NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers

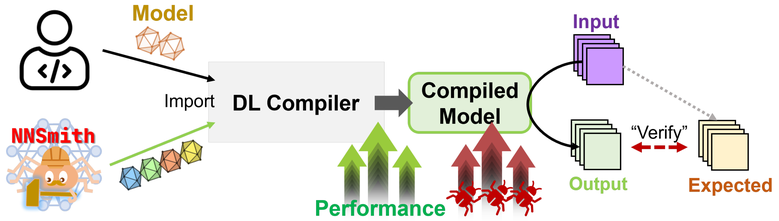
Jiawei Liu[I*] Jinkun Lin[*] Fabian Ruffy[] Cheng Tan[N] Jinyang Li[] Aurojit Panda[] Lingming Zhang[I]

Univiersity of Illinois Urbana-Champaign[I] New York University[] Northeastern University[N] *Co-primary

TL;DR NNSmith is a fuzzer that *automatically* generates *well-formed* models and their inputs for validating DL compilation

Introduction

1. Compilation technologies are increasingly used to optimize DL computation
2. The complex multi-layer compiler stack imposes challenges for correctness
3. Up-to 42% of the codebase are manually-written testing code



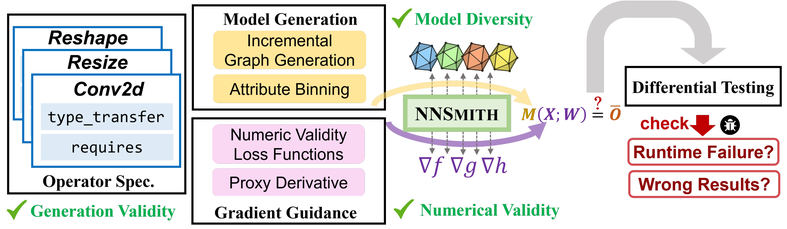
Can we test Deep-Learning compilers *automatically*?

A typical test-case in DL is $T = (M, (X; W), \bar{O})$ where *(i)* M is the DL model; *(ii)* $X; W$ is the model's input; *(iii)* $\bar{O}$ is the expected output for running $M(X; W)$

Test oracle: *(i)* M can be compiled and executed; and *(ii)* $M(X; W) = \bar{O}$

Well-formedness requires *(i)* the construction & "connection" of operators in M to be valid; and *(ii)* computation of $M(X; W)$ to not involve NaNs and ∞

Why *well-formedness*? An invalid model, *i.e.,* violating *(i)*, oftentimes leads the parser to reject the model, leaving other important components untested. While computing operators over NaNs/∞, violating *(ii)*, can lead to false positives (*e.g.,* `cast<int>(NaN)` is UB) and negatives (*i.e.,* model outputs are trivially NaNs/∞)



Overview. NNSmith finds bugs in DL compilation with following steps:

1. **Generation validity:** To generate valid M, for each operator we program a specification of input constraints and tensor type (shape & dtype) propagation
2. **Model diversity:** We construct M by incrementally inserting a randomly selected operator if its input constraints are satisfiable
3. **Numerical validity:** $X; W$ is "learnt" by doing gradient descent for a NaN/∞-inducing operator ϕ, *i.e.,* we minimize the value of a loss function defined from the domain of ϕ for penalizing out-of-domain input values of ϕ
4. **Diff. testing:** a bug is reported if M fails to compile/run or $M(X; W) \neq \bar{O}$

Approach

Operator Specification

We create valid M by incrementally adding an operator to an already-valid M, while preserving the validity. We formalize the validity essentials in the specification below.

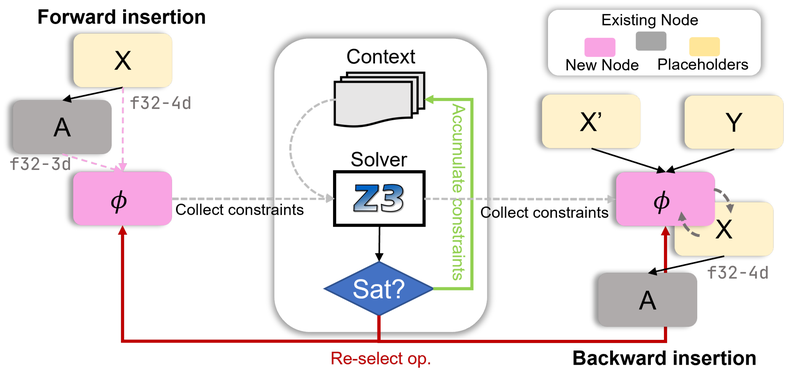
Input constraints of an operator can be described by its attributes and input shape dimensions (*i.e.,* symbolic integers). We use the `requires` method for making such constraints, by solving which a valid operator can be constructed from the solver-provided assignments.

```
class Pool2d(OpBase):
  def __init__(s, kh, kw, pad_h, ...):
    s.kh = kh  # height of kernel size
    …

  def requires(s, itensors) → List[Predicate]:
    ih, iw = itensors[0].shape[2:]
    return [
      0 < s.kh < 2 * s.pad_h + ih,
      0 < s.kw < 2 * s.pad_w + iw,
      0 < s.stride, … ]

  def type_transfer(s, itensors) → List[ATensor]:
    n, c, ih, iw = itensors[0].shape
    oh = (ih + 2*s.pad_h - s.kh) // s.stride + 1
    ow = (iw + 2*s.pad_w - s.kw) // s.stride + 1
    return [  ATensor(
                shape=(n, c, oh, ow),
                dtype=itensors[0].dtype)  ]
```

Type propagation. How to know `itensors` (*i.e.,* input shapes & dtypes) in `requires`? `type_transfer` is such a method to propagate the output tensor types, over these symbolic integers.

Incremental Model Construction

Starting from a placeholder, each time a random operator ϕ is constructed symbolically and inserted in two directions: *(i) forward*: as a consumer, ϕ takes existing tensors as inputs; and *(ii) backward*: as a producer, existing placeholders are replaced by ϕ, which grows new placeholders as inputs.



Searching Inputs with Gradients

Vulnerable operators. What computes NaNs/∞? Running operators with limited stable domain over out-of-domain inputs! *e.g.,* $\log_2(X)$ where $\exists x \in X, x \leq 0$. Such operators are regarded as vulnerable operators.

Gradient guidance. We use *gradient descent* to guide vulnerable operators' inputs to stay in-domain. Once an operator ϕ produces NaNs/∞, we apply a loss function $\mathcal{L}$ over the out-of-domain inputs and minimize the loss via gradient descent. $\mathcal{L}$ is defined by the normalized inequalities from ϕ's domain. For $\log_2$, the inequality of $f(x) = -x \leq 0$ derives $\mathcal{L} = \sum_{x \in X} \max(f(x), 0)$.

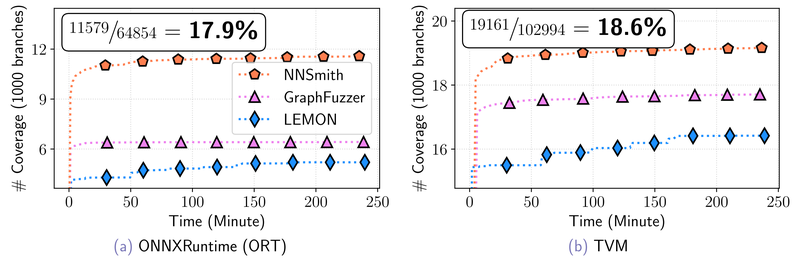
Proxy derivative. Some operators are non-differentiable or zero-derivative at certain regions. Under such circumstances, we proceed the gradient descents by applying constant derivatives whose sign complies with overall trend.

Result Highlights

Bug finding. NNSmith fuzzes the nightly builds of three DL compilers (and by product the PyTorch exporter), finding **72** bugs, **51** (71%) of which have been fixed.

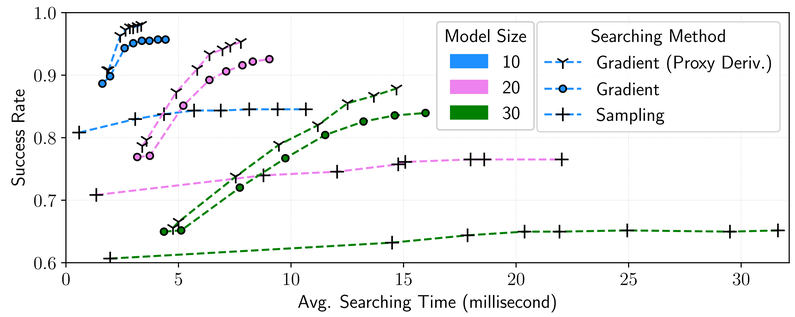
	Transformation	Conversion	Unclassified	**Total**
ONNXRuntime	10	0	2	12
TVM	29	11	0	40
TensorRT	4	2	4	10
PyTorch Exporter	–	10	–	10
Total	43	23	6	72

Branch coverage. After four-hour fuzzing, NNSmith covers around 18-19% system-wide *branch* coverage (more challenging than line cov.), outperforming the 2nd-best tester by **1.8×/1.08×** over ORT/TVM. The improvement over ORT is larger as it is more pattern-sensitive with more graph-level passes.



(a) ONNXRuntime (ORT) (b) TVM

Validity rate of $X; W$. Gradient guidance finds NaN/∞-free inputs for **98%** model samples in 3.5ms (each; on CPU), improving random sampling by up-to **34%**.



Try It Out

Our artifact and implementation are available on PyPI (`nnsmith`), GitHub (`ise-uiuc/nnsmith`), and DockerHub (`ganler/nnsmith-asplos23-ae`).

paper artifact code