An Empirical Study on Common Bugs in Deep Learning Compilers

Xiaoting Du
Beihang University, China
xiaoting_2015@buaa.edu.cn

Zheng Zheng*
Beihang University, China
zhengz@buaa.edu.cn

Lei Ma
University of Alberta, Canada
ma.lei@acm.org

Jianjun Zhao
Kyushu University, Japan
zhao@ait.kyushu-u.ac.jp

***Abstract*—The highly diversified deep learning (DL) frameworks and target hardware architectures bring big challenges for DL model deployment for industrial production. Up to the present, continuous efforts have been made to develop DL compilers with multiple state-of-the-arts available, e.g., TVM, Glow, nGraph, PlaidML, and Tensor Comprehensions (TC). Unlike traditional compilers, DL compilers take a DL model built by DL frameworks as input and generate optimized code as the output for a particular target device. Similar to other software, DL compilers are also error-prone. Buggy DL compilers can generate incorrect code and result in unexpected model behaviors. To better understand the current status and common bug characteristics of DL compilers, we performed a large-scale empirical study of five popular DL compilers covering TVM, Glow, nGraph, PlaidML, and TC, collecting a total of 2,717 actual bug reports submitted by users and developers. We made large manual efforts to investigate these bug reports and classified them based on their root causes, during which five root causes were identified, including *environment*, *compatibility*, *memory*, *document*, and *semantic*. After labeling the types of bugs, we further examined the important consequences of each type of bug and analyzed the correlation between bug types and impacts. Besides, we studied the time required to fix different types of bugs in DL compilers. Seven important findings are eventually obtained, with practical implications provided for both DL compiler developers and users.**

***Index Terms*—empirical study, deep learning compiler, bug, root cause, impact**

I. Introduction

Deep learning (DL) has been rapidly developed and successfully applied in many industrial domains, such as autonomous driving [1], smart city [2], disease diagnosis [3], and many others [4]–[6]. Various DL models with different architecture, such as Convolutional Neural Network (CNN) [7], Recurrent Neural Network (RNN) [8], transformer with attention mechanism [9], and Graph Neural Networks (GNN) [10] have been designed to deal with different kinds of tasks. As the foundation to support the training process and runtime inference, multiple popular DL frameworks have been developed so far, such as TensorFlow [11], PyTorch [12], MXNET [13] and CNTK [14]. These frameworks provide convenient interfaces for users to implement their DL model construction and training and deploy them on specific hardware devices that support the corresponding DL framework.

At the same time, we have witnessed an increasing trend and demand for deploying DL models to diverse end devices [15], [16]. In addition, quite a few DL chips with different architectures have been designed to boost performance, such as CPU, Google TPU [17], Hisilicon NPU [18], Apple Bonix [19], Intel NNP [20] as well as chips on small mobiles or edge devices [21]. However, it is still a challenge for DL frameworks to handle diverse hardware-specific transformations and DL model deployment across different target devices. To address this problem, several popular DL compilers have been proposed from both industry and academia, such as TVM [22], XLA [23], Glow [24], nGraph [25], PlaidML [26], and Tensor Comprehensions (TC) [27]. DL compilers take DL models constructed in a DL framework as input and generate optimized codes for diverse DL hardware as output, which performs an important role for DL model performance acceleration and diverse architecture hurdles.

Similar to traditional compilers, DL compilers also follow a layered design, including a frontend, an intermediate representation (IR) and a backend [28]. However, what makes DL compilers special is their multi-level IR and DL-specific optimization design. Specifically, DL compilers combine DL-oriented optimizations such as layer and operator fusion to achieve efficient code generation [29]. Like any software system, bugs can also commonly exist in DL compilers. The buggy DL compilers can cause misbehavior and even disaster when mis-compiled DL models are deployed to mission-critical applications. Therefore, the investigation of common bug characteristics in DL compilers should help design specific bug detection and localization methods, which are important for improving the quality and availability of DL compilers.

In this paper, we make a very early attempt to study the characteristics of bugs in DL compilers by analyzing 2,717 bugs collected from five popular DL compilers, including TVM, nGraph, Glow, PlaidML, and TC. The characteristics of bugs in DL compilers have been investigated from multiple aspects, including ❶ the root causes of bugs in DL compilers; ❷ the impacts of bugs in DL compilers and the correlation between bug types and impacts; ❸ the time it takes to fix bugs in DL compilers. The contribution of our work provides answers to the following three research questions.

RQ1: What common types of bugs exist in various DL compilers? To answer this question, we have classified the bugs in TVM, Glow, nGraph, PlaidML, and TC by analyzing their root causes. Through manual examination, five root causes are identified, including *environment*, *compatibility*,

*Corresponding author: Zheng Zheng.

2332-6549/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSRE52982.2021.00030

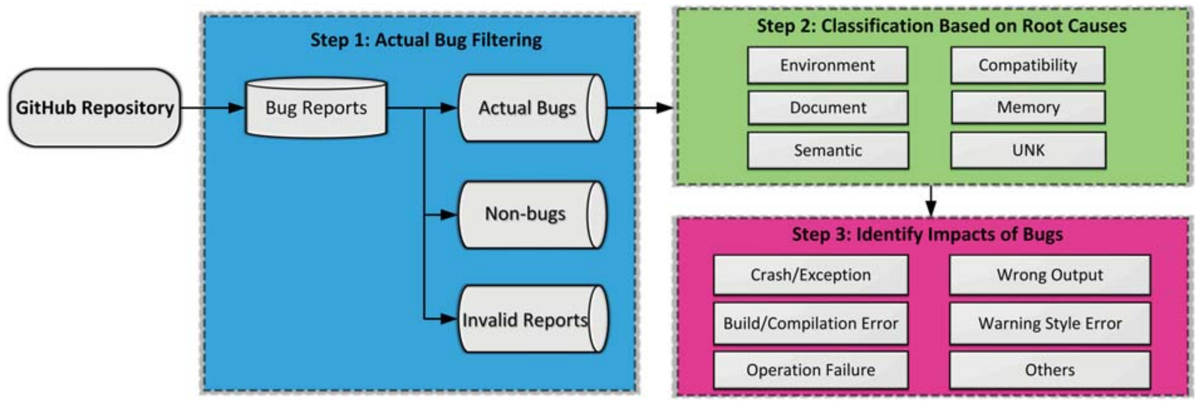

Fig. 1. The adopted procedure and workflow of bug report classification.

memory, *document*, and *semantic*. The distribution of different types of bugs is also studied. In addition, we have analyzed the evolution of proportions of different bug types over time. This RQ is answered in Section III.

RQ2: What are the impacts of different types of bugs in DL compilers? We have identified the impacts of bugs in DL compilers to understand the consequences caused by different types of bugs. Besides, we have studied the relationship between root causes and impacts of DL compiler bugs, which helps to understand which types of bugs are more likely to have a more severe impact. This RQ is answered in Section IV.

RQ3: How long does it take to fix bugs in DL compilers? We have performed a statistical analysis of the time it takes to fix the bugs in DL compilers. This RQ helps us notice the cost of repairing different types of bugs and therefore helps to allocate resources better. This RQ is answered in Section V.

Due to the page limit, in this paper, we provided summarized results and discussions and put more detailed information on the accompanied website of this paper, at https://sites.google.com/view/dlcompiler-common-bugs/, including the detailed information of all bug reports extracted from DL compilers' Github repositories, the closing time of all bug reports, and more details and concrete examples of each type of bugs, etc.

The rest of the paper is structured as follows. Section II presents the study design and methodology adopted in this paper. We give the answers for the three research questions in Section III, Section IV, and Section V, respectively. After discussing the threats to validity in Section VI and related work in Section VII, conclusions are given in Section VIII.

II. Study Design and Methodology

A. Data Collection

In this paper, we conduct our empirical study on five currently popular DL compilers, including TVM, Glow, nGraph, PlaidML, and Tensor Comprehensions (TC). We collect actual bug information of these compilers from their corresponding Github repositories. Table I summarizes the information of the collected bug reports.

In Table I, all bug reports are under the closed status. In Github's issue tracker, bug reports have two statuses: "open" and "closed". Considering that reports in the "open" state are still under discussion, their root causes may not have been found yet. Thus, only "closed" bug reports will be studied in this work. The second column shows the time period of bug reports. We collected as many closed bug reports as we could from TVM, Glow, nGraph, PlaidML, and TC. Among them, the time range of TVM, Glow, nGraph, and PlaidML is from 2017 to 2021, from when the first report is submitted to when the latest report is submitted. Different from the above four DL compilers, the time range of TC is from Feb. 2018 to Aug. 2019, a total of 18 months. It is because TC is no longer maintained, and no bug report has been closed since Aug. 27, 2019. After filtering, we collected 1,578 bug reports in TVM[1], 496 bug reports in Glow[2], 239 bug reports in nGraph[3], 290 bug reports in PlaidML[4], and 114 bug reports in TC[5]. Therefore, a total of 2,717 bug reports are obtained.

We obtain relevant bug information from these 2,717 bug reports through a designed Web-Crawler. The important information contained in bug reports is extracted, including both structural information and non-structural information. Among them, structural information includes the time of bug reports being submitted and closed. Non-structural information includes titles of bug reports, descriptions and comments of bugs, and linked commits and pull requests for resolving bugs.

B. Classification Procedure of Bug Reports

In this section, we manually review the description and comments of each bug report. Because the description in some bug reports can be ambiguous, we also explore related pull requests and commits for further confirmation. For a given bug report, the classification process is divided into three steps, as shown in Fig. 1.

Step 1: Actual bug filtering. A bug report should first be checked to make sure that it contains an actual bug. In our

[1]https://github.com/apache/tvm/issues
[2]https://github.com/pytorch/glow/issues
[3]https://github.com/NervanaSystems/ngraph/issues
[4]https://github.com/plaidml/plaidml/issues
[5]https://github.com/facebookresearch/TensorComprehensions/issues

TABLE I
SUMMARY OF SUBJECT OF DATA SET FOR STUDY (ALL BUG REPORTS ARE UNDER THE CLOSED STATUS)

Compiler	Time frame	# of reports
TVM	Oct. 13, 2017 - Jan. 7, 2021	1,578
Glow	Nov. 20, 2017 - Apr. 9, 2021	496
nGraph	Jul. 25, 2017 - Jan. 27, 2021	239
PlaidML	Oct. 20, 2017 - Apr.11, 2021	290
TC	Feb. 14, 2018 - Aug. 27, 2019	114
Total	2,717	

study, invalid bug reports will be filtered out first, i.e., the bug report contains too little information to determine whether it is a bug. For example, many bug reports were closed in DL compilers due to lack of activity, obsolescence, or cannot be reproduced. Then, valid bug reports that do not contain actual bugs will be filtered out, i.e., questions asked by users about the usage of the DL compilers, requests for new features or enhancements, operator errors and duplicate bug reports.

Step 2: Classification based on root causes. We carefully checked the bug report for each actual bug that was filtered from the last step to identify its root cause. According to our examination, we summarized five types of root causes, including *environment*, *compatibility*, *memory*, *document*, and *semantic*. The definition of each root cause will be given in section II-C. If there is not enough information to determine the root cause of a bug, it will be labeled as unknown (UNK).

Step 3: Identifying impacts of bugs. After classifying bugs according to their root causes, we aim to identify the impacts of different types of bugs. After analysis, five common impacts are found, including crash/exception, build/compilation error, operation failure, wrong output, and warning style error. If we cannot identify the impact of a bug, or if the impact does not belong to any of the five impacts we listed above, we will label it as others. The definition of each impact will be introduced in section II-D.

C. Root Causes of Bugs

This section aims to present some examples of different root causes (as listed in Table II). In addition, referring to the definitions in [30] and [31], we describe the following five root causes as follows.

- ***Environment*:** These errors are not directly related to DL compilers. They are errors that exist in frontend frameworks that export DL models (such as TensorFlow, MXNET, Pytorch, and Keras), underlying operating systems (such as MacOS), deployed tool-chains (such as LLVM, g++ and OpenCL) or dependent libraries (such as DMLC-Core).
- ***Compatibility*:** The program cannot normally run on specific hardware (e.g., CPU, MobileGPU), operating system (e.g., AMD, Windows), or a different endian type system (e.g., big-endian or little-endian). In addition, there are situations that a DL compiler cannot compile models from a specific framework (e.g., TensorFlow) or cannot run on a specified version of a toolkit (e.g., CUDA).
- ***Document*:** This kind of error is related to typos in examples, improperly recommended operations in instructions, outdated guidelines, missing items in guidelines or dead links in guidelines.
- ***Memory*:** These errors are caused by incorrect handling of memory objects, such as insufficient memory allocation, improper scheduling of shared memory or unreleased memory.
- ***Semantic*:** Inconsistent with the requirements or the programmer's intention and do not belong to the above categories. Including features that should be but not implemented, some boundary cases are incorrectly considered or ignored, improper handling of exceptions, incorrectly displayed output, equations processed improperly, and other *semantic* bugs that do not meet the design requirements.

D. Impacts of Bugs

In this section, we have investigated the impact of each bug to understand the consequences of bugs in DL compilers [32], [33]. The definition of each kind of impact is as follows.

- **Crash/exception:** If a DL compiler stops and exits unexpectedly, a crash/exception occurs. When this situation happens, the program typically throws an error message. For example, in Bug-1624 in TVM, an error was raised when the reporter uses auto-tvm to train a model that contained "conv2d_transpose".
- **Build/compilation error:** If a build or compilation error is printed, a build/compilation error occurs. For example, in Bug-4135 in Glow, the build of Glow failed when "-DGLOW_BUILD_TESTS=OFF".
- **Operation failure:** If unexpected behaviors, such as rejection of a task or multiple processing of a task, is encountered, it means an operation failure occurs. In DL compilers, installation failure, setup incomplete issue, and broke link caused download failure are considered as operation failures.
- **Wrong output:** If the program generates a wrong result and presents it to users, a wrong output occurs. For example, incorrect code output of DL compilers, wrong output from the scheduler, incorrect quantified nodes, not optimized graph output, incorrect classification results, and misleading log information, etc.
- **Warning style error:** If the program generates a warning message, a warning style error occurs. In addition, document issues such as incomplete and outdated documents, unnecessary warnings, performance degradation, and memory leak issues are considered as warning style errors.

III. DISTRIBUTION OF BUG TYPES

This section aims to answer RQ1. We first filter out the actual bugs from all reports and then classify them according to their root causes. Finally, the frequency distribution of bugs and the evolution trend of the proportions of different types of bugs are analyzed.

TABLE II
EXAMPLES OF ROOT CAUSES

Compiler	Bug ID	Root Cause	Description
TVM	998	Environment	"This might due to the problem of MXNet's DLPack layer creating wrong DLTensor"
nGraph	3907	Compatibility	"I faced an issue that as_type_ptr doesn't work for this cast: TensorIterator::InputDescription to TensorIterator::SliceInputDescription, ... This issue occurs only on Win10 platform."
Glow	366	Memory	"readPngImage() leaks memory when image fails to read"
PlaidML	237	Document	"OpenCL HAL example link is dead."
TC	193	Semantic	"autotuner should catch compilation failures"

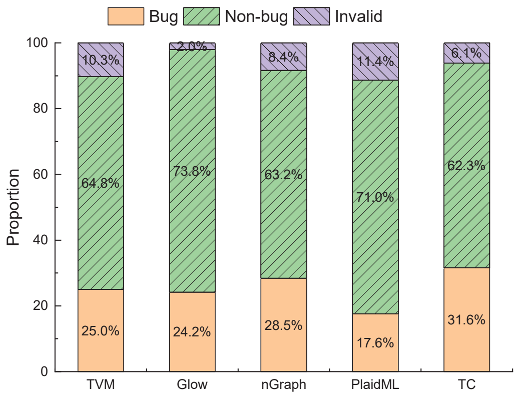

Fig. 2. Distribution of actual bugs, non-bugs, and invalid reports.

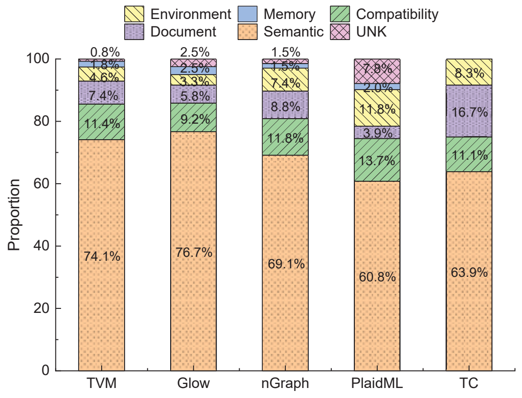

Fig. 3. Distribution of root causes of bugs.

A. Distribution of Actual bugs, Non-bugs, and Invalid Reports Among all Bug Reports

Finding #1: The ratio of actual bugs among all the submitted bug reports in DL compilers ranges from 17.6% to 31.6%.

From Fig. 2, we can see that among all the bug reports, the proportion of non-bugs is the highest. In TVM, Glow, nGraph, PlaidML, and TC, the percentages of non-bugs are 64.8%, 73.8%, 63.2%, 71.0%, and 62.3%, respectively. As for actual bugs, the proportion of actual bugs in TC is the highest, which is 31.6%, followed by nGraph (i.e., 28.5%), TVM (i.e., 25.0%), Glow (i.e., 24.2%), and PlaidML (i.e., 17.6%). In addition to actual bugs and non-bugs, the proportion of invalid reports in DL compilers is not negligible. In PlaidML, the proportion of invalid reports is the highest (i.e., 11.4%), followed by TVM (i.e., 10.3%). In nGraph and TC, the proportions of invalid reports are 8.4% and 6.1%, respectively. In Glow, the percentage of invalid reports is the lowest (i.e., 2.0%).

This finding shows that a large number of non-bugs and invalid reports were submitted, which could be a heavy burden for developers. According to our examination, in DL compilers, there are several typical situations in which non-bugs appear, including questions asked by reporters for the usage of DL compilers (e.g., in Bug-385 and Bug-493 in TVM, reporters asked "How to reproduce the Raspberry Pi experiment" and "How to install the libtvm.dll", respectively), reports submitted by developers related to the implementation of new features (for example, in Bug-3 and Bug-6 in nGraph, developers added a way to check code styling and added setup instructions for Ubuntu 14.04 LTS), feature requests proposed by users (for example, in Bug-104 and Bug-130 in TC, reporters asked developers to support TC on macOS and PyTorch master), and duplicate bug reports. Through manual inspection, the numbers of duplicate bug reports in TVM, Glow, PlaidML, and TC are 32, 9, 20, and 5, respectively. There is no duplicate bug report in nGraph.

B. Bug Type Distribution Among Different DL Compilers

Finding #2: The major bug types in DL compilers are *semantic* and *compatibility* bug.

Fig. 3 presents the distribution of the root causes of bugs in the five DL compilers we analyzed. According to the results, it is obvious that more than half of the bugs in DL compilers are *semantic* bugs. Glow has the highest proportion of *semantic* bugs, which is 76.7%, followed by TVM (i.e., 74.1%), nGraph (i.e., 69.1%), TC (i.e., 63.9%), and PlaidML (i.e., 60.8%). Compared with other software systems, the proportion of *semantic* bugs in DL compilers is lower than that in Mozilla and Apache. According to the study in [32], *semantic* bugs account for 87.0% in Mozilla and 82.5% in Apache. However, the results of DL compilers are similar to the blockchain systems. In [31], the authors investigated the frequency distribution of bug categories in 8 blockchain

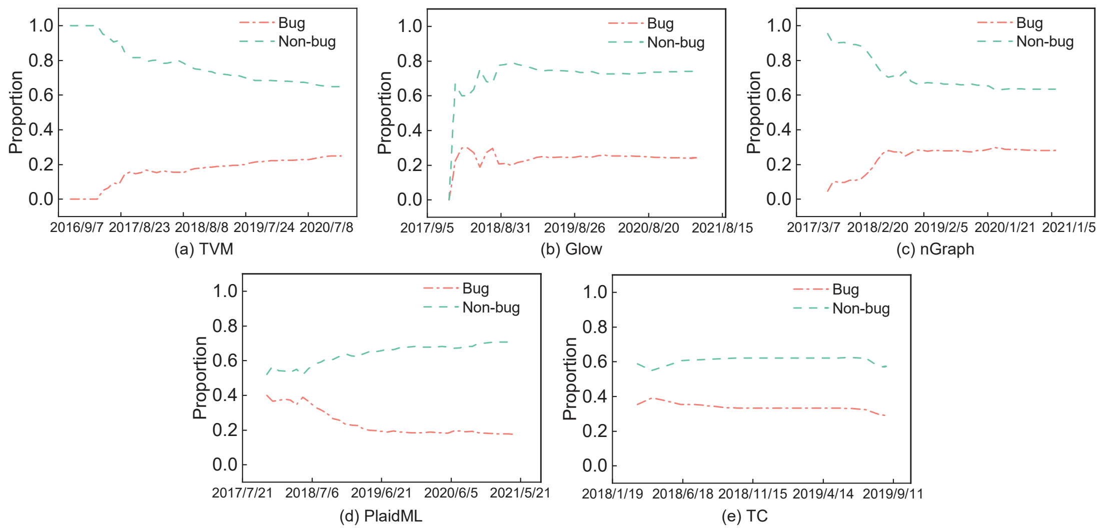

Fig. 4. Evolution of proportions of actual bugs and non-bugs.

systems. They found that 67.23% of the bugs in blockchain systems are *semantic* bugs.

Compatibility is the second most frequently appeared root cause in TVM, Glow, nGraph, and PlaidML, with the proportions being 11.4%, 9.2%, 11.8%, and 13.7% respectively. Different from the four DL compilers mentioned above, the second most common bug type in TC is *document*, accounting for 16.7%. *Environment* and *memory* bugs only account for a small portion. The percentages of *environment* bugs in TVM, Glow, nGraph, PlaidML, and TC are 4.6%, 3.3%, 7.4%, 11.8%, and 8.3%, respectively. *Memory* bugs account for 1.8%, 2.5%, 1.5%, and 2.0% in TVM, Glow, nGraph, and PlaidML, respectively. And no *memory* bug is reported in TC.

Through further examination, we found that in DL compilers, a number of *semantic* bugs are related to missing cases in the implementation of functions, also known as neglected condition bugs [34]. For example, in Bug-1592 in TVM, it was stated that "[Pass] Missing CHECK in storage rewrite". In this bug, if "max_num_bits<640", an allocation that exceeds the memory bounds will be triggered. In Bug-6239 in TVM, "NotImplementedError" occurred due to missing operator "is_floating_point". Installation and setup issues are also common occurrences in *semantic* bugs. For example, in Bug-306 in PlaidML, the PlaidML-setup could only recognize the CPU and fail to find the GPU. Exception handling is another important subtype of *semantic* bug. For example, in Bug-193 in TC, warnings about preconditions not being checked should be suppressed because non-initialized reductions are unnecessary when developing autotuning. In Bug-2855 in Glow, "RemoveNetwork" failed silently instead of returning error information about the failure. Finally, some system-specific *semantic* bugs occurred in DL compilers. For example, in Bug-1713 in TVM, the weight shape in the operator conv2 was incorrectly processed. As a result, the wrong weight's shape was obtained. In Bug-2013 in Glow, the graph optimizer failed to optimize the non-inverses consecutive transposes.

As for *compatibility* bugs, there can be many reasons for these bugs in DL compilers, including conflicts with DL framework frontends, hardware backends, operating systems, and hardware devices. Taking the TVM compiler as an example, it was stated in Bug-6287 that "TVMError: relay.frontend.from_keras() fails with models compiled in TensorFlow 2.3". This bug showed a conflict with the TensorFlow framework because TensorFlow 2.3 didn't support the attributes "node_indices" and "tensor_indices" in the "Node class" anymore. Bug-2355 is a *compatibility* bug caused by a Mobile GPU backend conflict. When TVM was deployed to Mobile GPUs, incorrect "Vulkan results" would be obtained. A windows *compatibility* bug happened in Bug-1007. It was reported that "ModuleNotFoundError: No module named "fcntl" when built TVM on the Windows operating system."

C. Evolution of Proportions of Different Types of Bugs

Finding #3: The evolution trends of various bugs differ greatly among different DL compilers.

Fig. 4 depicts the evolution of the proportions of actual bugs and non-bugs in TVM, Glow, nGraph, PlaidML, and TC. The results show that in TVM and nGraph, the proportions of actual bugs tend to increase over time, while the percentages of non-bugs tend to decrease. Contrary to TVM and nGraph, in PlaidML and TC, the proportions of actual bugs tend to decrease over time, while the ratios of non-bugs gradually increase. In Glow, the proportions of actual bugs and non-bugs have no obvious evolutionary trend.

In Fig. 5, we further present the evolution trend of the proportions of different bug types over time. In TVM, the proportions of different types of bugs have significant evolution trends. As time evolves, the proportions of *environment*, *memory* and *compatibility* bugs show an upward trend while *document* and *semantic* bugs show a downward trend. In Glow,

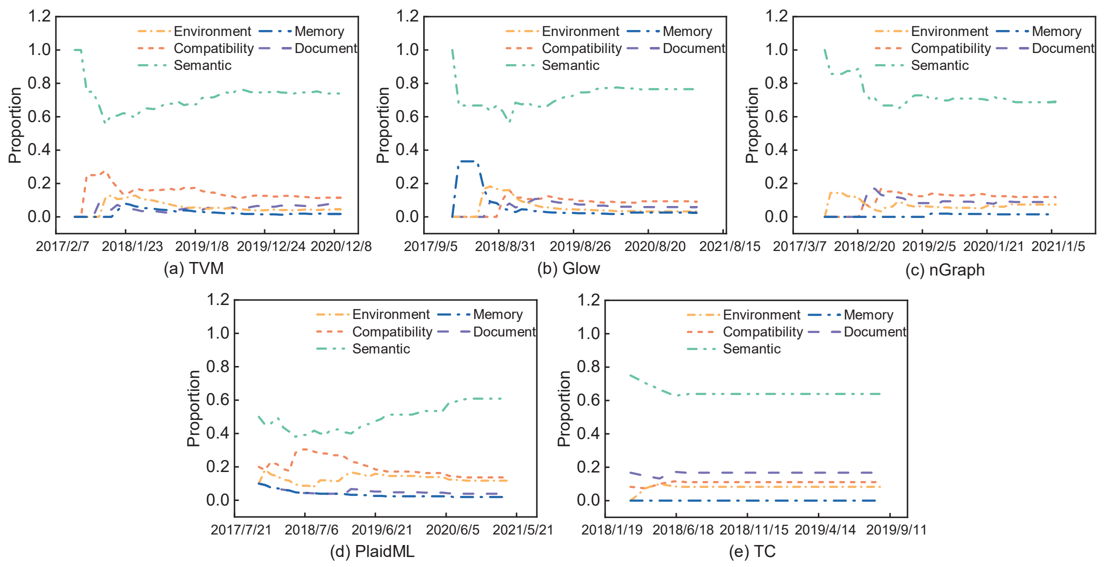

Fig. 5. Evolution of proportions of different bug types.

the ratios of *environment* and *memory* bugs tend to decrease. In contrast, the proportion of *semantic* bugs tends to increase, and there is no significant trend for *compatibility* and *document* bugs. In nGraph, the ratio of *memory* bugs has increased significantly over time, but the percentage of *semantic* bugs decreases. As for *environment*, *compatibility*, and *document* bugs, trends are not obvious. The ratios of *memory*, *compatibility* and *document* bugs decrease obviously in PlaidML, while the *semantic* bugs' proportion is significantly increased, and the trend of *environment* bugs is not obvious. In TC, no significant trend is observed for all types of bugs. Above conclusions have been tested through the Mann-Kendall [35] trend test ($alpha = 0.05$) and the results are shown in Table III.

D. Implications

Since non-bugs account for more than half of all bug reports, we first propose suggestions from two aspects to help developers deal with non-bugs. On the one hand, a question tracker could be provided for users to submit their questions and find relevant answers. For example, in TVM, it is encouraged to post general questions on https://discuss.tvm.ai, where more people are expected to join the discussion of the questions. In addition, stack overflow[6] could be another important place for asking questions and finding detailed answers. On the other hand, a tool can be integrated to identify duplicate bug reports. For example, in [36], a just-in-time duplicate detection method was proposed to prevent duplicate bug reports from being submitted by means of the continuous query. By recommending relevant bug reports to reporters, it helps users find solutions effectively and reduces the developers' workloads at the same time.

As for actual bugs, more effort should be made on *semantic* and *compatibility* bugs because these two types of bugs are the most numerous in DL compilers. To address *semantic* bugs, automatic bug detection tools could be used. For example, MUVI (Multi-variable inconsistency) [37], a method developed to automatically detect *semantic* bugs by identifying the multi-variable access correlations from programs. In addition, since the number of *compatibility* bugs is the second most in DL compilers, specific methods for detecting *compatibility* bugs in DL compilers should be developed. For example, researchers have done quite a few work to study the detection methods of *compatibility* bugs in Android Apps [38], [39].

[6]https://stackoverflow.com/

IV. Impacts of Different Types of Bugs

In this section, the results of RQ2 are presented. We analyze the impacts of bugs to understand the consequences caused by bugs in DL compilers.

A. Distribution of Impacts Caused by Bugs in Different DL Compilers

Finding #4: More than one-third of bugs in DL compilers result in crashes or exceptions.

According to the results in Fig. 6, the major impact of bugs in DL compilers is crash/exception. In TVM, 47.8% of bugs result in crashes/exceptions, which is the highest among the five DL compilers. It is 46.3% in PlaidML, 45.0% in Glow, 36.1% in TC, and 35.3% in nGraph. Crashes/exceptions usually occur when the DL compilers stop and exit unexpectedly. For example, in TVM, an error was printed when using cache_read in order in the Bug-802. Similarly, in Bug-1043 in TVM, an error occurred when using cache_read in the situation of reusing the same buffer in multiple stages. Except for crash/exception, build/compilation error and wrong output are two other important impacts. 29.4% of bugs in nGraph cause build/compilation errors, which is the highest among the five DL compilers. In TC, 19.4% of bugs can

TABLE III
RESULTS OF MANN-KENDALL TREND DETECTION FOR FIG. 4 AND FIG. 5 ("$\nearrow$" MEANS UPWARD TREND, "$\searrow$" MEANS DOWNWARD TREND, AND "$\longrightarrow$" MEANS NO SIGNIFICANT TREND)

Compiler	Bug type	p value	Trend
TVM	Bug	<0.001	$\nearrow$
	Bon-bug	<0.001	$\searrow$
	Environment	0.001	$\searrow$
	Memory	0.02	$\searrow$
	Compatibility	<0.001	$\searrow$
	Document	<0.001	$\nearrow$
	Semantic	<0.001	$\nearrow$
Glow	Bug	0.95	$\longrightarrow$
	Non-bug	0.88	$\longrightarrow$
	Environment	<0.001	$\searrow$
	Memory	<0.001	$\searrow$
	Compatibility	0.94	$\longrightarrow$
	Document	0.11	$\longrightarrow$
	Semantic	<0.001	$\nearrow$
nGraph	Bug	< 0.001	$\nearrow$
	Non-bug	< 0.001	$\searrow$
	Environment	0.49	$\longrightarrow$
	Memory	0.03	$\nearrow$
	Compatibility	0.85	$\longrightarrow$
	Document	0.53	$\longrightarrow$
	Semantic	<0.001	$\searrow$
PlaidML	Bug	< 0.001	$\searrow$
	Non-bug	< 0.001	$\nearrow$
	Environment	0.12	$\longrightarrow$
	Memory	<0.001	$\searrow$
	Compatibility	<0.001	$\searrow$
	Document	<0.001	$\searrow$
	Semantic	<0.001	$\nearrow$
TC	Bug	<0.001	$\searrow$
	Non-bug	0.02	$\nearrow$
	Environment	0.93	$\longrightarrow$
	Compatibility	0.35	$\longrightarrow$
	Document	0.70	$\longrightarrow$
	Semantic	0.32	$\longrightarrow$

generate and present wrong results to users. It is higher than that of Glow (i.e., 15.8%), TVM (i.e., 11.4%), nGraph (i.e., 7.4%), and PlaidML (i.e., 5.6%). Taking Bug-47 in PlaidML as an example, the computation results obtained after using PlaidML were different from those obtained directly from the TensorFlow framework (i.e., a wrong output was given).

In order to analyze which kind of bug is more likely to cause severe consequences, we have investigated the impact distribution among different bug types (see Fig. 7). According to the results, the major impact of the *environment* bug is crash/exception. 47.22% (17 out of 36) of *environment* bugs would cause crashes/exceptions, followed by build/compilation errors (i.e., 19.44%), wrong outputs (i.e., 11.11%), warning style errors (i.e., 5.56%), and operation failures (i.e., 2.78%). *Memory* bugs only result in crashes/exceptions and warning style errors. The proportions of both are 50%. Crash/exception and build/compilation error are the two major impacts caused by *compatibility* bugs. The

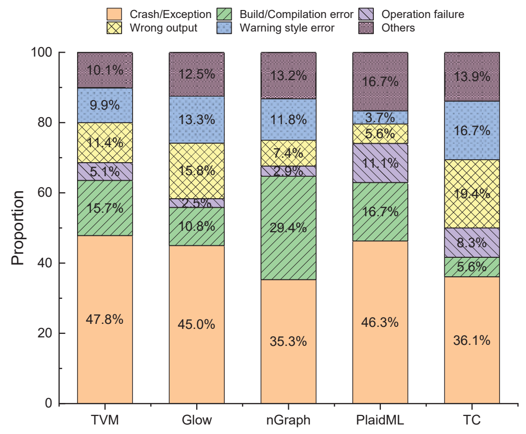

Fig. 6. The distribution of impacts in DL compilers.

proportions are 40% (30 out of 75) and 35.06% (28 out of 75), respectively. In addition, 8% and 6.67% of *compatibility* bugs lead to operation failures and wrong outputs, respectively. As for *document* bugs, 72% (36 out of 50) of them result in warning style errors. Almost half of the *semantic* bugs, i.e., 49.07% (238 out of 485), lead to crashes/exceptions. The proportions of wrong outputs, build/compilation errors, warning style errors and operation failures caused by *semantic* bugs are 14.23%, 14.02%, 5.57%, and 4.54%, respectively.

For further analysis, we have calculated the $lift$ correlation [40] between bug types and impacts. $lift$ is a statistical metric used to indicate the correlation between two types of bugs. If the $lift$ value between two types equals 1, it means that there is no correlation between them. If the $lift$ value between two types is larger than 1, it means that the two types are positively correlated. Conversely, if the $lift$ value between two types is less than 1, there is a negative correlation between the two types. Results of the $lift$ correlation between bug types and impacts are presented in Table IV. Numbers greater than 1 (i.e., with positive correlation) are highlighted in bold font.

B. *Correlation Between Bug Types and Impacts*

Finding #5: *Environment* and *compatibility* bugs are prone to result in build/compilation errors; *memory* and *document* bugs are prone to lead to warning style errors; *semantic* bugs are more likely to cause crashes or exceptions.

Table IV summarizes the values of $lift$ correlation between bug types and impacts. The results show that *environment* and *compatibility* bugs are most prone to result in build/compilation errors. The $lift$ value between *environment* bug and build/compilation error is 1.29. Between *compatibility* bug and build/compilation error, it is 2.32. *Memory* bugs are most likely to cause warning style errors, and the $lift$ correlation between *memory* bugs and warning style errors is 3.55. The reason is that many *memory* bugs are memory leaks that related to memory resource release failure. The running of the program may not be disturbed immediately, but the

TABLE IV
THE $lift$ CORRELATIONS BETWEEN BUG TYPES AND IMPACTS

Type	Environment	Memory	Compatibility	Document	Semantic
Crash/exception	**1.04**	0.97	0.83	0.56	**1.09**
Build/compilation error	**1.29**	0.41	**2.32**	0	0.93
Operation failure	0.54	**1.22**	**1.52**	**1.45**	0.88
Wrong output	0.89	**1.22**	**1.52**	**1.45**	0.88
Warning style error	0.53	**3.55**	0.61	**4.52**	0.53

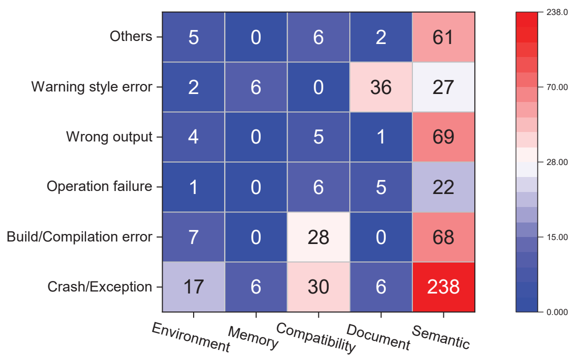

Fig. 7. The impacts of different bug types.

code still needs improvement. For example, in Bug-2646 in nGraph, when the "remove_compiled_function" was called on the CPU backend, "cleanup_runtime_context()" was not called to clean up the runtime context and led to a memory leak. In addition to *memory* bugs, *document* bugs are also more likely to cause warning style errors. The correlation between *document* bugs and warning style errors is 4.52, indicating that a strong correlation exists between warning style errors and *document* bugs. For example, in Bug-29 and Bug-176 in TC, the typos in examples make it hard for users to understand. Finally, *semantic* bugs are prone to cause crashes/exceptions. The $lift$ value between *semantic* bug and crash/exception impact is 1.09. Therefore, we consider these two types to be weakly correlated.

C. Implications

A large number of bugs in DL compilers lead to serious consequences, such as crashes or exceptions, accounting for 35.3-47.8% of all the actual bugs. In order to improve the quality of the DL compilers, a lot of effort should be put into handling the crash or exception errors. For example, crash reporting tools similar to Mozilla Socorro [41] could be embedded in DL compilers to collect information relevant to crashes. Using these tools, crash reports will be generated when programs stop running normally in the user environment. The information available in crash reports could be used to locate and rank buggy files. For example, Wang et al. [41] proposed an algorithm based on crash correlation groups, which locates and ranks buggy files by analyzing stack traces in correlated crash types. In addition, the information in crash reports could be used to predict top crashes in a specific version of the DL compiler. For example, in [42], the authors employed features extracted from crash reports and source code to train a machine learning model to effectively predict the top crashes before a new version release.

In addition to crash/exception, build/compilation error is another important impact of bugs in the DL compilers. Developers spend a substantial amount of time repairing code that does not compile. For example, according to our analysis, it takes an average of 34.3 days to fix a build/compilation error in TVM. In order to reduce the burden on developers, we recommend using automatic build/compilation error repair methods. For example, DeepDelta [43] was proposed to repair the build-time compilation errors automatically. It learns the fix patterns of errors through learning the changes between failed and resolved snapshots of code.

V. FIXING TIME OF BUGS

To answer RQ3, we analyze the time required to close bug reports in this section.

A. Fixing Time of Actual Bugs, Non-bugs, and Invalid Reports

Finding #6: It takes the most time to close bug reports in nGraph and the least time to close bug reports in TVM.

In this section, we have calculated the average fixing time of actual bugs, non-bugs, and invalid reports in TVM, Glow, nGraph, PlaidML, and TC. According to the results in Fig. 8, it is obvious that closing reports in nGraph takes more time than closing reports in the other four compilers. In nGraph, it takes the most time (i.e., 433.1 days on average) to close invalid reports. The average time spends on closing non-bugs is 213.7 days. However, the time required to fix actual bugs is the least, which is 164.8 days on average. Similar to nGraph, invalid reports also take the most time (i.e., 118.2 days on average) to be closed in PlaidML, while the time required to close actual bugs and non-bugs is 51.2 days and 49.9 days, respectively. Bug reports in TVM are closed fastest among all the five DL compilers we analyzed. In TVM, it takes an average of 20.6 days to fix an actual bug, 36.0 days to close a non-bug and 11.8 days to close an invalid report. From the above analysis, we can also conclude that in addition to fixing actual bugs, amounts of time are spent on dealing with non-bugs and invalid reports. Taking nGraph as an example, the time it takes to close invalid reports is 2.6 times the time it takes to fix actual bugs. A similar situation happens in

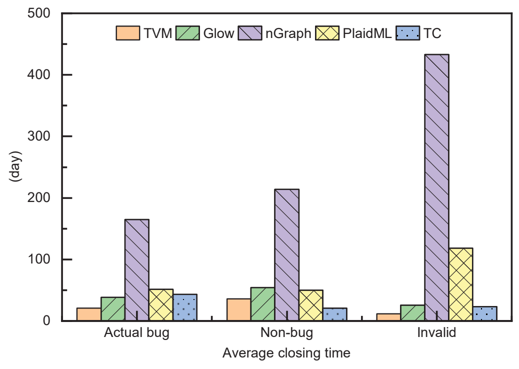

Fig. 8. Average closing time of actual bugs, non-bugs and invalid reports.

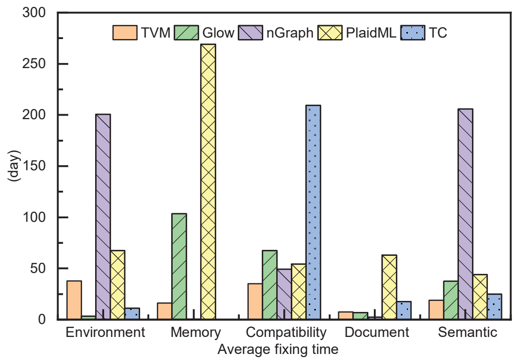

Fig. 9. Average fixing time of different types of bugs.

PlaidML. The time it takes to close invalid reports is 2.3 times the time it takes to fix actual bugs.

B. Fixing Time of Environment, Compatibility, Memory, Document, and Semantic Bugs

Finding #7: It takes the least time to resolve *document* bugs.

In this section, we study the difference between the fixing time of different types of bugs. From Fig. 9, we can observe that *document* bugs often consume much less time to be fixed than the other four types of bugs. In PlaidML, the fixing time of *document* bugs is the longest among the five DL compilers, which is 63.0 days on average, followed by TC (17.6 days on average), TVM (7.4 days on average), Glow (6.8 days on average), and nGraph (2.3 days on average). The time it takes to fix *environment*, *memory*, *compatibility*, and *semantic* bugs is quite different for different DL compilers. In TVM, it takes the most time to fix *environment* bugs (on average of 37.8 days), which is more than 5 times the fixing time of *document* bugs and more than 2 times the fixing time of *memory* bugs. In Glow and PlaidML, the fixing time of *memory* bugs is the longest, which is 103.6 days and 269.2 days, respectively. In nGraph, it takes 205.9 days on average to fix *semantic* bugs. However, it only takes an average of 0.2 days to resolve *memory* bugs and an average of 2.3 days to solve *document* bugs. As for TC, *compatibility* bugs spend the most time to fix (i.e., 209.5 days on average), followed by *document* bugs (17.6 days) and *environment* bugs (11.2 days).

C. Implication

Due to the longest fixing time of bugs in nGraph, we have conducted a more in-depth study on bug reports in nGraph. After the manual analysis, we find that 21.9% of bug reports in nGraph take more than a year to close. The main reason for the extremely long fixing time is the failure to respond to and close bug reports in a timely manner. For example, in Bug-552 in nGraph, the last comment was submitted on Feb. 28, 2018, but it was not closed until Oct. 7, 2020, which took 951.7 days. The same thing happens in Bug-765, Bug-957, and Bug-1077. In Bug-2917, it took developers 46.8 days to first respond and assign the report, and then it took another 465.8 days to close the bug report. We suggest deploying a bot in nGraph to help with the maintenance of issues in the Github repository. For example, Tensorflow-Butler is used in TensorFlow's Github repository to assign bug reports to the corresponding developers and update the state of bug reports to improve the efficiency of processing reports.

Besides the suggestions for developers, we recommend that users should submit high-quality bug reports to help developers reproduce bugs as soon as possible. It is difficult to reproduce a bug only with the error message. Detailed environment information is required, such as the hardware, the operating system, and the specific version of the DL compiler. In addition to the environment information, a self-contained example would be helpful and necessary to accelerate the fixing process.

VI. Threats to Validity

Similar to other empirical research, our study naturally has potential validity issues. We identify potential threats to the effectiveness of our study from the following three aspects:

Threats to Construct Validity. In this study, we concentrate on closed bug reports because bug reports that have not been closed are still under discussion and contain incomplete information. If future closed bug reports are considered, the distribution of bug types could be influenced.

Threats to Internal Validity. In the process of manual classification, We tried our best to avoid classification errors. We carefully checked the information contained in bug reports, including bug descriptions, comments, related pull requests, and commits. Two authors separately took the classification of all bug reports to reduce the threat. During the process, cross-checks were performed, and conflict cases were eliminated through discussion to reach a consensus.

Threats to External Validity. The bug reports studied in this paper are collected from TVM, Glow, nGraph, PlaidML, and TC. Although they are widely-used DL compilers, some findings and implications may not be generalized to other DL compilers. To reduce this threat, we try not to extend the conclusions to other DL compilers, nor do we intend to provide any general implication to all DL compilers. Even though, we believe that this paper still provides many interesting findings and suggestions for DL compiler developers and users, the five studied DL compilers are also among the most popular ones.

VII. Related Work

Bugs in Traditional Compilers and DL Frameworks. Compilers are foundational and widely-used software [28], [44], the bugs in which can have a significant impact since they can almost affect all software build on them. Especially for safety- and security-critical applications, the consequences can be disastrous [45]. Therefore, it is essential to detect bugs in compilers. However, compiler bugs are difficult to recognize because they usually manifest indirectly as application failures. It is hard for developers to determine whether a software failure is caused by the program they are developing or the compilers they are using [46]–[48].

An in-depth understanding of bugs in compilers can help detect and fix them. In [45], the authors conducted the first empirical study on the characteristics of the bugs in two traditional compilers, GCC and LLVM. They studied the location of bugs, properties of bug-revealing test cases and the bug fixes, duration of bugs, and priorities of bugs. To improve the quality of traditional compilers, some methods have been proposed [49]–[52]. In [53], the authors conducted an empirical study on the testing methods of traditional compilers from four aspects, including approaches of constructing test programs [54], methods to address the test-oracle problems [54], [55], methods for optimizing the test process [56], and methods developed for post-processing test results [57]. Different from the above research, we focus on studying the characteristics of bugs in DL compilers instead of traditional compilers in this paper.

In addition, some recent progress has been made to analyze bug characteristics in DL frameworks [58]–[61]. Islam et al. [58] studied bugs in Caffe, Keras, TensorFlow, Theano, and Torch from the perspectives of bug type, root cause, and effects of bugs. Zhang et al. [59] performed an empirical study on deep learning applications programmed on the TensorFlow framework. They collected 175 bugs from stack overflow and Github to analyze the symptoms and root causes of bugs. Different from DL frameworks, DL compilers aim to provide a general-purpose optimization and compilation of a DL model for diverse target devices to accelerate DL model deployment at the industrial production level. Our study also confirms that the common bug characteristics of DL compilers distribute quite differently from DL frameworks, calling for attention for DL compiler developers and researchers to provide quality assurance techniques.

Deep Learning Compilers. With the growing demand to deploy various DL models on diverse DL hardware, multiple DL compilers have been proposed from both industry and academia, such as TVM [22], XLA [23], Glow [24], nGraph [25], PlaidML [26] and Tensor Comprehensions (TC) [27]. Among them, XLA [23] is developed by the Google team and is a compiler for TensorFlow. XLA uses JIT compilation techniques to analyze the TensorFlow graph created by the user at runtime. TVM [22] is an end-to-end optimizing compiler stack proposed by Apache. It provides a machine learning-based optimization system that can search for optimized tensor operators automatically. Glow [24] and TC [27] are two DL compiler released by Facebook. The main technique of Glow is graph-lowering, which could be used to generating efficient code. TC provides a high-level language for neural network developers to express their networks. PlaidML [26] and nGraph [25] are two DL compilers designed by Intel. nGraph consumes the compute graph obtained from the DL framework to construct the nGraph intermediate representation (IR) and then lowers that to different backends, such as cuDNN and MKL-DNN. PlaidML, as a compiler for DL, is also available as a component of the Intel nGraph compiler stack, which is useful mainly for supporting a new kernel.

Although the functional components (such as frontend and backend) in DL compilers are similar to traditional compilers, these two types of compilers have different characteristics. Focusing on the design architecture of DL compilers, authors in [29] performed a survey of TVM, nGraph, TC, Glow, and XLA. The authors dissected the commonly adopted design architecture of the existing DL compilers and analyzed the critical design components. In addition, they performed a quantitative performance comparison among DL compilers. However, to the best of our knowledge, the research on the bug analysis of DL compilers is still at a very early stage. This paper makes an early attempt and conducts a comprehensive empirical study on bugs in five DL compilers, including TVM, Glow, nGraph, PlaidML, and TC.

VIII. Conclusions

This paper performs a large-scale study on actual bugs in five widely-used deep learning compilers, including TVM, Glow, nGraph, PlaidML, and TC. We manually analyzed 2,717 bug reports collected from their corresponding Github repositories, which are actual bug information submitted by users and developers. Our analysis is conducted from three perspectives, including ❶ the frequency distribution of different types of bugs and the proportion evolution of bugs with the increase of time, ❷ severe consequences caused by bugs and the correlation between bug types and impacts, and ❸ the time required to close bug reports. Our study found that the main root causes of bugs in DL compilers are *semantic* and *compatibility* bugs. Over one-third of bugs in DL compilers would result in crashes/exceptions. It takes the longest time to fix bugs in the nGraph compiler among all the five compilers studied in this paper. Finally, practical implications were provided for both developers and users to develop and use DL compilers with higher quality.

Acknowledgment

This work was supported by the National Natural Science Foundation of China under Grant 61772055 and Grant 61872169.

References

[1] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah, "A survey of deep learning applications to autonomous vehicle control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 2, pp. 712–733, 2021.

[2] M. Mohammadi, A. Al Fuqaha, M. Guizani, and J. S. Oh, "Semisupervised deep reinforcement learning in support of iot and smart city services," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 624–635, 2017.

[3] A. Puente Castro, E. Fernandez Blanco, A. Pazos, and C. R. Munteanu, "Automatic assessment of alzheimer's disease diagnosis based on deep learning techniques," *Computers in Biology and Medicine*, vol. 120, p. 103764, 2020.

[4] H. Panwar, P. Gupta, M. K. Siddiqui, R. Morales Menendez, and V. Singh, "Application of deep learning for fast detection of covid-19 in x-rays using ncovnet," *Chaos, Solitons & Fractals*, vol. 138, p. 109944, 2020.

[5] F. Murat, O. Yildirim, M. Talo, U. B. Baloglu, Y. Demir, and U. R. Acharya, "Application of deep learning techniques for heartbeats detection using ecg signals-analysis and review," *Computers in biology and medicine*, p. 103726, 2020.

[6] Q. Guo, S. Jin, M. Li, Q. Yang, K. Xu, Y. Ju, J. Zhang, J. Xuan, J. Liu, Y. Su *et al.*, "Application of deep learning in ecological resource research: Theories, methods, and challenges," *Science China Earth Sciences*, pp. 1–18, 2020.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

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

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[10] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021.

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[14] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 2135–2135.

[15] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "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*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 750–762. [Online]. Available: https://doi.org/10.1145/3368089.3409759

[16] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 810–822.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[18] H. Liao, J. Tu, J. Xia, and X. Zhou, "Davinci: A scalable architecture for neural network computing," in *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, 2019, pp. 1–44.

[19] A. Kingsley-Hughes, "Inside apple's new a11 bionic processor," *ZDNet, September*, 2017.

[20] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, and S. Avancha, "Intel nervana neural network processor-t (nnp-t) fused floating point many-term dot product," in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2020, pp. 133–136.

[21] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. V. Gool, "AI benchmark: All about deep learning on smartphones in 2019," *CoRR*, vol. abs/1910.06663, 2019.

[22] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: end-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, vol. 11, p. 20, 2018.

[23] C. Leary and T. Wang, "Xla: Tensorflow, compiled," *TensorFlow Dev Summit*, 2017.

[24] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.

[25] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi *et al.*, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," *arXiv preprint arXiv:1801.08058*, 2018.

[26] Intel, "Plaidml," 2018. [Online]. Available: https://www.intel.ai/reintroducing-plaidml

[27] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[28] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.

[29] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2020.

[30] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, "An empirical study on real bugs for machine learning programs," in *2017 24th Asia-Pacific Software Engineering Conference*. IEEE, 2017, pp. 348–357.

[31] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: a large-scale empirical study," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*. IEEE, 2017, pp. 413–424.

[32] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[33] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, "A comprehensive study on real world concurrency bugs in node.js," *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 520–531, 2017.

[34] R. Y. Chang, A. Podgurski, and J. Yang, "Discovering neglected conditions in software by mining dependence graphs," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 579–596, 2008.

[35] H. B. Mann, "Nonparametric tests against trend," *Econometrica: Journal of the econometric society*, pp. 245–259, 1945.

[36] A. Di Sorbo, J. Spillner, G. Canfora, and S. Panichella, "Won't we fix this issue? qualitative characterization and automated identification of wontfix issues on github," *arXiv preprint arXiv:1904.02414*, 2019.

[37] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 103–116.

[38] H. Huang, L. Wei, Y. Liu, and S.-C. Cheung, "Understanding and detecting callback compatibility issues for android applications," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 532–542.

[39] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in android apps," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 167–177.

[40] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[41] S. Wang, F. Khomh, and Y. Zou, "Improving bug management using correlations in crash reports," *Empirical Software Engineering*, vol. 21, no. 2, pp. 337–367, 2016.

[42] D. Kim, X. Wang, S. Kim, A. Zeller, S. C. Cheung, and S. Park, "Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 430–447, 2011.

[43] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: learning to repair compilation errors," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 925–936.

[44] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[45] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 294–305.

[46] Y. Tang, Z. Ren, W. Kong, and H. Jiang, "Compiler testing: a systematic literature analysis," *Frontiers of Computer Science*, vol. 14, no. 1, pp. 1–20, 2020.

[47] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 65–76, 2015.

[48] A. Groce, C. Zhang, M. A. Alipour, E. Eide, Y. Chen, and J. Regehr, "Help, help, i'm being suppressed! the significance of suppressors in software testing," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 390–399.

[49] H. Jiang, Z. Zhou, Z. Ren, J. Zhang, and X. Li, "Ctos: Compiler testing for optimization sequences of llvm," *IEEE Transactions on Software Engineering*, 2021.

[50] B. Jiang, X. Wang, W. Chan, T. Tse, N. Li, Y. Yin, and Z. Zhang, "Cudasmith: A fuzzer for cuda compilers," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 861–871.

[51] R. Schumi and J. Sun, "Spectest: Specification-based compiler testing," *Fundamental Approaches to Software Engineering*, vol. 12649, p. 269, 2021.

[52] J. Chen, H. Ma, and L. Zhang, "Enhanced compiler bug isolation via memoized search," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 78–89.

[53] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[54] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[55] K. Nakamura and N. Ishiura, "Random testing of c compilers based on test program generation by equivalence transformation," in *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 2016, pp. 676–679.

[56] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction: delta debugging, even without bugs," *Software Testing, Verification and Reliability*, vol. 26, no. 1, pp. 40–68, 2016.

[57] J. Holmes and A. Groce, "Causal distance-metric-based assistance for debugging after compiler fuzzing," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 166–177.

[58] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "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*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 510–520.

[59] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–140.

[60] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1110–1121.

[61] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 788–799.