Seedzer: A Full-Stack Pipeline for Fuzzing Deep Learning Compilers

NIKHIL KAMATH, University of Maryland, USA
MILIJANA SURBATOVICH, University of Maryland, USA

This is an Undergraduate Abstract. Nikhil Kamath's ACM Student Member Number is 4549354.

1 INTRODUCTION

Deep Neural Networks (DNNs) are becoming increasingly common, spanning different target devices. Critically, neural networks must use hardware-specific optimizations to improve performance and efficiency. Deep-learning (DL) compilers bridge the gap between the DNN software and deployed hardware. 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.

Fuzzing is a popular approach to finding bugs in DL compilers by generating test cases as inputs and comparing outputs with some reference. Several tools have successfully found bugs in TVM, a popular DL compiler with multiple optimization levels. However, all of these tools focus on fuzzing a certain optimization level of TVM: NNSmith only generates high-level models, and Tzer and TVMFuzz only fuzz the low-level optimization passes of TVM. We explore whether a full-compile-stack fuzzing approach outperforms state-of-the-art fuzzing approaches.

We propose Seedzer, a novel 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.

2 BACKGROUND

Apache TVM is a DL compiler designed to generate optimized low-level code for a variety of hardware platforms for inputted deep-learning models [1]. It performs a variety of steps that optimize the model both at the high- and low levels. At the high level, neural networks are represented as computation graphs. TVM's computation graph intermediate representation (IR) is called Relay. The Relay IR is transformed into Tensor IR, during which TVM optimizes the computation graph. These transformation passes may include passes that simplify the graph or merge certain nodes 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. A target-specific pipeline such as LLVM handles compilation from the target IR to machine code.

Due to the complexity of DL compilers, fuzzing is a common approach to automate test-case generation. Fuzzing a DL compiler involves generating some model, compiling it, and asserting that the compiled model performs equivalently to an oracle, such as the PyTorch interpreter. Several works have explored the best ways to improve the coverage and bug-finding capabilities of DL compiler fuzzers. NNSmith describes an approach that generates high-level models for various deep learning frameworks and compilers [2]. NNSmith critically considers structural validity when generating neural networks, only generating valid high-level models and minimizing floating-point exceptions.

Authors' addresses: Nikhil Kamath, University of Maryland, USA; Milijana Surbatovich, University of Maryland, USA, nikhil.k123234@gmail.com.

A drawback that the NNSmith authors note is that *low-level* fuzzing approaches 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. Tzer is one such low-level fuzzer [3]. Tzer begins with a seed pool of low-level models and runs various optimization levels on them, asserting that the model should perform equivalently regardless of how optimized it is. More optimized models should also not perform slower than lesser optimized versions.

Tzer's initial seed pool is an oversight that the paper does not discuss. Tzer is seeded with models from the TVM model zoo, which only contains commonly used architectures such as VGG and ResNet. We suspect that due to the popularity of these initial seeds, many of these optimization paths are commonly encountered and are less likely to have unfixed bugs. We explore the benefits of a diverse yet valid high-level neural network generator, NNSmith, to better seed the low-level fuzzer to uncover uncommon optimization paths and IRs.

3 OUR APPROACH

One limitation of Tzer is that its effectiveness depends on the initial seed pool. 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. Operator fusion combines layers at the high level to create optimized, unique low-level operators.

Seedzer 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.

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, NNSmith used TVM v0.11, while Tzer used v0.8.

Seedzer implements a heuristic-based system to choose which high-level models are best to propagate to Tzer. 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 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. Our heuristics were arbitrarily chosen, though ongoing work is being done in a design-space exploration to find which heuristics may lead to more fragile 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.

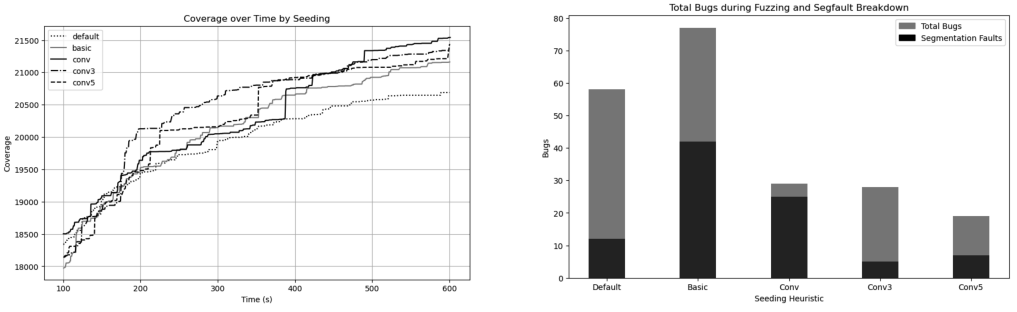
4 EVALUATION

Coverage is a standard metric for evaluating fuzzing techniques, measuring the code paths of the target program that the fuzzing achieves. Higher coverage typically suggests that more unique cases were tested. We outline the coverage of TVM and the number of bugs found by Seedzer. Our coverage measurements were taken from Tzer's instrumentation, though our true coverage may be slightly higher due to the un-instrumented Relay model loading functions Seedzer uses.

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 1a, the default Tzer seed pool resulted in the least coverage after ten minutes. Our heuristic-selected seed pools from NNSmith all had higher coverage than the default seed pool, though the choice of heuristic did not seem to make a substantial difference in coverage. A more thorough exploration of which heuristics are more performant is ongoing.

Figure 1b shows the number of buggy test cases generated during each 10-minute run. We also include a breakdown of which were segmentation faults, rather than inconsistencies or other crashing cases. Only the `basic` heuristic outperformed the default seeding in the absolute number of bugs encountered. 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. The seed pools generated by the `big` and `small` heuristics are not graphed as bugs in the Relay loader prevented them from running Tzer for all ten minutes. We are still investigating the cause of this.

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. In conclusion, our experiments demonstrate that Seedzer's integration of NNSmith's high-level model generation with Tzer's low-level fuzzing allows for a more comprehensive bug detection process. Full-compiler fuzzing techniques are thus promising approaches for DL compilers.



(a) Coverage for each Tzer run.

(b) Bug and SegV breakdown for each Tzer run.

Fig. 1. Quantitative results of the Seedzer pipeline.

5 FUTURE WORK

We are refining the pipeline to use a consistent version of TVM throughout, before re-running Seedzer for a longer period and manually exploring the bugs. We are also working on a design space exploration to more thoroughly investigate which heuristics suggest neural networks with more fragile qualities. Lastly, we are exploring the possibilities of continuous full-compiler feedback, with an epoch-based system that fuzzes the next epoch of Tzer with a seed pool generated based on how the previous epoch's coverage performed.

REFERENCES

[1] 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.

[2] 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.

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

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.