ORTHRUS: Detecting Deep Learning Compiler Bugs via Optimization Resistance Transformations

Tongwei Zhang Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
twzhang99@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—Deep learning compilers are essential for deploying deep learning applications across heterogeneous hardware platforms. To improve execution efficiency, they employ sophisticated optimizations, which inevitably introduce bugs due to their considerably large code size and complex logic. Therefore, effectively detecting optimization bugs is essential to guarantee the correctness and trustworthiness of deep learning compilers.**

In this paper, we present ORTHRUS, an automatic approach to effectively detect optimization bugs in deep learning compilers. Conceptually, our approach develops a non-optimizing reference compiler to an optimizing compiler, then to detect optimization bugs by comparing the discrepancies in the two compilers' outputs. Obtaining a non-optimizing reference compiler is challenging, because existing deep learning compilers provide limited control over optimizations. We thus propose a novel approach dubbed optimization resistance transformation that structurally transforms an input deep learning model from an optimizable form to an unoptimizable form that the deep learning compiler can no longer perform the potential optimizations. We build a prototype for our approach and evaluate it in an extensive testing campaign on two widely-used deep learning compilers TVM and ONNXRuntime. ORTHRUS detects 21 bugs, of which 9 are non-crash optimization bugs and 1 is missed by the state-of-the-art tool NNSmith even with its cross reference feature enabled. Meanwhile, ORTHRUS introduces negligible execution overhead.

***Index Terms*—deep learning compilers, compiler testing, optimization bugs**

I. Introduction

Over the past decade, deep learning compilers, such as XLA [1], TensorRT [2], ONNXRuntime [3], and TVM [4], have emerged as essential techniques to fulfill the ever-increasing demand for efficient and scalable execution of deep learning models, by compiling these models to target code on diverse heterogeneous architectures including CPU, GPU, ASIC, and FPGAs, among others. To generate efficient target code that fully exploits the underlying hardware's computing capability, deep learning compilers employ sophisticated optimizations such as graph rewriting, operator fusion, and tensor optimizations during its translation of input tensors into low-level code [5]. Unfortunately, while critical and powerful, these optimizations inevitably contain bugs [6] [7] [8] [9], which not only undermine the trustworthiness of the deep learning compilers themselves but also affect downstream applications relying on deep learning compilers. Therefore, detecting deep learning compiler optimization bugs is important and imperative.

* The corresponding author.

Despite this importance and imperative, effectively detecting optimization bugs in deep learning compilers remains challenging. First, deep learning compilers leverage unique and complicated optimizations such as graph rewriting and operator fusion on distinct and deep learning-specific intermediate representation (IRs) such as Relay [10] in TVM and MLIR [11] in IREE. Therefore, these unique characteristics of deep learning compilers makes it challenging to directly apply bug detection techniques that have been proposed for traditional non-deep learning compilers [12] [13]. Second, optimization bugs in deep learning compilers may lead to crashes [6] [7] and logic bugs [8] [9]. While detecting crash bugs is relatively straightforward because they often present clear symptoms such as compiler crashes or segmentation faults during compilation, detecting logic bugs is more challenging because they often emerge silently to produce incorrect results which in turn propagate into downstream applications. Even more concerning, logic bugs in deep learning compilers—even when they produce correct functional outputs—can also introduce novel issues that are rarely a concern for traditional optimizing compilers on CPUs. These issues include architecture-specific performance degradation [14] and quantization-related problems [15].

To address the challenge of detecting optimization bugs in deep learning compilers, researchers have conducted a board range of studies [16] [17] [18] [19] [20] [21] [22]. While these studies offer valuable contributions, they do not fully address this challenge. First, some studies leverage fuzz testing to detect bugs, but struggle to locate and diagnose bug root causes. For example, NNSmith [18] leverages fuzz and differential testing, by generating diverse and valid deep learning models. Unfortunately, NNSmith cannot provide any information about bug root causes. Consequently, compiler developers struggle to locate and fix the uncovered bugs even with considerable efforts. Even more concerning, NNSmith leverages PyTorch as an oracle to detect compiler optimization bugs thus considerably enlarges it trusted computing base, because PyTorch itself still contains serious bugs [23]. Consequently, using PyTorch as an oracle cannot fully guarantee the correctness of its verdicts. Second, some studies leverage metamorphic testing, but struggle to design effective metamorphic relationships. For example, MT-DLComp [16] detect compiler bugs by constructing equivalent models based on equivalent model mutations. However, MT-DLComp is still limited in bug detection because designing effective relationship in an automated manner remains difficult. Moreover, MT-DLComp cannot control the specific optimization to test, because it generates models fully randomly.

In this paper, we propose a novel approach called Optimization Resistance Transformation (ORTHRUS), a general, fine-grained, and cost-effective technique to detect optimization bugs in deep learning compilers. The high-level idea of our approach is to build a reference non-optimizing compiler that does not perform any optimizations and leverage it as an oracle to differential test the candidate optimizing compiler. To this end, an optimization bug is uncovered when the optimizing compiler crashes during compilation or produces diverging execution outputs from the non-optimizing reference compiler.

However, developing such a reference non-optimizing deep learning compiler remains challenging. A direct approach is to build a new reference deep learning compiler from scratch. Nevertheless, this approach is labor intensive and technically daunting, even for a limited set of input tensor operators and specific architectures. Another viable approach is to leverage compiler options to control optimization levels (*e.g.*, `-O0`, `-O2`), but these options are limited and specific to different deep learning compilers. Worse yet, while optimization options could determine which optimization pipeline is applied during compilation, they are inherently coarse-grained and opaque, because the specific optimization levels (*e.g.*, `-O2`) represent bundled configurations of multiple optimization passes, often with complex interactions and undocumented dependencies. Consequently, relying on optimization levels for bug detection offers limited visibility and control.

To address this challenge, we propose, in our ORTHRUS approach, to structurally transform a given deep learning model (*i.e.*, the deep learning compiler input) from an optimizable form to an unoptimizable form that the deep learning compiler can no longer perform the potential optimizations. It remains a challenge that the transformation mechanism should preserve the original model's semantics while making optimizations inapplicable. To address this challenge, we propose to instrument semantics-preserving but optimization-blocking deep learning operators including transpose and matrix multiplication, at strategic points in the computation graph. As a result, we can prevent the deep learning compiler from recognizing specific deep learning optimization opportunities such as graph rewriting, operator fusion, and algebraic simplification.

We then leverage differential testing to test the deep learning compiler with the two optimizable and unoptimizable models as inputs. Specifically, both the optimizable model and its unoptimizable counterpart are compiled by the same deep learning compiler with the same compiling options enabled, then executed in the same runtime environment using identical inputs. As a result, we uncover a potential semantic inconsistencies bug, if the two models produce diverging results.

We argue that our approach ORTHRUS is general, fine-grained, and cost-effective. First, ORTHRUS is general because it supports not only different deep learning compilers (*e.g.*, TVM or ONNXRuntime), but also diverse deep learning models in various formats (*e.g.*, PyTorch or ONNX). Second, ORTHRUS is fine-grained because it can handle any specific buggy optimizations by employing a syntax-directed transformation to resist that optimization. Third, ORTHRUS is cost-effective because once a potential bug is uncovered, compiler developers or testers always have clear information about which optimization trigger that bug, facilitating subsequent bug root cause diagnosis and rectification.

We implement ORTHRUS as a practical software prototype and apply it to two widely used and well-tested deep learning compilers, namely ONNXRuntime and TVM. To conduct the evaluation, we first create a micro-benchmark **OptBench** consisting of real-world issues from top security conferences and GitHub issues. First, our evaluation of ORTHRUS on OptBench demonstrates that ORTHRUS is effective in detecting the optimization bugs, achieving a recall of 90% and a precision of 100%. Second, to evaluate the practical usefulness of ORTHRUS, we conduct a 12-hour test campaign, during which ORTHRUS detects 21 bugs in TVM and ONNXRuntime, among which 9 are non-crash optimization bugs and 1 is missed by NNSmith, a state-of-the-art fuzzing test tool. Third, to evaluate the overhead ORTHRUS introduced, we measure the transformation time, model generation time, and model execution time, and the results demonstrate that the transformation overhead is less than 1.4%. Finally, a developer study shows ORTHRUS enables developers to identify the root causes of bugs more effectively and cost-effectively.

In summary, our work makes the following contributions:

- We propose a new approach ORTHRUS of optimization resistance transformations to detect optimization bugs in deep learning compilers.
- We design and implement a practical software prototype for ORTHRUS and conduct systematic evaluations with it.
- We conducted extensive experiments to evaluate ORTHRUS in terms of effectiveness, usefulness, overhead, and efficiency.

The remainder of this paper is organized as follows. Section II introduces the background and motivation. Section III presents our approach. Section IV presents the experimental evaluation of ORTHRUS. Section V discusses limitations and future directions. Section VI reviews related work, and Section VII concludes.

II. Background and Motivation

To be self-contained, in this section, we first present the necessary background knowledge on deep learning compilers and their optimizations (§ II-A), then present our motivation through a real-world optimization bug uncovered in the deep learning compiler TVM (§ II-B).

A. Background

Deep learning compilers. Deep learning compilers transform deep learning models into executable implementations across

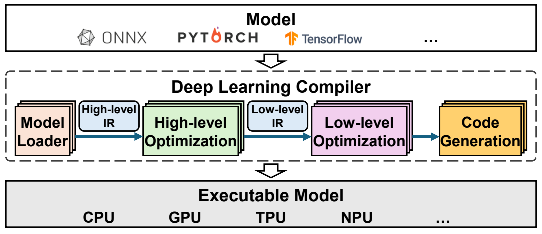


Fig. 1: An overview of deep learning compilers' workflow.

various hardware platforms. As Fig. 1 illustrates, the compilation process typically comprises four main stages.

First, deep learning compilers convert the input model from frameworks such as ONNX [24], PyTorch [25], and TensorFlow [26] into a high-level intermediate representation (IR), e.g. Relay [10] or MLIR [11]. Second, deep learning compilers employ hardware-independent optimizations on high-level IRs. Third, deep learning compilers perform hardware-aware optimizations to lower optimized high-level IR to low-level IRs. Finally, deep learning compilers generate target-specific executable codes, which are then deployed across diverse hardware backends including CPU, GPU, TPU, NPU, among others.

Optimizations in deep learning compilers. Deep learning compilers employ a series of high-level and low-level optimization transformations to optimize deep learning models into highly efficient executable code on diverse hardware architectures.

High-level optimizations are performed on high-level IRs, and focus on transforming the computation graph. High-level optimizations typically include node elimination, node replacement, algebraic simplification [27] [28] [29], and operator fusion [30] [31] [32].

Low-level optimizations [33] [34] [35] [36] target hardware-aware performance tuning and scheduling, and comprise hardware intrinsic mapping which transforms low-level IR instructions into specialized hardware kernels, memory allocation and access optimization, and loop-level optimizations to improve data locality and throughput.

B. Motivation

Optimizations in deep learning compilers are inherently complex, which can threaten compiler correctness and reliability. Take algebraic simplification as an example: developers must carefully account for both operator semantics and nuances of high-dimensional data structures under different numerical precisions. However, insufficient checks may introduce subtle bugs, producing incorrect outputs without crashes or other visible symptoms.

Fig. 2 illustrates a motivating example of an optimization bug in TVM [8], where an incorrect expression simplification merges the two `casts` (*i.e.*, ① and ②) into a single `cast` (② in Fig. 2(b)). This model is then compiled and optimized by the passes as represented by the solid blue arrows in Fig. 2(d). It should be noted that detecting bugs like this is challenging because it emerges silently without introducing any symptoms like compilation crashes or segmentation faults, and diagnosing its root causes still remains challenging, given the considerable number of optimization passes to triage.

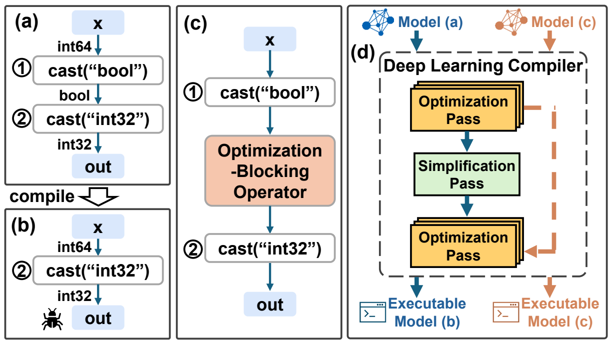


Fig. 2: An illustrative example of optimization bug in TVM. Blue rectangles represent input tensors, and white rectangles represent operators. Blue arrows denote the dataflow between operators, and parentheses next to the arrow denote the data type of the tensor. Our approach ORTHRUS of optimization resistant transformation is highlighted in orange.

To address these challenges, our high-level idea of optimization resistance transformation is to structurally transform the model from an optimizable form to an unoptimizable form that blocks the potential optimizations. Specially, for this example, our approach first transforms the model in Fig. 2(a) to an equivalent model as introduced in Fig. 2(c), where optimization-blocking operators are introduced into the original model as optimization "barriers". Next, the transformed model (*i.e.*, Fig. 2(c)) is fed to TVM again (Fig. 2(d)) to produce an executable model (c). It should be noted that this round of compilation will exercise the orange dashed line in Fig. 2(d) that bypasses the buggy simplification pass because TVM no longer recognizes the optimization opportunity due to the optimization resistance. Finally, we can leverage differential testing to compare the output executable model (c) against the model (b), and any discrepancies represent specific bugs in the compiler's simplification pass.

Nevertheless, designing optimization resistance transformation remains challenging because these transformations should be not only effective in blocking specific optimizations but also semantically equivalent without changing model behaviors. We will present how we conduct optimization resistance transformation generally in § III and how to apply our approach to this example specifically in § IV (Fig. 4).

III. APPROACH

This section provides a detailed description of ORTHRUS, a technique for effectively detecting optimization bugs in deep learning compilers. Our core insight is that a given deep learning model with great optimization opportunities, can be transformed to an equivalent deep learning model that is less amenable to optimizations. Therefore, we can leverage these two models to perform differential test on the target compiler,

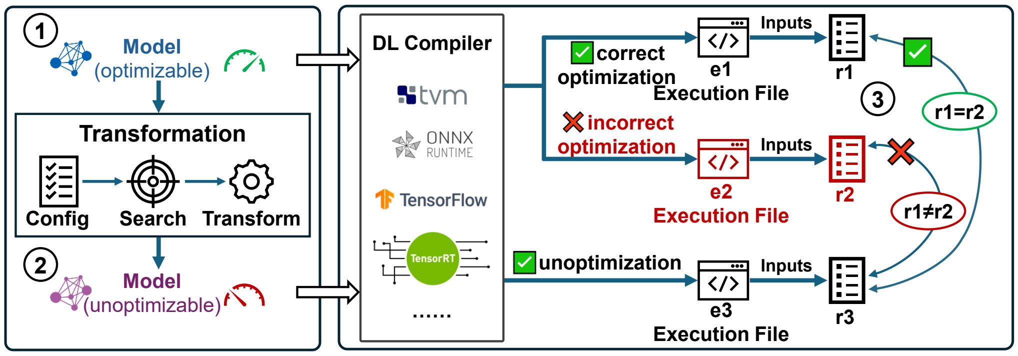


Fig. 3: An overview of ORTHRUS's workflow.

uncovering potential bugs. For brevity, we refer to the deep learning models that are potentially optimized by the deep learning compilers as *optimizable models*, and the models that are not or less optimized as the *unoptimizable models*. While our transformation does not guarantee the complete elimination of optimizations, our empirical observations indicate that it is broadly effective in suppressing them in practice.

A. Overview

Fig. 3 presents an overview of our approach, comprising three key steps. In step ①, we begin with an optimizable deep learning model and feed it to the target deep learning compiler under test (*e.g.*, TVM). The deep learning compiler compiles and optimizes the input model to generate binary models. Assuming that the compiler performs optimizations correctly without introducing bugs, it will produce an executable file e_1, which, when run, generates a correct output r_1. Otherwise, if the compiler contains bugs in its optimization, it produces a buggy executable e_2, which yields an incorrect output r_2 when run.

It should be noted that, as we have discussed in the introduction, uncovering such optimization bugs is not trivial because the specific optimization levels (*e.g.*, -O3) represent bundled configurations of multiple optimization passes, often with complex interactions and undocumented dependencies. Consequently, turning on or off specific optimization levels for bug detection offers limited visibility and control. Moreover, different deep learning compilers such as TVM or ONNXRuntime offer different optimization options making option control-based approach both ineffective and non-portable.

In step ②, we transform the original optimizable deep learning model to an unoptimizable deep learning model using our approach of optimization resistance transformations (§ III-B). ORTHRUS first searches and identifies model sub-structures that potentially trigger compiler optimizations according to a configuration, and applies corresponding optimization resistance transformations accordingly. The configuration allow end users to specify in fine-grained manner compiler optimization passes that are testing targets. The transformed unoptimizable model is then fed to the deep learning compiler and is expected to evade the compiler's optimization transformations during compilation. After compiling the unoptimizable model, we obtain an executable e_3, which produces output r_3 when run.

In step ③, we leverage differential testing to compare the results r_1 (or r_2) against r_3 for consistency. Since the executables $e1$ (or e_2) and e_3 are run with the same input, equivalence $r_1 = r_3$ indicates correct compilation, while a discrepancy $r_2 \neq r_3$ signals a potential compiler optimization bug. Furthermore, to accommodate minor numerical differences, we apply absolute tolerance (atol) and relative tolerance (rtol) thresholds [18], because even correct optimizations may introduce small variations in precision within the same runtime environment. Importantly, such precision-induced errors are significantly smaller than discrepancies caused by differences in execution environments.

B. Optimization Resistance Transformation

Transforming an optimizable deep learning model into an unoptimizable yet semantically equivalent form is a challenging task. Our approach ORTHRUS follows two fundamental principles during the transformation: 1) preserving the original model's input/output behaviors, and 2) preventing the compiler from applying specific optimizations. The core methodology we describe can be extended to address a wide range of optimization techniques implemented by current production deep learning compilers such as TVM and ONNXRuntime.

Transformation. Algorithm 1 outlines the overall transformation workflow of ORTHRUS, which converts an optimizable deep learning model M_{opt} into a semantically equivalent but unoptimizable deep learning model M_{unopt}.

Initially, ORTHRUS loads the ORT configuration C that specifies the optimization resistance targets, such as operator fusion, algebraic simplification, along with matching conditions and corresponding transformation strategies. By adjusting these rules, the user of ORTHRUS can either simultaneously detect multiple categories of optimization bugs or focus on a specific class. Upon loading the configuration, ORTHRUS constructs the computation graph of M_{opt} (line 3), explicitly representing the dataflow and operator dependencies.

Then, ORTHRUS leverages a depth-first search to identify potential optimization opportunities within the graph (line

Algorithm 1: Transformation

```
Input: $C$: configuration,
       $M_{\text{opt}}$: an optimizable deep learning model
Output: $M_{\text{unopt}}$: an unoptimizable deep learning model
1 Function ORT ($M_{opt}$) :
2     $rules \leftarrow$ LoadConfig($C$);
3     $graph \leftarrow$ GenerateComputationGraph($M_{\text{opt}}$);
4     $opportunities \leftarrow$ SearchOpportunities($graph$);
5     for $opportunity\ in\ opportunities$ do
6         Transform($graph$, $opportunity$)
7     $M_{\text{unopt}} \leftarrow$ RebuildModel($graph$);
8     return $M_{\text{unopt}}$;
```

4), which are collected into a set *opportunities*. For each detected opportunity, ORTHRUS applies the transformation rules (line 5-6), that maintain semantic equivalence but effectively block the compiler's ability to perform the targeted optimization.

Finally, ORTHRUS reassembles the modified graph into the unoptimized model M_{unopt} (line 7), which preserves the original functionality while exhibiting reduced susceptibility to optimization.

To put the discussion into perspective, we will showcase two specific transformations of operator fusion and algebraic simplifications. We select these two transformations because 1) they are important and representative optimization found in nearly all production deep learning compilers, and 2) they are susceptible to bugs due to their complex logics and tricky validity checks. But it is worthy noting that our approach can be applied to a broad range of optimizations besides these two.

Operator fusion. To detect bugs related to operator fusion, we leverage a strategy that inserts specific intermediate operators into the computation graph to disrupt fusion patterns. The key first step to this strategy is to gain a thorough understanding of operator fusion patterns and types that are widely employed in current production deep learning compilers. To this end, we first conduct an empirical investigation of operator fusions in current deep learning compilers by inspecting their source code and documentation.

Our investigation reveals important facts about operator fusion as shown in Table I. Specifically, operators that are used in deep learning models can be classified into six categories based on their fusion capabilities: element-wise, broadcast, injective, common-reduce, outwise-fusable, and opaque. Each category comprises representative operator examples. For example, the ElemWise category comprises relu, cast and neg operators, whereas the Broadcast operators consists of where, add, mul, among others. Furthermore, each category is ordered in a decreasing order of fusion capability, where a smaller number (*e.g.*, i) indicates stronger fusion potential. Next, for brevity, we write an operator O with capability c as O_c. For example, we write the operator `add` as $\texttt{add}_{\text{ii}}$.

When an operator pair (O_i, O_j) is identified as a candidate

TABLE I: Operator patterns and kinds in production deep learning compilers.

Operator category	Operator example	Fusion capability
ElemWise	relu, cast, neg, ...	i
Broadcast	where, add, mul, ...	ii
Injective	pad, reshape, transpose, ...	iii
CommReduce	argmin, sum, mean, ...	iv
OutEWiseFusable	conv2d, matmul, ...	v
Opaque	others	vi

for fusion, we insert an intermediate operator O_k between O_i and O_j with $k \geq i$ and $k \geq j$. The two inequalities specify that O_k can rarely fused with O_i or O_j because it has a low fusion capability. Therefore, this insertion prevents the compiler from matching the fusion pattern while preserving the model's semantics. For example, consider a model containing the operators `neg` of category ifollowed by `argmin` of category iv, which normally triggers operator fusion in deep learning compilers. To prevent this operator fusion, we insert a matrix multiplication operator `matmul` of category vbetween them, where the output of `neg` is multiplied by an identity matrix. This operator is mathematically neutral but alters the dataflow sufficiently to block the operator fusion optimization.

This approach is not limited to the example above but can be systematically extended to other fusion patterns. By referencing the fusion rules of the target compiler, we can select appropriate insertion operators to build optimization resistance rules.

Algebraic simplification. Deep learning compilers often leverage algebraic simplification to simplify consecutive identical expressions or repeated tensors to improve efficiency. However, such transformations can introduce semantic errors due to the tricky algebraic properties of tensors. To detect bugs caused by incorrect algebraic simplification optimizations, we introduce intermediate tensors and arithmetic operations that preserve mathematical equivalence but disrupt simplification heuristics.

To better illustrate our idea, we consider a sample algebraic simplification optimization that automatically replaces repeated addition operations (*e.g.*, $add(x, x)$) with multiplication (*e.g.*, $multiply(x, 2)$). As the argument x may be a large tensor, this optimization can reduce redundant nodes and save considerable memory. To block such an optimization, we introduce an intermediate tensor y that holds the same value as the tensor x (*i.e.*, $y = x$). Then, we apply identity-preserving transformations to perform a mathematically neutral operation on y. One feasible such operation is to multiply y by an identity matrix I of the same dimensions (*i.e.*, $y = matmul(y, I)$). Finally, we construct unoptimized model $add(x, y)$, where $y = x$ and $y = matmul(y, I)$. Since y remains mathematically equivalent to x, the model's output should stay unchanged. Meanwhile, deep learning compiler optimizers will have to retain the addition operation $add(x, x)$ instead of converting it to $multiply(x, 2)$, because they cannot recognize the equivalence between $add(x, y)$ and $add(x, x)$

Algorithm 2: Differential Testing

```
Input:  $M_{\text{opt}}$: optimized deep learning model
        $M_{\text{unopt}}$: unoptimized deep learning model
        $Inputs$: a set of test inputs
Output: $Bug$
1 Function TestEngine($M_{opt}$, $M_{unopt}$, $Inputs$):
2     for input in Inputs do
3         $e_{\text{opt}} \leftarrow$ Compile($M_{\text{opt}}$);
4         $e_{\text{unopt}} \leftarrow$ Compile($M_{\text{unopt}}$);
5         $res_{\text{opt}} \leftarrow e_{\text{opt}}(input)$;
6         $res_{\text{unopt}} \leftarrow e_{\text{unopt}}(input)$;
7         if $res_{opt} \neq res_{unopt}$ then
8             return Bug(Semantic Divergence);
9     return NoBug;
```

any more. As a result, if the optimized model produces a different output from the original one, a potential bug in the deep learning compiler's algebraic simplification pass manifest.

C. Differential Testing

We leverage differential testing to effectively detect bugs (step ③ in Fig. 3), by comparing the optimized and unoptimized deep learning models with the identical inputs. A semantic error is flagged when the optimized model produces results that deviate from those of the unoptimized model, thereby violating the expected functional behavior. Furthermore, we leverage numerical comparison metrics, specifically absolute and relative tolerances, to determine equivalence. Significant deviations beyond these thresholds are indicative of optimization bugs.

Algorithm 2 illustrates the key steps of the differential testing. The algorithm takes as inputs the optimized deep learning model M_{opt} and unoptimized deep learning models M_{unopt}, and a set of model inputs, and outputs a detected bug report. First, both models, M_{opt} and M_{unopt} are compiled using the same deep learning compiler with same configurations and compiling options, to generate two executable files, e_{opt} and e_{unopt} respectively. Then, e_{opt} and e_{unopt} are executed under the same runtime environment with identical inputs, to collect outputs e_{opt} and e_{unopt}, respectively. If the outputs e_{opt} and e_{unopt} differ, it indicates the presence of semantic inconsistency bugs. Finally, the test engine returns a bug report based on these results for subsequent bug diagnosis and rectification.

D. Implementation

To validate our design, we implement a software prototype for ORTHRUS. Next, we highlight some implementation details.

Model generation. We leverage NNSmith [18], a powerful and state-of-the-art deep learning compiler fuzzing test tool to produce diverse and valid deep learning models. We leverage NNSmith's random model synthesis to ensure wide coverage of operator combinations, data types, and tensor shapes. To maximize compatibility, ORTHRUS is designed to support three types of input model format: PyTorch, ONNX, and TensorFlow, allowing for seamless collaboration with NNSmith.

Transformation. We implement the optimization resistance transformation part of ORTHRUS as transformation plugins. For models in PyTorch or TensorFlow formats, we leverage their official export APIs to obtain a uniform ONNX representation. We then leverage ONNXScript [37] to transform the ONNX form, allowing precise, efficient, and programmatic graph rewriting. ORTHRUS identifies potential optimization triggering patterns based on rules in the configuration to inject transformation operators. The configuration architecture ensures that new transformation rules can be easily added or removed without modifying the core framework.

Testing engine. We leverage and extend the mitigation strategy in NNSmith [18] to handle the numerical instability inherent in floating-point computations. We check output equivalence by comparing the absolute difference and relative difference between two outputs with high error tolerance.

IV. Evaluation

The goal of our evaluation is to demonstrate the effectiveness of our approach. To this end, we apply ORTHRUS to real-world deep learning compilers to assess its effectiveness in identifying optimization bugs. Specifically, our evaluation aims to answer the following research questions:

RQ1: Effectiveness. Is ORTHRUS effective in detecting optimization bugs in deep learning compilers?

RQ2: Usefulness. Is ORTHRUS useful in uncovering optimization bugs in real-world deep learning compilers?

RQ3: Overhead. Does ORTHRUS introduce significant overhead during the testing process?

RQ4: Cost-effectiveness. As ORTHRUS is introduced to help deep learning compiler developers to locate bug and analyze root causes, is it cost-effective to help them achieve this?

A. Experiment Setup

Datasets. To evaluate the effectiveness of ORTHRUS by calculating the precision and recall of a bug detection needs a benchmark suite that comes with ground truth for the optimization bugs. Yet such a benchmark suite, to the best of our knowledge, is not available, while curating the ground truth directly from the source code of large/complex, real-world deep learning compilers may not be feasible. We thus take the first step to manually create **OptBench**, a microbench for optimization bugs in deep learning compilers. We create this microbench by collecting real-world issues from two sources: 1) top security conferences (*e.g.*, NNSmith [18], Polyjuice [21] and Scuzer [22]), and 2) GitHub issues of optimization bugs. As shown in Table II, OptBench consists of 10 benchmarks, covering the testcase, the deep learning compiler from which the bug manifest, root causes, and links to resources. Currently, we are still maintaining and augmenting OptBench by including more benchmarks when new bugs are covered.

Baselines. For our comparative study, we use NNSmith, a state-of-the-art deep learning compiler fuzzer, as a baseline.

Tested deep learning compilers. We select TVM [4] and ONNXRuntime [3], as our evaluation target compilers, because 1) they are actively maintained open-source deep learning compilers, continuously evolving with support from the research community and industry, and 2) they are commonly used as benchmark in prior studies on bug detection and evaluation [18] [21], which ensures that our experimental results are comparable and carry strong reference value. However, our approach (see Fig. 3) is general and thus can be applied to other deep learning compilers as well.

Experiment configuration. We conduct our evaluation on a machine equipped with an Intel i7 CPU with 12 cores and a NVIDIA GPU, running Ubuntu 22.04 LTS.

B. Effectiveness

To answer RQ1 by investigating ORTHRUS's effectiveness, we first conduct an evaluation of ORTHRUS on our micro-benchmarks **OptBench**. We repeat each experiment 3 rounds to avoid potential bias. Column 5 in Table II presents the experimental results, where "✔" indicates that ORTHRUS successfully detects the bug and "✘" denotes that ORTHRUS misses that bug. The experimental results demonstrate that ORTHRUS effectively detects 9 testcases among all 10 cases but misses 1 testcase. Consequently, ORTHRUS achieves a recall of 90% ($\frac{\#detected}{\#all}$) and a precision of 100% ($\frac{\#detected}{\#true\ positive}$), which illustrates that ORTHRUS is effective in detecting real-world bugs in deep learning compilers.

Furthermore, to investigate the root cause of the failed case, we conduct a manual inspection. This inspection reveals the detection failure is because the optimization occurs within the operator itself during the low-level optimization, which means the transformation performed on the computation graph cannot expose that bug. This is not caused by the design defects of ORTHRUS itself, but the limitations of transformation on the computation graph used by ORTHRUS.

C. Usefulness

To answer RQ2 by showing ORTHRUS's practical usefulness in uncovering previously unknown bugs, we apply ORTHRUS to real-world deep learning compilers ONNXRuntime and TVM. We run ORTHRUS on each compiler for 12 hours, and repeat the experiment three times. We record all bugs detected by ORTHRUS and classify them into three categories: result inconsistencies, compilation failures, and runtime crashes.

Table III summarizes the detected bugs. We detect a total of 22 bugs, including 9 bugs that result in output inconsistencies, 3 bugs that cause runtime crashes, and 10 compilation failure bugs. We next focus on the bugs that trigger inconsistent outputs between optimizable and unoptimizable deep learning models, as our primary goal is to uncover optimization bugs.

To further investigate whether our approach surpasses current state-of-the-art techniques, we compare ORTHRUS with NNSmith. Recall that NNSmith detects optimization bugs by cross-checking results against PyTorch as the oracle. However, PyTorch itself may contain latent bugs that could lead to incorrect outputs. So when NNSmith's cross-backend checking functionality is disabled, it is limited to detecting only compilation failures and runtime crashes but not optimization bugs as our work does. Even when the crosschecking feature in NNSmith is enabled, ORTHRUS also detects one more bug than NNSmith, which is caused by TVM's data layout optimization.

In addition, inconsistency bugs identified by NNSmith cannot be directly confirmed as optimization bugs, since discrepancies may also arise from operator implementation errors or precision differences introduced during cross-checking. In contrast, our approach can directly classify result inconsistency bugs as optimization bugs. This is because we compile all models using the same compiler with same configurations and options, ensuring that operator implementation errors do not lead to output discrepancies. Moreover, our differential testing is conducted in identical runtime environments, and any precision loss introduced by optimization is negligible—below 10^{-6}. Therefore, the bug category can be directly determined.

D. Overhead

To answer RQ3 by investigating the overhead of ORTHRUS, we measure the test throughput, a critical test metric, to evaluate the overhead of our approach. Specifically, we conduct a 12-hour testing campaign on TVM and ONNXRuntime to evaluate the overhead introduced by ORTHRUS. Table IV provides a detailed breakdown of the time consumption during testing. The model transformation column represents the overhead introduced by ORTHRUS. The result shows that ORTHRUS could transform deep learning model for about 30ms, that means ORTHRUS could complete transformation efficiently. The model generation column denotes the time spent to generate the initial deep learning models. From the table, we can observe that majority of the time is spent in the execution of the compiler and the model generation. It is also noteworthy that the number of graph nodes does not significantly influence the time spent by ORTHRUS, but it does impact the execution time of deep learning compiler. Therefore, as the number of graph nodes increases, the overhead introduced by ORTHRUS becomes negligible.

E. Cost-Effectiveness

We conduct a developer study to quantify the manual effort required to locate optimization bugs and identify their root causes, which is then leveraged to evaluate the cost-effectiveness of ORTHRUS. We hire three graduate students to conduct this study, and all of them have extensive experience in developing deep learning compilers.

Throughout the study, we ask students to complete a task: analyzing three test cases that trigger optimization bugs and identifying the operators where the bugs occur. We measure the time each student takes to complete the task, evaluating the effort required to pinpoint an optimization bug. The three test cases containing the deep learning models 1, 6, and 9 from our

TABLE II: The micro-benchmark created to evaluate the effectiveness of ORTHRUS. This benchmark is collected from prior works and real-world GitHub issues related to optimization bugs in production deep learning compilers.

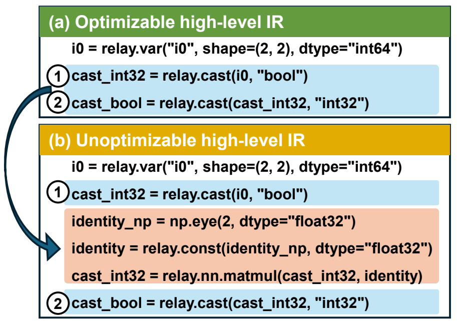
#	TestCase	Deep learning compiler	Root cause	ORTHRUS	Resource
01	NNSmith	ONNXRuntime	incorrect expression simplification	✔	https://github.com/microsoft/onnxruntime/issues/11994
02	NNSmith	ONNXRuntime	incorrect optimization	✔	https://github.com/microsoft/onnxruntime/issues/11870
03	NNSmith	TVM	incorrect expression simplification	✔	https://github.com/apache/tvm/issues/13048
04	NNSmith	TVM	incorrect algebraic simplification	✘	https://github.com/apache/tvm/pull/10336
05	PJ	TVM	incorrect expression simplification	✔	https://github.com/apache/tvm/pull/14571
06	PJ	TVM	incorrect operator fusion	✔	Fig. 8
07	Scuzer	TVM	incorrect inline optimization	✔	https://github.com/cxx122/Scuzer/blob/main/bugs/bug2.md
08	Scuzer	TVM	incorrect operator replacement	✔	https://github.com/cxx122/Scuzer/blob/main/bugs/bug6.md
09	Scuzer	TVM	mis-optimization of asymmetric operations	✔	https://github.com/cxx122/Scuzer/blob/main/bugs/bug8.md
10	Scuzer	TVM	incorrect operations fusion	✔	Fig. 15

TABLE III: Statistics of bugs uncovered by ORTHRUS.

Deep learning compiler	Optimization bug	Crash	Compilation
TVM	6	2	7
ONNXRuntime	3	1	3
Total	9	3	10

TABLE IV: Average time taken during ORTHRUS's testing.

Deep learning compiler	Graph node	Model generation	Model transformation	Differential testing
TVM	5	12.12ms	0.32ms	617.35ms
	10	22.79ms	0.33ms	1156.29ms
	15	65.40ms	0.33ms	1942.01ms
ONNXRuntime	5	7.22ms	0.21ms	8.66ms
	10	14.73ms	0.23ms	14.99ms
	15	23.82ms	0.25ms	22.41ms


(a) Optimizable high-level IR
(b) Unoptimizable high-level IR

```
i0 = relay.var("i0", shape=(2, 2), dtype="int64")
①cast_int32 = relay.cast(i0, "bool")
②cast_bool = relay.cast(cast_int32, "int32")
```

```
i0 = relay.var("i0", shape=(2, 2), dtype="int64")
①cast_int32 = relay.cast(i0, "bool")
identity_np = np.eye(2, dtype="float32")
identity = relay.const(identity_np, dtype="float32")
cast_int32 = relay.nn.matmul(cast_int32, identity)
②cast_bool = relay.cast(cast_int32, "int32")
```

Fig. 4: A bug uncovered by ORTHRUS from the TVM compiler. This bug is caused by incorrect expression simplification optimization.

micro-benchmark (**OptBench**) and their corresponding inputs. The students first identify the root causes of optimization bugs without ORTHRUS, and spend an average of 25 minutes to analyze one bug. The students then finish the same task leveraging ORTHRUS, and spend an average of 3 minutes to analyze one bug. These evaluation results demonstrate that ORTHRUS is cost-effective in helping end developers or testers locate real bugs.

F. Bug Study

To illustrate the ORTHRUS's ability of bug detection, we demonstrate, through concrete bug case studies, how our approach can detect diverse optimization bugs from both micro-benchmark and real-world deep learning compilers. Meanwhile, these bug studies are essential for offering a deep understanding of how optimization bugs may manifest in practical deep learning compilers, thus shedding light on future studies in this direction.

Bug study 1. Fig. 4 illustrates how ORTHRUS detects bugs in the example from the motivation (Fig. 2(a) in § II-B). Specifically, Fig. 4(a) shows the Relay IR of the model, where the two `cast` operators are incorrectly simplified into single kernel (Fig. 2(b)), due to a TVM's bug in the datatype checking before the simplification.

To detect potential bugs as this one in the simplification optimization, ORTHRUS performs an optimization resistance transformation on the input model to generate a new model as shown in 4(b). Specifically, ORTHRUS inserts a `matmul` operator between the two cast operators, where the `matmul` computes the product of the `cast` output and an `identity` matrix.

Next, we compile the transformed model (Fig. 4(b)) using TVM which generates an output model that is same to the input model without triggering the simplification optimization because TVM does not recognize the optimization opportunity. As a result, ORTHRUS successfully detects this bug, even though it does not trigger any compiling peculiarity.

Bug study 2. Fig. 5 illustrates how ORTHRUS detects a operator fusion optimization bug. Specifically, Fig. 5(a) shows a fragment of the input model that triggers that bug, which contains `transpose`, `resize`, `sigmoid`, `floor`, and `argmin` operators. During the compilation, the compiler performs operator fusion optimization on the `transpose`, `resize`, and `sigmoid` operators (①), into a single kernel `fused_transpose_image_resize2d_sigmoid` (Fig. 5(b)).

To detect such bugs, ORTHRUS performs an optimization resistance transformation on the model to produce an unoptimizable model as Fig. 5(c) shows. Specifically, ORTHRUS inserts a `matmul` operator, to multiply the output of `resize` with the `identity` matrix that serves as the input to `sigmoid`.

Next, we compile the transformed model (Fig. 5(b)) us-

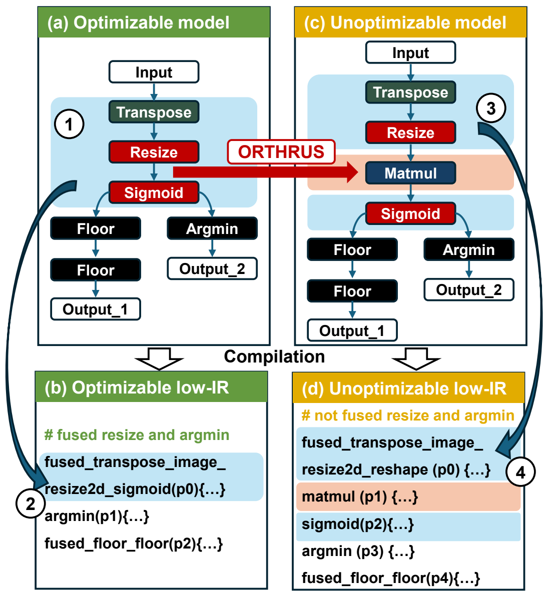


Fig. 5: A bug uncovered by ORTHRUS from TVM. This bug is caused by incorrect operator fusion optimization.

ing TVM which in turn generates an output model that is shown in Fig. 5(d) ④. Specifically, our optimization resistance transformation successfully prevents the fusion optimization between `resize` and `sigmoid`, thus uncovering the bug in that optimization. It is worth noting that, in Fig. 5(d), the `transpose` and `resize` operators still undergo operator fusion optimization, because they are not the target of the configure rules of optimization resistance, showing the fine granularity and flexibility of ORTHRUS.

V. Discussion

In this section, we discuss some possible enhancements to this work, along with directions for future work.

Generality. ORTHRUS is a universal technique for a wide range of deep learning compilers regardless of their code availability. Our core observation is that, despite the highly diverse concrete implementations of optimizations in various deep learning compilers, their underlying techniques are similar. Therefore, by transforming input models rather than optimization implementations, our technique can be applied across different deep learning compilers, even those yet to be developed.

Limitation. We identify several limitations of ORTHRUS. Since it is based on analyzing computational graphs, it is more effective for frontend optimizations that can be blocked through graph rewriting. However, optimizations occurring within atomic operators, such as data layout transformations in conv2d, remain challenging, and we are actively exploring mechanisms to address them. Nevertheless, the core idea of optimization resistance transformations remains applicable to such cases.

Future work. Enhancing transformation rules represents a promising direction for future work. ORTHRUS provides a configuration for researchers to effortlessly experiment with various transformation rules, which could contribute significantly to the robustness of deep learning compilers.

VI. Related Work

Deep learning compiler testing. Testing deep learning compilers has attracted considerable attention due to their increasing complexity and the critical need for correctness. Early works such as MT-DLComp [16] employ metamorphic testing by mutating existing models. NNSmith [18] and GenCoG [38] generate diverse and valid computation graphs to enhance test coverage. HirGen [19] focuses on high-level optimizations using operator-related coverage metrics, while Tzer [39] applies feedback-driven mutation to low-level IRs. However, ORTHRUS introduces optimization resistance transformation on computation graphs. Some works focus on specific classes of bugs, such as OPERA [40] detecting model loading bugs, TracNe [41] identifying subtle numerical errors, and Scuzer [22] targeting scheduling optimization bugs. In contrast, ORTHRUS targets optimization bugs.

VII. Conclusion

We present a novel approach, ORTHRUS, based on optimization resistance transformations to detect optimization bugs in deep learning compilers. The key idea of this approach is to transform optimizable models into semantically equivalent but unoptimizable models that block potentially buggy optimizations. The evaluation results of a software prototype we realize for ORTHRUS demonstrate that ORTHRUS is effective in detecting optimization bugs and practical in uncovering real-world bugs in production and well-tested deep learning compilers, outperforming the state-of-the-art. Our future work represents a new step towards effective bug detection in deep learning compilers, making them more reliable and trustworthy.

References

[1] Google, “Xla: Optimizing compiler for machine learning,” 2017.
[2] NVIDIA. Corporation, “Nvidia tensorrt: High performance deep learning inference,” 2023.
[3] Microsoft Corporation, “Onnx runtime: Accelerating machine learning inference,” 2023.
[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Tvm: An automated end-to-end optimizing compiler for deep learning,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594.
[5] 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, Mar. 2021.
[6] “[bug] alteroplayout failed on conv->transpose->conv · issue #10109 · apache/tvm,” https://github.com/apache/tvm/issues/10109.
[7] “[bug] layout error when putting ‘argmin‘ after ‘conv2d‘ · issue #9813 · apache/tvm,” https://github.com/apache/tvm/issues/9813.
[8] “[bug] wrong results of ‘cast<int32>(cast<bool>(-1i64))‘ · issue #13048 · apache/tvm,” https://github.com/apache/tvm/issues/13048//.
[9] “[arith][bugfix] fix a bug of iter map floormod(x,2) simplify by tqchen · pull request #14571 · apache/tvm · github,” https://github.com/apache/tvm/pull/14571.

[10] J. Roesch, S. Lyubomirsky, M. Kirisame, L. Weber, J. Pollock, L. Vega, Z. Jiang, T. Chen, T. Moreau, and Z. Tatlock, "Relay: A high-level compiler for deep learning," 2019.

[11] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[13] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 216–226, Jun. 2014.

[14] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "Deltann: Assessing the impact of computational environment parameters on the performance of image recognition models," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct. 2023, pp. 414–424.

[15] M. Guo, "Analyzing quantization in tvm," Aug. 2023.

[16] D. Xiao, Z. Liu, Y. Yuan, Q. Pang, and S. Wang, "Metamorphic testing of deep learning compilers," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–28, Feb. 2022.

[17] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–26, Apr. 2022.

[18] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Vancouver BC Canada: ACM, Jan. 2023, pp. 530–543.

[19] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Seattle WA USA: ACM, Jul. 2023, pp. 248–260.

[20] K. Lin, X. Song, Y. Zeng, and S. Guo, "Deepdiffer: Find deep learning compiler bugs via priority-guided differential fuzzing," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. Chiang Mai, Thailand: IEEE, Oct. 2023, pp. 616–627.

[21] C. Zhou, B. Qian, G. Go, Q. Zhang, S. Li, and Y. Jiang, "Polyjuice: Detecting mis-compilation bugs in tensor compilers with equality saturation based rewriting," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1309–1335, Oct. 2024.

[22] X. Chen, X. Lin, J. Wang, J. Sun, J. Wang, and W. Wang, "Scuzer: A scheduling optimization fuzzer for tvm," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–28, 2025.

[23] N. S. Harzevili, J. Shin, J. Wang, S. Wang, and N. Nagappan, "Characterizing and understanding software security vulnerabilities in machine learning libraries," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. Melbourne, Australia: IEEE, May 2023, pp. 27–38.

[24] "Github - onnx/onnx: Open standard for machine learning interoperability," https://github.com/onnx/onnx.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[26] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: A system for {Large-Scale} machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[27] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Huntsville Ontario Canada: ACM, Oct. 2019, pp. 47–62.

[28] H. Wang, J. Zhai, M. Gao, Z. Ma, S. Tang, L. Zheng, Y. Li, K. Rong, Y. Chen, and Z. Jia, "Pet: Optimizing tensor programs with partially equivalent transformations and automated corrections," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 37–54.

[29] Y. Yang, P. Phothilimthana, Y. Wang, M. Willsey, S. Roy, and J. Pienaar, "Equality saturation for tensor graph superoptimization," in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 255–268.

[30] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimizing dnn computation with relaxed graph substitutions," in *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia, Eds., vol. 1, 2019, pp. 27–39.

[31] J. Zhao and P. Di, "Optimizing the memory hierarchy by compositing automatic transformations on computations and data," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Athens, Greece: IEEE, Oct. 2020, pp. 427–441.

[32] W. Jung, T. T. Dao, and J. Lee, "Deepcuts: A deep learning optimization framework for versatile gpu workloads," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Virtual Canada: ACM, Jun. 2021, pp. 190–205.

[33] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 863–879.

[34] H. Zhu, R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui, F. Yang, M. Yang, L. Zhou, A. Cidon, and G. Pekhimenko, "Roller: Fast and efficient tensor compilation for deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 233–248.

[35] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 559–578.

[36] C. Zhang, L. Ma, J. Xue, Y. Shi, Z. Miao, F. Yang, J. Zhai, Z. Yang, and M. Yang, "Cocktailer: Analyzing and optimizing dynamic control flow in deep learning," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 681–699.

[37] "Github - microsoft/onnxscript." https://github.com/microsoft/onnxscript.

[38] Z. Wang, P. Nie, X. Miao, Y. Chen, C. Wan, L. Bu, and J. Zhao, "Gencog: A dsl-based approach to generating computation graphs for tvm testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Seattle WA USA: ACM, Jul. 2023, pp. 904–916.

[39] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–26, 2022.

[40] Q. Shen, Y. Tian, H. Ma, J. Chen, L. Huang, R. Fu, S.-C. Cheung, and Z. Wang, "A tale of two dl cities: When library tests meet compiler," 2024.

[41] Z. Xia, Y. Chen, P. Nie, and Z. Wang, "Detecting numerical deviations in deep learning models introduced by the tvm compiler," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, 2024, pp. 73–83.