posts and the inconsistent cases are resolved with the help of a arbitrator. Besides, the inter-rater reliability is relatively high, exhibiting the reliability of the coding procedure.

9 Related Work

In this section, we discuss the following lines of closely related research.

Deep Learning Compilers The development and adoption of deep learning compilers have drawn much attention in both academia and industry due to the increasing interests to perform hardware-specific optimization on model deployment. Many deep learning compilers have been proposed and evaluated till this end, e.g., TVM (Chen et al. 2018a), nGraph (Cyphers et al. 2018), Tensor Comprehension (Vasilache et al. 2018), Glow (Rotem et al. 2018), XLA (Leary and Wang 2017) and DLA (Abdelfattah et al. 2018). Such deep learning compilers differ in their design architectures, IR designs and optimization methods, which are studied and elaborated in a recent survey paper on deep learning compilers (Li et al. 2021). Xing et al. (2019) concluded a general DL compiler flow and furthermore performed an in-depth comparison among DL compilers regarding the internal components, e.g., optimization strategies and intermediate representations.

Follow-up research has been proposed to further improve the efficiency of deep learning compilers based on the common architectures. Chen et al. (2018b) present AutoTVM that optimizes tensor programs based on workloads. Boemer et al. (2019) extended nGraph to perform computation on encrypted data so that data privacy can be preserved. These prior studies focus on the development and improvement of deep learning compilers, but have not studied the usage and development of deep learning compilers, especially the challenges users and developers may encounter during the process are not yet studied. Through this study, we concluded the common types of challenges users may have when using one of the most popular deep learning compiler, i.e., TVM. Our research findings call for future research efforts on providing better tool support for TVM users.

Studies on Developing and Deploying ML/DL Applications Due to the increasing popularity of machine learning applications, many empirical studies have been performed to obtain a deep understanding on the new challenges posed during the development of machine learning applications. A recent study by Zhang et al. (2019) categorize common challenges in developing deep learning applications through manually investigating 715 Stack Overflow questions. In particular, their study categorized and concluded the top three challenges are related to program crashes, model deployment and implementation. Chen et al. (2020) investigate the challenges in deploying machine learning applications and present a taxonomy of such challenges, through an analysis of 769 StackOverflow posts. Different from previous work on general development and deployment of machine learning software, this study focuses on the usage and development of one deep learning compiler and the common challenges shared by users and developers. While a deep learning compiler like TVM is used for model optimization in model deployment, general-purpose knowledge sharing websites, such as StackOverflow, may have very limited discussions on TVM usage (as shown in Section 3.1). Hence, our analysis on TVM complements these prior studies on the aspect of deep learning compilers, which is not yet studied.

In addition, many studies are performed to characterize the defects that occur in machine learning applications. Zhang et al. (2018) examined the root causes and symptoms of the bugs found in TensorFlow programs. Furthermore, Islam et al. (2019) et al. present a comprehensive study of bugs (e.g., the root causes, the impact) from a larger set of deep learning libraries, i.e., Caffe, Keras, Tensorflow, Theano, and Torch. A following work by Humbatova et al. (2020) present a taxonomy of real faults in deep learning systems through a manual examination of 1,059 commits and bug issues. Zhang et al. (2020) analyzed the impact of adversarial defects on DL models and concluded the patterns of such adversarial defects. In this study, through an analysis of user-reported bugs, i.e., bugs in either TVM or TVM usage, we investigate the impacts of TVM bugs may have on DL software. Our preliminary investigation on TVM bugs calls for specialized testing techniques for DL compilers.

Studies on Bug Reports and Feature Requests Issue-tracking systems such as Jira and Bugzilla provide rich information to improve many lines of software engineering research, such as fault localization and debugging (Wang and Lo 2016; Saha et al. 2013), automated program repair (Liu et al. (2013), and feature tracking (Fischer et al. (2003). The quality, components, and characteristics of issue reports and they impact software engineering tasks have been well studied by previous work. Zimmermann et al. (2010) studied the essential criteria and elements to produce good reports and proposed a tool to measure the quality of bug reports. Chen and Chen (2021) studied the quality of logs in the bug reports and how that may affect fault localization techniques. As TVM does not employ Apache Jira for development communication (e.g., reporting bugs and requesting features), hence in our study, we use the posts from the TVM discuss forum, which is an active platform used by both users and developers. Through our manual analysis, we find that TVM forum posts provide complementary knowledge on critical software development procedures and decisions, such as fixing bugs and requesting new features, in addition to Github issues. Future research should consider using forum posts as an alternative data source in addition to Github issues and StackOverflow posts.

10 Conclusion

This paper initiates the first step towards the usage and development challenges of DL compilers. We manually inspect 347 posts from the official discussion forum of TVM and identify a taxonomy of challenges about the usage of TVM consisting of 15 categories and seven types of common topics which TVM developers discuss. Among all the categories, procedure, configuration, and how-to questions are the top three most frequently asked questions. Furthermore, four kinds of bug symptoms are summarized to better understanding the common defects of TVM. Finally, we discuss our implications for developers and researchers in order to help them to have a better understanding of developing DL compilers.

Acknowledgements This work was supported by JST, the establishment of university fellowships towards the creation of science technology innovation, Grant Number JPMJFS2132.

Funding No funding was received to assist with the preparation of this manuscript

Declarations

Competing Interests : The authors have no competing interests to declare that are relevant to the content of this article. The authors declare no conflict of interests with the suggested reviewers for this article.

References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X (2016) Tensorflow: a system for large-scale machine learning. In: Proceedings of the 12th USENIX conference on operating systems design and implementation, USENIX Association, USA, OSDI'16, pp 265–283

Abdelfattah MS, Han D, Bitar A, DiCecco R, O'Connell S, Shanker N, Chu J, Prins I, Fender J, Ling AC, Chiu GR (2018) Dla: compiler and fpga overlay for neural network inference acceleration. In: 2018 28th International conference on field programmable logic and applications (FPL), pp 411–4117. https://doi.org/10.1109/FPL.2018.00077

Badue C, Guidolini R, Carneiro RV, Azevedo P, Cardoso VB, Forechi A, Jesus L, Berriel R, Paixão TM, Mutz F, de Paula Veronese L, Oliveira-Santos T, De Souza AF (2021) Self-driving cars: a survey. Expert Syst Appl 113816:165. https://doi.org/10.1016/j.eswa.2020.113816. http://www.sciencedirect.com/science/article/pii/S095741742030628X

Bagherzadeh M, Khatchadourian R (2019) Going big: a large-scale study on what big data developers ask. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2019. Association for Computing Machinery, New York, pp 432–442. https://doi.org/10.1145/3338906.3338939

Berg BL, Lune H, Lune H (2004) Qualitative research methods for the social sciences, vol 5. Pearson, Boston

Boemer F, Lao Y, Cammarota R, Wierzynski C (2019) Ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In: Proceedings of the 16th ACM international conference on computing frontiers, CF '19. Association for Computing Machinery, New York, pp 3–13. https://doi.org/10.1145/3310273.3323047

Chen T (2020) Language server tool to navigate across packedfunc ffi for ides like vscode and emacs. https://bit.ly/3oO6Qso, retrieved on December 9, 2020

Chen A, Chen PTH (2021) Demystifying the challenges and benefits of analyzing user-reported logs in bug reports. Empirical Software Engineering

Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Cowan M, Shen H, Wang L, Hu Y, Ceze L, Guestrin C, Krishnamurthy A (2018a) Tvm: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of the 13th USENIX conference on operating systems design and implementation, USENIX Association, USA, OSDI'18, pp 579–594

Chen T, Zheng L, Yan E, Jiang Z, Moreau T, Ceze L, Guestrin C, Krishnamurthy A (2018b) Learning to optimize tensor programs. In: Proceedings of the 32nd international conference on neural information processing systems, NIPS'18. Curran Associates Inc., Red Hook, pp 3393–3404

Chen Z, Cao Y, Liu Y, Wang H, Xie T, Liu X (2020) A comprehensive study on challenges in deploying deep learning based software. In: Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2020. Association for Computing Machinery, New York, pp 750–762. https://doi.org/10.1145/3368089.3409759

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Meas 20(1):37–46. https://doi.org/10.1177/001316446002000104

Cyphers S, Bansal AK, Bhiwandiwalla A, Bobba J, Brookhart M, Chakraborty A, Constable W, Convey C, Cook L, Kanawi O, Kimball R, Knight J, Korovaiko N, Vijay VK, Lao Y, Lishka CR, Menon J, Myers J, Narayana SA, Procter A, Webb TJ (2018) Intel ngraph: an intermediate representation, compiler, and executor for deep learning. CoRR arXiv:1801.08058. 1801.08058

Deng L, Liu Y (2018) Deep learning in natural language processing. Springer, Berlin

Fakoor R, Ladhak F, Nazi A, Huber M (2013) Using deep learning to enhance cancer diagnosis and classification. In: Proceedings of the international conference on machine learning, vol 28. ACM, New York

Fischer M, Pinzger M, Gall H (2003) Analyzing and relating bug report data for feature tracking. In: 10th Working conference on reverse engineering, 2003. WCRE 2003. Proceedings, pp 90–99. https://doi.org/10.1109/WCRE.2003.1287240

Foundation A (2020) Mxxnet. https://mxnet.apache.org/, retrieved on January 4, 2021

Garcia J, Feng Y, Shen J, Almanee S, Xia Y, Chen QA (2020) A comprehensive study of autonomous vehicle bugs. In: 2020 IEEE/ACM 42nd international conference on software engineering (ICSE), pp 385–396

Github search api (2021) https://docs.github.com/en/rest/reference/search, retrieved on May 10, 2021

Gulzar MA, Interlandi M, Yoo S, Tetali SD, Condie T, Millstein T, Kim M (2016) Bigdebug: debugging primitives for interactive big data processing in spark. In: Proceedings of the 38th international conference on software engineering, ICSE '16. Association for Computing Machinery, New York, pp 784–795. https://doi.org/10.1145/2884781.2884813

Hemanth DJ, Estrela VV (2017) Deep learning for image processing applications, vol 31. IOS Press

Hoang VCD, Koehn P, Haffari G, Cohn T (2018) Iterative back-translation for neural machine translation. In: Proceedings of the 2nd workshop on neural machine translation and generation, pp 18–24

Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P (2020) Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, ICSE '20. Association for Computing Machinery, New York, pp 1110–1121. https://doi.org/10.1145/3377811.3380395

Intel (2020) Accelerate fast math with intel oneapi math kernel library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html, retrieved on December 9, 2020

Islam MJ, Nguyen G, Pan R, Rajan H (2019) A comprehensive study on deep learning bug characteristics. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and

symposium on the foundations of software engineering. ESEC/FSE 2019. Association for Computing Machinery, New York, pp 510–520. https://doi.org/10.1145/3338906.3338955

Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, Boyle R, Cantin PL, Chao C, Clark C, Coriell J, Daley M, Dau M, Dean J, Gelb B, Ghaemmaghami TV, Gottipati R, Gulland W, Hagmann R, Ho CR, Hogberg D, Hu J, Hundt R, Hurt D, Ibarz J, Jaffey A, Jaworski A, Kaplan A, Khaitan H, Killebrew D, Koch A, Kumar N, Lacy S, Laudon J, Law J, Le D, Leary C, Liu Z, Lucke K, Lundin A, MacKean G, Maggiore A, Mahony M, Miller K, Nagarajan R, Narayanaswami R, Ni R, Nix K, Norrie T, Omernick M, Penukonda N, Phelps A, Ross J, Ross M, Salek A, Samadiani E, Severn C, Sizikov G, Snelham M, Souter J, Steinberg D, Swing A, Tan M, Thorson G, Tian B, Toma H, Tuttle E, Vasudevan V, Walter R, Wang W, Wilcox E, Yoon DH (2017) In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th annual international symposium on computer architecture, ISCA '17. Association for Computing Machinery, New York, pp 1–12. https://doi.org/10.1145/3079856.3080246

Kingsley-Hughes A (2017) Inside apple's new a11 bionic processor. ZDNet, September

Lacey G, Taylor GW, Areibi S (2016) Deep learning on fpgas: past, present, and future. arXiv:160204283. https://arxiv.org/abs/1602.04283

Leary C, Wang T (2017) XLA: TensorFlow, Compiled!. TensorFlow Dev Summit 2017

LeCun Y (2019) 1.1 deep learning hardware: Past, present, and future. In: 2019 IEEE International Solid-State Circuits Conference—(ISSCC), pp 12–19. https://doi.org/10.1109/ISSCC.2019.8662396

Lecun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324. https://doi.org/10.1109/5.726791

Li M, Liu Y, Liu X, Sun Q, You X, Yang H, Luan Z, Gan L, Yang G, Qian D (2021) The deep learning compiler: a comprehensive survey. IEEE Trans Parallel Distrib Syst 32(3):708–727. https://doi.org/10.1109/TPDS.2020.3030548

Liu C, Yang J, Tan L, Hafiz M (2013) R2fix: automatically generating bug fixes from bug reports. In: 2013 IEEE Sixth international conference on software testing, verification and validation, pp 282–291. https://doi.org/10.1109/ICST.2013.24

Liu S, Du Z, Tao J, Han D, Luo T, Xie Y, Chen Y, Chen T (2016) Cambricon: an instruction set architecture for neural networks. In: 2016 ACM/IEEE 43rd annual international symposium on computer architecture (ISCA). IEEE, pp 393–405

Moreau T, Chen T, Jiang Z, Ceze L, Guestrin C, Krishnamurthy A (2018) VTA: an open hardware-software stack for deep learning. CoRR. arXiv:1807.04188. http://arxiv.org/abs/1807.04188, 1807.04188

Nguyen G, Dlugolinsky S, Bobák M, Tran V, López García A, Heredia I, Malík P, Hluchy L (2019) Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. Artif Intell Rev 52(1):77–124. https://doi.org/10.1007/s10462-018-09679-z

NVIDIA (2020) Nvidia cudnn. https://developer.nvidia.com/cudnn, retrieved on December 9, 2020

ONNX (2020) Open nueral network exchange. https://onnx.ai/, retrieved on December 9, 2020

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) Pytorch: an imperative style, high-performance deep learning library. In: Wallach HM, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox EB, Garnett R (eds) Advances in neural information processing systems 32: annual conference on neural information processing systems 2019, neurIPS 2019, December 8–14, 2019, Vancouver, pp 8024–8035. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S (2013) Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. SIGPLAN Not 48(6):519–530. https://doi.org/10.1145/2499370.2462176

Roesch J, Lyubomirsky S, Weber L, Pollock J, Kirisame M, Chen T, Tatlock Z (2018) Relay: a new ir for machine learning frameworks. In: Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages, MAPL 2018, p 58–68. Association for Computing Machinery, New York. https://doi.org/10.1145/3211346.3211348

Rotem N, Fix J, Abdulrasool S, Deng S, Dzhabarov R, Hegeman J, Levenstein R, Maher B, Satish N, Olesen J, Park J, Rakhov A, Smelyanskiy M (2018) Glow: graph lowering compiler techniques for neural networks. CoRR arXiv:1805.00907. http://arxiv.org/abs/1805.00907, 1805.00907

Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. Nature 323(6088):533–536

Saha RK, Lease M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: 2013 28th IEEE/ACM International conference on automated software engineering (ASE), pp 345–355. https://doi.org/10.1109/ASE.2013.6693093

Schmidhuber J, Hochreiter S (1997) Long short-term memory. Neural Comput 9(8):1735–1780

Tompson J, Schlachter K (2012) An introduction to the opencl programming model
TVM (2020a) About vta. https://tvm.apache.org/vta, retrieved on December 9, 2020
TVM (2020b) Dive into deep learning compiler. https://tvm.d2l.ai/, retrieved on December 9, 2020
TVM (2020c) Tvm language reference. https://tvm.apache.org/docs/langref/index.html, retrieved on December 9, 2020
TVM (2020d) Tvm start up tutorial. https://tvm.apache.org/docs/tutorials, retrieved on December 9, 2020
TVMConf (2019) Tvm conference keynote. https://tvmconf.org/slides/2019/tvmconf-keynote-dec19.pdf, retrieved on December 9, 2020
TvmDeveloper (2018a) Datalayout structure. https://discuss.tvm.apache.org/t/datalayout-structure/80, retrieved on December 9, 2020
TvmDeveloper (2018b) Int8 quantization proposal. https://discuss.tvm.apache.org/t/int8-quantization-proposal/516, retrieved on December 9, 2020
TvmDeveloper (2018c) Tvm benchmark. https://github.com/apache/tvm/wiki/Benchmark, retrieved on December 9, 2020
TVMDeveloper (2019) Improving quantization accuracy with more precise bias. https://discuss.tvm.apache.org/t/improving-quantization-accuracy-with-more-precise-bias/2444, retrieved on December 9, 2020
TvmDeveloper (2020a) Naming consistency: util vs utils. https://discuss.tvm.apache.org/t/naming-consistency-util-vs-utils/6434, retrieved on December 9, 2020
TvmDeveloper (2020b) Tvm docs: deploy and integration. https://tvm.apache.org/docs/deploy/index.html, retrieved on December 9, 2020
TvmDeveloper (2020c) Tvm documents: runtime. https://tvm.apache.org/docs/dev/runtime.html#module, retrieved on December 9, 2020
TvmDocs (2020) Pattern matching in relay. https://tvm.apache.org/docs/langref/relay_pattern.html#pattern-language-design, retrieved on December 9, 2020
TvmUser (2018a) [autotvm] localrunner not working on windows. https://discuss.tvm.apache.org/t/autotvm-localrunner-not-working-on-windows/969, retrieved on December 9, 2020
TvmUser (2018b) [compiler] cannot compile operator mean for cuda target. https://discuss.tvm.apache.org/t/compiler-cannot-compile-operator-mean-for-cuda-target/1110, retrieved on December 9, 2020
TvmUser (2018c) Global sync across different blocks in ir builder. https://discuss.tvm.apache.org/t/global-sync-across-different-blocks-in-ir-builder/477, retrieved on December 9, 2020
TvmUser (2018d) How to load and read the.params in big endian system? https://discuss.tvm.apache.org/t/how-to-load-and-read-the-params-in-big-endian-system/203, retrieved on December 9, 2020
TvmUser (2018e) Int8 quantization proposal. https://discuss.tvm.apache.org/t/int8-quantization-proposal/516, retrieved on December 9, 2020
TvmUser (2018f) Registering operator attributes for relay pattern matching. https://discuss.tvm.apache.org/t/registering-operator-attributes-for-relay-pattern-matching/8894, retrieved on December 9, 2020
TvmUser (2018g) [solved] how to export model library to so file instead of tar for armv7 on x86 box. https://discuss.tvm.apache.org/t/solved-how-to-export-model-library-to-so-file-instead-of-tar-for-armv7-on-x86-box/970, retrieved on December 9, 2020
TvmUser (2019a) Compile official mobilenet onnx, get a very slow performance. https://discuss.tvm.apache.org/t/compile-official-mobilenet-onnx-get-a-very-slow-performance/2839, retrieved on December 9, 2020
TvmUser (2019b) [cuda]got error: Cuda error launch out of resources. https://discuss.tvm.apache.org/t/cuda-got-error-cuda-error-launch-out-of-resources/4173, retrieved on December 9, 2020
TvmUser (2019c) An error when doing autotune tensorflow net uisng opencl on intel integrated graphics. https://discuss.tvm.apache.org/t/an-error-when-doing-autotune-tensorflow-net-uisng-opencl-on-intel-integrated-graphics/4143, retrieved on December 9, 2020
TvmUser (2019d) [frontend] tensorflow op: Fifoqueuev2 and queuedequeuemanyv2 are not supported. https://discuss.tvm.apache.org/t/frontend-tensorflow-op-fifoqueuev2-and-queuedequeuemanyv2-are-not-supported/1596, retrieved on December 9, 2020
TvmUser (2019e) Limit cpu cores for auto tuned model. https://discuss.tvm.apache.org/t/limit-cpu-cores-for-auto-tuned-model/2384, retrieved on December 9, 2020
TvmUser (2019f) [quantization] how to quantize transpose and nn.pad operators? https://discuss.tvm.apache.org/t/quantization-how-to-quantize-transpose-and-nn-pad-operators/3861, retrieved on December 9, 2020
TvmUser (2019g) Ssd gluoncv: incorrect inference result on opencl arm mali. https://discuss.tvm.apache.org/t/ssd-gluoncv-incorrect-inference-result-on-opencl-arm-mali/2848, retrieved on December 9, 2020
TvmUser (2019h) Tensorarray globalvar and globaltypevar confusion. https://discuss.tvm.apache.org/t/tensorarray-globalvar-and-globaltypevar-confusion/4567, retrieved on December 9, 2020

TvmUser (2019i) What's the difference between build() and create_executor() in tvm.relay.build_module? https://discuss.tvm.apache.org/t/whats-the-difference-between-build-and-create-executor-in-tvm-relay-build-module/1967, retrieved on December 9, 2020
TvmUser (2019j) What's the model bias in tvm paper. https://discuss.tvm.apache.org/t/whats-the-model-bias-in-tvm-paper/2963, retrieved on December 9, 2020
TvmUser (2020a) Can tvm now support batched inference? autotvm runs twice as long as tensorflow. https://discuss.tvm.apache.org/t/can-tvm-now-support-batched-inference-autotvm-runs-twice-as-long-as-tensorflow/6405, retrieved on December 9, 2020
TvmUser (2020b) Complex pattern matching in relay. https://discuss.tvm.apache.org/t/complex-pattern-matching-in-relay/5633, retrieved on December 9, 2020
TvmUser (2020c) [resolved][performance regression] migrate all low-level passes to the pass manager pr causing regression. https://discuss.tvm.apache.org/t/resolved-performance-regression-migrate-all-low-level-passes-to-the-pass-manager-pr-causing-regression/6246, retrieved on December 9, 2020
TvmUser (2020d) [rfc] improve pull requests with respect to bug fixes. https://discuss.tvm.apache.org/t/rfc-improve-pull-requests-with-respect-to-bug-fixes/6529, retrieved on December 9, 2020
TvmUser (2020e) Rocm 'segmentation fault' error when auto-tuning. https://discuss.tvm.apache.org/t/autotvm-stuck-during-tuning/6011, retrieved on December 9, 2020
TvmUser (2020f) Rpc error for large arrays. https://discuss.tvm.apache.org/t/rpc-error-for-large-arrays/6591, retrived on December 9, 2020
TvmUser (2020g) Same shape pattern. https://discuss.tvm.apache.org/t/same-shape-pattern/7012, retrieved on December 9, 2020
TvmUser (2020h) Support for tvm relay with wasm runtime. https://discuss.tvm.apache.org/t/support-for-tvm-relay-with-wasm-runtime/6765, retrieved on December 9, 2020
TvmUser (2020i) Where is dldatatype defined? https://discuss.tvm.apache.org/t/where-is-dldatatype-defined/6071, retrieved on December 9, 2020
Vasilache N, Zinenko O, Theodoridis T, Goyal P, DeVito Z, Moses WS, Verdoolaege S, Adams A, Cohen A (2018) Tensor comprehensions: framework-agnostic high-performance machine learning abstractions. arXiv:1802.04730. http://arxiv.org/abs/1802.04730, 1802.04730
Wang S, Lo D (2016) Amalgam+: composing rich information sources for accurate bug localization. J Softw Evol Process 28(10):921–942. https://doi.org/10.1002/smr.1801
Wang X, Tu Z, Wang L, Shi S (2020) Tencent ai lab machine translation systems for the wmt20 biomedical translation task. In: Proceedings of the fifth conference on machine translation, pp 881–886
Xing Y, Weng J, Wang Y, Sui L, Shan Y, Wang Y (2019) An in-depth comparison of compilers for deep neural networks on hardware, pp 1–8. https://doi.org/10.1109/ICESS.2019.8782480
Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018) An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2018. Association for Computing Machinery, New York, pp 129–140. https://doi.org/10.1145/3213846.3213866
Zhang T, Gao C, Ma L, Lyu M, Kim M (2019) An empirical study of common challenges in developing deep learning applications. In: 2019 IEEE 30th international symposium on software reliability engineering (ISSRE), pp 104–115. https://doi.org/10.1109/ISSRE.2019.00020
Zhang X, Xie X, Ma L, Du X, Hu Q, Liu Y, Zhao J, Sun M (2020) Towards characterizing adversarial defects of deep learning software from the lens of uncertainty. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, ICSE '20. Association for Computing Machinery, New York, pp 739–751. https://doi.org/10.1145/3377811.3380368
Zimmermann T, Premraj R, Bettenburg N, Just S, Schroter A, Weiss C (2010) What makes a good bug report? IEEE Trans Softw Eng 36(5):618–643. https://doi.org/10.1109/TSE.2010.63