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

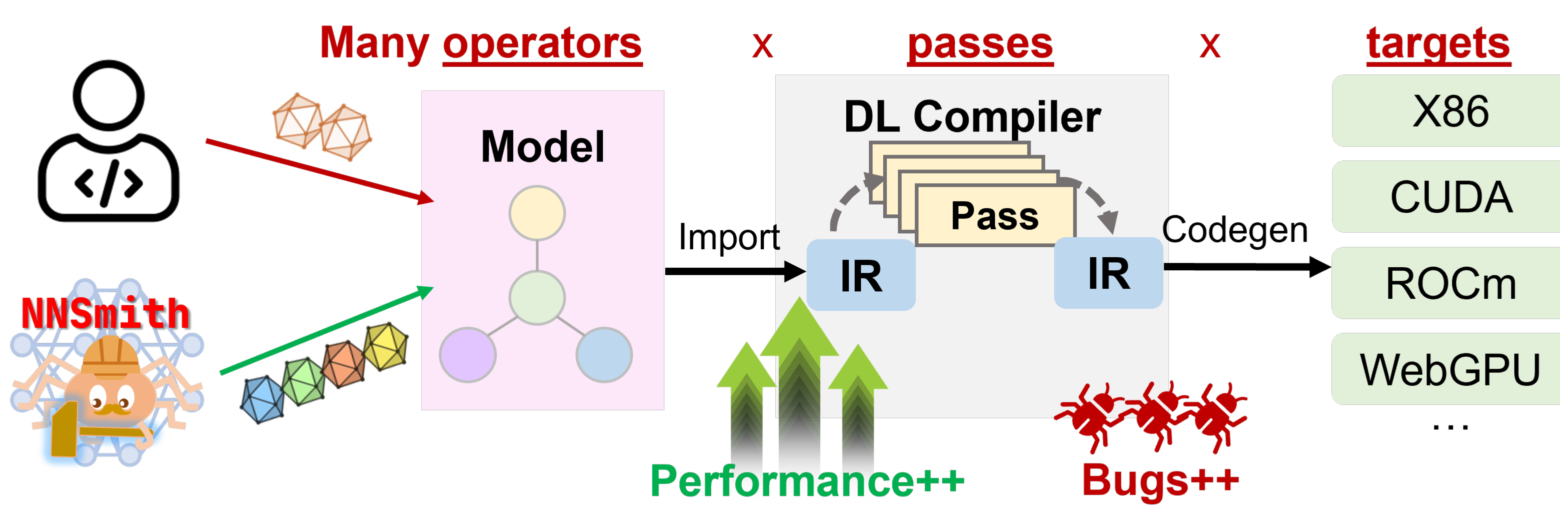
Jiawei Liu[I]* Jinkun Lin[NYU]* Fabian Ruffy[NYU] Cheng Tan[N] Jinyang Li[NYU] Aurojit Panda[NYU] Lingming Zhang[I]

Univeristy of Illinois Urbana-Champaign[I] New York University[NYU] 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*?

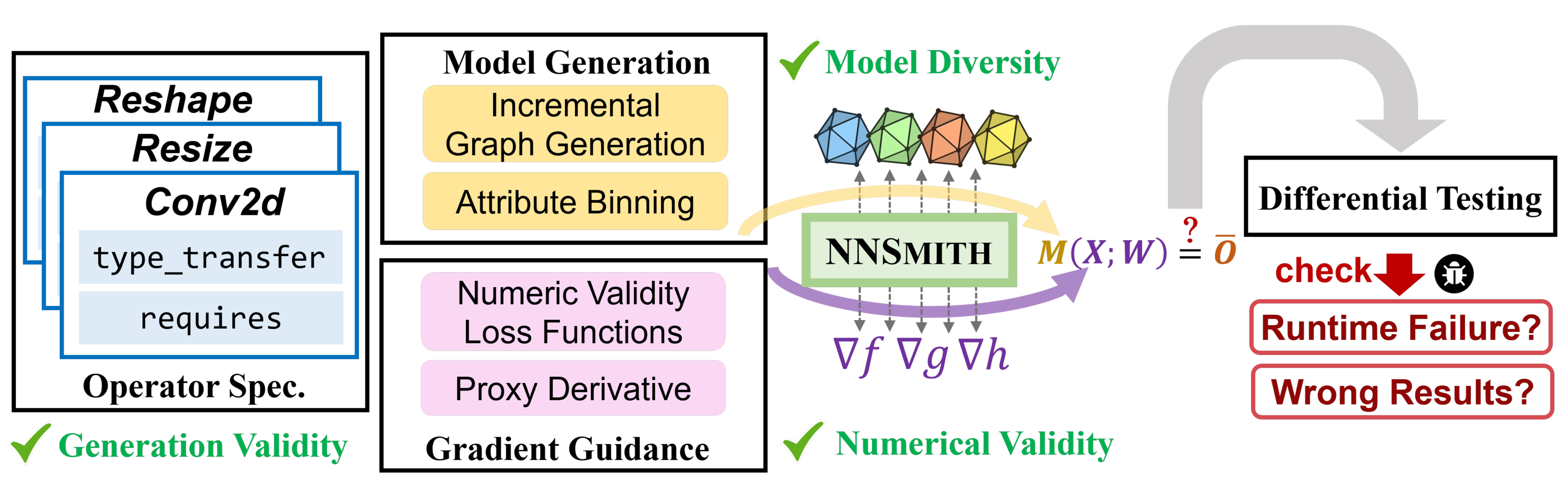
Test oracle: *(i)* M can be compiled and executed; and *(ii)* $M(X;W) = \bar{O}$

$$\mathrm{T} = (M, (X; W), \bar{O})$$

Model, Input, Test-case, Expected output

Well-formedness requires *(i)* 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.*, models always output 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 penalize out-of-domain input values by defining and minimizing loss functions 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.

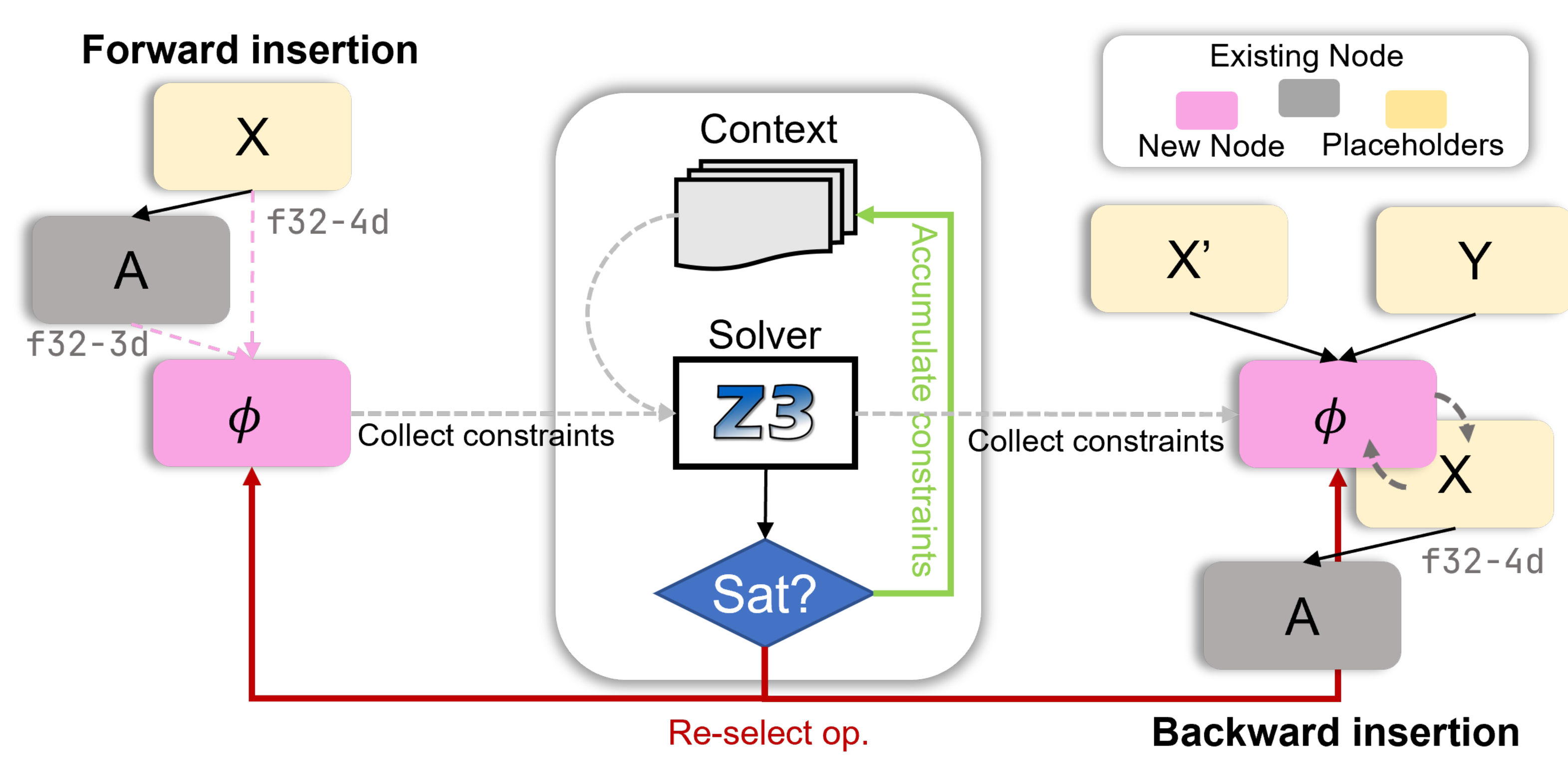
Input constraints of an operator can be described by its attributes and input shape dimensions (*i.e.*, symbolic integers). We use the `requires` method for making such constraints, by solving which a valid operator can be constructed from the solver-provided assignments.

```
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.

Model Construction

Starting from a placeholder, each time a random operator ϕ is symbolically constructed and inserted one of the 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.



Gradient-based Input Search

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

Gradient guidance. 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 according to the inequalities of ϕ's domain. For $\log_2$, the inequality $f(x) = -x \leq 0$ derives $\mathcal{L} = \sum_{x \in X} \max(f(x), 0)$.

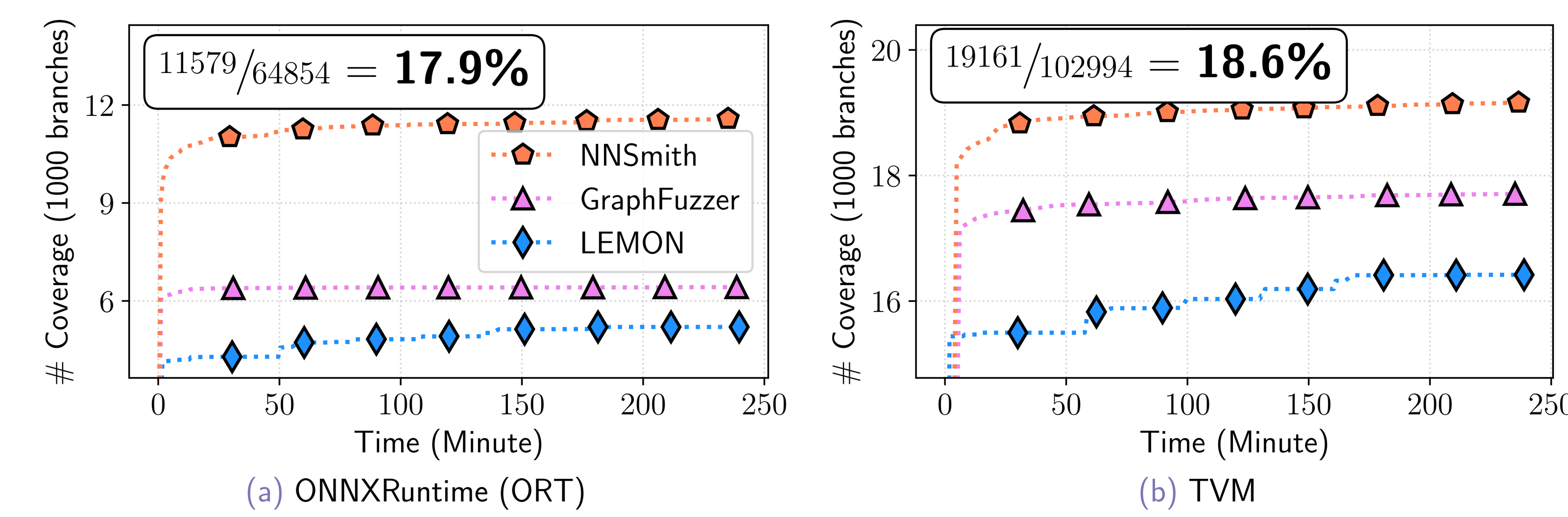
Proxy derivative. Some operators are non-differentiable or zero-derivative at certain regions. Under such circumstances, we proceed the gradient descents by applying constant derivatives whose sign complies with overall trend.

Result Highlights

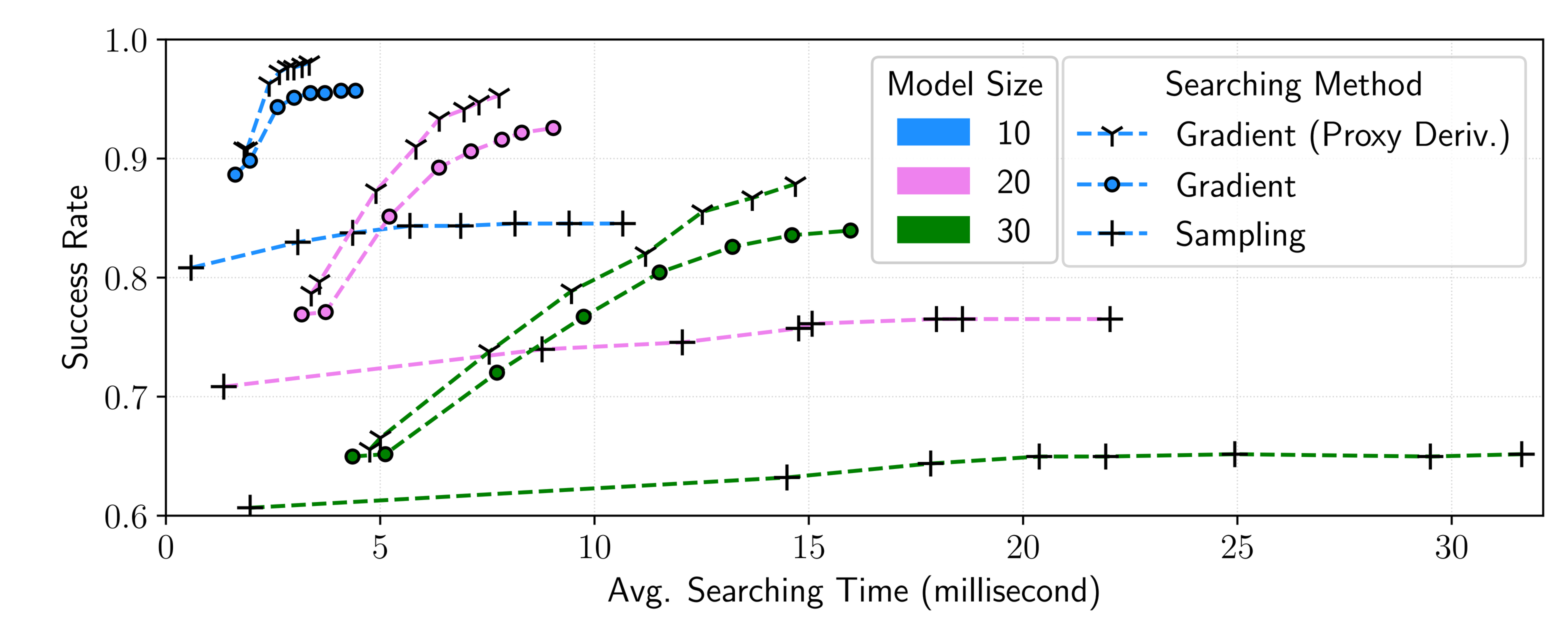
Bug finding. NNSmith fuzzes the nightly builds of three DL compilers (and by product the PyTorch exporter), finding **72** bugs, **51** (71%) of which have been fixed.

- Around 24% are semantic bugs (others are crashing bugs)
- Around 60% are transformation bugs (others are conversion bugs or unclassified)

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.



Validity rate of $X;W$. Gradient guidance finds NaN/∞-free $X;W$ for **98%** model samples in 3.5ms (each; on CPU), improving random sampling by up-to **34%**. However, if we simply do *default initialization*, the validity rate will be only **30-48%**.



Try It Out

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

```
pip install "nnsmith[torch,onnx]"   # install nnsmith
nnsmith.model_gen model.type=onnx   # gen random model
```

paper artifact code