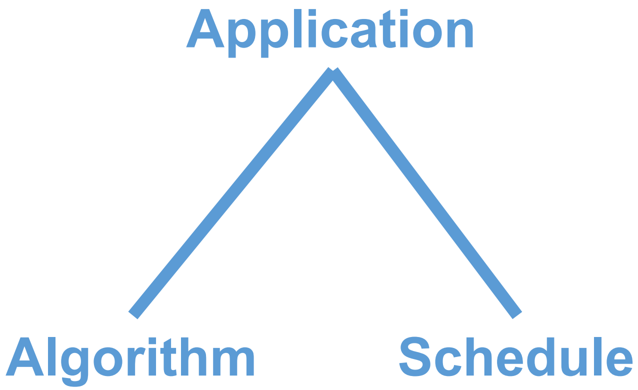
Comparing Halide, TVM, and Ansor



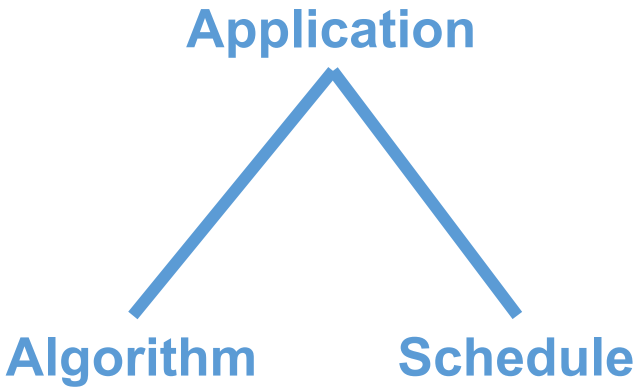
Halide: separate algorithm and schedule, require manual schedules

```
Func halide_blur(Func in) {
 Func tmp, blurred;
 Var x, y, xi, yi;

 // The algorithm
 tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

 // The schedule
 blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
 tmp.chunk(x).vectorize(x, 8);

 return blurred;
}
```



Application

Algorithm

Schedule

Halide: separate algorithm and schedule, require manual schedules

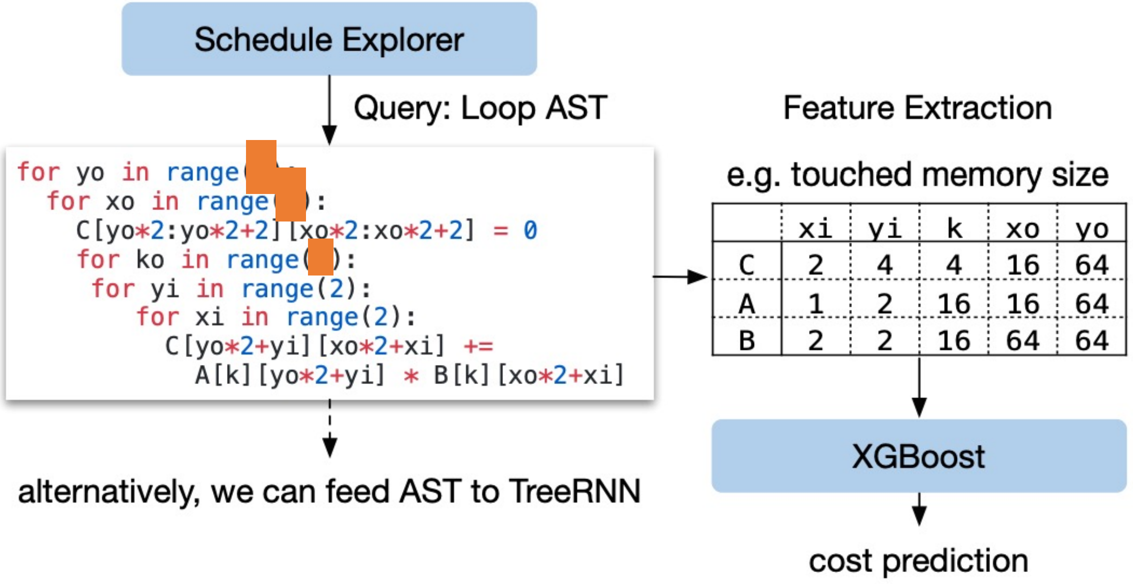
```
Func halide_blur(Func in) {
 Func tmp, blurred;
 Var x, y, xi, yi;

 // The algorithm
 tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

 // The schedule
 blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
 tmp.chunk(x).vectorize(x, 8);

 return blurred;
}
```

TVM: require users to specify a schedule space, use ML to explore the space



Schedule Explorer

Query: Loop AST

```
for yo in range([redacted]):
  for xo in range([redacted]):
    C[yo*2:yo*2+2][xo*2:xo*2+2] = 0
    for ko in range([redacted]):
     for yi in range(2):
        for xi in range(2):
          C[yo*2+yi][xo*2+xi] +=
            A[k][yo*2+yi] * B[k][xo*2+xi]
```

alternatively, we can feed AST to TreeRNN

Feature Extraction

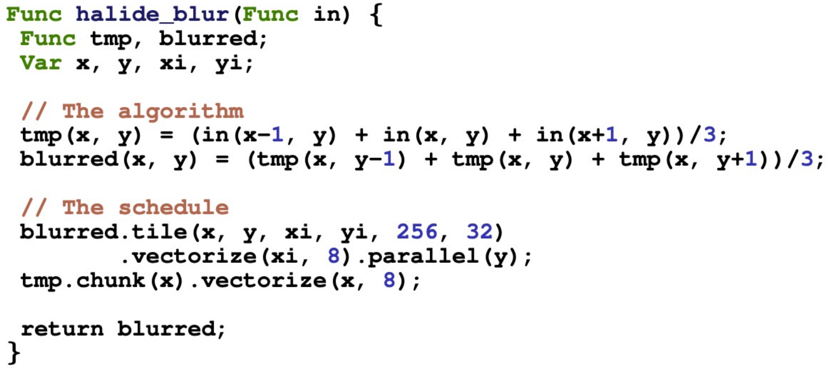
e.g. touched memory size

	xi	yi	k	xo	yo
C	2	4	4	16	64
A	1	2	16	16	64
B	2	2	16	64	64

XGBoost

cost prediction

Halide: separate algorithm and schedule, require manual schedules

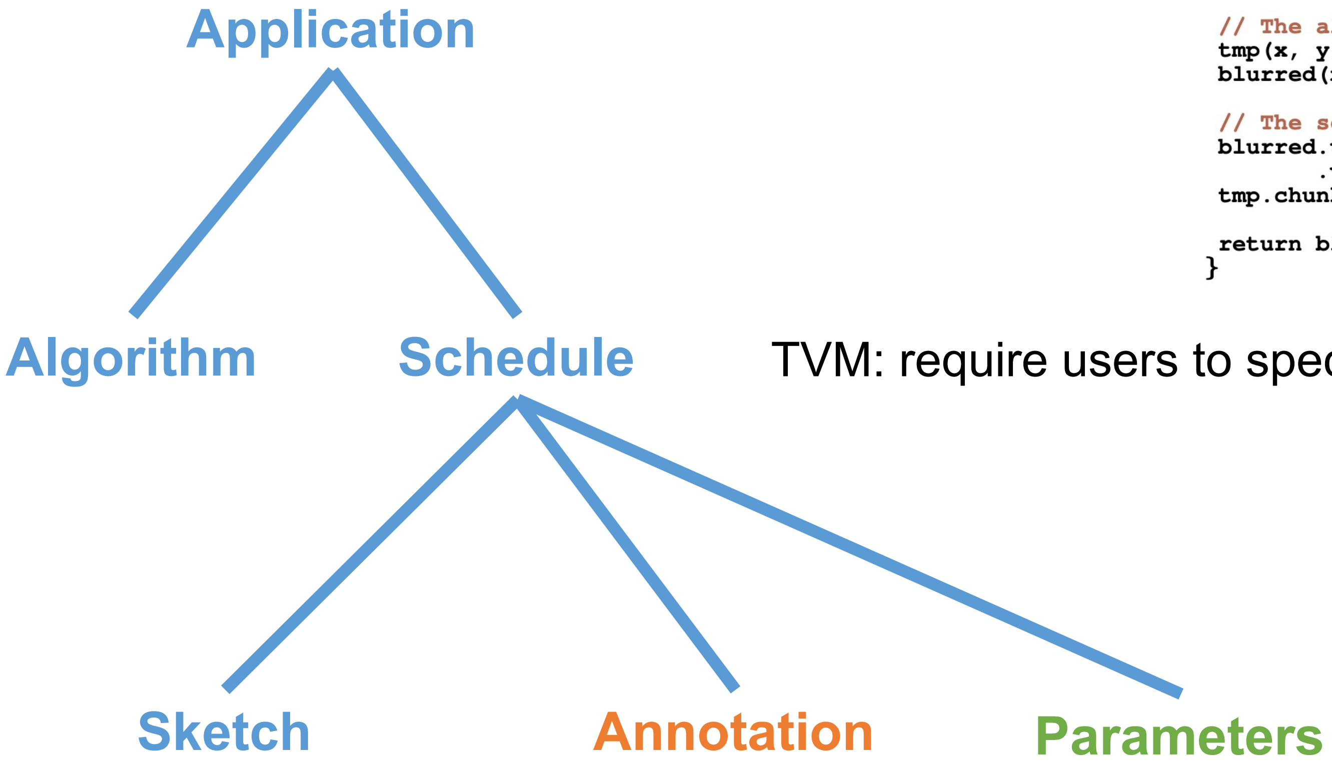


```
Func halide_blur(Func in) {
 Func tmp, blurred;
 Var x, y, xi, yi;

 // The algorithm
 tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

 // The schedule
 blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
 tmp.chunk(x).vectorize(x, 8);

 return blurred;
}
```



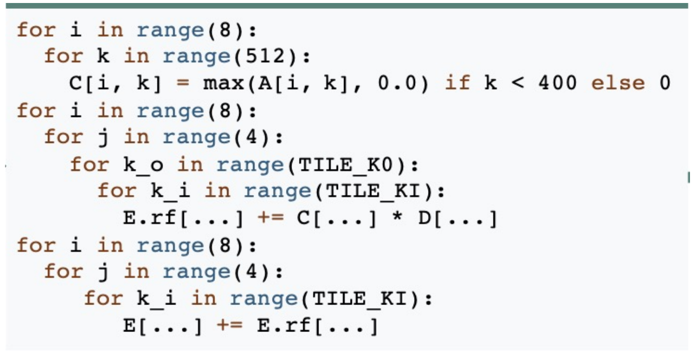
Application

Algorithm **Schedule**

TVM: require users to specify a schedule space, use ML to explore the space

Sketch **Annotation** **Parameters**

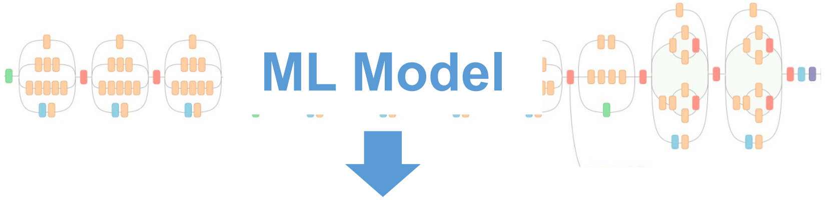
Ansor: generate random sketches and annotations, auto-tune parameters



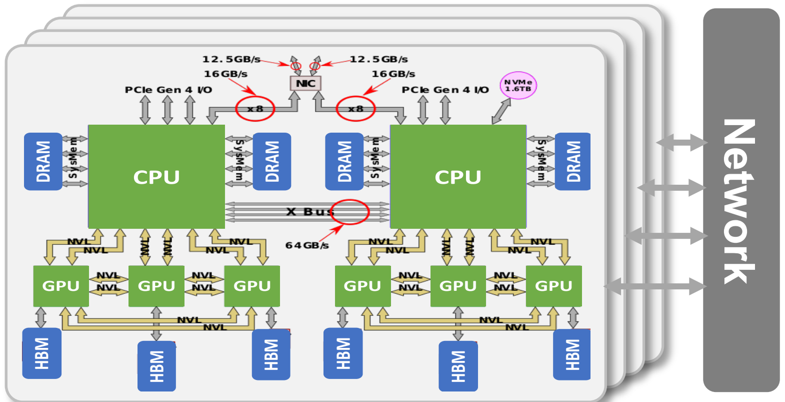
```
for i in range(8):
  for k in range(512):
    C[i, k] = max(A[i, k], 0.0) if k < 400 else 0
for i in range(8):
  for j in range(4):
    for k_o in range(TILE_K0):
      for k_i in range(TILE_KI):
        E.rf[...] += C[...] * D[...]
for i in range(8):
  for j in range(4):
    for k_i in range(TILE_KI):
      E[...] += E.rf[...]
```

```
parallel i in range(8):
  for k in range(512):
    C[i, k] = ...
  for j in range(4):
    unroll k_o in range(32):
      vectorized k_i in range(16):
        E.rf[...] += C[...] * D[...]
parallel i in range(8):
  for j in range(4):
    unroll k_i in range(16):
      E[...] += E.rf[...]
```

Recap: An Overview of Deep Learning Systems



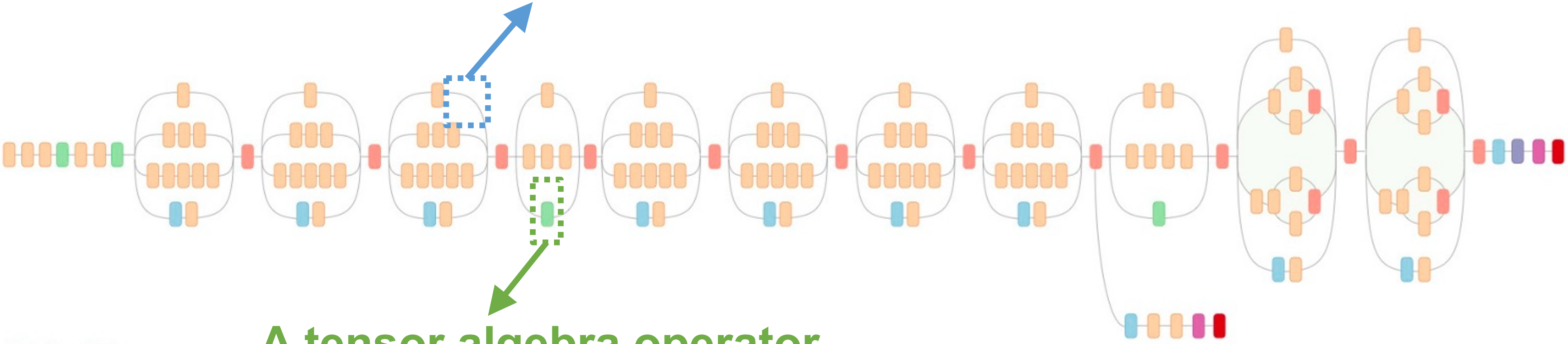
- Automatic Differentiation
- **Graph-Level Optimization** ← **Week 5: this lecture**
- Parallelization / Distributed Training ← **Week 6: Data, Model, Pipeline Parallelism**
- Kernel Generation ← **Week 4: Halide, TVM, Ansor**
- Memory Optimization ← **Week 7: Zero-Redundancy, Tensor Rematerialization**



Recap: Deep Neural Network

- Collection of simple trainable mathematical units that work together to solve complicated tasks

A tensor (i.e., *n*-dimensional array)

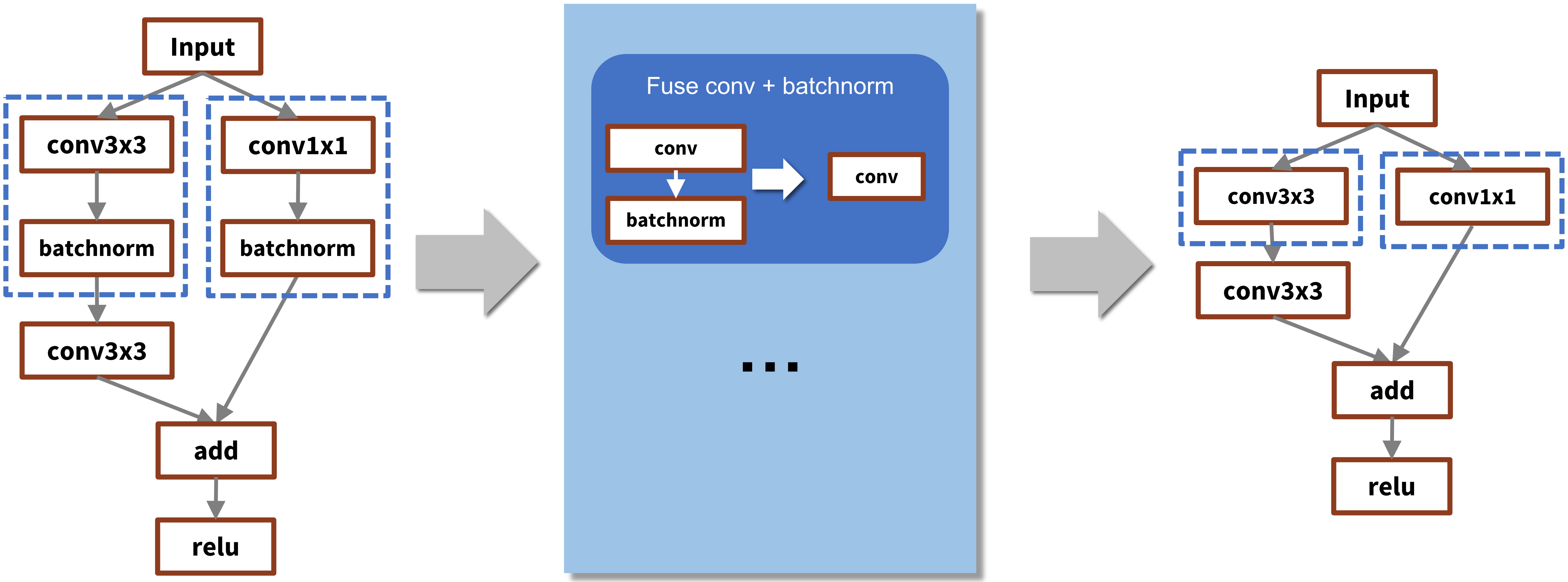


A tensor algebra operator (e.g., convolution, matrix mul)

TASO: Optimizing Deep Learning with Automatic Generation of Graph Substitutions

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

Graph-Level Optimizations

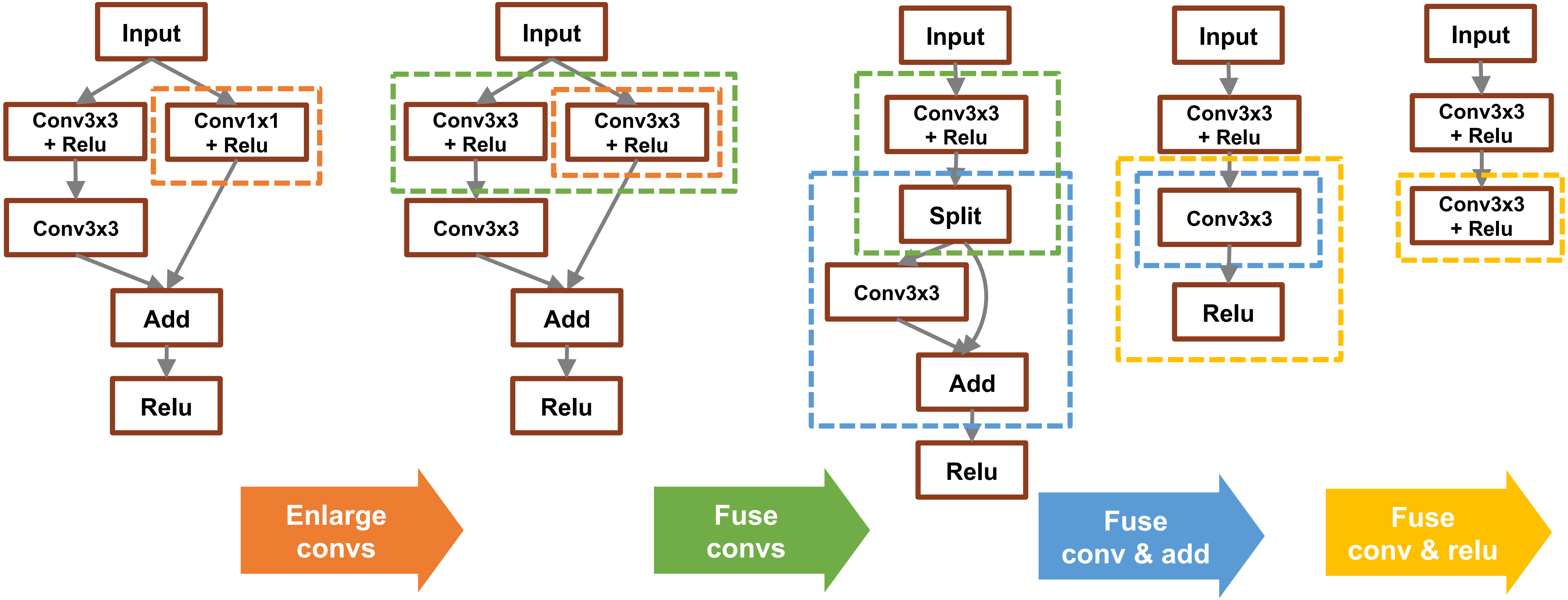


Input Computation Graph

Potential graph transformations

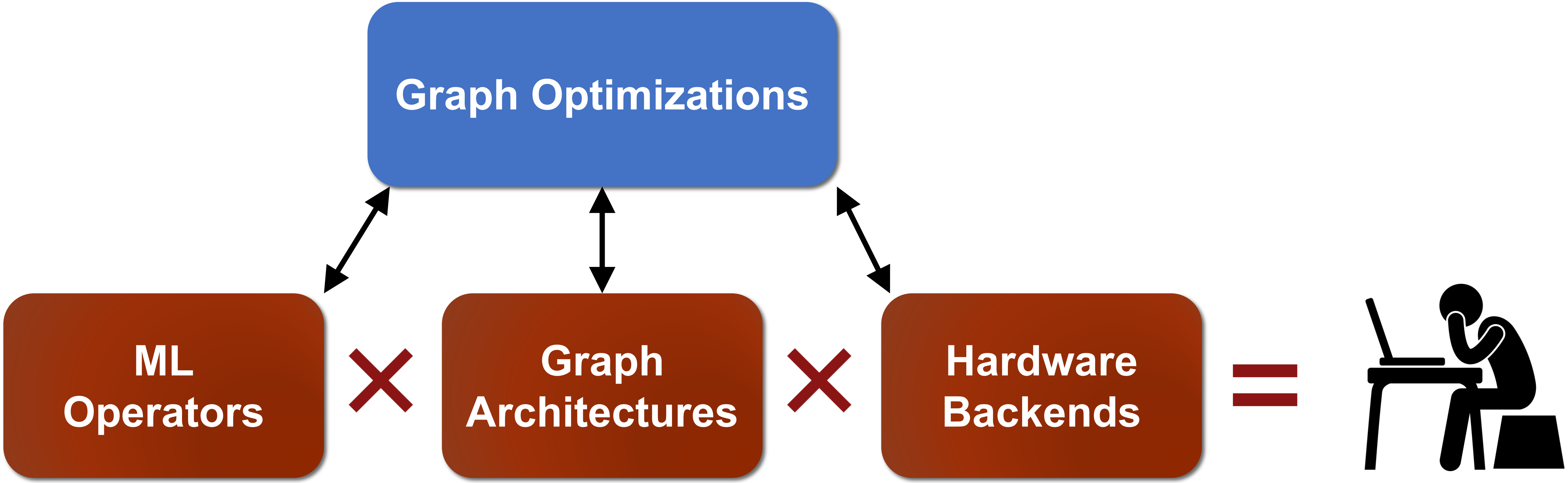
Optimized Computation Graph

Recap: ResNet Example



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

Challenge of Graph Optimizations for ML



Graph Optimizations

ML Operators × Graph Architectures × Hardware Backends =

Infeasible to manually design graph optimizations for all cases

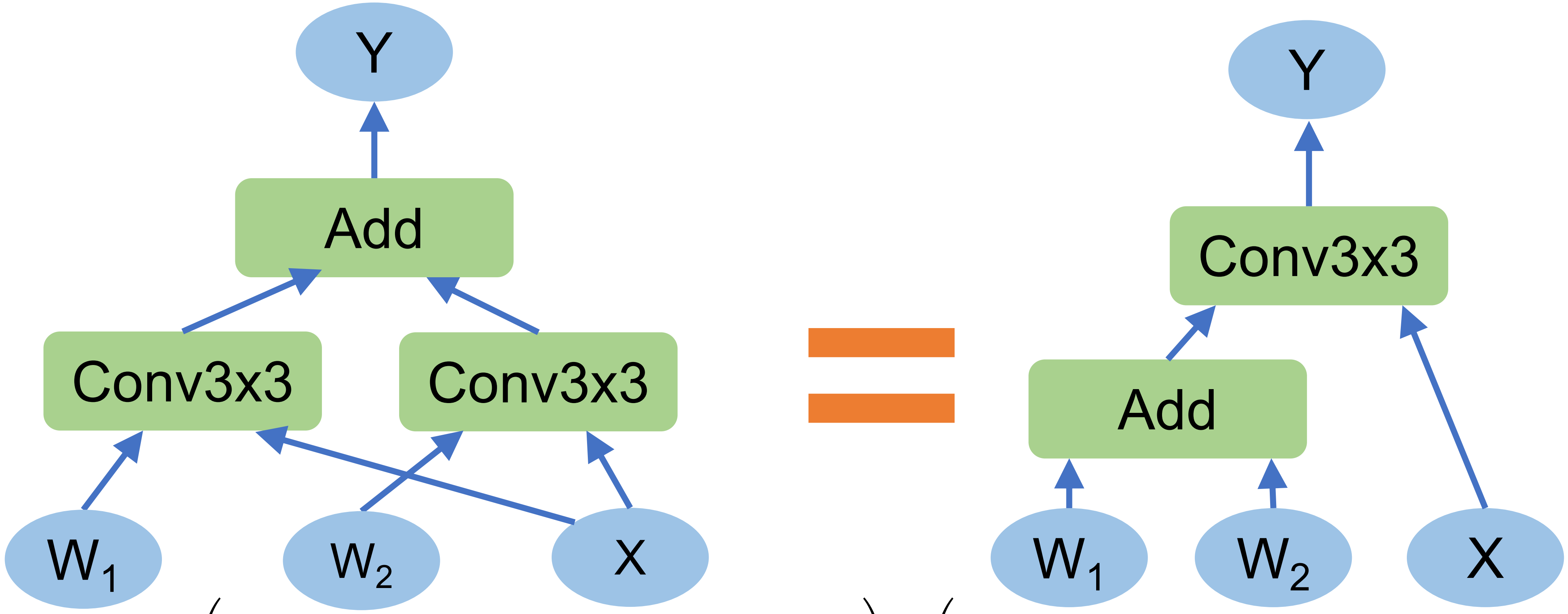
TASO: Tensor Algebra SuperOptimizer

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

- **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 **3x**
- **Stronger correctness:** formally verify all generated substitutions

TASO: Optimizing Deep Learning Computation with Automated Generation of Graph Substitutions. SOSP'19.

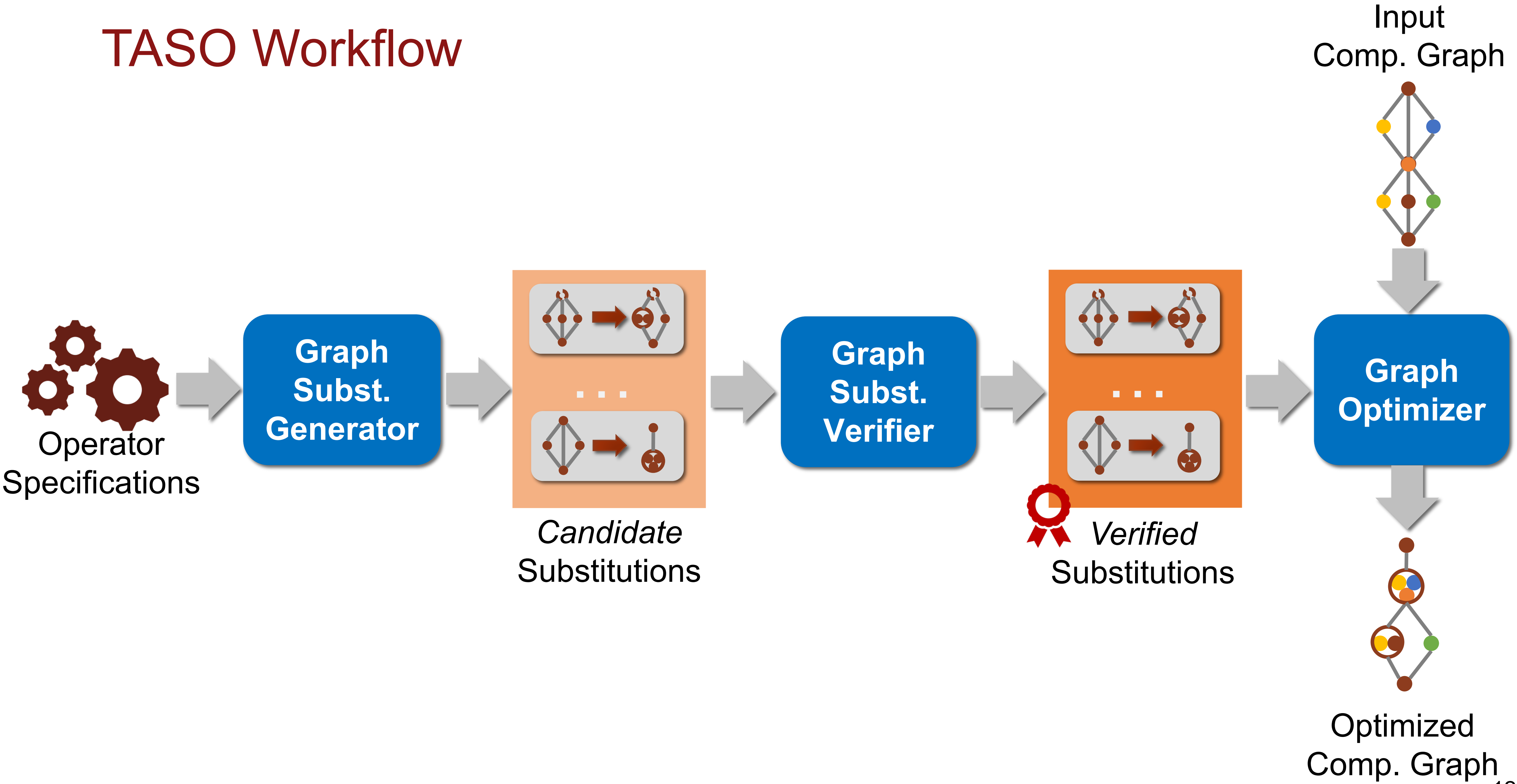
Graph Substitution



$$Y(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W1(c,d,u,v)\right) + \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W2(c,d,u,v)\right)$$

$$\Leftrightarrow Y(n,c,h,w) = \sum_{d,u,v} X(n,d,h+u,w+v) * \big((W_1\,(c,d,u,v) + W_2\,(c,d,u,v))\big)$$

TASO Workflow



Input
Comp. Graph

Operator
Specifications

Graph Subst. Generator

Candidate
Substitutions

Graph Subst. Verifier

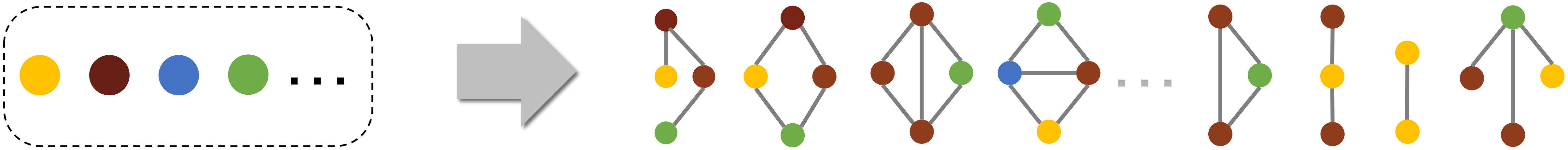
Verified
Substitutions

Graph Optimizer

Optimized
Comp. Graph

Graph Substitution Generator

Enumerate **all possible** graphs up to a fixed size using available operators



Operators supported by hardware backend

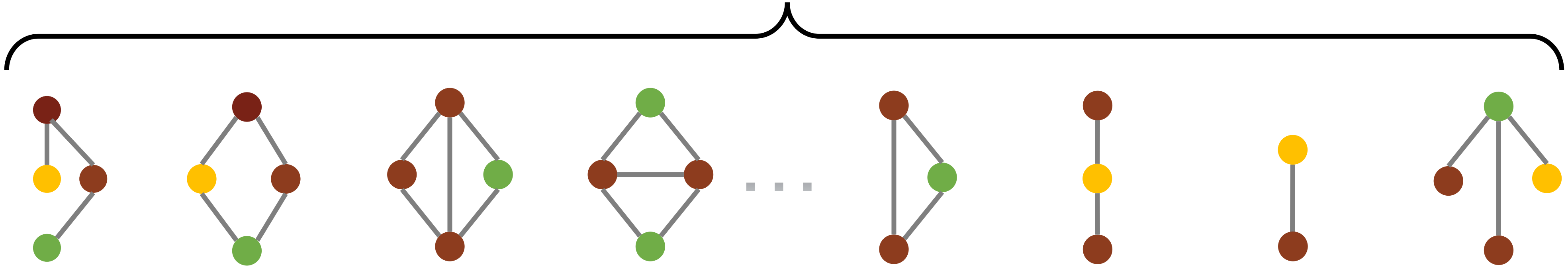
Subst. Generator

Subst. Verifier

Graph Optimizer

Graph Substitution Generator

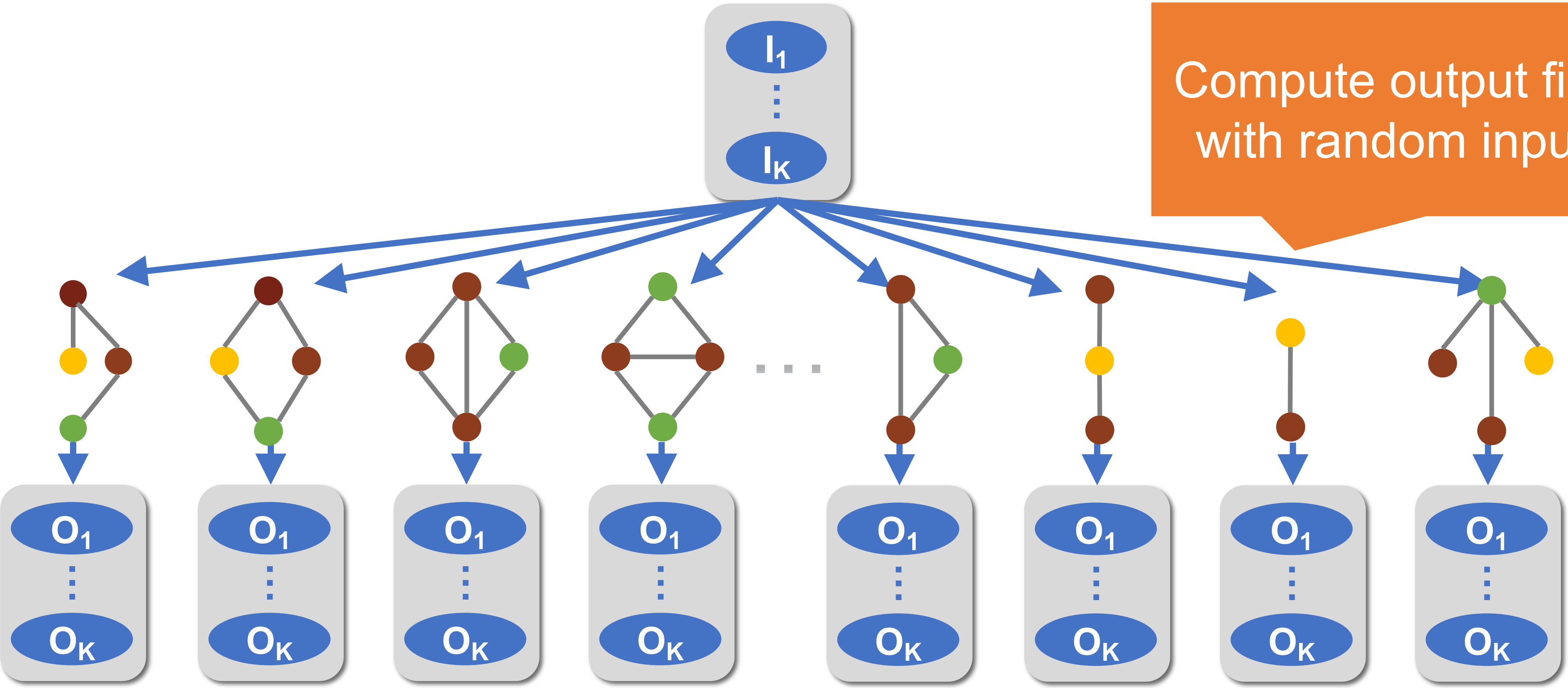
66M graphs with up to **4** operators



A substitution = a pair of equivalent graphs

Explicitly considering all pairs does not scale

Graph Substitution Generator



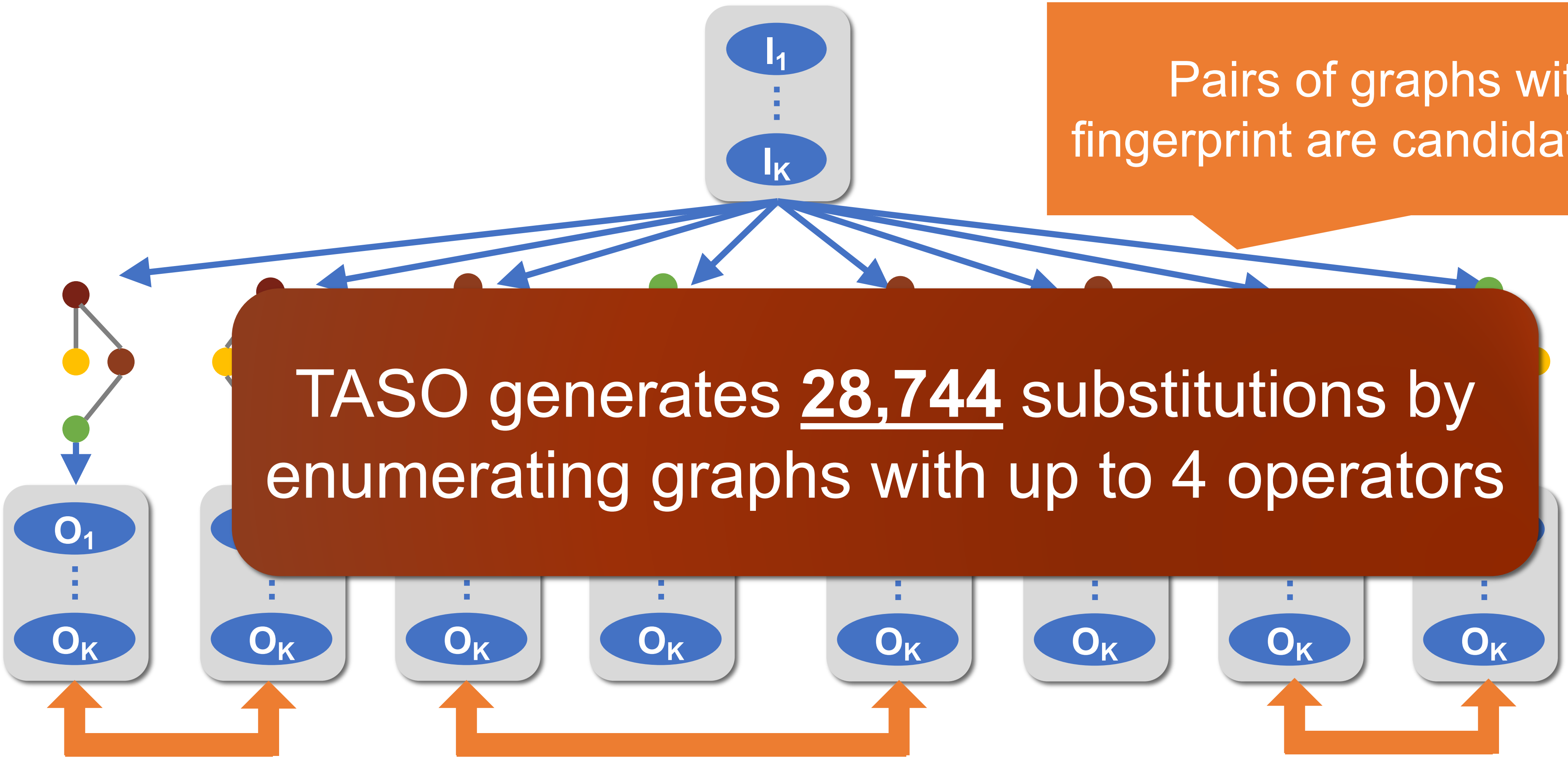
Compute output fingerprints with random input tensors

Subst. Generator

Subst. Verifier

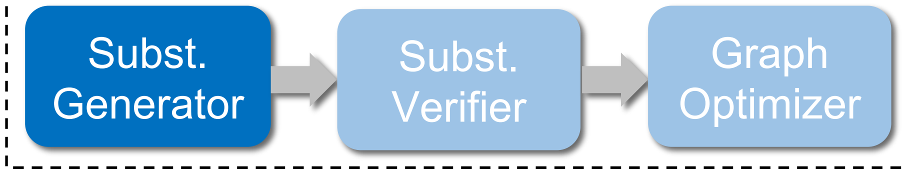
Graph Optimizer

Graph Substitution Generator



Pairs of graphs with identical fingerprint are candidate substitutions

TASO generates **28,744** substitutions by enumerating graphs with up to 4 operators

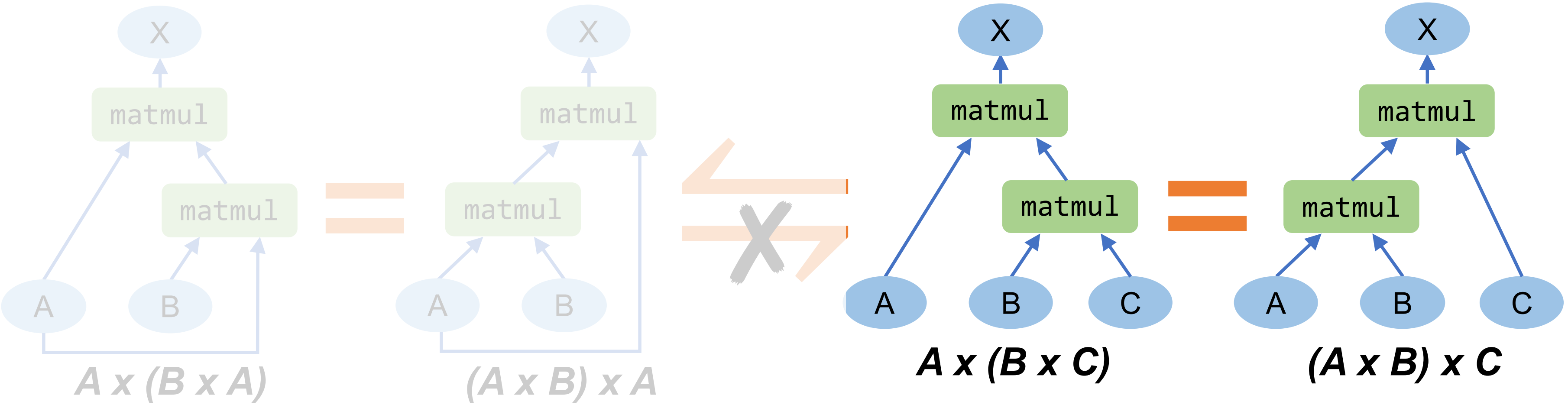


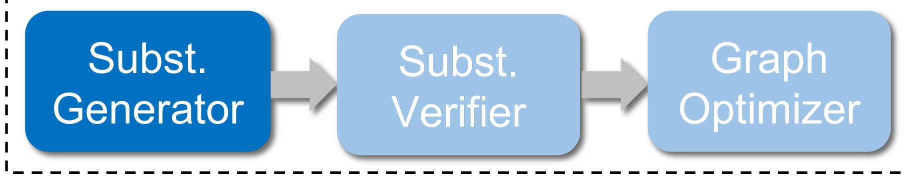
Pruning Redundant Substitutions

28,744 substitutions

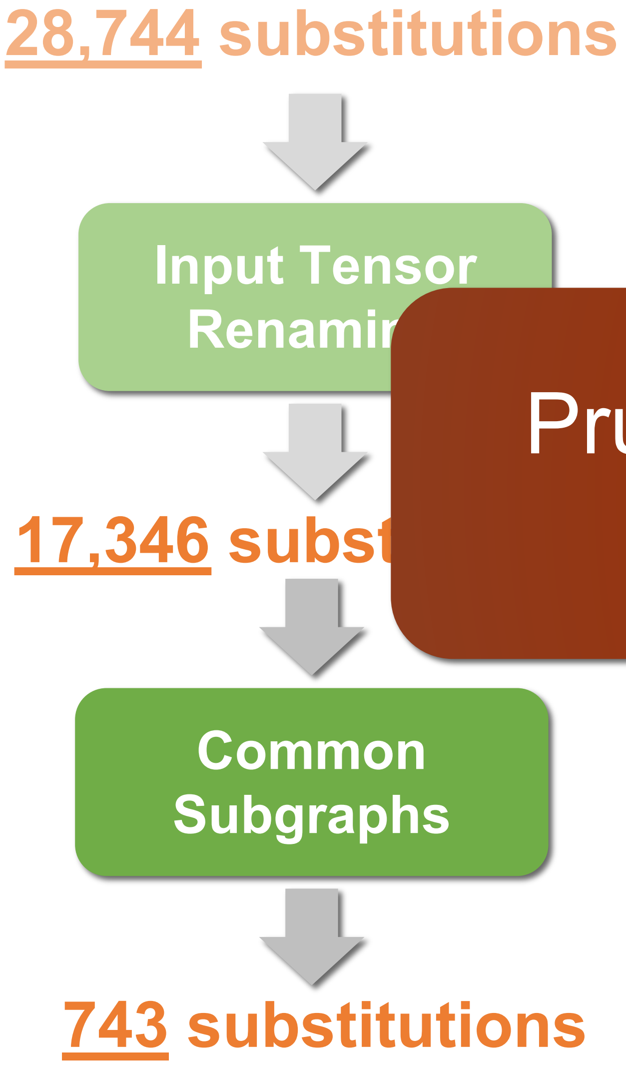
Input Tensor Renaming

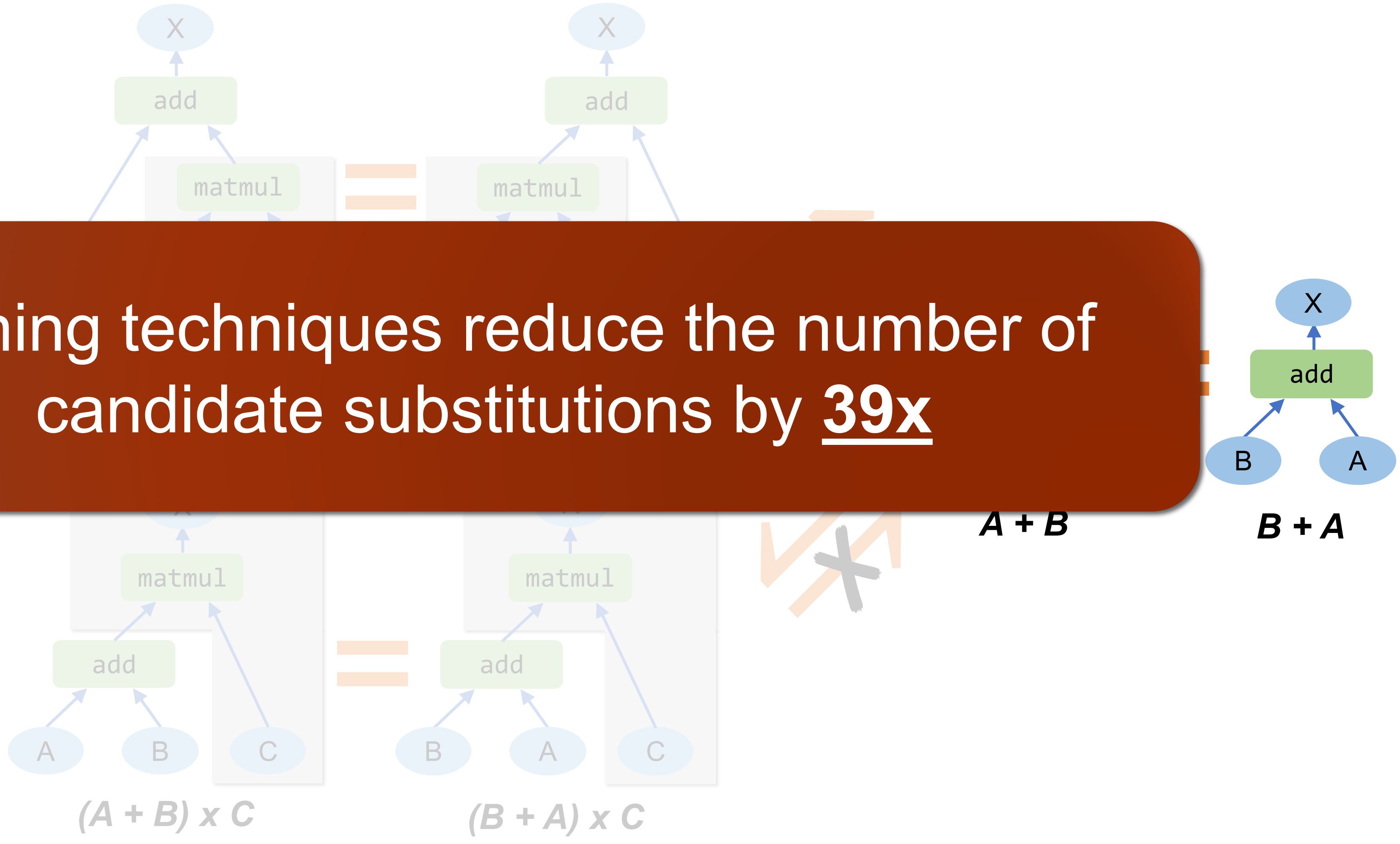
17,346 substitutions





Pruning Redundant Substitutions

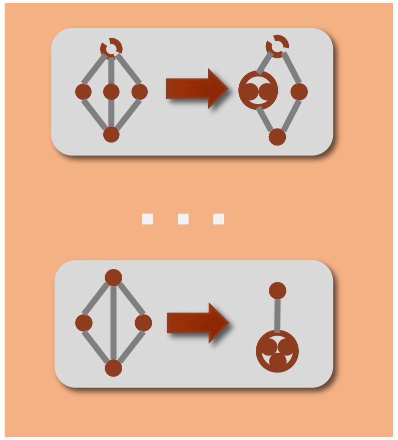




Pruning techniques reduce the number of candidate substitutions by **39x**

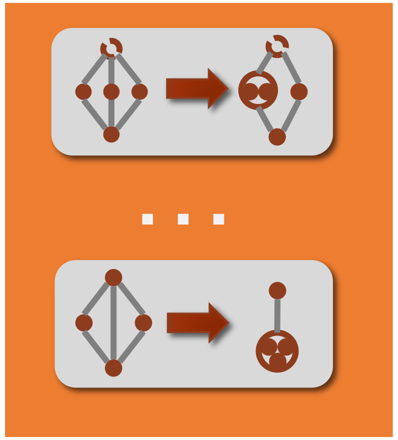
Subst. Generator → Subst. Verifier → Graph Optimizer

Graph Substitution Verifier



Candidate Substitutions

Graph Subst. Verifier



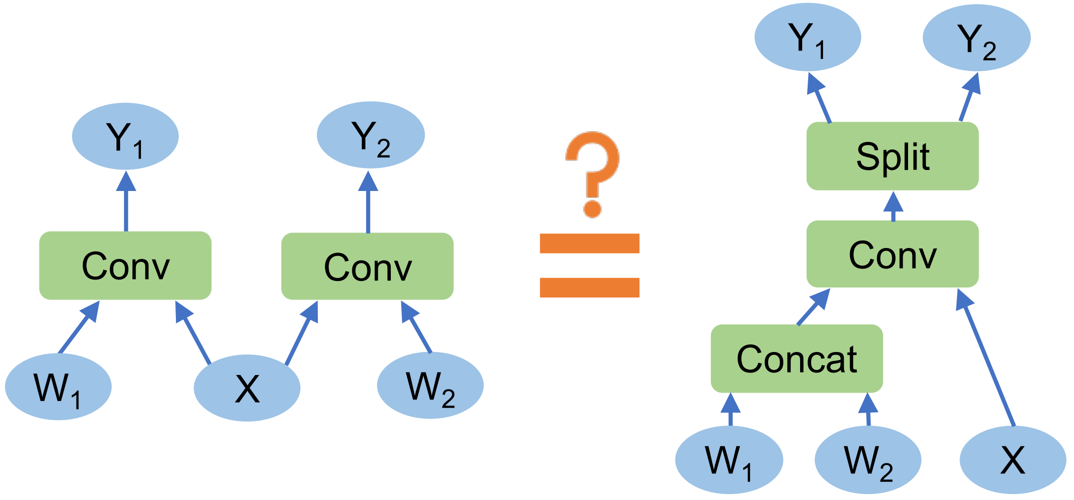
Verified Substitutions

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

Operator Specifications

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

Verification Workflow



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

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

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

Automated Theorem Prover

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. …

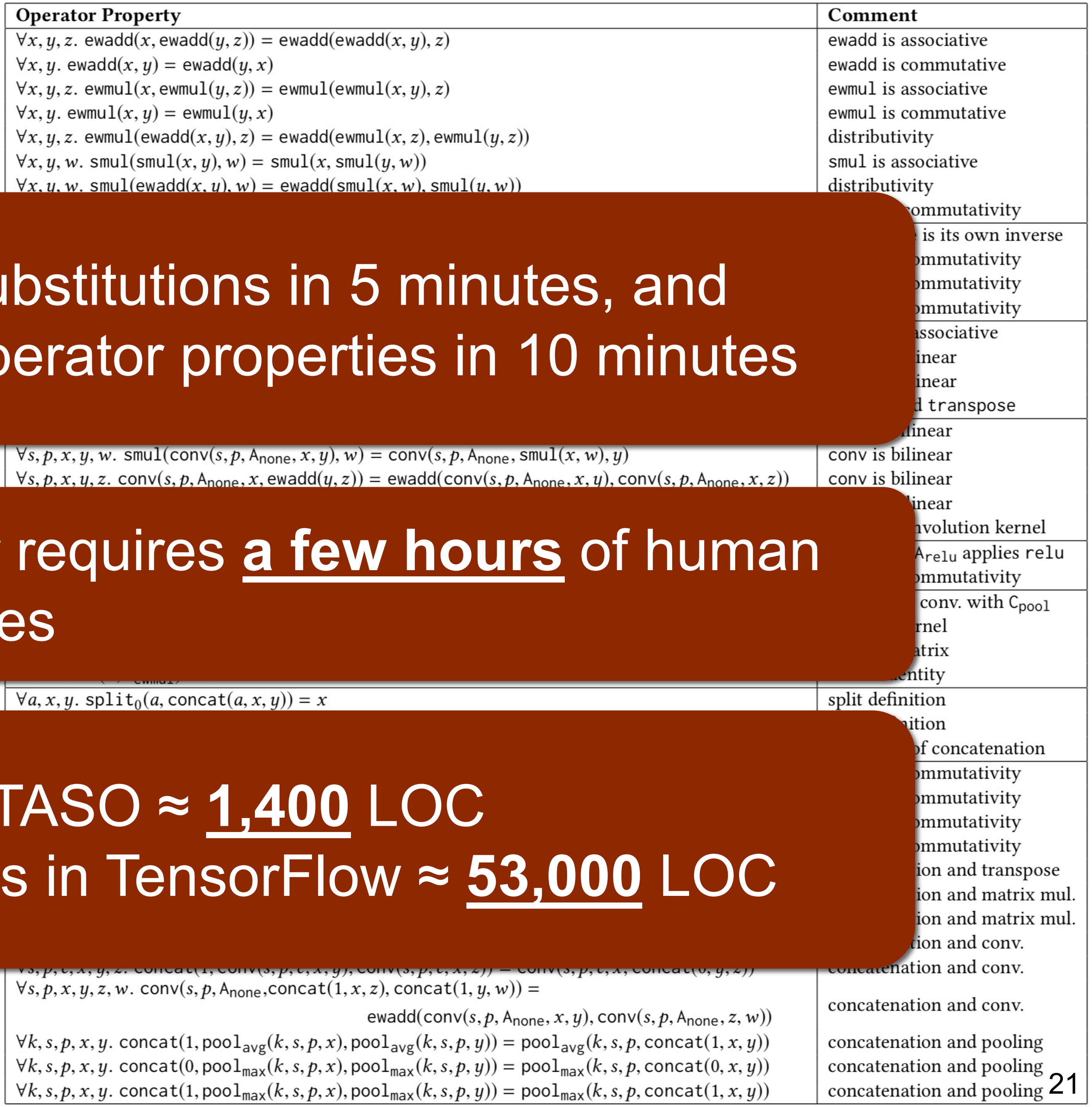
Operator Specifications

Verification Effort

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

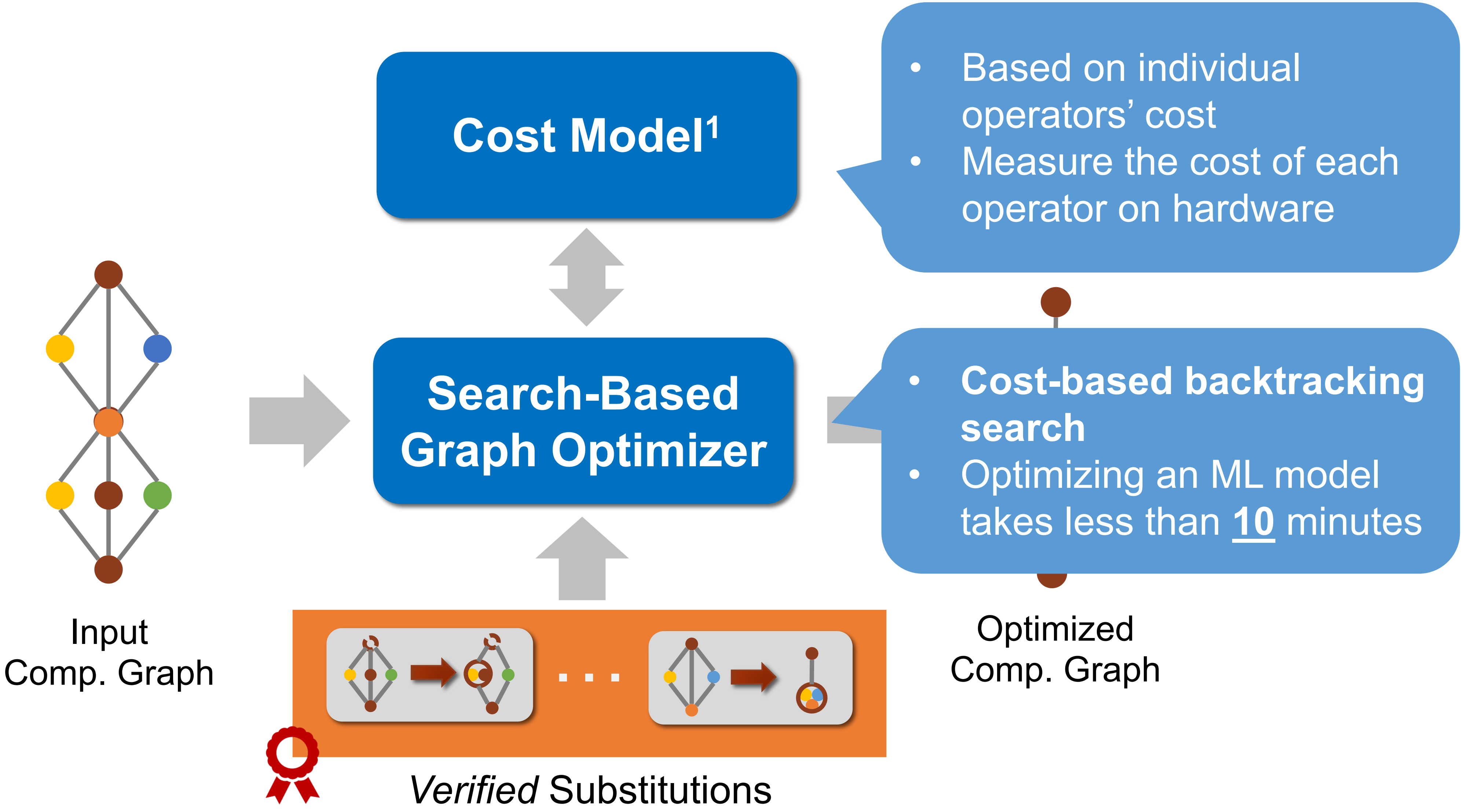
Supporting a new operator requires **a few hours** of human effort to specify its properties

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



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

Search-Based Graph Optimizer



Cost Model[1]

- Based on individual operators' cost
- Measure the cost of each operator on hardware

Search-Based Graph Optimizer

- **Cost-based backtracking search**
- Optimizing an ML model takes less than **10** minutes

Input
Comp. Graph

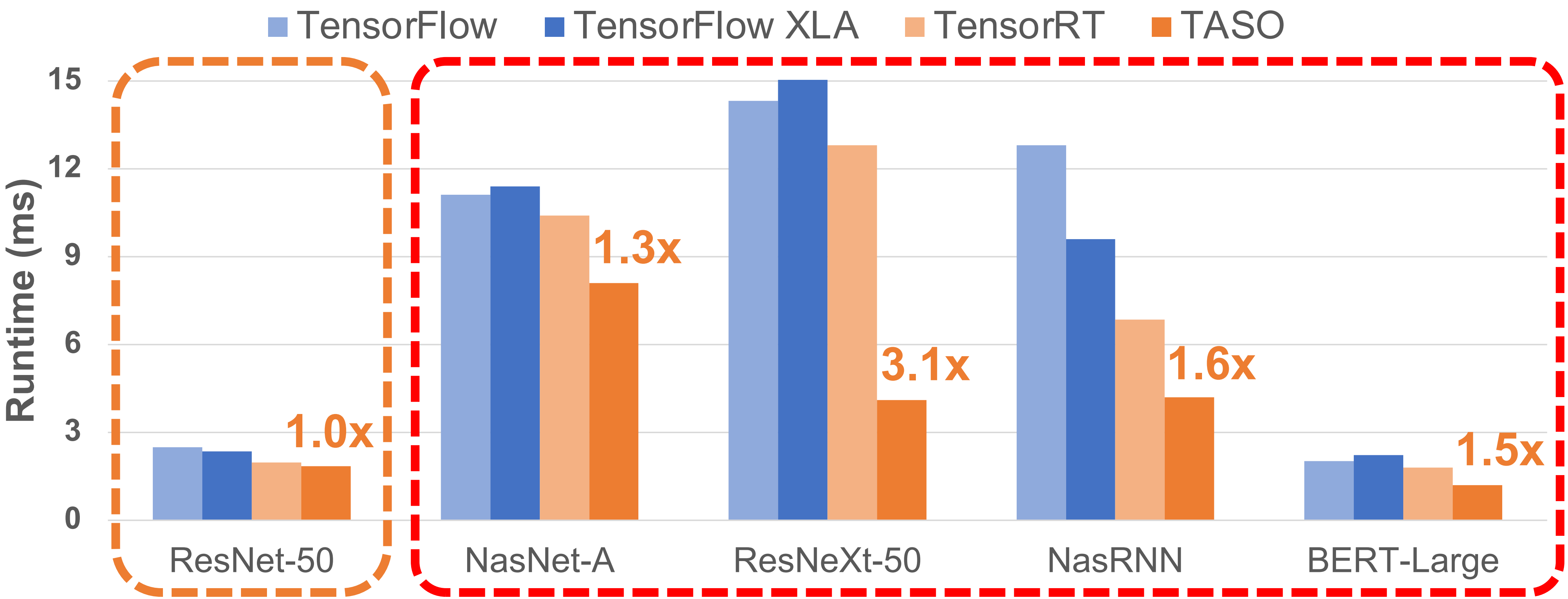
Verified Substitutions

Optimized
Comp. Graph

---

1. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. ICML'18.

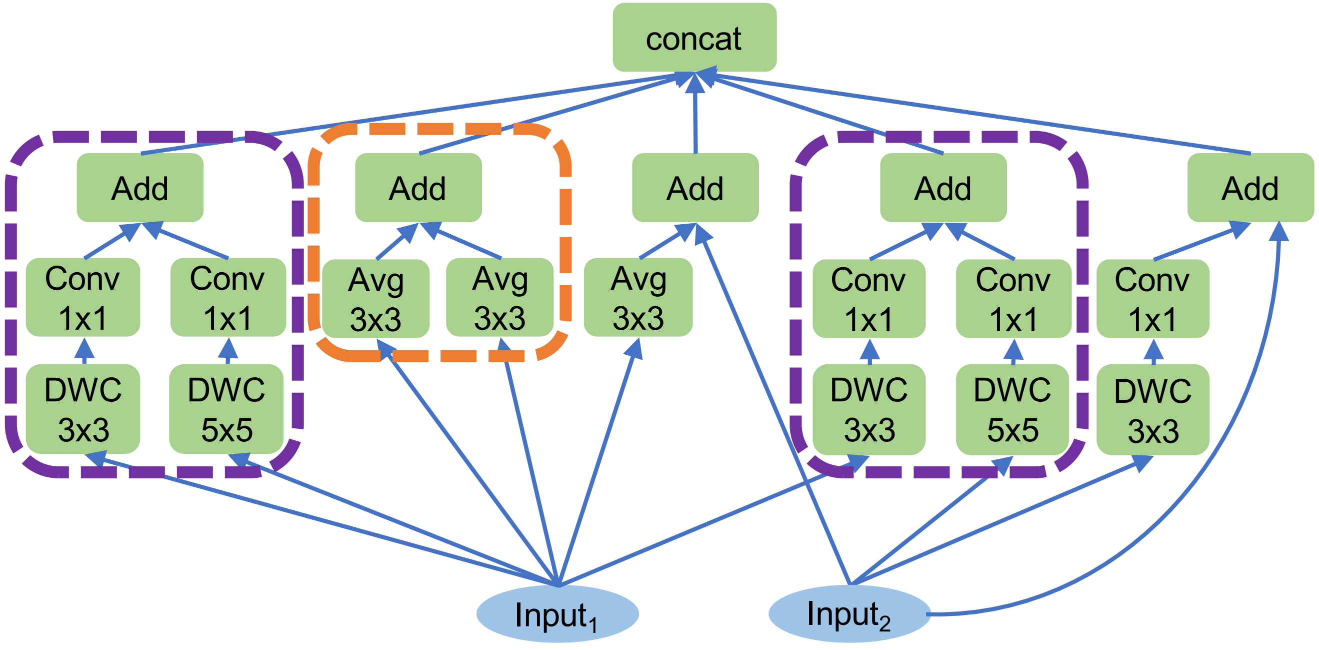
End-to-end Inference Performance (Nvidia V100 GPU)



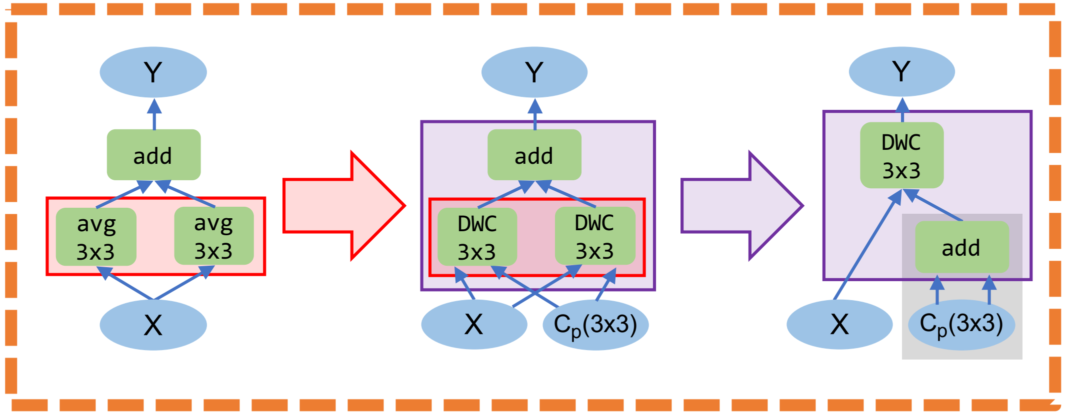
Competitive on standard models

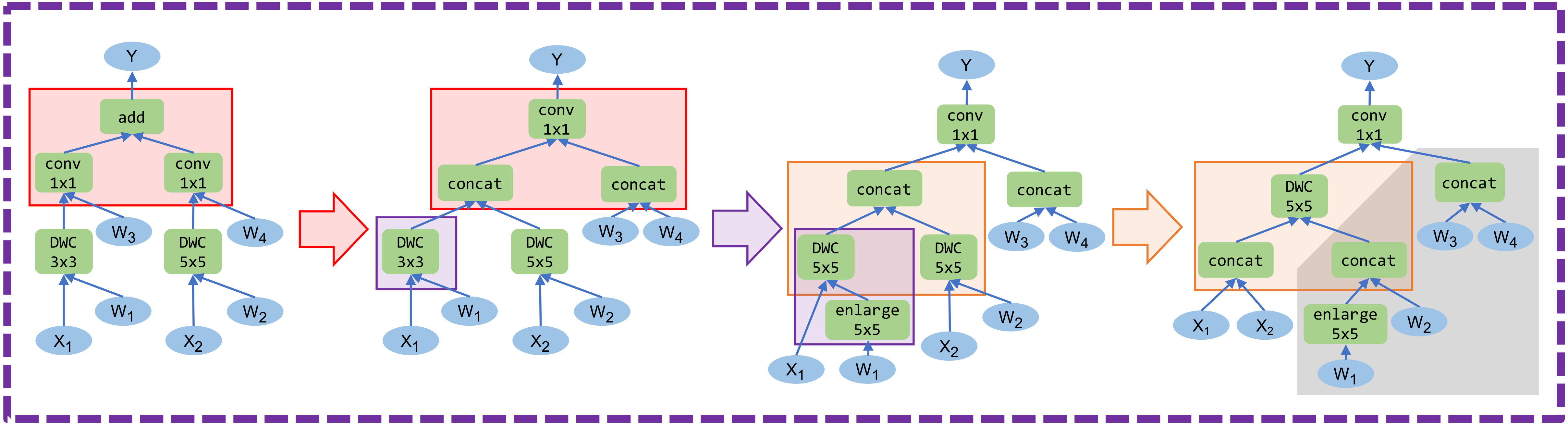
Larger speedups on emerging models

Case Study: NASNet



*DWC: depth-wise convolution





Why TASO is a SuperOptimizer?

What is the difference between optimizer and super-optimizer?

Goal: gradually *improve* an input program by greedily applying optimizations

Goal: automatically find an *optimal* program for an input program

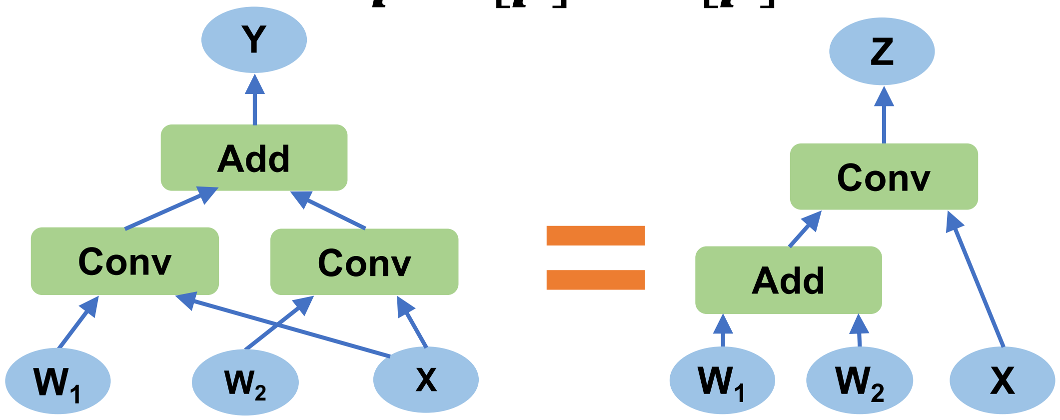
PET:
Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections

Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang,
Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, Zhihao Jia

Tsinghua University Carnegie Mellon University Facebook

Motivation: Current Systems Consider only Fully Equivalent Transformations

$$\forall p.\ Y[p] = Z[p]$$

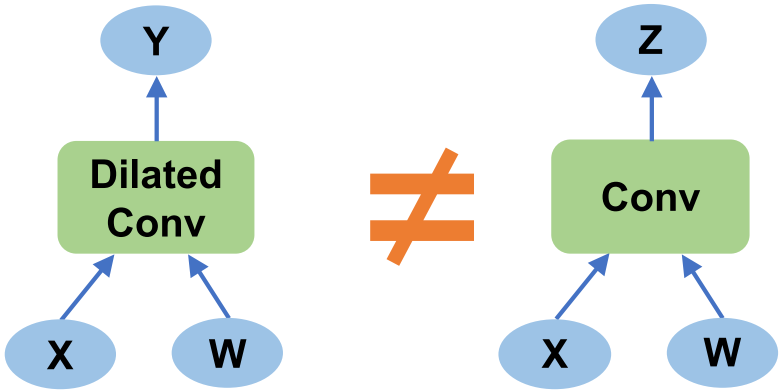


Fully Equivalent Transformations

👍 Pro: preserve functionality

👎 Con: miss optimization opportunities

$$\exists p.\ Y[p] \neq Z[p]$$



Partially Equivalent Transformations

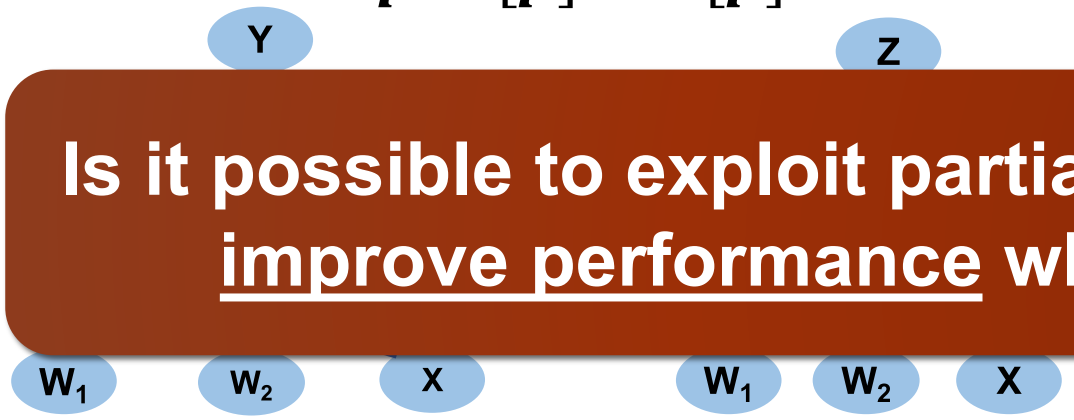
👍 Pro: better performance

- Faster ML operators
- More efficient tensor layouts
- Hardware-specific optimizations

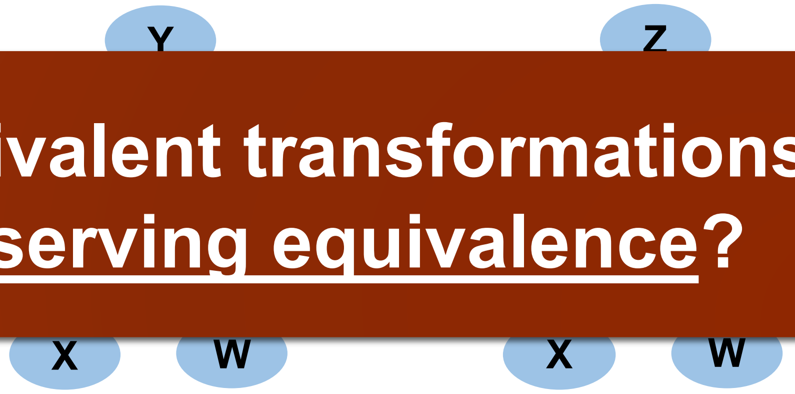
👎 Con: potential accuracy loss

Motivation: Current Systems Consider only Fully Equivalent Transformations

$\forall p.\ Y[p] = Z[p]$



$\exists p.\ Y[p] \neq Z[p]$



Is it possible to exploit partially equivalent transformations to improve performance while preserving equivalence?

Fully Equivalent Transformations

Pro: preserve functionality

Con: miss optimization opportunities

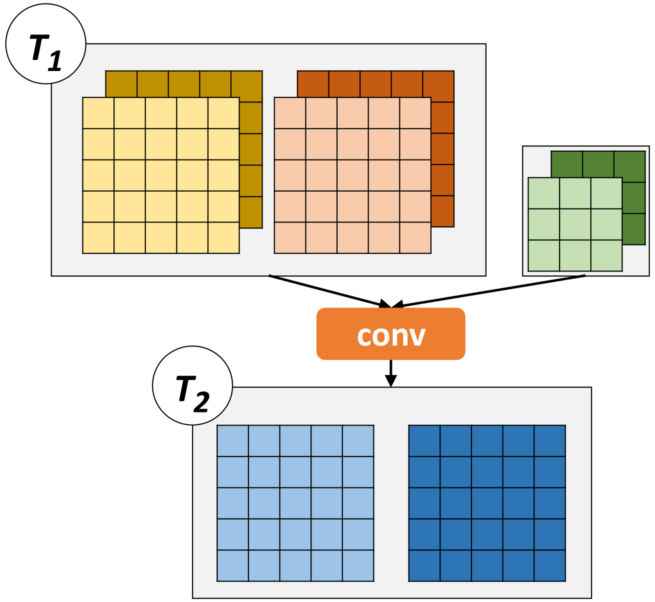
Partially Equivalent Transformations

Pro: better performance

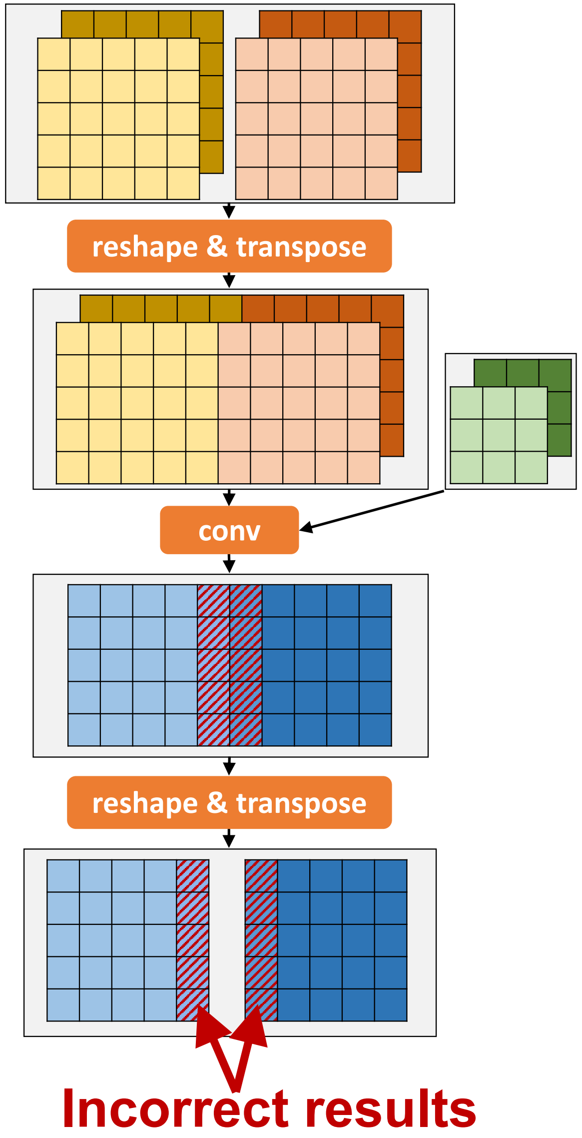
- Faster ML operators
- More efficient tensor layouts
- Hardware-specific optimizations

Con: potential accuracy loss

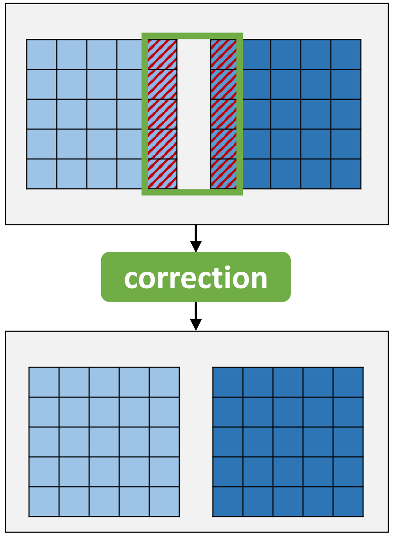
Motivating Example



Input Program

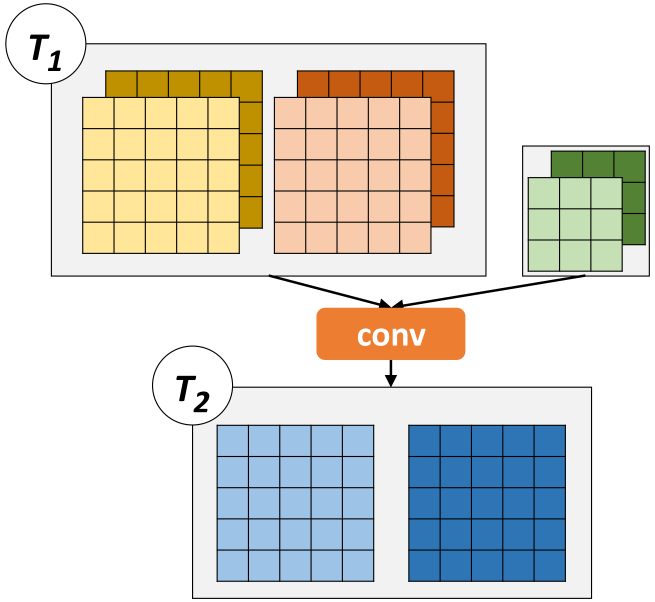


Partially Equivalent Transformation

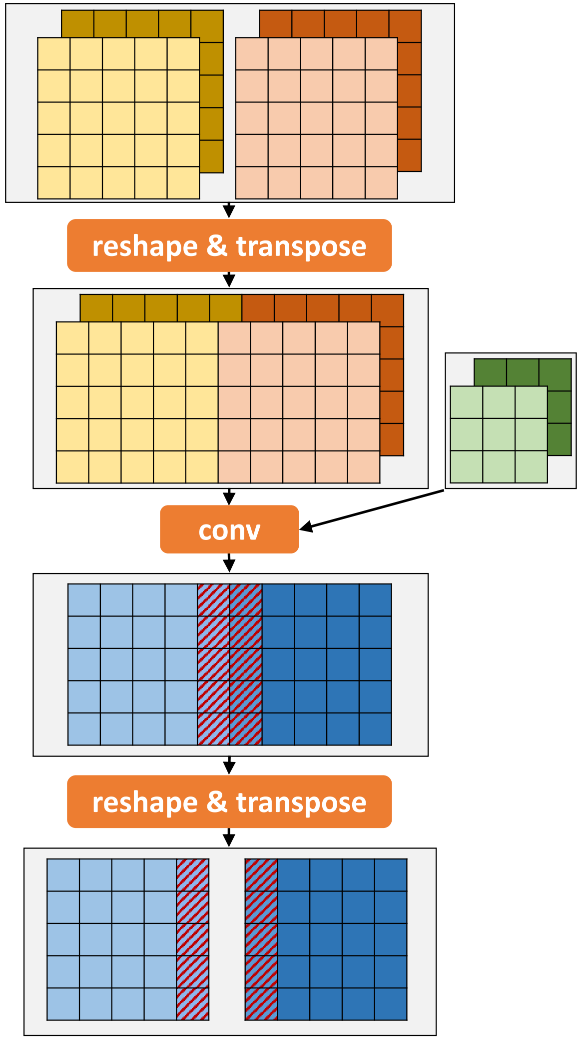


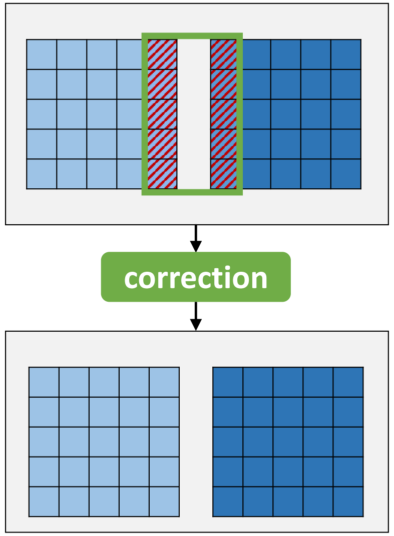
Correcting Results

Motivating Example



Input Program





Correcting Results

- Transformation and correction lead to **<u>1.2x</u>** speedup for ResNet-18
- Correction preserves end-to-end equivalence

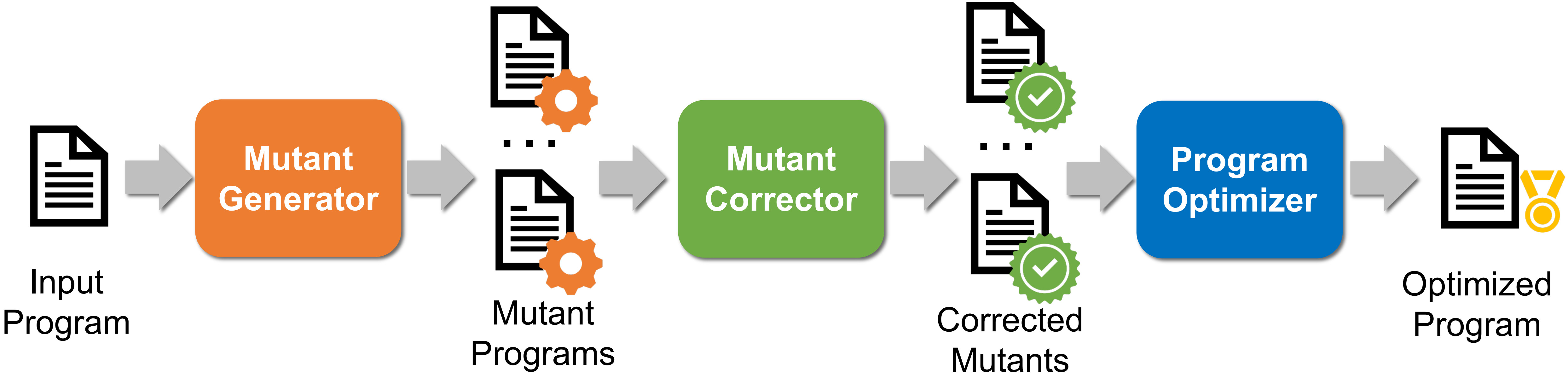
PET

- **First tensor program optimizer** with partially equivalent transformations

- **Larger optimization space** by combining fully and partially equivalent transformations
- **Better performance**: outperform existing optimizers by up to **2.5x**
- **Correctness**: automated corrections to preserve end-to-end equivalence

PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. OSDI'21

PET Overview



Input Program

Mutant Generator

Mutant Programs

Mutant Corrector

Corrected Mutants

Program Optimizer

Optimized Program

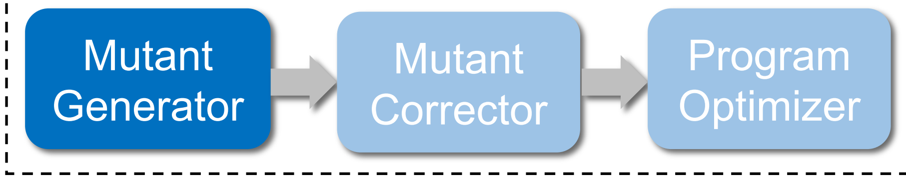
Key Challenges

1. How to generate partially equivalent transformations?

Superoptimization

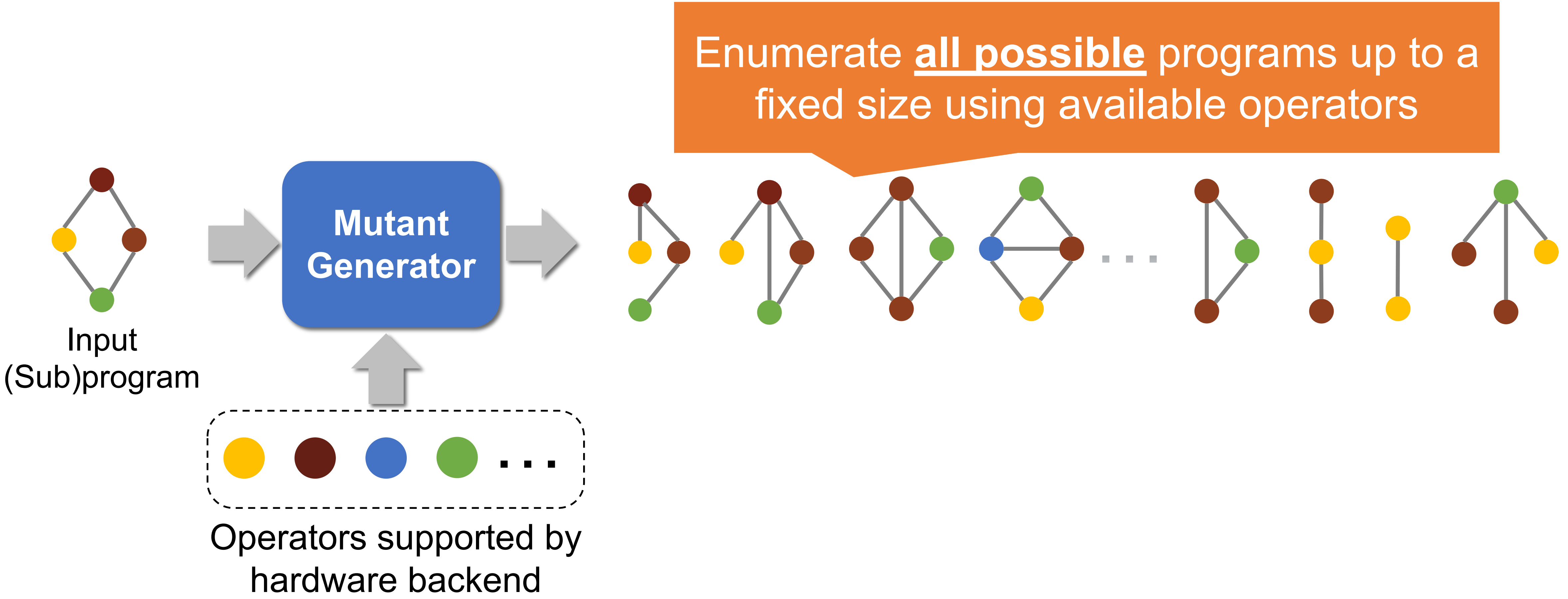
2. How to correct them?

Multi-linearity of DNN computations



Mutant Generator

Superoptimization adapted from TASO[1]



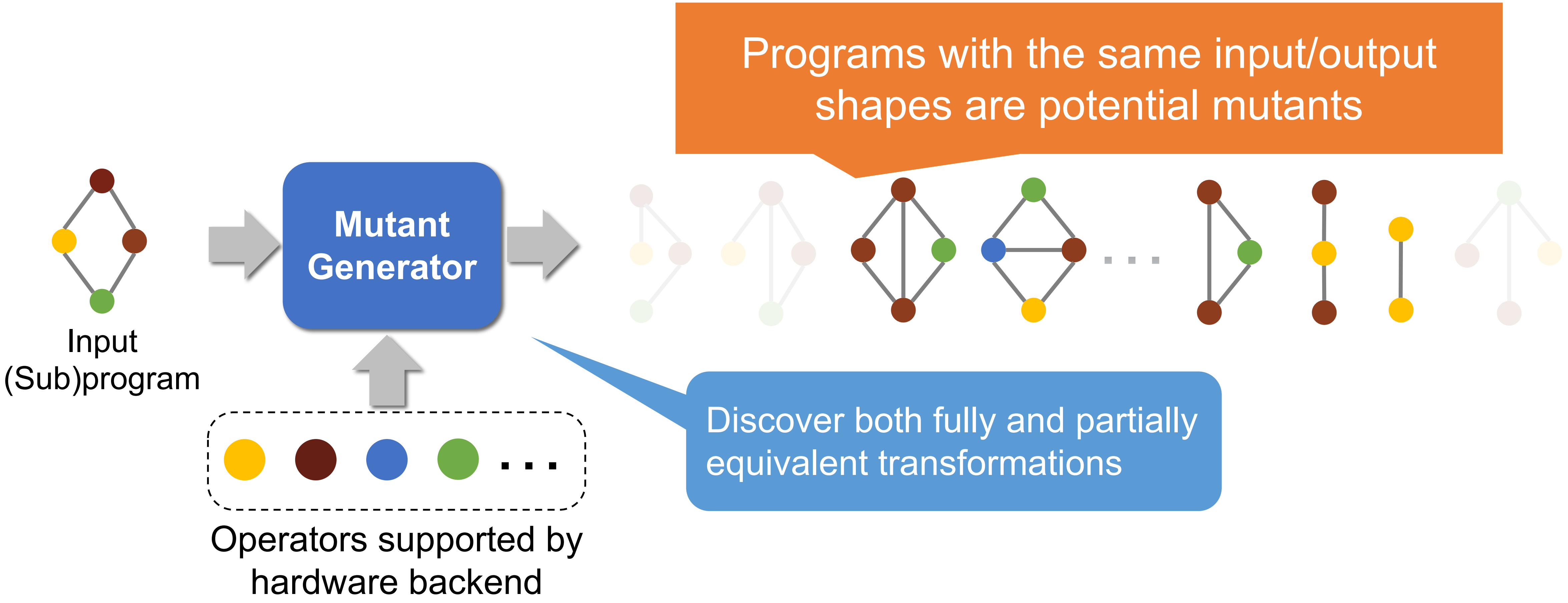
Enumerate **all possible** programs up to a fixed size using available operators

---

1. TASO: Optimizing Deep Learning Computation with Automated Generation of Graph Substitutions. SOSP'19.

Mutant Generator

Superoptimization adapted from TASO[1]



Programs with the same input/output shapes are potential mutants

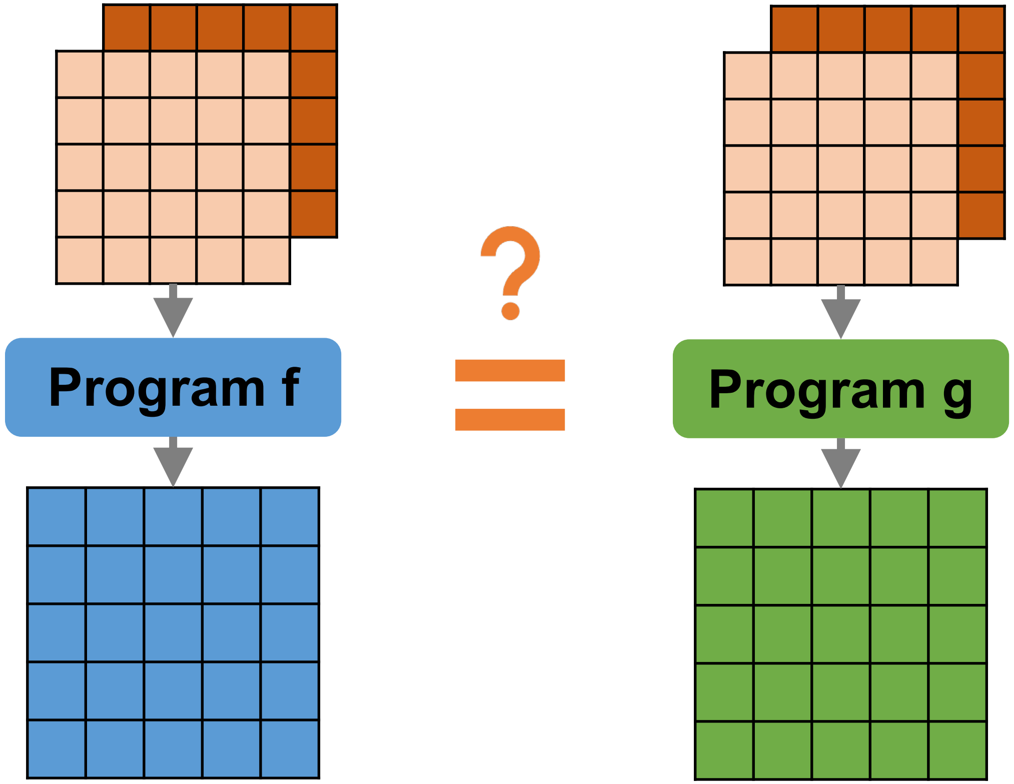
Discover both fully and partially equivalent transformations

---

[1] TASO: Optimizing Deep Learning Computation with Automated Generation of Graph Substitutions. SOSP'19.

Mutant Generator → Mutant Corrector → Program Optimizer

Challenges: Examine Transformations

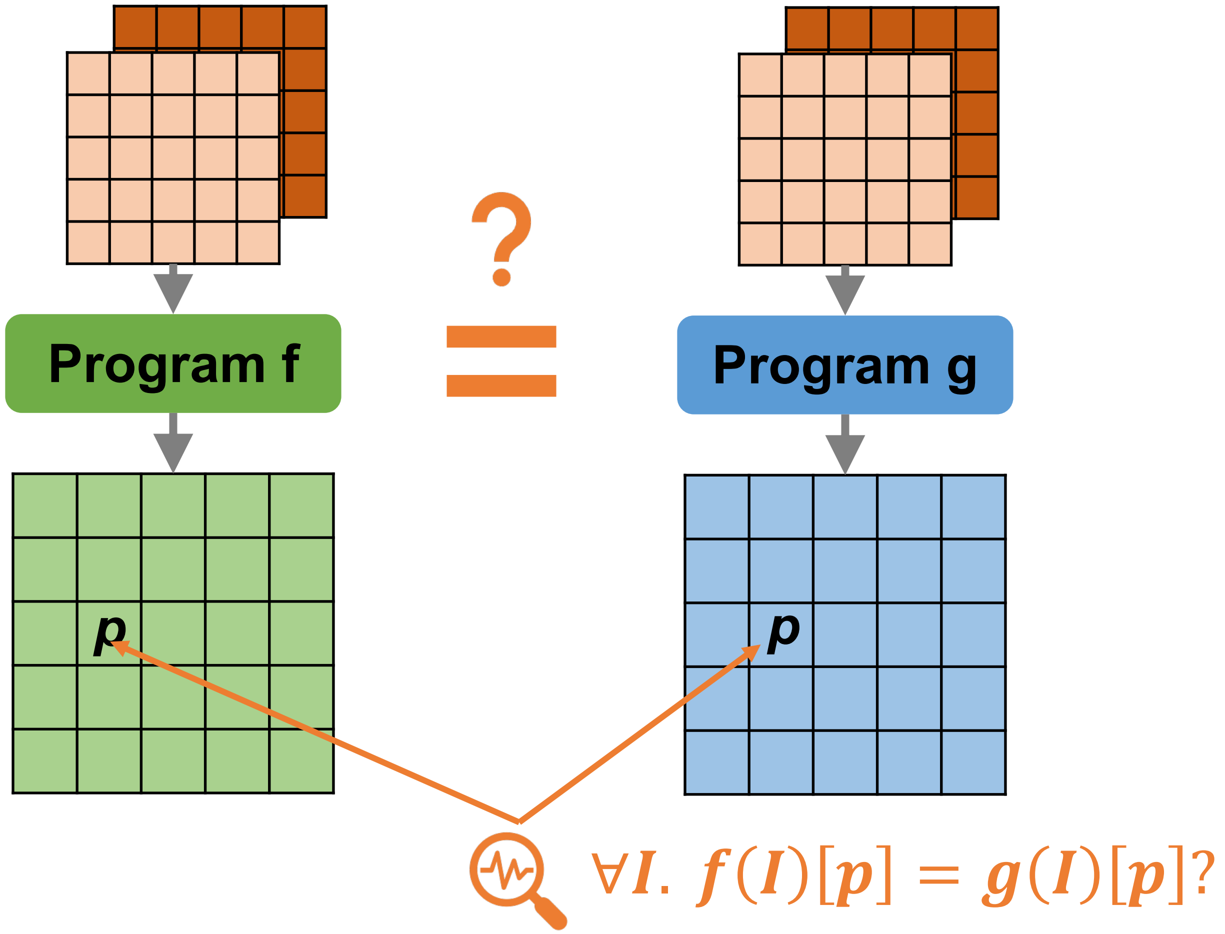


1. Which part of the computation is not equivalent?
2. How to correct the results?

A Strawman Approach

- **Step 1**: Explicitly consider all output positions (m positions)

- **Step 2**: For each position ***p***, examine all possible inputs (n inputs)



$\forall I.\ f(I)[p] = g(I)[p]?$

Require O(m * n) examinations, but both m and n are too large to explicitly enumerate

Multi-Linear Tensor Program (MLTP)

- A program $\boldsymbol{f}$ is multi-linear if the output is linear to all inputs
  - $\boldsymbol{f}(I_1, \dots, X, \dots, I_n) + \boldsymbol{f}(I_1, \dots, Y, \dots, I_n) = \boldsymbol{f}(I_1, \dots, X + Y, \dots, I_n)$
  - $\alpha \cdot \boldsymbol{f}(I_1, \dots, X, \dots, I_n) = \boldsymbol{f}(I_1, \dots, \alpha \cdot X, \dots, I_n)$

- DNN computation = MLTP + non-linear activations

Majority of the computation

O(m * n) examinations in strawman approach

MLTP

O(1) examinations in PET’s approach

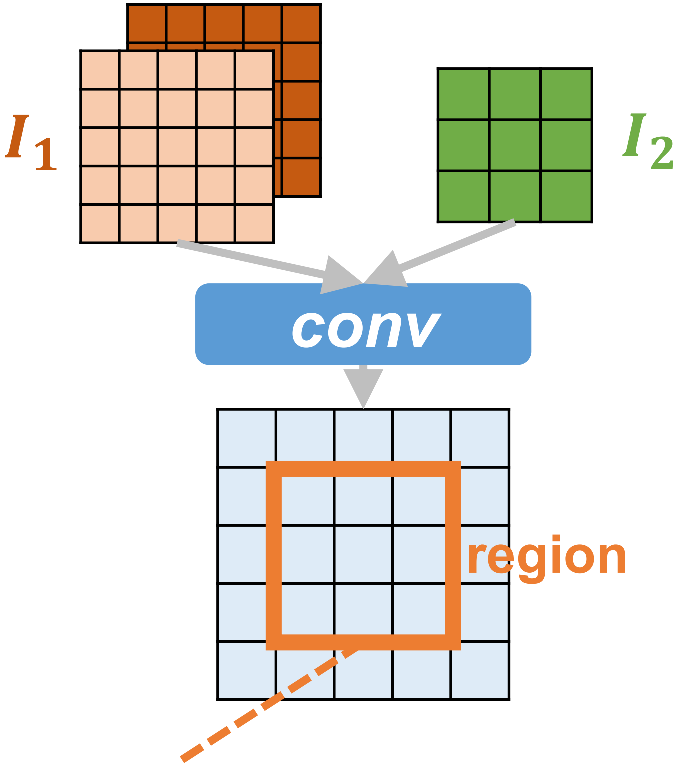
No Need to Enumerate All Output Positions

Group all output positions with an identical summation interval into a region

***Theorem 1**: For two MLTPs **f** and **g**, if **f=g** for O(1) positions in a region, then **f=g** for all positions in the region

Only need to examine O(1) positions for each region.

Complexity: **O(m * n) → O(n)**



$$conv(c,h,w) = \boxed{\sum_{d=0}^{D-1} \sum_{x=-1}^{1} \sum_{y=-1}^{1}} I_1(d, h+x, w+y) \times I_2(d, c, x, y)$$

Summation interval

*Proof details available in the paper

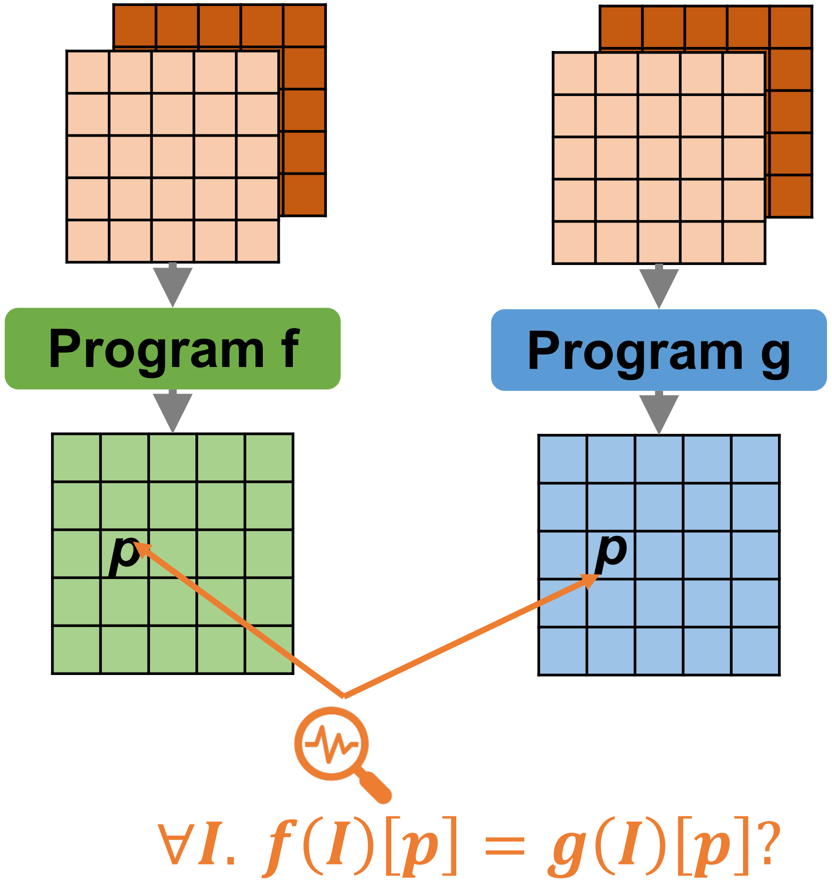
No Need to Consider All Possible Inputs

Examining equivalence for a single position is still challenging

Theorem 2**: If $\exists I.\ f(I)[p] \neq g(I)[p]$, then the probability that **f** and **g** give identical results on ***t random integer inputs is $(\frac{1}{2^{31}})^t$

Run ***t*** random tests for each position $\boldsymbol{p}$

Complexity: **O(n) → O(*t*) = O(1)**

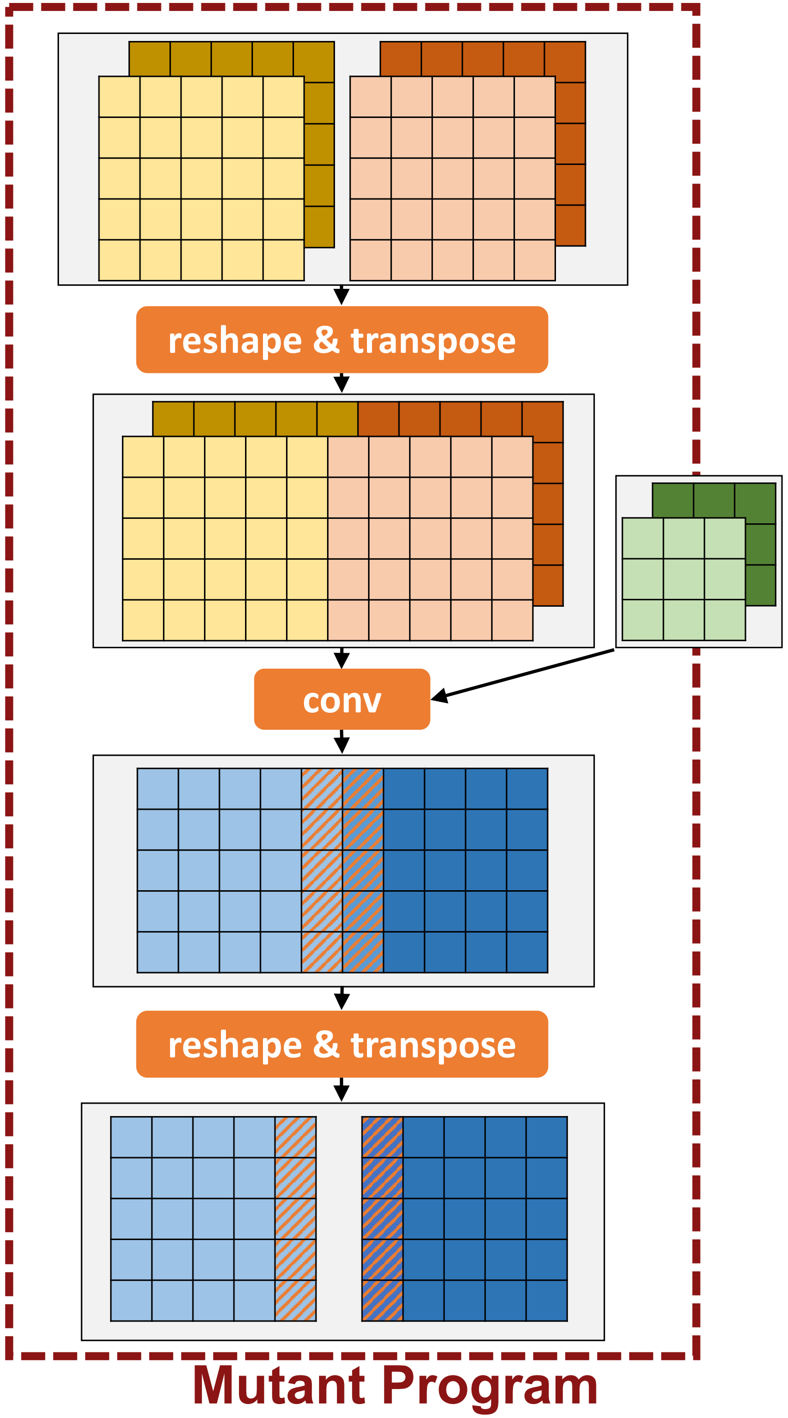


$\forall \boldsymbol{I}.\ \boldsymbol{f}(\boldsymbol{I})[\boldsymbol{p}] = \boldsymbol{g}(\boldsymbol{I})[\boldsymbol{p}]?$

*Proof details available in the paper

Mutant Corrector

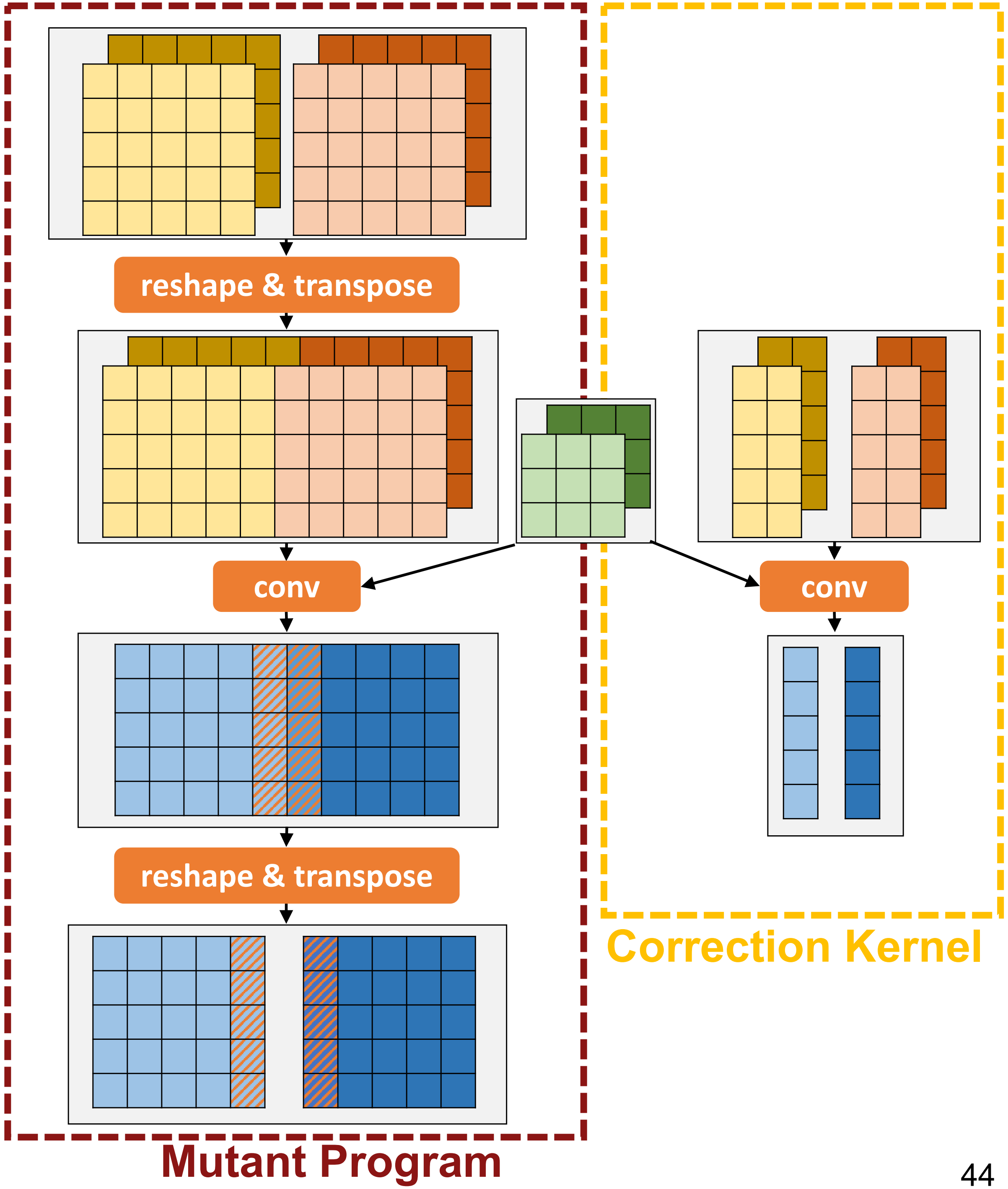
Goal: **quickly** and **efficiently** correcting the outputs of a mutant program



Mutant Corrector

Goal: **quickly** and **efficiently** correcting the outputs of a mutant program

Step 1: recompute the incorrect outputs using the original program



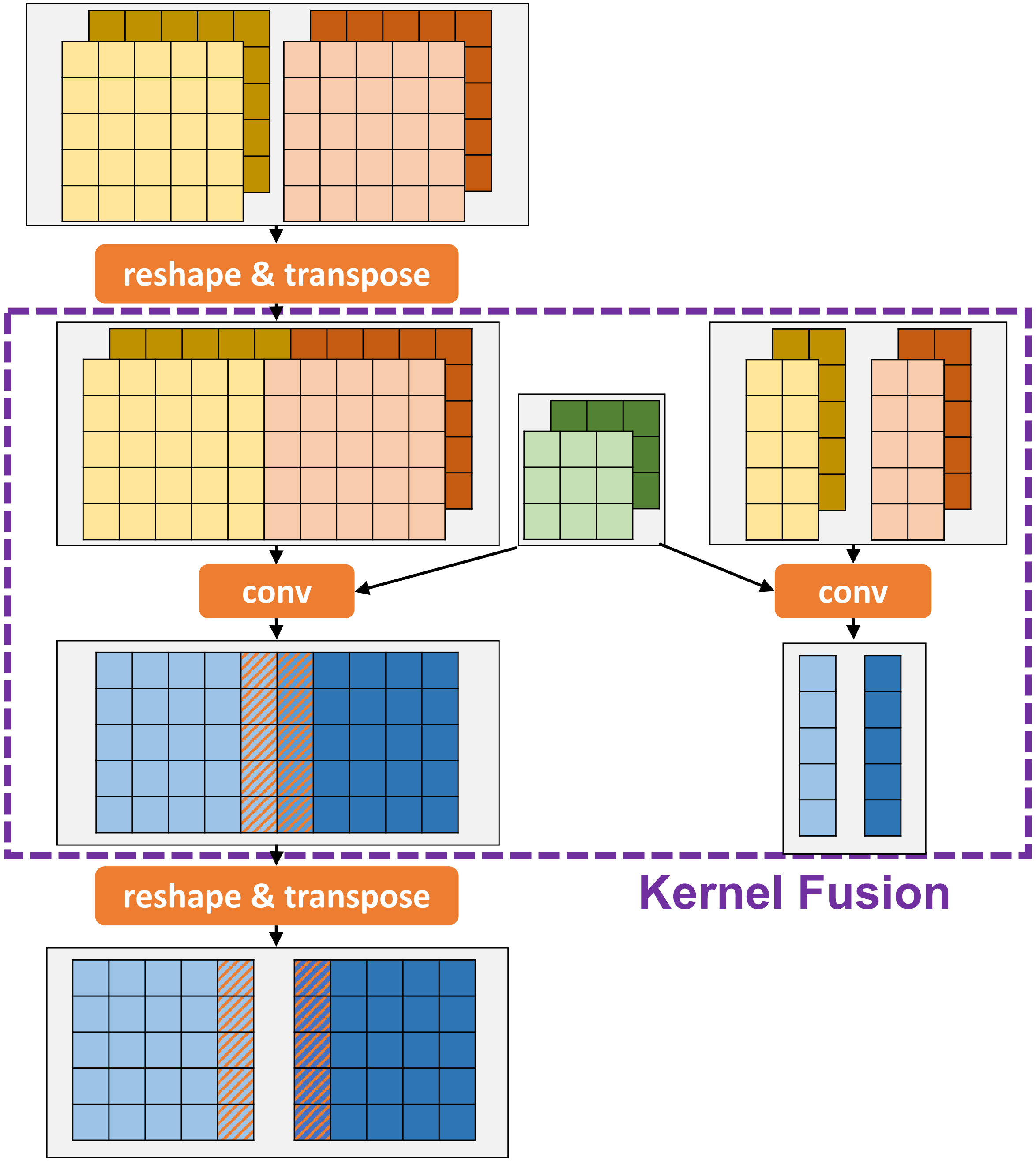
Mutant Corrector

Goal: **quickly** and **efficiently** correcting the outputs of a mutant program

Step 1: recompute the incorrect outputs using the original program

Step 2: opportunistically fuse correction kernels with other operators

Correction introduces less than 1% overhead



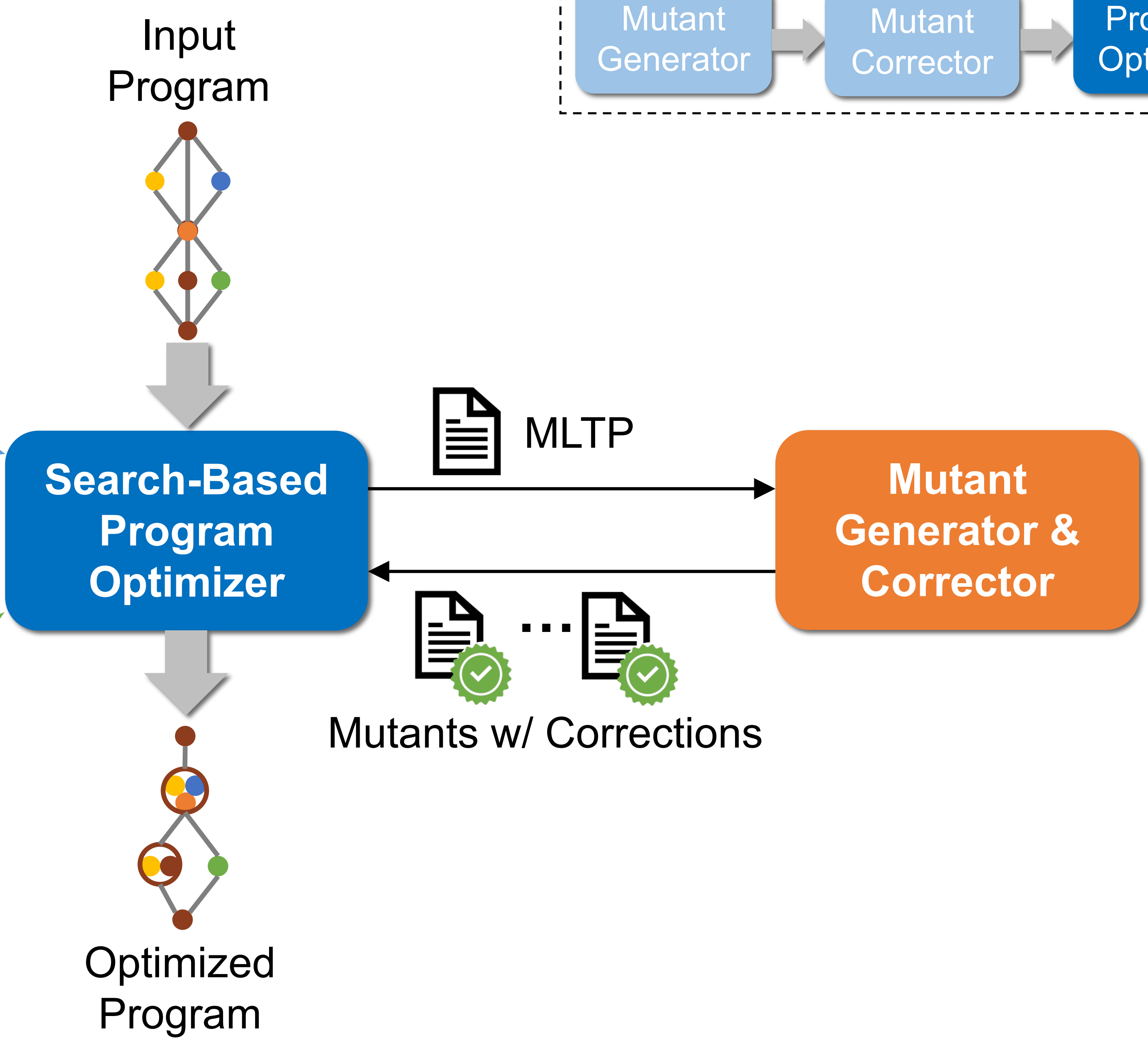
Mutant Generator → Mutant Corrector → **Program Optimizer**

Program Optimizer

- **Beam search**
- Optimizing a DNN architecture takes less than **30** minutes

Other optimizations:

- Operator fusion
- Constant folding
- Redundancy elimination



Input Program

Search-Based Program Optimizer

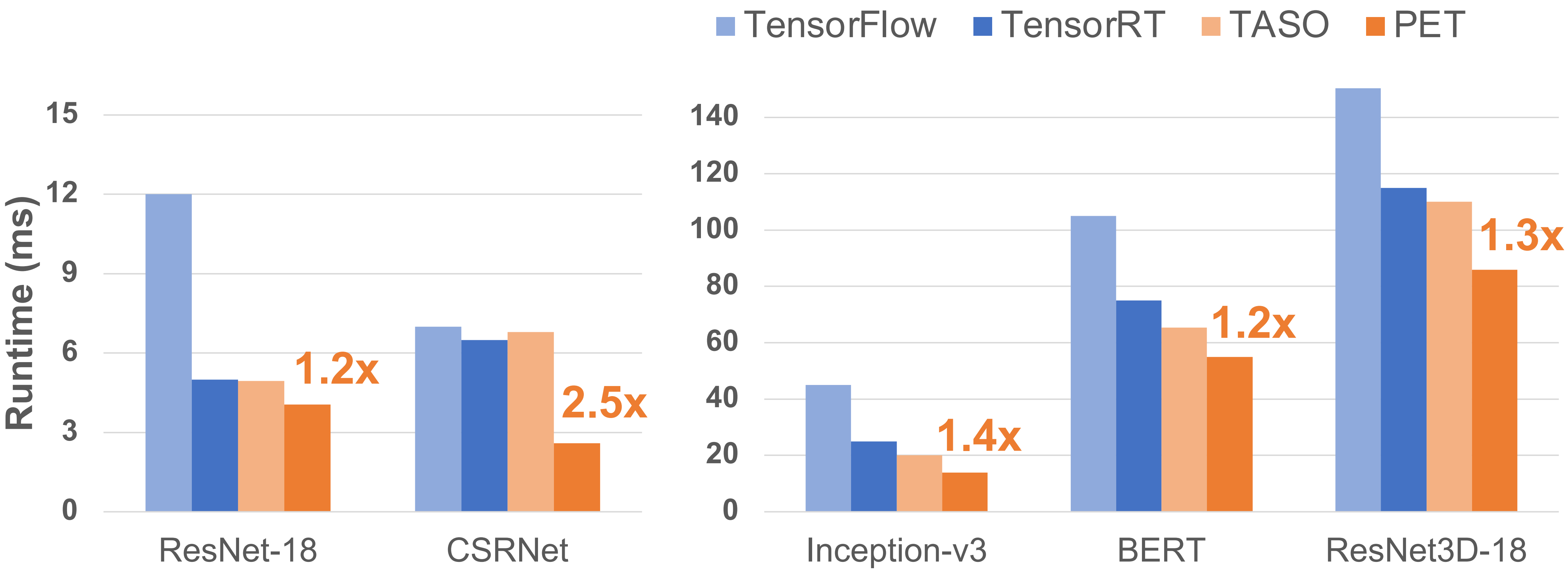
MLTP

Mutant Generator & Corrector

Mutants w/ Corrections

Optimized Program

End-to-end Inference Performance (Nvidia V100 GPU)



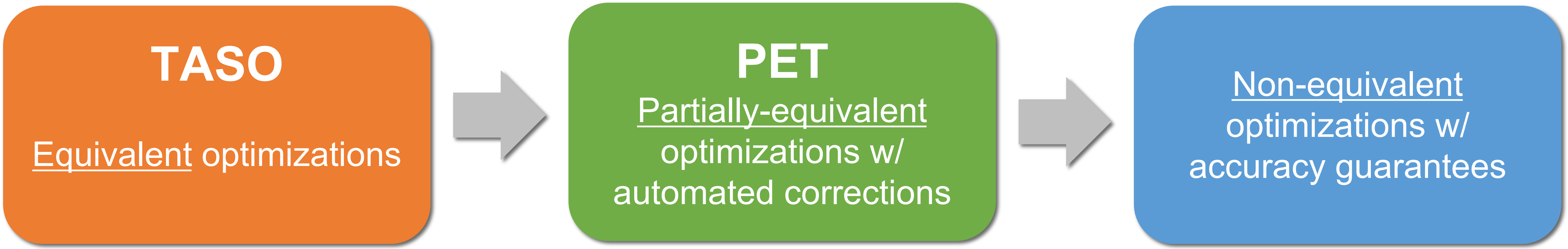
PET outperforms existing optimizers by 1.2-2.5x
by combining fully and partially equivalent transformations

PET

- A tensor program optimizer with partially equivalent transformations and automated corrections

- **Larger optimization space** by combining fully and partially equivalent transformations
- **Better performance**: outperform existing optimizers by up to **2.5x**
- **Correctness**: automated corrections to preserve end-to-end equivalence

From Equivalent to Non-Equivalent Optimizations for ML



TASO

Equivalent optimizations

PET

Partially-equivalent optimizations w/ automated corrections

Non-equivalent optimizations w/ accuracy guarantees

Week 11: Model Pruning, Quantization, Distillation, etc.

Questions to Discuss

1. How does PET differ from TASO in generating graph transformations?

2. How does PET differ from TASO in verifying/correcting transformations?

3. How can we combine graph optimizations with kernel optimizations?