Seedzer: A Pipeline for Fuzzing in Deep Learning Compilers

Jerry Gui
University of Maryland
College Park, MD

***Abstract*—Deep-learning (DL) compilers, such as TVM and TensorRT, are widely used in optimizing Deep Learning Models to increase performance and reach across devices. However, bugs in the compiler may result in generated models that behave differently from input models. This may lead to a loss in accuracy and correctness in applications that rely on generated models. Fuzzing is an automated method to detect bugs in software by generating random test cases as inputs and checking for inconsistent behavior. Our work focuses on TVM, a popular deep-learning compiler developed by Apache. Existing fuzzers for TVM target particular optimization levels. However, bug detection may be improved by a full-compiler fuzzing analysis. We propose a pipeline to detect DL compiler bugs by combining these existing tools.**

***Index Terms*—Deep Learning Compiler, Fuzzing, Coverage**

I. Introduction

Deep Neural Networks (DNN) are becoming increasingly common in our lives. Applications that involve DNNs range from image enhancers in phone cameras to generative AI that generates images from text. One such generative AI service, ChatGPT, has more than 100 million users every week. [1] As the use cases for neural networks grow, so do the range of devices that they are run on. Devices have different power limits and different hardware architectures. Critically, Neural Networks must take advantage of hardware-specific optimizations to improve performance and efficiency across a range of devices. Deep-learning (DL) compilers bridge the gap between the DNN software and deployed hardware. Apache TVM, ONNXRuntime, TensorRT, OpenXLA, NGraph, and Glow are examples of DL compilers maintained by companies such as Apache, Intel, and Meta. DL compilers take a neural network model as input and output an optimized representation for a specific architecture. Bugs in these compilers can slow outputted models to unusable extents or result in unexpected prediction results compared to the initial model.

We focus on Apache TVM, a deep-learning compiler popular both in industry and academia. TVM is a multi-stage compiler that performs both high- and low-level optimizations on a neural network before passing it down to a machine code compilation toolchain. Several tools have been developed to automatically find bugs in TVM, including NNSmith, Tzer, and TVMFuzz. These tools are fuzzers, which generate test cases as inputs and compare outputs with some reference. However, all of these tools focus on fuzzing a certain optimization level of TVM: NNSmith only generates high-level models to pass to TVM, and Tzer and TVMFuzz only fuzz the low-level optimization passes of TVM. This raises the question of whether a full-compile-stack fuzzing approach would outperform the state-of-the-art fuzzing approaches.

We propose Seedzer, a pipeline that aims to close the disconnection between low-level and high-level fuzzers. Seedzer aims to intelligently propagate NNSmith's high-level model generation to Tzer's low-level fuzzing. We investigate the potential of Seedzer on TVM and compare it to Tzer individually.

Section II describes the background of DL compilers in more detail, specifically TVM. Section III describes prior work done by NNSmith and Tzer. Section IV details our approach to combining this fuzzing pipeline. Section V describes our setup for experiments. Section VI provides measurements of our pipeline's performance. Section VII describes observations and limitations of our current setup. Finally, Section VIII concludes the takeaways of our work.

II. Apache TVM

Apache TVM is a DL compiler designed to generate optimized low-level code for a variety of hardware platforms for inputted deep-learning models [2]. TVM performs a variety of steps that optimize the model both at the high- and low levels. At the high level, Deep Neural network models are in a form called a computation graph. This form abstracts the tensor operations of a NN model as nodes of a graph. It is also independent of any particular deep learning framework, such as PyTorch or TensorFlow. In TVM, this intermediate representation (IR) is called Relay. Then the Relay IR is transformed into Tensor IR, during which several optimizations are performed to optimize the computation graph. These transformation passes may include passes that simplify the graph or merge certain nodes together into operations that may be easier to compute mathematically. Finally, the Tensor IR is translated to a target IR (such as LLVM), performing target-specific optimizations. Compilation from the target IR is handled by the target's pipeline to generate machine code, such as by using the LLVM toolchain. A breakdown of the stages of the pipeline that TVM handles is shown in Figure 1.

TVM's evaluation shows that TVM generates operators in neural networks that outperform hand-tuned operators. It showed 1.6× to 3.8× speedup in server class GPUs compared to high-level frameworks Tensorflow and MXNet [2]. It also

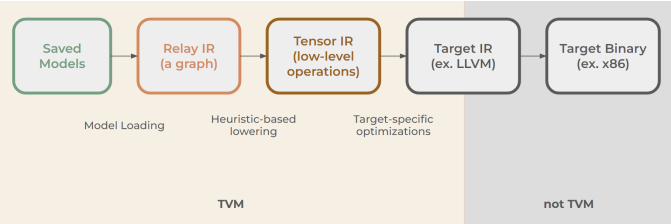

Fig. 1. The TVM compiler pipeline.

improved performance on CPUs as compared to a Tensorflow Lite baseline.

III. Fuzzing

To find bugs without manually combing through TVM's large codebase, automated testing tools are used to detect possible bugs. Fuzzing is one such automated bug detection technique. Fuzzers generate large amounts of random inputs which are executed through the tested program. The outputs are compared to an oracle, or expected output for each input.

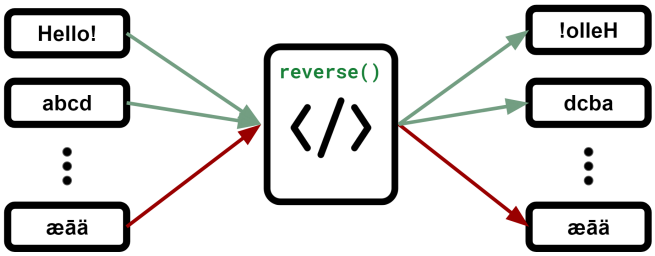

Fig. 2. Example of fuzzing a simple reverse function.

Fig. 2 shows an example of fuzzing a function. A fuzzer generates inputs for the reverse function that produce expected outputs. However, the last input generates an incorrect value, meaning there is a bug in the reverse function.

Fuzzers must generate diverse inputs that cover the input space. Otherwise, very few bugs will be detected, as the fuzzer does not generate inputs with traits that lead to bugs. Additionally, because DL compilation is split into multiple stages, fuzzers for DL compilers can start generation in other stages.

A. NNSmith

NNSmith, published in late 2022, describes an approach that generates high-level models for various deep learning front- and backends [3]. A frontend represents a deep learning framework that users can use, such as PyTorch, TensorFlow, and ONNX. A backend is a compiler that inputs a model and outputs optimized machine code, in our case, TVM. NNSmith's novelty was that prior approaches had not considered structure validity when generating neural networks, thus resulting in nets with invalid shapes or operations or networks with floating-point exceptions like infinities or NaNs. To generate high-level networks that abide by these restrictions NNSmith defines constraints for various network layers that specify each layer's input and output shape.

In essence, NNSmith uses an SMT solver to generate random *high-level* network architecture, then uses a gradient-guided value search to find weights that minimize floating-point exceptions. It passes this "valid" network to a backend, checking its compiled output's behavior against a "correct" implementation, typically using the PyTorch interpreter.

With NNSmith and its add-ons being the current state-of-the-art high-level neural network generator, we will utilize their tooling in our full compiler stack fuzzing tool. Their approach achieved far higher bug-finding capabilities than many previous works, though they leave significant room for improvement. A critical oversight that the NNSmith authors note is that *low-level* fuzzing approaches such as Tzer introduce several times as many unique coverage paths as high-level fuzzers due to their operation on low-level IRs, which contain operands that cannot be explicitly included at the high level.

This sparsity is where our work will be focused. We can run NNSmith on our target compiler to find inputs that cause inconsistent behavior or crashes. Using these inputs, we will apply the approaches from the low-level fuzzing approaches which we detail further.

B. Tzer

Tzer was one of the first approaches to fuzzing deep learning compilers, published in early 2022 [4]. It targets the TVM backend while only mutating the low-level IR. It leverages various mutation strategies to store both the IR and a pass mutation series in the seed list. Its novelty comes from focusing on *pass sequences*. Essentially, once a high-level seed is lowered into TVM's internal representation, the compiler runs a sequence of optimization passes to improve performance or set up further passes. Critically, each of these passes should still maintain an operationally equivalent form of the neural network.

Critically, storing both the optimization passes P and the original IR F allows Tzer to have two new oracles. Firstly, the optimized version $P(F)$ should have results consistent with the original F when tested on the same data. Secondly, the optimized model should not be slower than the original. Due to its better depth, apparent ease of reproducibility, strong results, and community acceptance, we chose to use Tzer.

A significant oversight that the paper fails to mention is the initial seed pool. They mention that the source of their initial seeding is the TVM model zoo. Looking at its documentation reveals that this only contains commonly used architectures such as VGG and ResNet. We predict that, due to the popularity of these initial seeds, many of these optimization paths are the common case, which are unlikely to have unfixed bugs. The zoo is also a fairly limited set of neural nets. As such, we find it promising to use a diverse yet valid high-level neural network generator, NNSmith, to better seed the low-level fuzzer to uncover uncommon optimization paths and IRs.

IV. Our Approach

One limitation of Tzer is that its bug detection depends on the initial seed pool. Many lower-level generations may not correspond to some real neural network model. Furthermore, many optimizations that low-level compilation introduces depend on certain high-level qualities of the seeds that may not be present when simply randomly fuzzing them at the low level. For example, a high-level model with a `softmax` layer following a `conv2d` combines the two layers at the high level to create unique low-level operators. As such, we can use a high-level model generator to create models that may have more of these qualities.

Our approach combines NNSmith's model generation with Tzer's low-level mutations. We modify Tzer to take any arbitrary ONNX files as initial seeds, rather than its hard-coded seed pool. We also program NNSmith to output its random, valid high-level neural networks as ONNX files. We can then run a variety of heuristics to determine which of the high-level models we deem most effective at finding bugs when fuzzed at the low level. These ONNX files can be loaded and lowered into the Relay IR, TVM's high-level intermediate representation, and used in Tzer's initial seed pool.

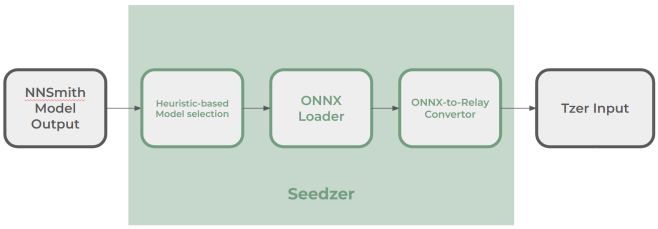

Fig. 3. The Seedzer Pipeline.

V. Experimental Setup

We ran NNSmith on Google Colab with Intel Xeon CPUs running at 2.20GHz. We ran Tzer locally on the prebuilt Docker container provided in the Tzer artifact. Due to difficulties rebuilding both tools, we had to settle with the prebuilt versions of TVM for both, so NNSmith used TVM v0.11, while Tzer used v0.8.

We designed a variety of heuristics implemented in different ways. We have three heuristics prioritizing certain convolutional neural networks (CNNs). The `conv` heuristic prioritizes 1x1 kernel convolutions, the `conv3` heuristic prioritizes 3x3 kernel convolutions, and the `conv5` heuristic prioritizes CNNs with 5x5 kernels or larger. These three heuristics could be implemented as patches to NNSmith, forcing it to only generate models according to the heuristic. We also had the `small` and `big` heuristics, which ran NNSmith with uncapped size. The models were then downloaded locally from Google Colab, and a script was run to sort models according to the number of weights and layers they contained. Lastly, our `basic` heuristic ran NNSmith with only the default parameters, allowing it to generate all models.

Once the models were generated and sorted, their ONNX files were pushed to Tzer's prebuilt Docker container. We also pushed a script to the container that converted these ONNX models to Relay and seeded Tzer. We could then run Tzer for ten-minute runs for each seed pool that each heuristic generated.

As a by-product of our multi-stage pipeline, we were not only able to find bugs that Tzer's fuzzing ran into but also bugs in the library functions we were using to load and convert the ONNX models to Relay.

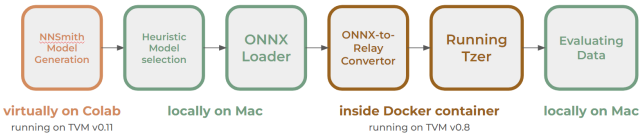

Fig. 4. Our setup.

VI. Evaluation

For our evaluation, we selected the number of bugs found and the diversity of the bugs as our performance metric. The other standard metric used to evaluate fuzzing approaches is coverage, which measures the number of code paths within the TVM library that our runs triggered. We include coverage graphs over the ten-minute runtime spans for each heuristic, though our coverage measurements are slightly flawed. Due to our difficulties building and patching TVM, we could only track Tzer's coverage, rather than the full high-to-low-level pipeline (ie, the functions used in lowering ONNX models to Relay was *not* included in our coverage evaluation).

Experiments were run using ten-minute fuzzing for all runs. Tzer was run with the default initial seed pool, and then with the seed pools generated by each of our heuristics. As shown in Figure 5, the default Tzer seed pool resulted in the least coverage after completing the ten-minute fuzzing. Our heuristic selected seed pools from NNSmith had a significant difference in coverage compared to the default seed pool. However, the choice of heuristic did not make a substantial difference among the different ones we decided for coverage. This similarity in coverage is likely because the selected seeds are generated from the same source, NNSmith.

Figure 6 shows the number of buggy test cases that were generated during the 10-minute run of Tzer for each of the heuristics. We also include a breakdown of which of these cases were segmentation faults, rather than inconsistencies or other crashing cases. This time, only the `basic` heuristic outperformed the default seeding. It's important to note that this is only a measure of the number of buggy cases, and is *not* the number of unique bugs found, which may skew the results. Another trend to mention is that even though the `conv` and the `conv3` heuristics encountered the same number of total bugs, the `conv` heuristic, which uses solely 1x1 convolutions, faced far more segmentation faults than its 3x3 counterpart.

An explanation for this may be due to 3x3 convolutions being a more commonly used operation, which suggests that TVM has been more carefully scrutinized for 3x3 convolutions.

The seed pools generated by the `big` and `small` heuristics are not graphed as bugs in the loader caused them to not be able to run Tzer for a full ten-minute run. We are still investigating the cause of this.

We are still in the process of investigating the root cause of many of these bugs, so a complete breakdown of unique bugs is left for future work. We have included a few bugs that we manually investigated in Appendix A, some of which have recently been encountered in practice and posted by users on both the TVM forum and its GitHub.

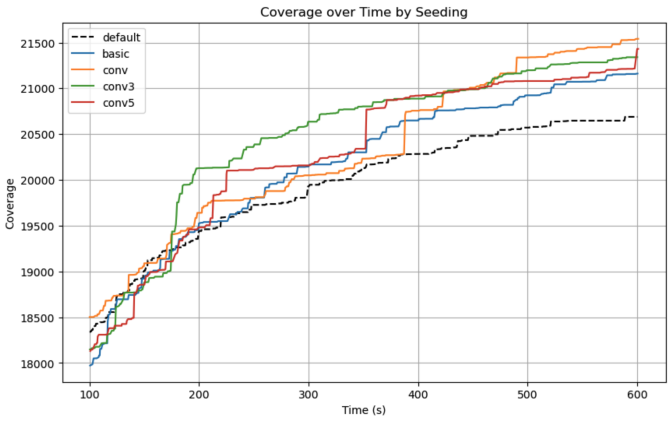

Fig. 5. TVM coverage when running Tzer with different initial seeds.

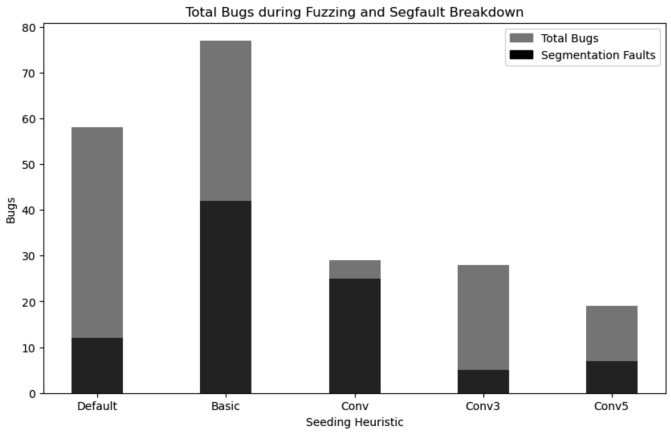

Fig. 6. Number of TVM bugs found for different seeds.

VII. Future Work

The first step of future work in fuzzing TVM using our approach is to refine the pipeline. There are a variety of changes we would make. Running all the tools on the same machine, using the same version of TVM throughout, would be the first step. This way, we could patch TVM to measure the coverage throughout the entire codebase, rather than just the part that TVM measures itself. Once the entire pipeline was automated, we could then implement some sort of feedback-guided NNSmith generation techniques. For example, on Epoch 1, we could seed Tzer randomly, and then based on its performance and coverage paths, we could intelligently tune the models that NNSmith generates to seed Epoch 2 more effectively. This cycle could provide better coverage and remove the need for manually selected heuristics.

We did not do so in this paper, but we also could have run the experiments with varying lengths of time. The original Tzer paper runs its fuzzing experiments for six hours, though the majority of its coverage paths are encountered within the first ten minutes. We can test whether our full-fuzzing pipeline results in better performance over a longer period.

In a broader scope, our approach to full-compiler fuzzing is currently bound by the selections of NNSmith and Tzer. Though Tzer operates solely on the TVM deep learning compiler, NNSmith generates models that are compiler-independent. Thus, we could replicate this process to seed other low-level fuzzers for different DL compilers to see if similar benefits are possible. On the other hand, NNSmith can be substituted for other high-level fuzzers in model generation. By leaving Tzer constant, we can measure the impact that the model generation algorithm has on the both coverage and bugs found in the Seedzer pipeline.

VIII. Conclusion

In conclusion, our paper presents a pipeline, Seedzer, to enhance bug detection in deep-learning (DL) compilers, focusing on TVM, a widely used DL compiler. By combining the strengths of existing high-level and low-level fuzzing approaches, Seedzer aims to detect bugs more effectively across the entire compiler stack. We integrated NNSmith's high-level model generation with Tzer's low-level fuzzing, allowing for a more comprehensive bug detection process. Our experiments demonstrated that Seedzer outperforms the default seeding method, uncovering a higher number of bugs within TVM. While our results show promising bug-finding capabilities, there is room for refinement and future work. Automated feedback-guided techniques and extending our approach to other DL compilers could further enhance bug detection and improve the robustness of DL compiler optimizations.

References

[1] Sam Altman. Opening keynote. OpenAI DevDay, 2023.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018.

[3] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming 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*, ASPLOS '23. ACM, January 2023.

[4] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation, 2022.

APPENDIX

A. Sample of bugs found

```
Check failed: (!check_type.defined()) is false:
Expected Array[IntImm], but got relay.Constant
```

Type mismatch bug similar to https://github.com/apache/tvm/pull/5276

```
In particular dimension 4 conflicts: 55 does not match (int64)1.
The Relay type checker is unable to show the following types match.
In particular ‘Tensor[(1, 1, 19, 60, 1), bool]‘
does not match ‘Tensor[(1, 1, 19, 60, 55), bool]‘
```

Quantized convolution bug, similar to one found https://github.com/apache/tvm/issues/7878.

```
Check failed: (n.defined()) is false: Found null pointer node while traversing AST.
The previous pass may have generated invalid data.
```

An unsupported operator that should be supported, similar to https://discuss.tvm.apache.org/t/bug-onnx-found-null-pointer-node-while-traversing-ast/14745.

```
Check failed: (false) is false: relay.concatenate requires
all tensors have the same shape on non-concatenating axes
```

A concatenation shape check fails on a valid model.

```
KeyError:’axes’
```

A bug in the loader affects some models from the `big` heuristic. This bug seems to be fixed in the newest version of TVM.

```
tvm.error.OpNotImplemented: The following operators
are not supported for frontend ONNX: Trilu
```

An unimplemented function from TVM is perhaps not a bug but something to note.