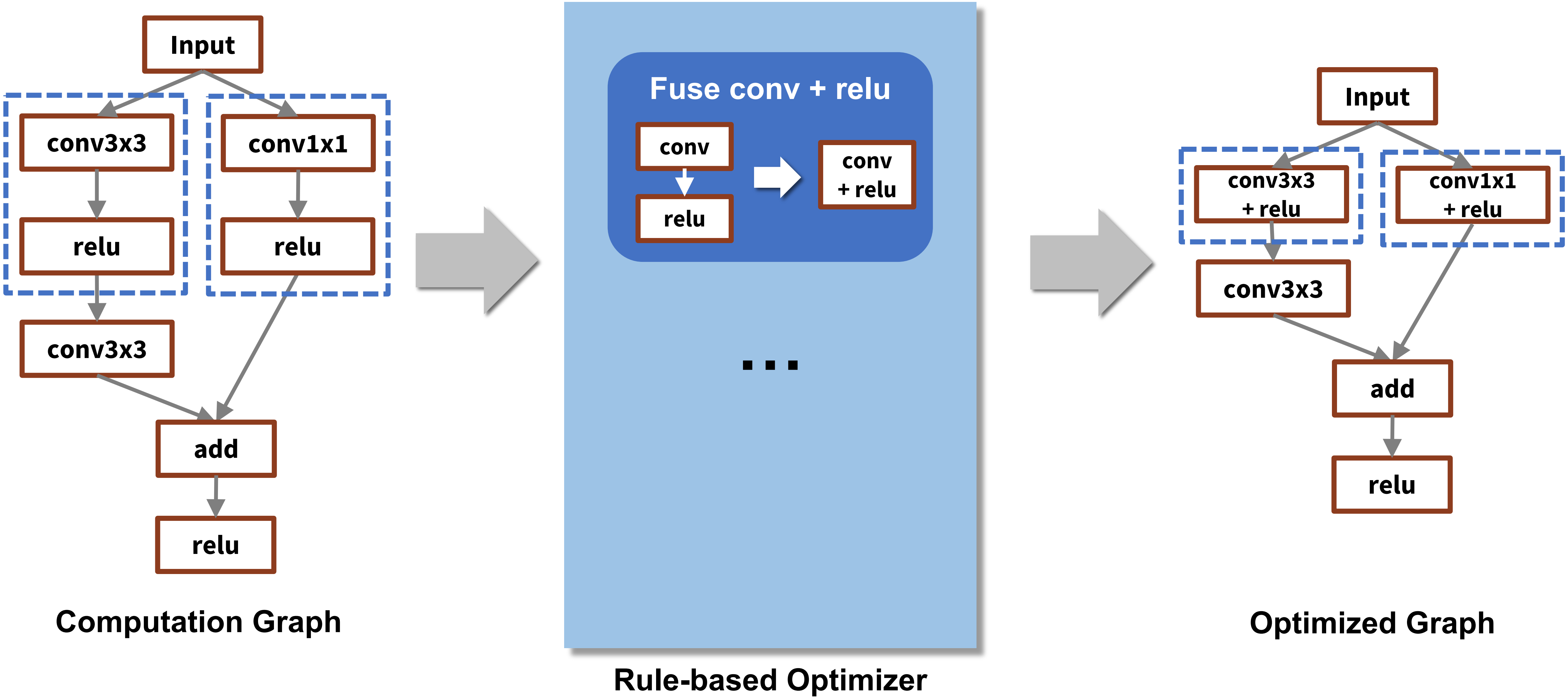
TASO: Optimizing Deep Learning with Automatic Generation of Graph Substitutions

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken

Stanford University

Current Rule-based DNN Optimizations



Current Rule-based DNN Optimizations

TensorFlow currently includes ~200 rules (~53,000 LOC)

Fuse conv + relu

Fuse conv + batch normalization

Fuse multi. convs

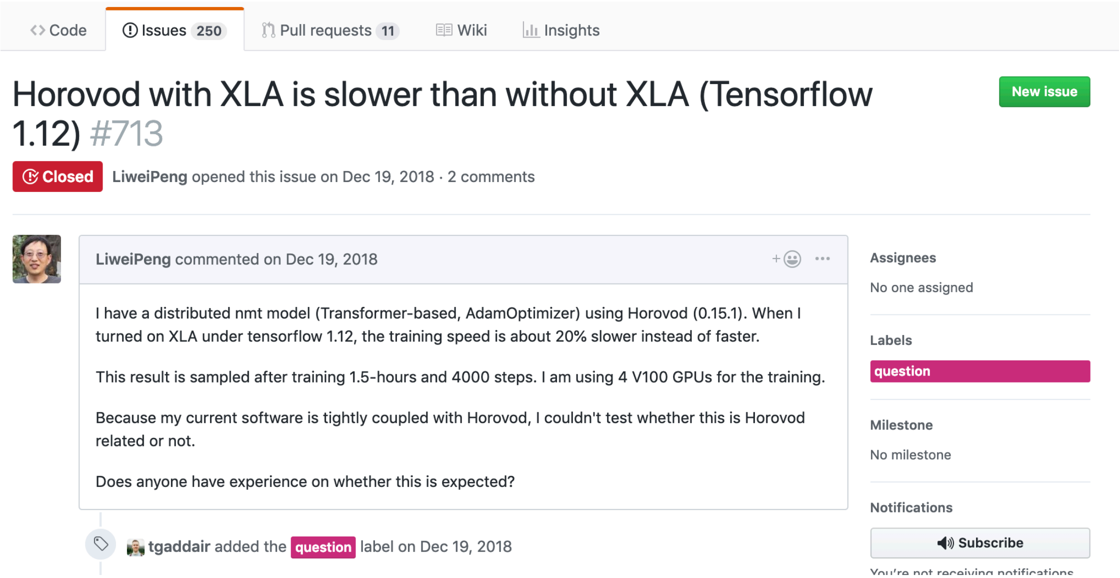
...

Rule-based Optimizer

```cpp
namespace tensorflow {
namespace graph_transforms {

// Converts Conv2D or MatMul ops followed by column-wise Muls into equivalent
// ops with the Mul baked into the convolution weights, to save computation
// during inference.
Status FoldBatchNorms(const GraphDef& input_graph_def,
                      const TransformFuncContext& context,
                      GraphDef* output_graph_def) {
  GraphDef replaced_graph_def;
  TF_RETURN_IF_ERROR(ReplaceMatchingOpTypes(
      input_graph_def,  // clang-format off
      {"Mul",                // mul_node
        {
          {"Conv2D|MatMul|DepthwiseConv2dNative",  // conv_node
            {
              {"*"},         // input_node
              {"Const"},     // weights_node
            }
          },
          {"Const"},         // mul_values_node
        }
      },  // clang-format on
      [](const NodeMatch& match, const std::set<string>& input_nodes,
         const std::set<string>& output_nodes,
         std::vector<NodeDef>* new_nodes) {
        // Find all the nodes we expect in the subgraph.
        const NodeDef& mul_node = match.node;
        const NodeDef& conv_node = match.inputs[0].node;
        const NodeDef& input_node = match.inputs[0].inputs[0].node;
        const NodeDef& weights_node = match.inputs[0].inputs[1].node;
        const NodeDef& mul_values_node = match.inputs[1].node;

        // Check that nodes that we use are not used somewhere else.
        for (const auto& node : {conv_node, weights_node, mul_values_node}) {
          if (output_nodes.count(node.name())) {
            // Return original nodes.
            new_nodes->insert(new_nodes->end(),
                              {mul_node, conv_node, input_node, weights_node,
                               mul_values_node});
            return Status::OK();
          }
        }

        Tensor weights = GetNodeTensorAttr(weights_node, "value");
        Tensor mul_values = GetNodeTensorAttr(mul_values_node, "value");

        // Make sure all the inputs really are vectors, with as many entries as
        // there are columns in the weights.
        int64 weights_cols;
        if (conv_node.op() == "Conv2D") {
          weights_cols = weights.shape().dim_size(3);
        } else if (conv_node.op() == "DepthwiseConv2dNative") {
          weights_cols =
              weights.shape().dim_size(2) * weights.shape().dim_size(3);
        } else {
          weights_cols = weights.shape().dim_size(1);
        }
        if ((mul_values.shape().dims() != 1) ||
            (mul_values.shape().dim_size(0) != weights_cols)) {
          return errors::InvalidArgument(
              "Mul constant input to batch norm has bad shape: ",
              mul_values.shape().DebugString());
        }

        // Multiply the original weights by the scale vector.
        auto weights_vector = weights.flat<float>();
        Tensor scaled_weights(DT_FLOAT, weights.shape());
        auto scaled_weights_vector = scaled_weights.flat<float>();
        for (int64 row = 0; row < weights_vector.dimension(0); ++row) {
          scaled_weights_vector(row) =
              weights_vector(row) *
              mul_values.flat<float>()(row % weights_cols);
        }

        // Construct the new nodes.
        NodeDef scaled_weights_node;
        scaled_weights_node.set_op("Const");
        scaled_weights_node.set_name(weights_node.name());
        SetNodeAttr("dtype", DT_FLOAT, &scaled_weights_node);
        SetNodeTensorAttr<float>("value", scaled_weights, &scaled_weights_node);
        new_nodes->push_back(scaled_weights_node);

        new_nodes->push_back(input_node);

        NodeDef new_conv_node;
        new_conv_node = conv_node;
        new_conv_node.set_name(mul_node.name());
        new_nodes->push_back(new_conv_node);

        return Status::OK();
      },
      {}, &replaced_graph_def));
  *output_graph_def = replaced_graph_def;
  return Status::OK();
}

REGISTER_GRAPH_TRANSFORM("fold_batch_norms", FoldBatchNorms);

}  // namespace graph_transforms
}  // namespace tensorflow
```

Limitations of Rule-based Optimizations

Robustness
Experts' heuristics do not apply to all DNNs/hardware


Code | Issues 250 | Pull requests 11 | Wiki | Insights

Horovod with XLA is slower than without XLA (Tensorflow 1.12) #713

New issue

Closed **LiweiPeng** opened this issue on Dec 19, 2018 · 2 comments

LiweiPeng commented on Dec 19, 2018

I have a distributed nmt model (Transformer-based, AdamOptimizer) using Horovod (0.15.1). When I turned on XLA under tensorflow 1.12, the training speed is about 20% slower instead of faster.

This result is sampled after training 1.5-hours and 4000 steps. I am using 4 V100 GPUs for the training.

Because my current software is tightly coupled with Horovod, I couldn't test whether this is Horovod related or not.

Does anyone have experience on whether this is expected?

tgaddair added the question label on Dec 19, 2018

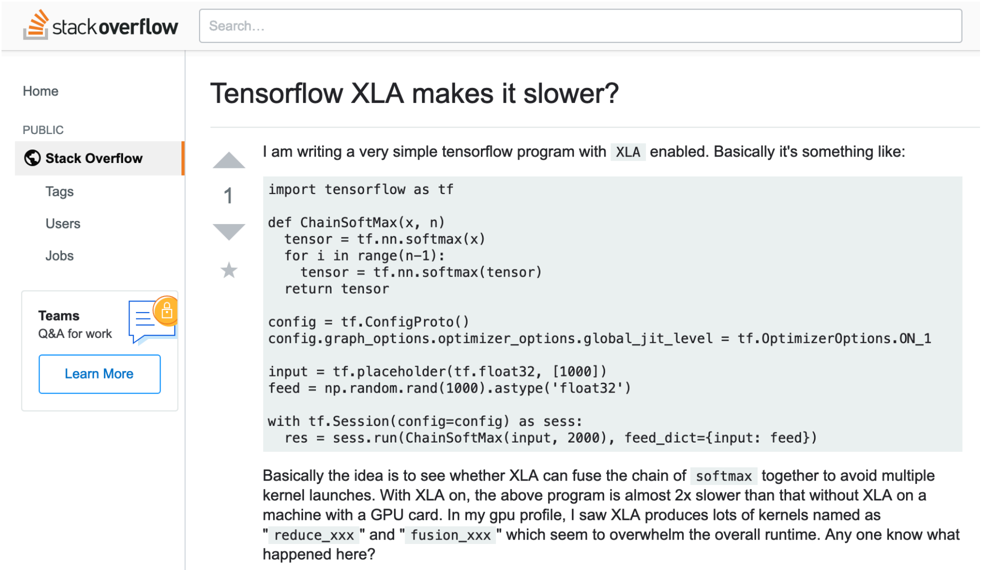
Assignees
No one assigned

Labels
question

Milestone
No milestone

Notifications
Subscribe

When I turned on XLA (TensorFlow's graph optimizer), the training speed is **about 20% slower**.


stackoverflow Search...

Home
PUBLIC
Stack Overflow
Tags
Users
Jobs
Teams
Q&A for work
Learn More

Tensorflow XLA makes it slower?

1

I am writing a very simple tensorflow program with `XLA` enabled. Basically it's something like:

```
import tensorflow as tf

def ChainSoftMax(x, n)
  tensor = tf.nn.softmax(x)
  for i in range(n-1):
    tensor = tf.nn.softmax(tensor)
  return tensor

config = tf.ConfigProto()
config.graph_options.optimizer_options.global_jit_level = tf.OptimizerOptions.ON_1

input = tf.placeholder(tf.float32, [1000])
feed = np.random.rand(1000).astype('float32')

with tf.Session(config=config) as sess:
  res = sess.run(ChainSoftMax(input, 2000), feed_dict={input: feed})
```

Basically the idea is to see whether XLA can fuse the chain of `softmax` together to avoid multiple kernel launches. With XLA on, the above program is almost 2x slower than that without XLA on a machine with a GPU card. In my gpu profile, I saw XLA produces lots of kernels named as " `reduce_xxx` " and " `fusion_xxx` " which seem to overwhelm the overall runtime. Any one know what happened here?

With XLA, my program is **almost 2x slower than** without XLA

Limitations of Rule-based Optimizations

Robustness
Experts' heuristics do not apply to all DNNs/hardware

Scalability
New operators and graph structures require more rules

TensorFlow currently uses ~4K LOC to optimize convolution

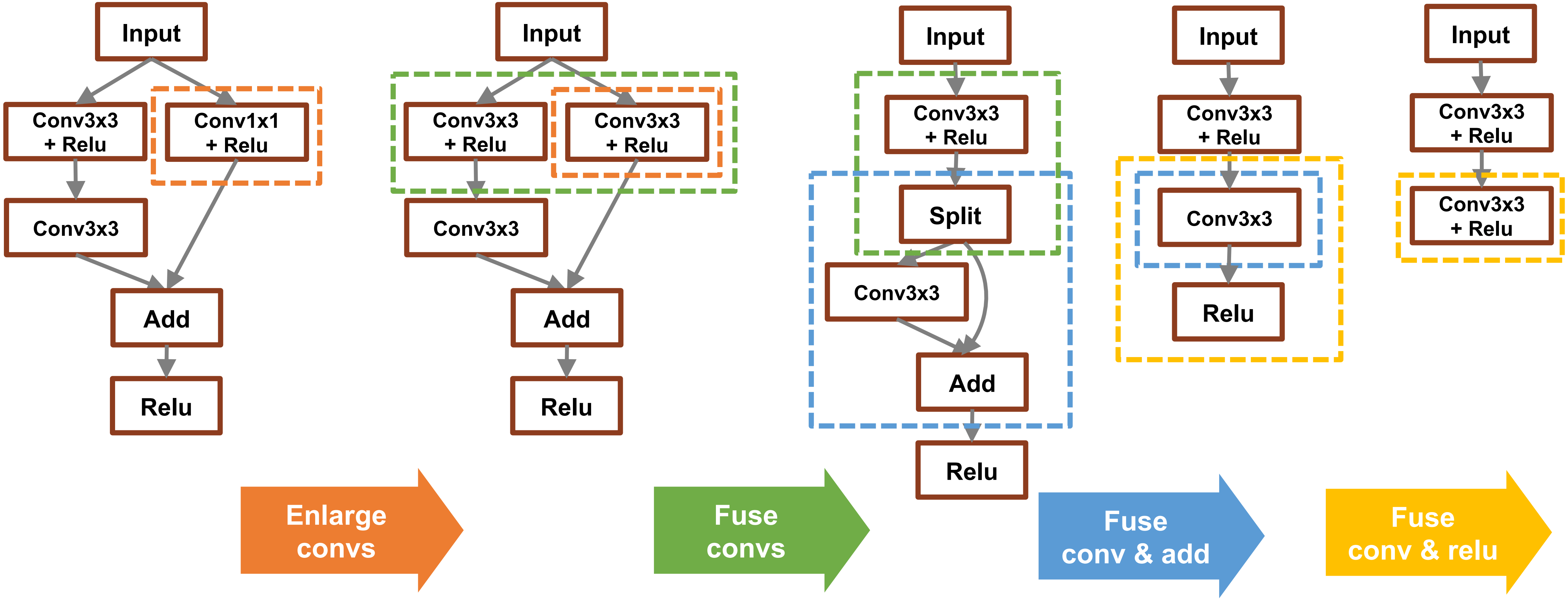
Limitations of Rule-based Optimizations

Robustness
Experts' heuristics do not apply to all DNNs/hardware

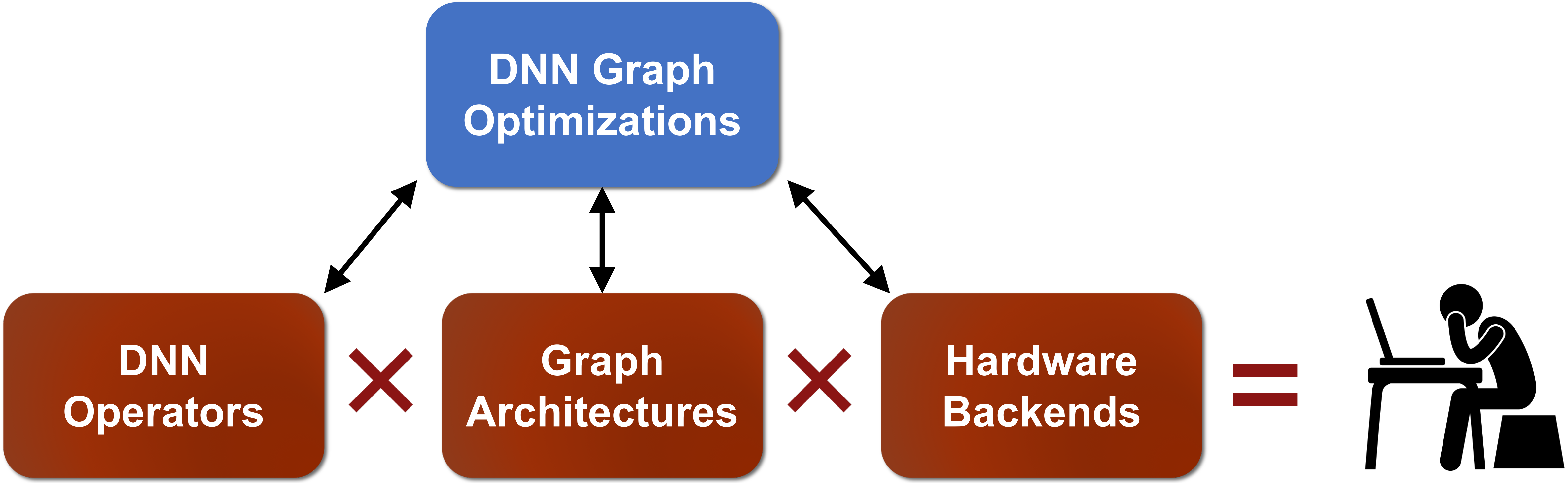
Scalability
New operators and graph structures require more rules

Performance
Miss subtle optimizations for specific DNNs/hardware

Motivating Example



The final graph is <u>30% faster</u> on V100 but <u>10% slower</u> on K80.



How should we address the complexity of designing DNN graph optimizations?

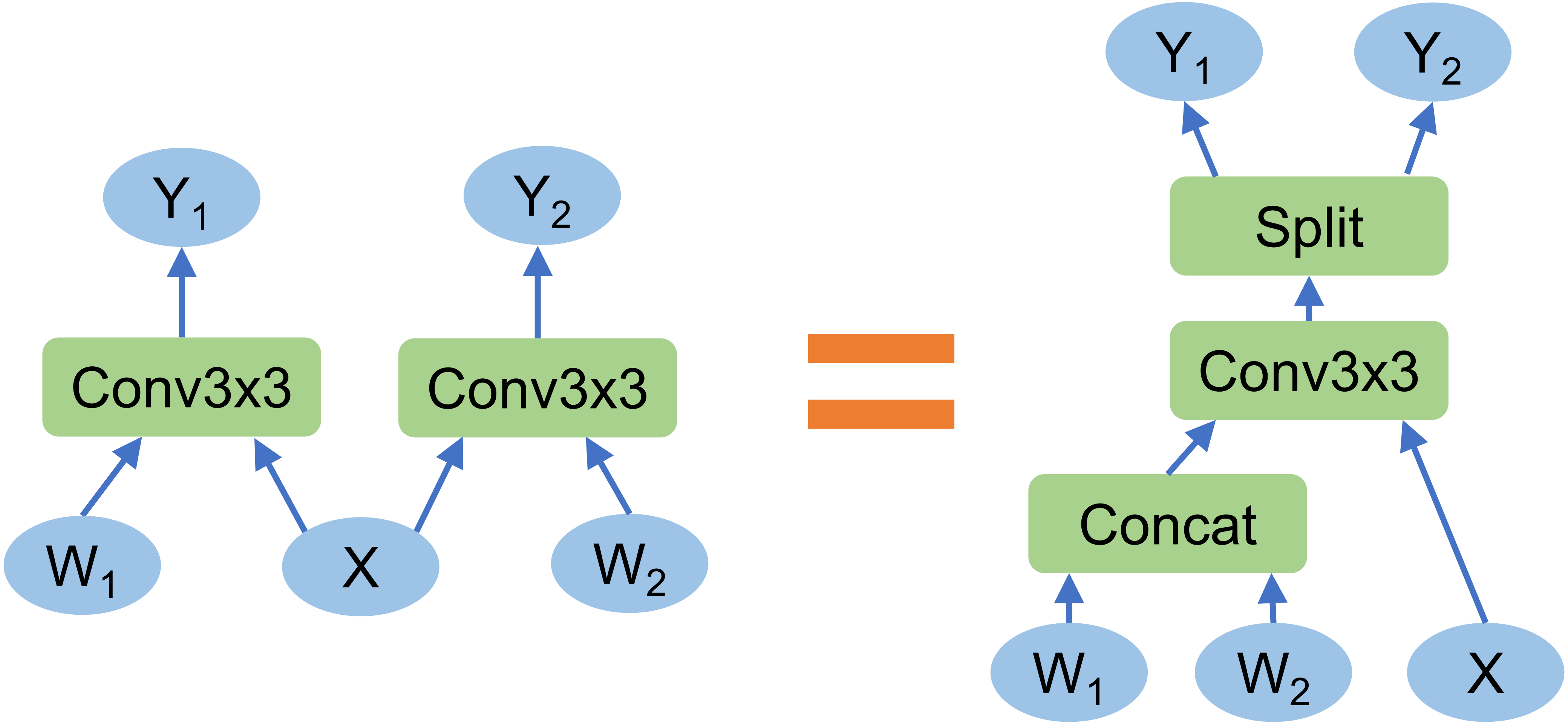
TASO: Tensor Algebra SuperOptimizer

- Key idea: replace manually-designed graph optimizations with ***automated generation and verification*** of graph substitutions for deep learning

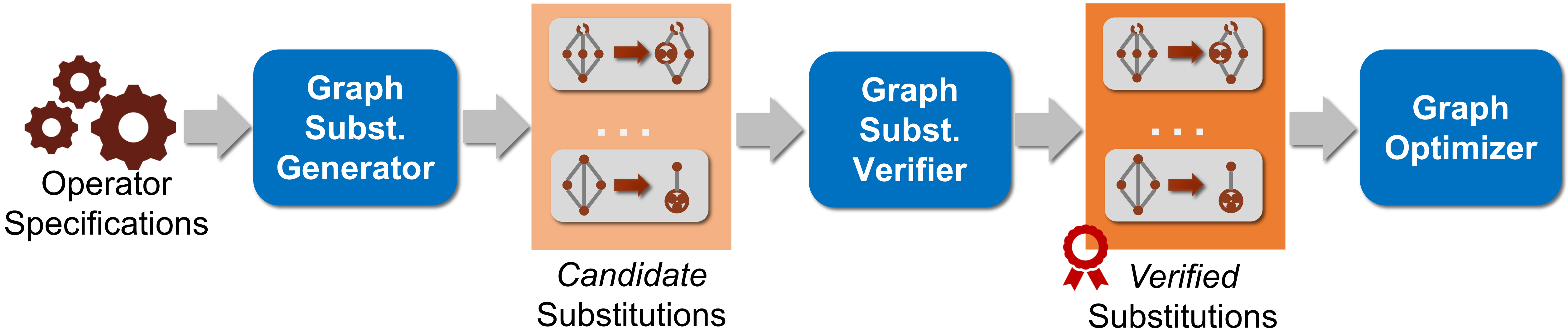
- **Less engineering effort:** **53,000** LOC for manual graph optimizations in TensorFlow → **1,400** LOC in TASO

- **Better performance:** outperform existing optimizers by up to **2.8x**

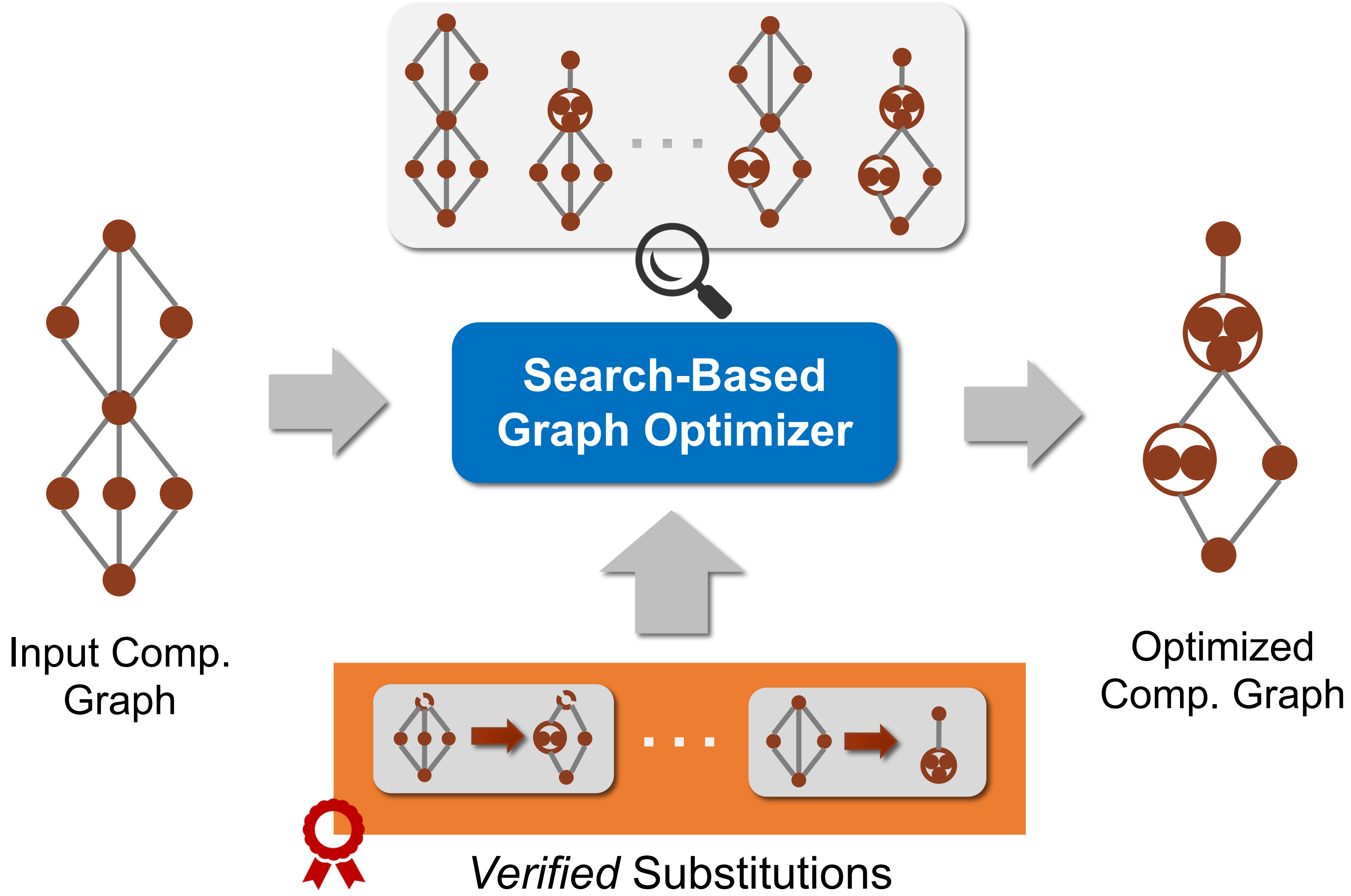
Graph Substitution



TASO Workflow



TASO Workflow



Key Challenges

1. How to generate potential substitutions?

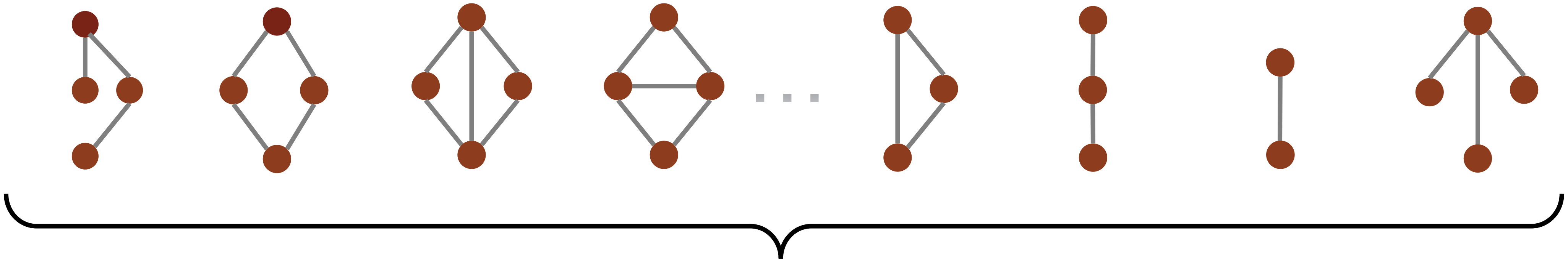
Graph fingerprints

2. How to verify their correctness?

Operator specifications + theorem prover

Graph Substitution Generator

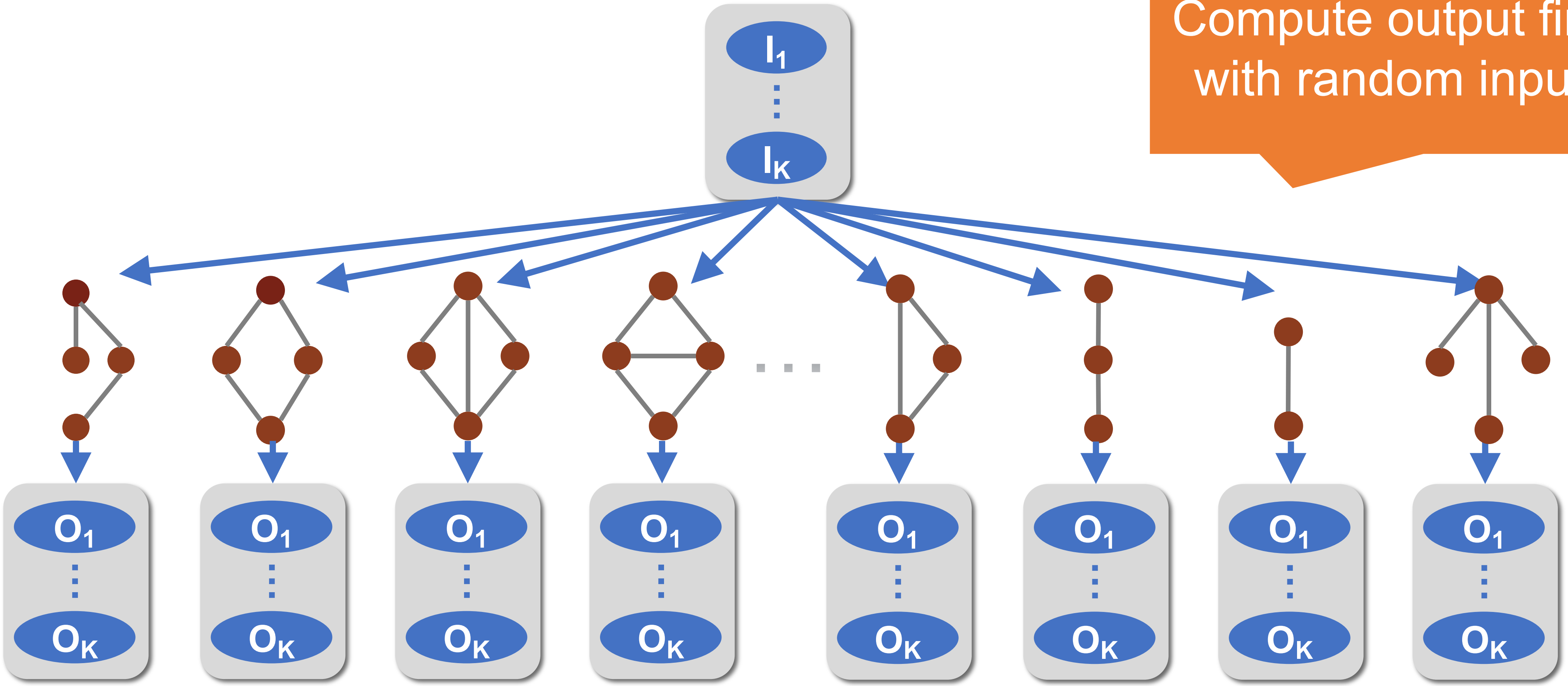
Enumerate all possible subgraphs up to a fixed size using available operators



~66M graphs with up to 4 operators

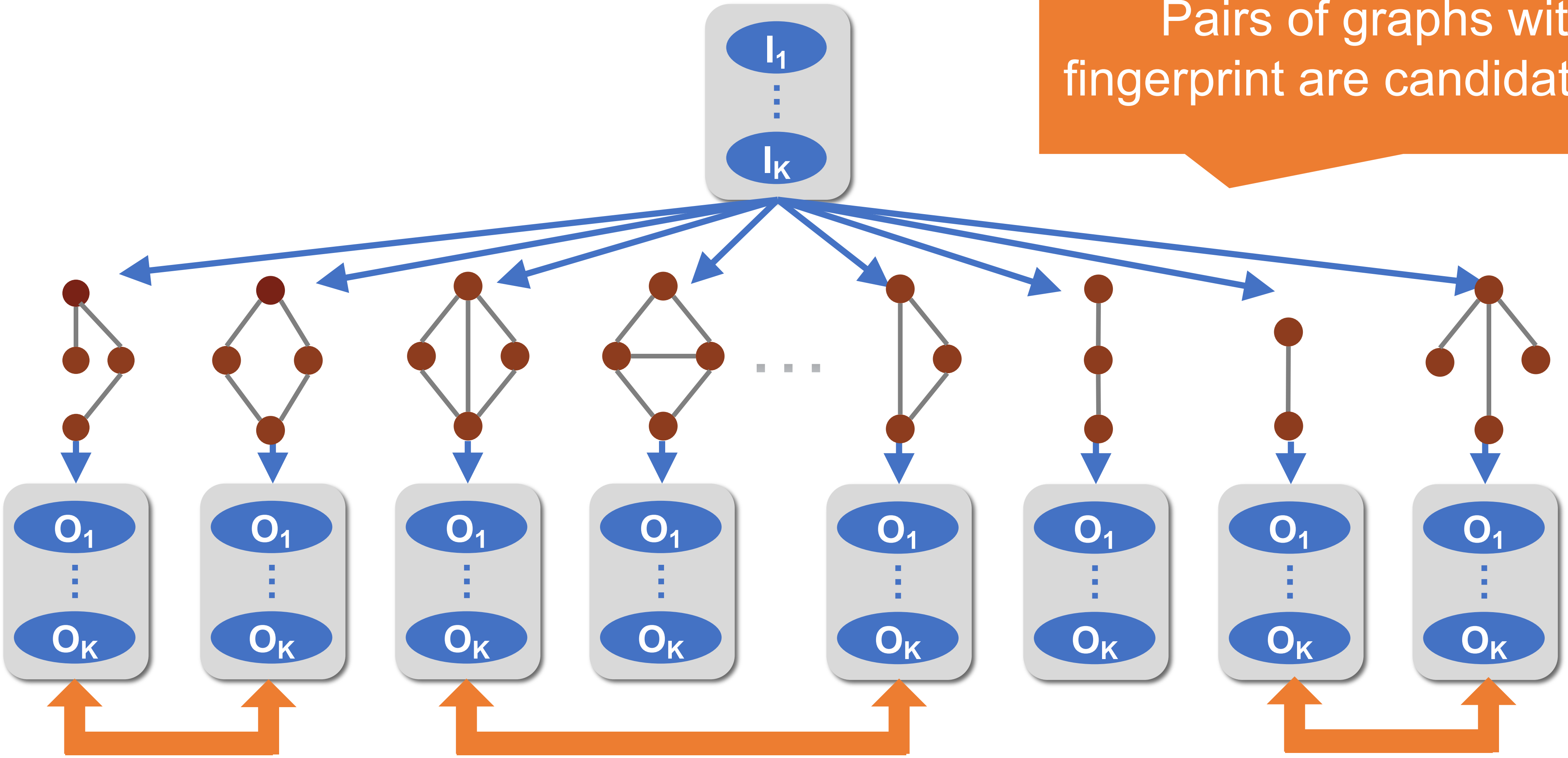
Graph Substitution Generator

Compute output fingerprints with random input tensors



Graph Substitution Generator

Pairs of graphs with identical fingerprint are candidate substitutions

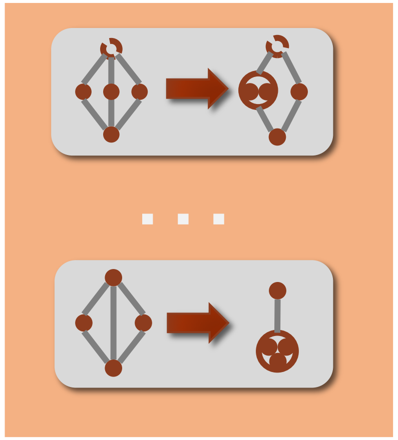


Graph Substitution Generator

TASO generates **~<u>29,000</u>** substitutions by enumerating graphs w/ up to 4 operators

<u>743</u> substitutions remain after applying pruning techniques to eliminate redundancy

Graph Substitution Verifier



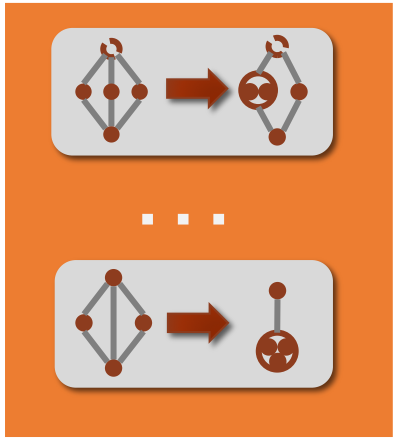
Candidate
Substitutions

Graph Subst. Verifier

P1. conv is distributive over concatenation
P2. conv is bilinear
…
Pn.

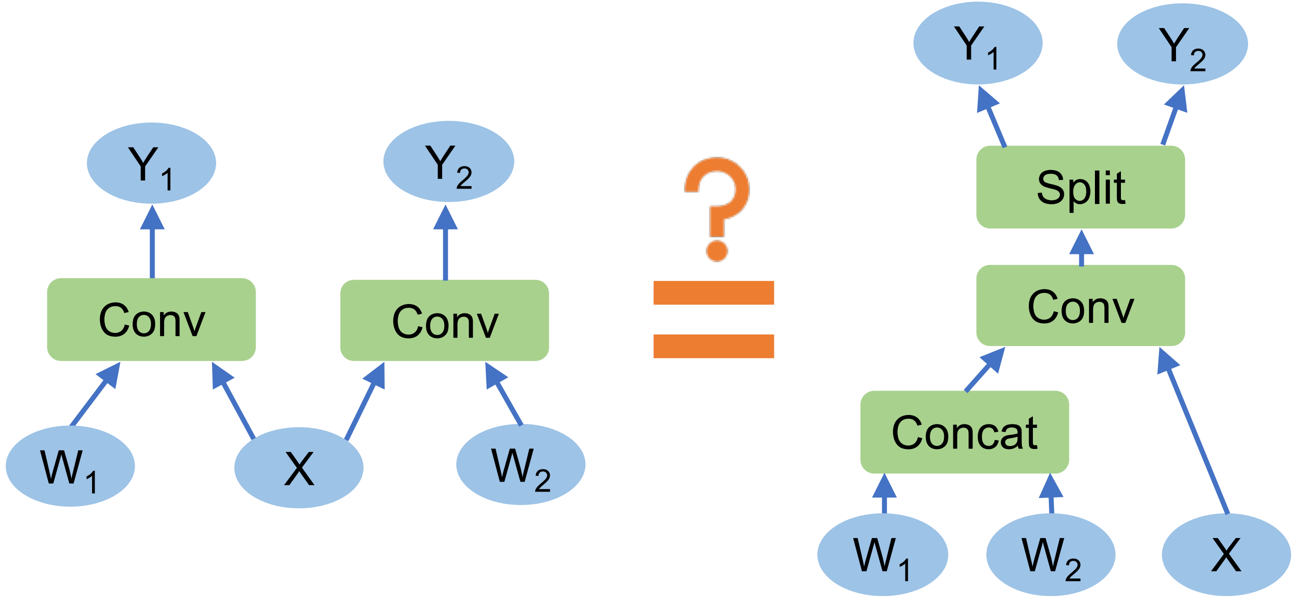
Operator Specifications

$$\forall x, w_1, w_2 \,.$$
$$Conv(x, Concat(w_1, w_2)) = Concat(Conv(x, w_1), Conv(x, w_2))$$



Verified
Substitutions

Verification Workflow



(Conv(x, w_1), Conv (x, w_2))

Split(Conv(x, Concat(w_1, w_2)))

$$\exists x, w_1, w_2 \,.$$
$$\big(Conv(x, w_1), Conv(x, w_2)\big)$$
$$\neq Split\Big(Conv\big(x, Concat(w_1, w_2)\big)\Big)$$

P1. $\forall x, w_1, w_2 \,.$

$Conv\big(x, Concat(w_1, w_2)\big) =$

$Concat\big(Conv(x, w_1), Conv(x, w_2)\big)$

P2. …

Operator Specifications

Theorem Prover

UNSAT

Verification Effort

TASO generates all **743** substitutions in 5 minutes, and verifies them against **43** operator properties in 10 minutes

Supporting a new operator requires **a few hours** of human effort to discover its properties

Operator specifications in TASO ≈ **1,400** LOC
Manual graph optimizations in TensorFlow ≈ **53,000** LOC

Operator Property	Comment
$\forall x, y, z.\ \texttt{ewadd}(x, \texttt{ewadd}(y, z)) = \texttt{ewadd}(\texttt{ewadd}(x, y), z)$	ewadd is associative
$\forall x, y.\ \texttt{ewadd}(x, y) = \texttt{ewadd}(y, x)$	ewadd is commutative
$\forall x, y, z.\ \texttt{ewmul}(x, \texttt{ewmul}(y, z)) = \texttt{ewmul}(\texttt{ewmul}(x, y), z)$	ewmul is associative
$\forall x, y.\ \texttt{ewmul}(x, y) = \texttt{ewmul}(y, x)$	ewmul is commutative
$\forall x, y, z.\ \texttt{ewmul}(\texttt{ewadd}(x, y), z) = \texttt{ewadd}(\texttt{ewmul}(x, z), \texttt{ewmul}(y, z))$	distributivity
$\forall x, y, w.\ \texttt{smul}(\texttt{smul}(x, y), w) = \texttt{smul}(x, \texttt{smul}(y, w))$	smul is associative
$\forall x, y, w.\ \texttt{smul}(\texttt{ewadd}(x, y), w) = \texttt{ewadd}(\texttt{smul}(x, w), \texttt{smul}(y, w))$	distributivity
	commutativity
	is its own inverse
	ommutativity
	ommutativity
	ommutativity
	associative
	inear
	inear
	d transpose
	inear
	conv is bilinear
$\forall s, p, x, y, z.\ \texttt{conv}(s, p, A_{\texttt{none}}, x, \texttt{ewadd}(y, z)) = \texttt{ewadd}(\texttt{conv}(s, p, A_{\texttt{none}}, x, y), \texttt{conv}(s, p, A_{\texttt{none}}, x, z))$	conv is bilinear
	inear
	nvolution kernel
	$A_{\texttt{relu}}$ applies relu
	ommutativity
	conv. with $C_{\texttt{pool}}$
	rnel
	atrix
	entity
$\forall a, x, y.\ \texttt{split}_0(a, \texttt{concat}(a, x, y)) = x$	split definition
	ition
	f concatenation
	ommutativity
	ommutativity
	ommutativity
	ommutativity
	ion and transpose
	ion and matrix mul.
	ion and matrix mul.
	tion and conv.
	concatenation and conv.
$\forall s, p, x, y, z, w.\ \texttt{conv}(s, p, A_{\texttt{none}}, \texttt{concat}(1, x, z), \texttt{concat}(1, y, w)) = \texttt{ewadd}(\texttt{conv}(s, p, A_{\texttt{none}}, x, y), \texttt{conv}(s, p, A_{\texttt{none}}, z, w))$	concatenation and conv.
$\forall k, s, p, x, y.\ \texttt{concat}(1, \texttt{pool}_{\texttt{avg}}(k, s, p, x), \texttt{pool}_{\texttt{avg}}(k, s, p, y)) = \texttt{pool}_{\texttt{avg}}(k, s, p, \texttt{concat}(1, x, y))$	concatenation and pooling
$\forall k, s, p, x, y.\ \texttt{concat}(0, \texttt{pool}_{\texttt{max}}(k, s, p, x), \texttt{pool}_{\texttt{max}}(k, s, p, y)) = \texttt{pool}_{\texttt{max}}(k, s, p, \texttt{concat}(0, x, y))$	concatenation and pooling
$\forall k, s, p, x, y.\ \texttt{concat}(1, \texttt{pool}_{\texttt{max}}(k, s, p, x), \texttt{pool}_{\texttt{max}}(k, s, p, y)) = \texttt{pool}_{\texttt{max}}(k, s, p, \texttt{concat}(1, x, y))$	concatenation and pooling

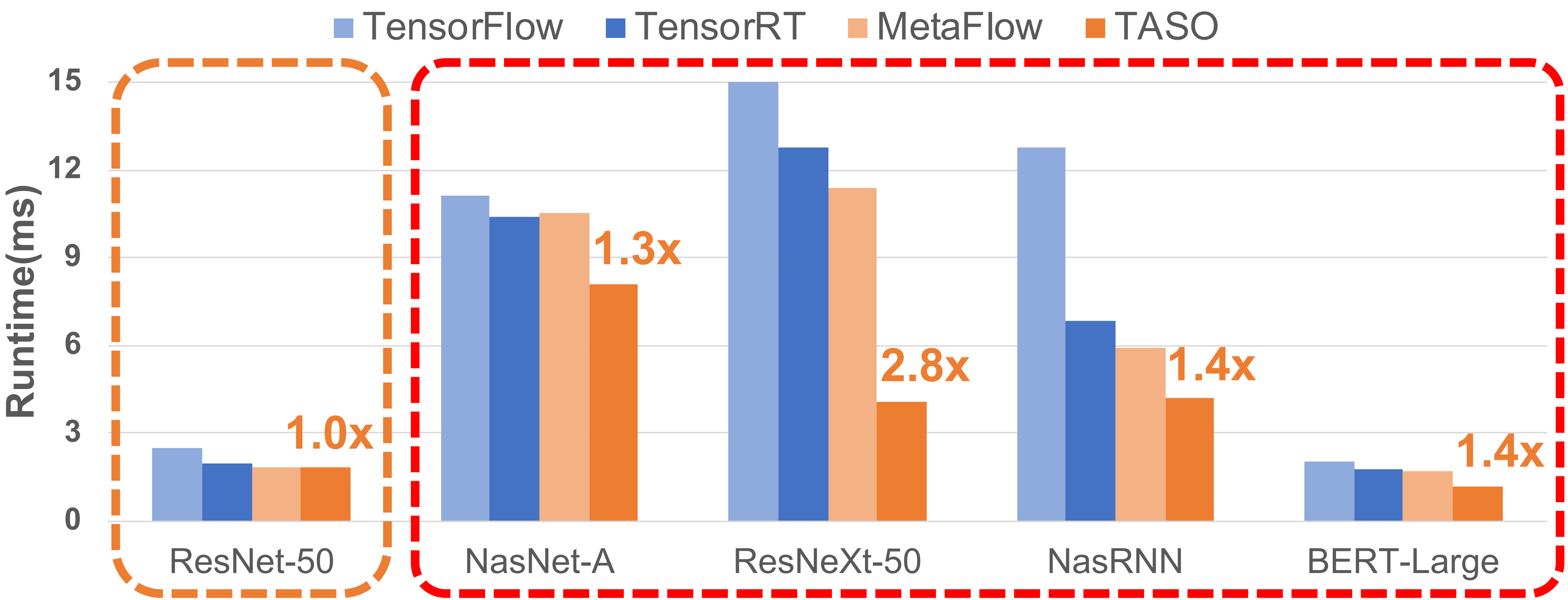
Search-Based Graph Optimizer (MetaFlow [SysML19])

- **Goal**: applying verified substitutions to obtain an optimized graph

- **Cost model**
  - Based on the sum of individual operators' cost
  - Measure the cost of each operator on hardware

- **Cost-based backtracking search**
  - Backtrack local optimal solutions
  - Optimizing a DNN model takes less than **10** minutes

End-to-end Inference Performance (V100 GPU w/ cuDNN)



Competitive on standard models

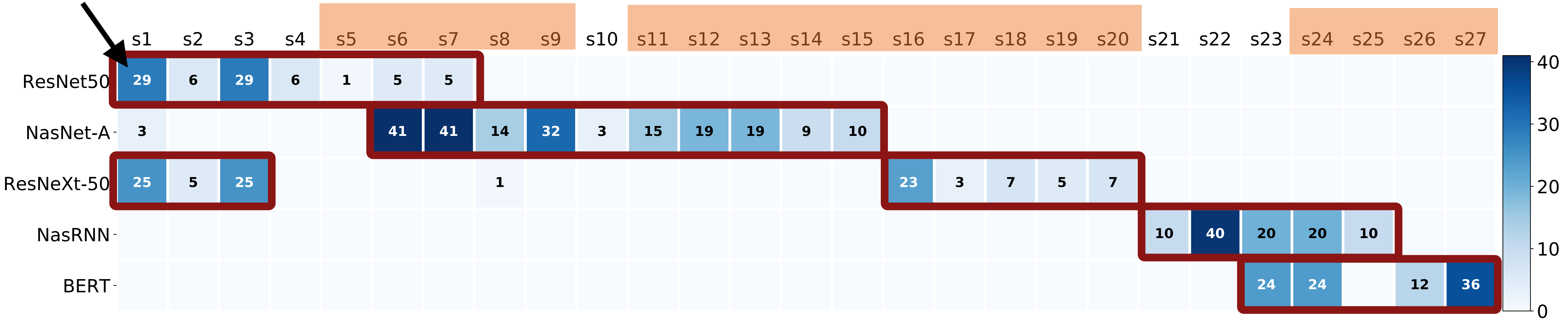
Larger speedups on emerging models

Similar speedups on the TVM backend

Heatmap of Used Substitutions

Not covered in TensorFlow

How many times a subst. is used to optimize a DNN



Different DNN models require **different** substitutions.

Conclusion

TASO is the first DNN optimizer that automatically generates substitutions

- Less engineering effort
- Better performance
- Formal verification

https://github.com/jiazhihao/taso

- Support DNN models in ONNX, TensorFlow, and PyTorch