https://hao-ai-lab.github.io/dsc291-s24/

DSC 291: ML Systems
Spring 2024

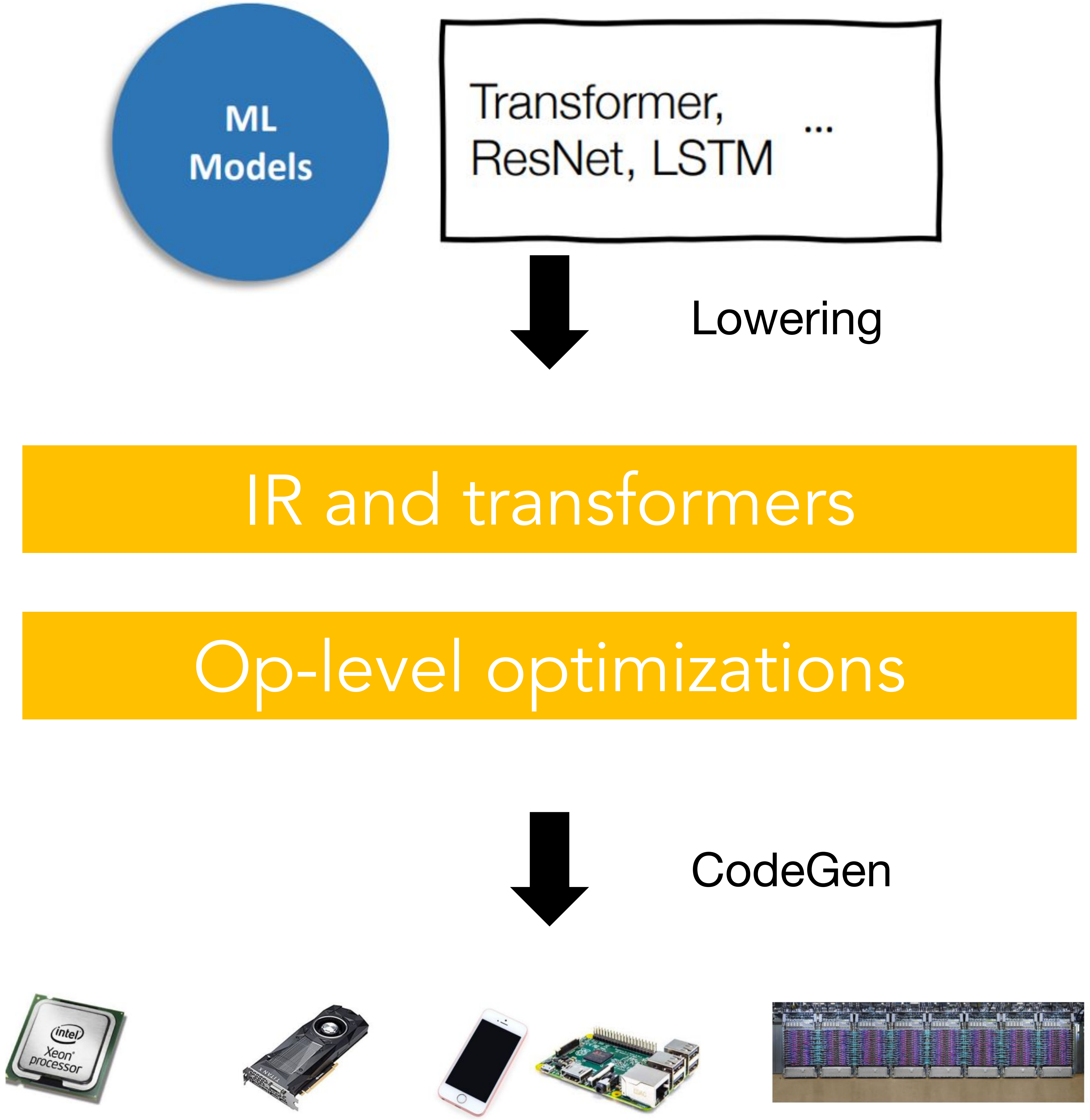
LLMs

Parallelization

Single-device Optimization

Basics

Compilation Process



Compilation Process

ML Models

Transformer, ResNet, LSTM ...

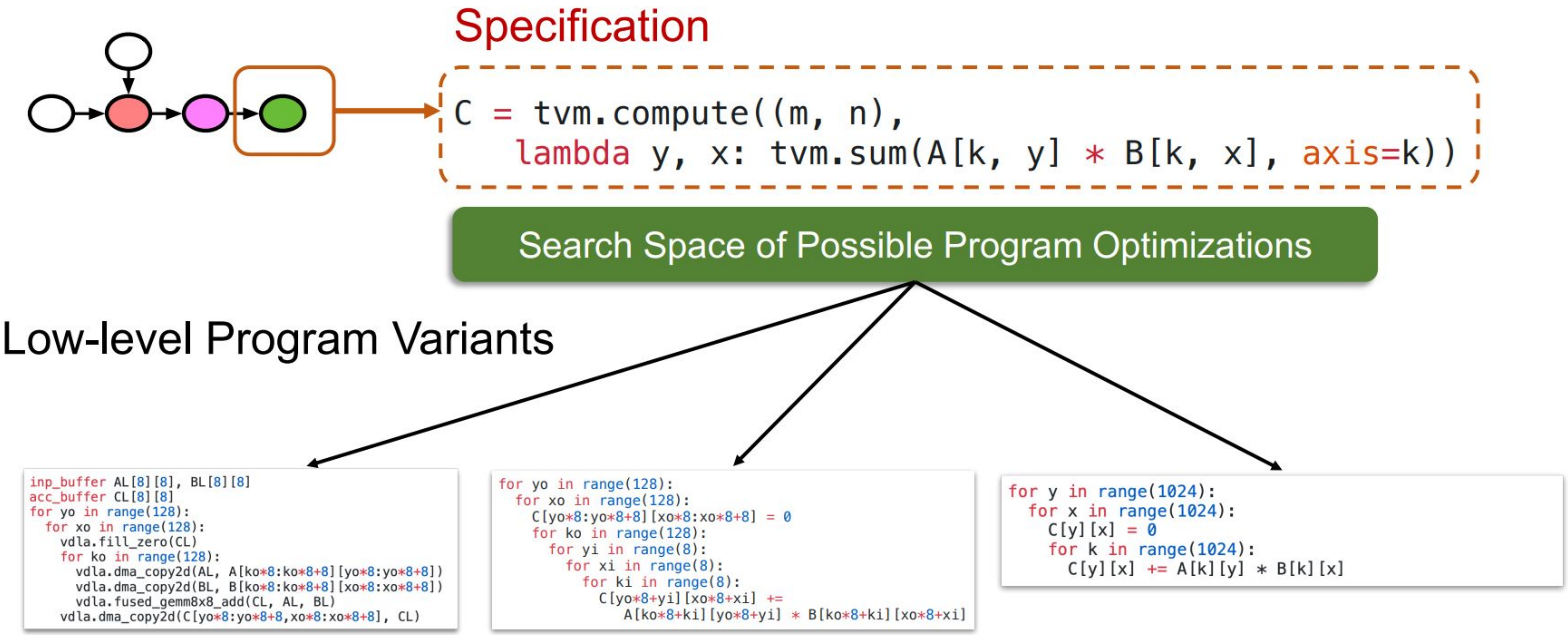
Lowering

IR and transformers

Op-level optimizations

CodeGen

Lower-level code optimization



Low-level Loop Representation

@dot-add(x, w, b)

```
for i, j in grid(16, 16):
  Y[i, j] = 0
for i, j, k in grid(16, 16, 16):
  Y[i, j] += x[i, k] * w[k, j]
for i, j in grid(16, 16):
  Z[i, j] = Y[i, j] + b[j]
```

Multi-dimensional buffer

Loop nests

Array computation

Transforming Loops: Loop Splitting

Code

```
for x in range(128):
  C[x] = A[x] + B[x]
```

```
for xo in range(32):
  for xi in range(4):
    C[xo * 4 + xi]
     = A[xo * 4 + xi] + B[xo * 4 + xi]
```

```
for xi in range(4):
  for xo in range(32):
    C[xo * 4 + xi]
     = A[xo * 4 + xi] + B[xo * 4 + xi]
```

```
def gpu_kernel():
  C[threadId.x * 4 + blockIdx.x] = . . .
```

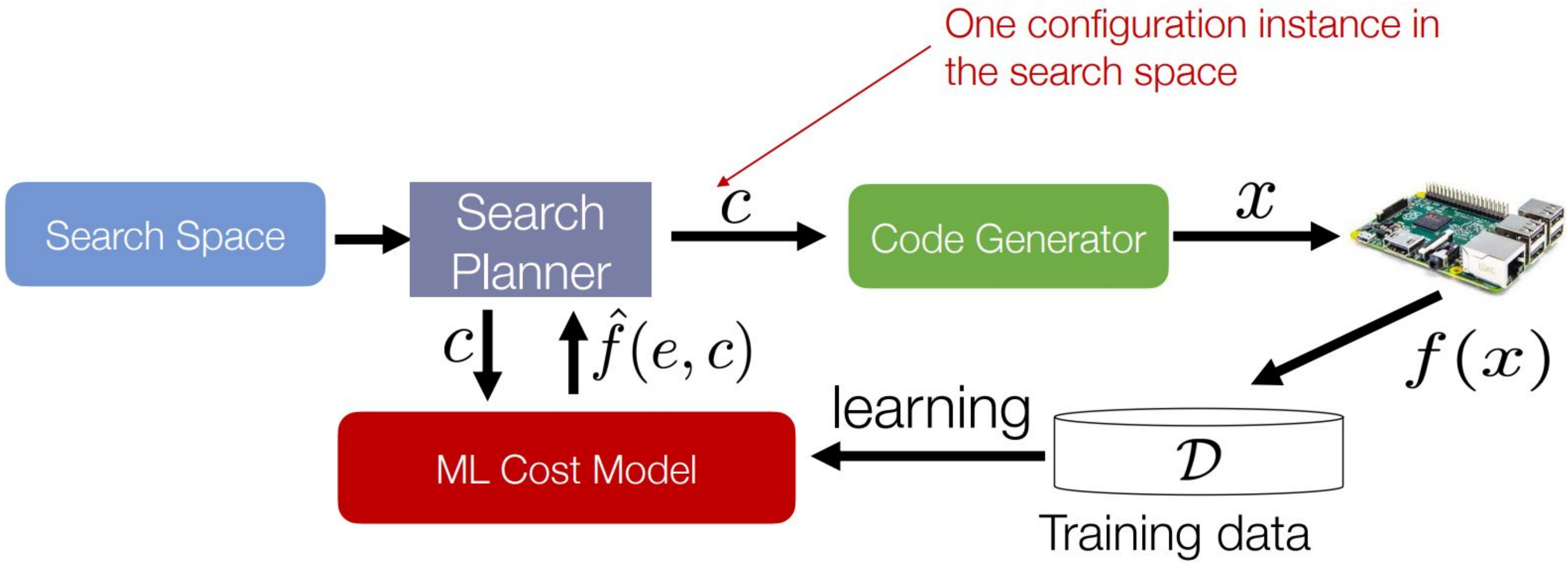
Problems

- We need to enumerate so many possibilities
- We need to fit with each device (register/cache sizes)
- We need to apply this to so many operators

Core Research Problems

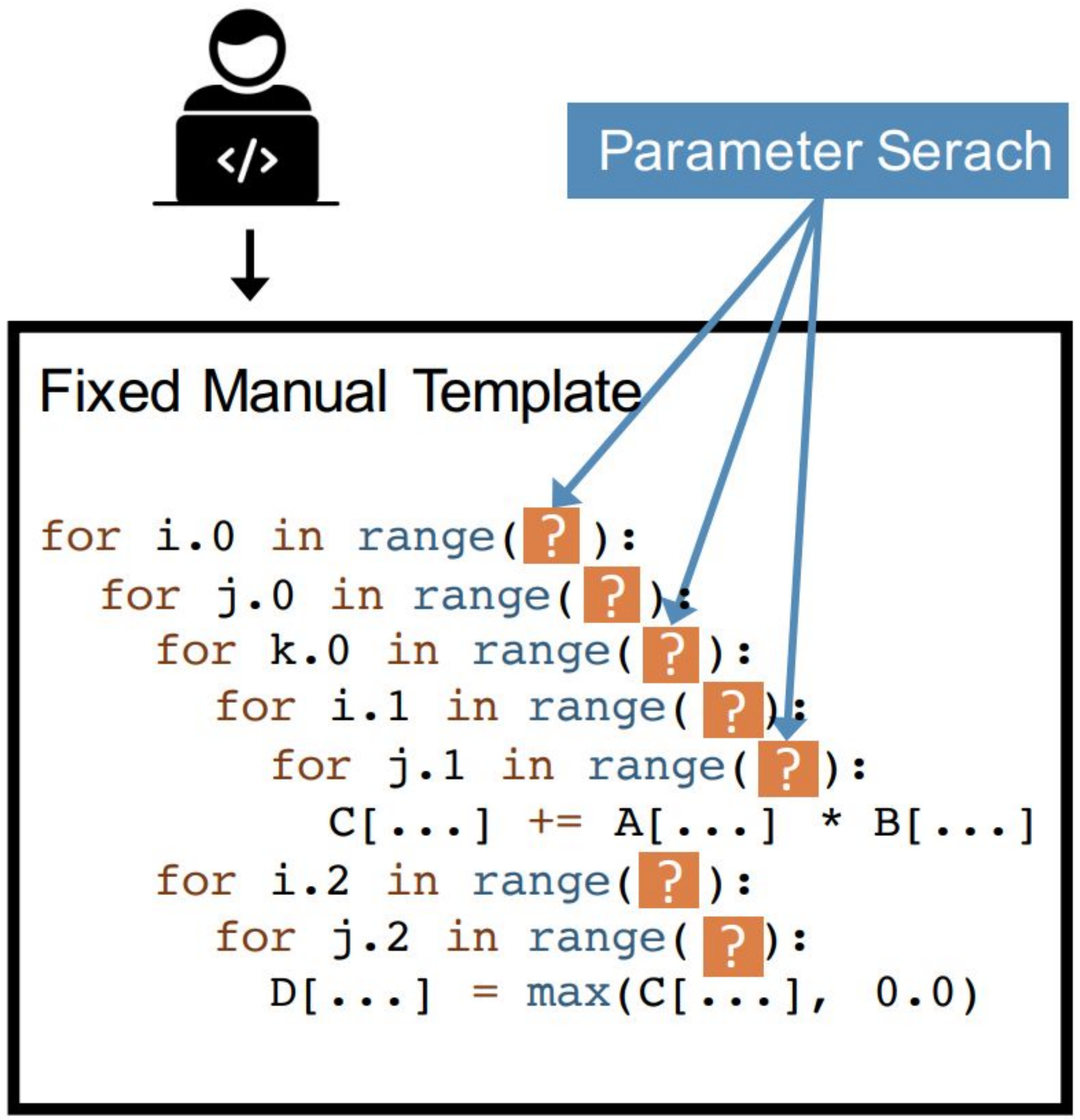
- We need to enumerate so many possibilities
  - How to represent all possibilities
  - What is the problem of missing some possibilities?
- We need to find the (close-to-)optimal values(register/cache sizes)
  - How to search?
- We need to apply this to so many operators and devices
  - How to reduce search space
  - How to generalize?

Search via Learned Cost Model



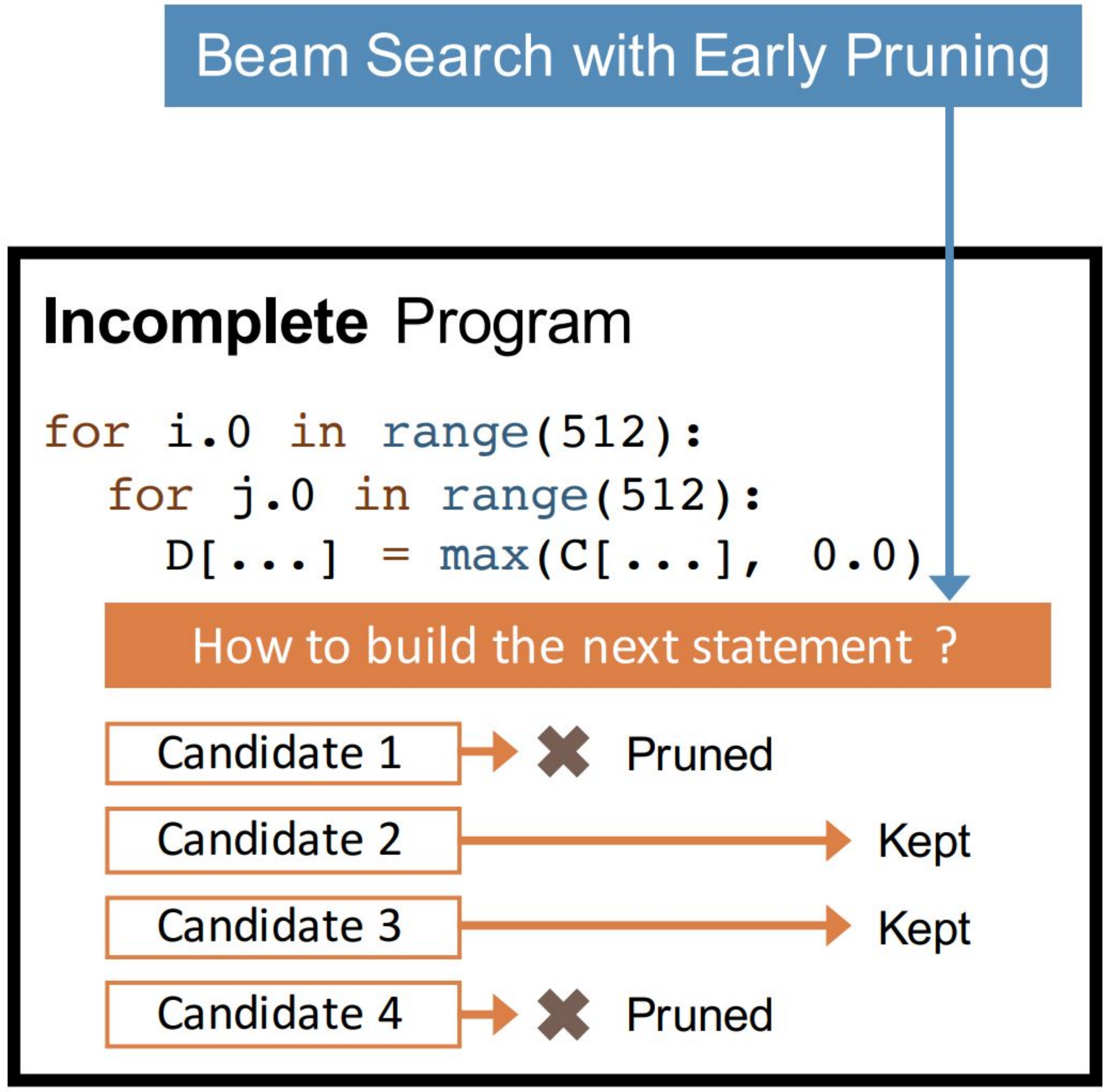
Search Space Definition e.g. Template based

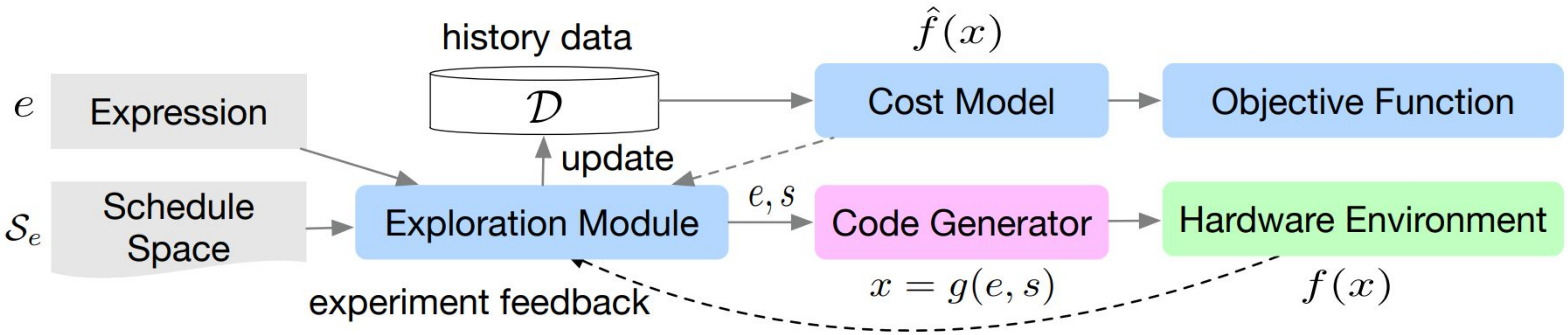
- Issue: still need experts to write templates



How to Search

- Sequential Construction using Early pruning
- Cost Model





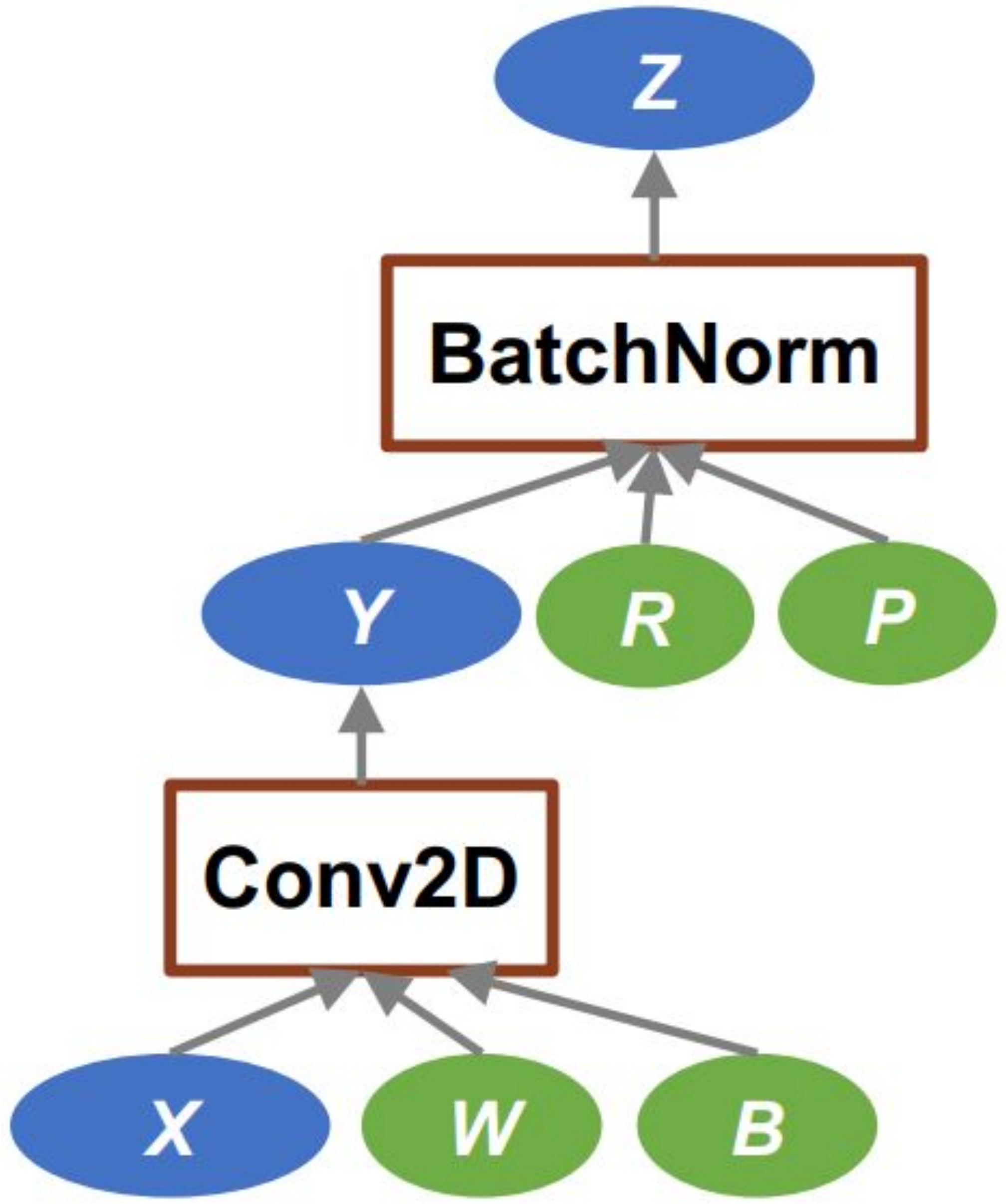
Summary: Operator Compiler

- Program abstraction
  - Represent the program/optimization of interest
- Build Search space through a set of transformations
  - Good coverage of common optimizations like tiling
- Effective Search
  - Accurate cost models
  - Transferability

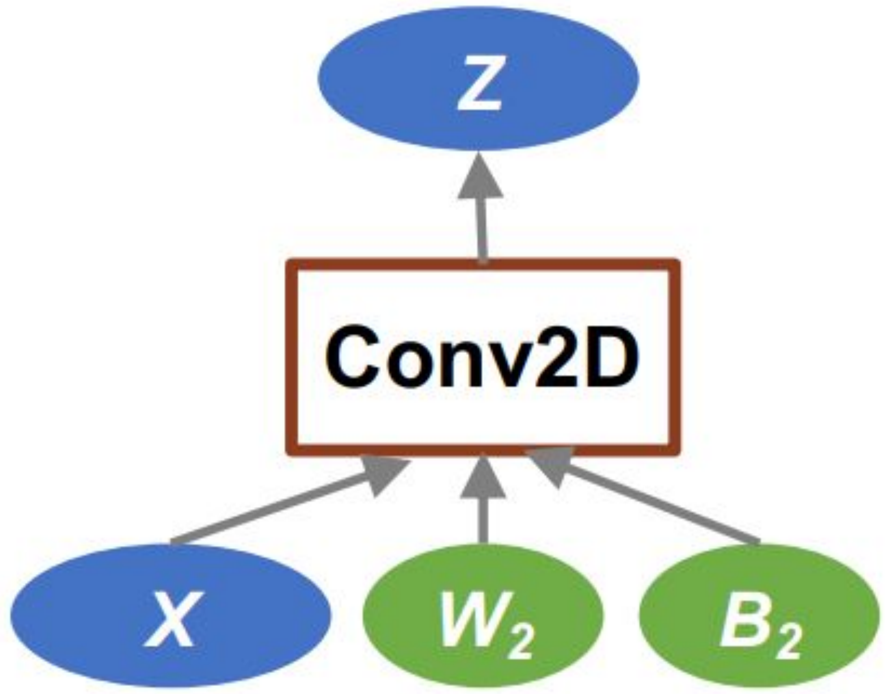
Agenda on this part

- ML Compilation Overview
  - Operator compilation
  - **Graph optimization**
- Memory Optimization
  - Activation checkpointing
  - Quantization and Mixed precision
- Two Guest Talks covering details in compilation, JIT, graph fusion, and beyond:
  - Meta PyTorch lead developer: Jason Ansel

Recall: fusing conv and bn



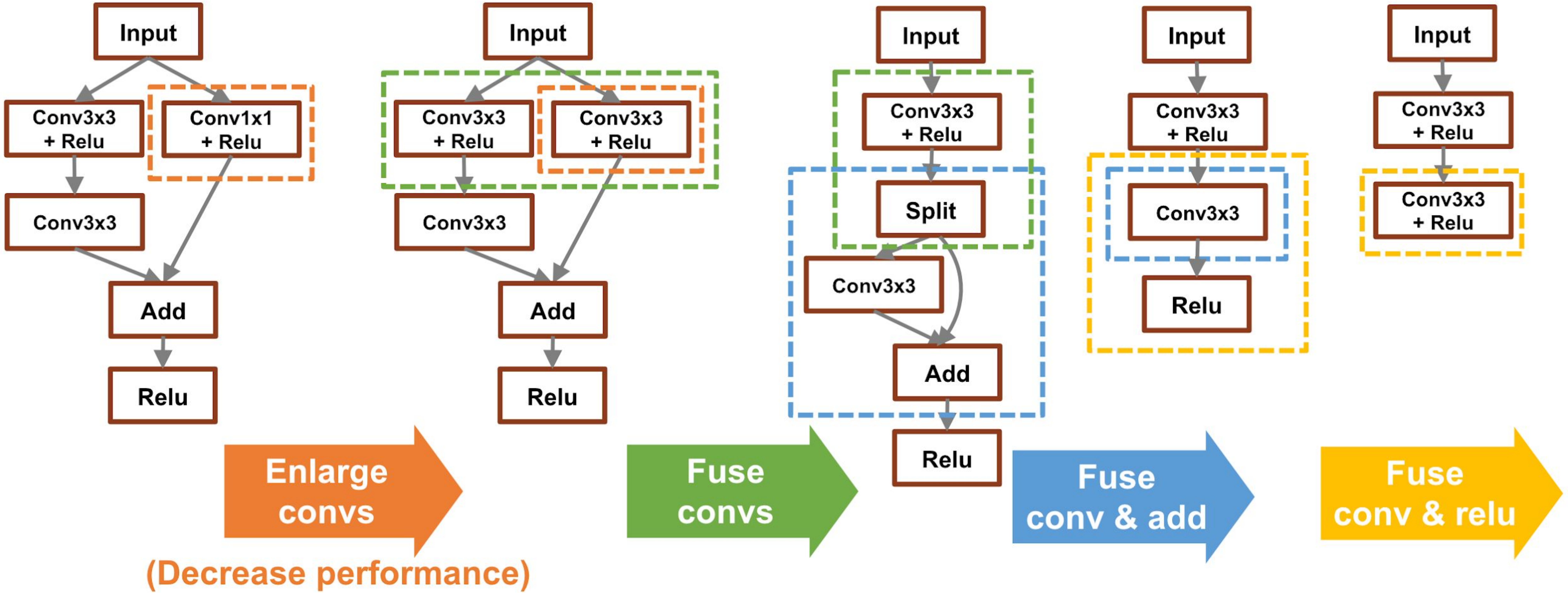
$$Z(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W_2\,(c,d,u,v) \right) + B_2\,(n,c,h,w)$$



$$W_2(n,c,h,w) = W(n,c,h,w) * R(c)$$

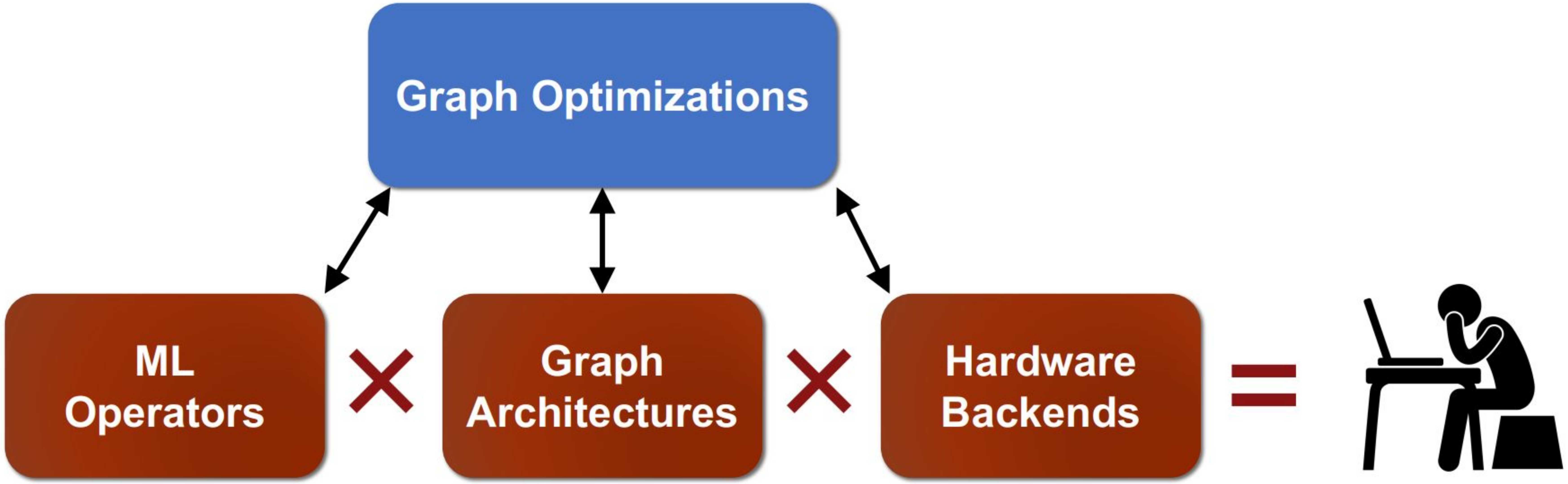
$$B_2(n,c,h,w) = B(n,c,h,w) * R(c) + P(c)$$

Recall: ResNet



- The final graph is 30% faster on V100 but 10% slower on K80.

Problems of High-level Graph Optimizations



200 - 300 1000s 10s

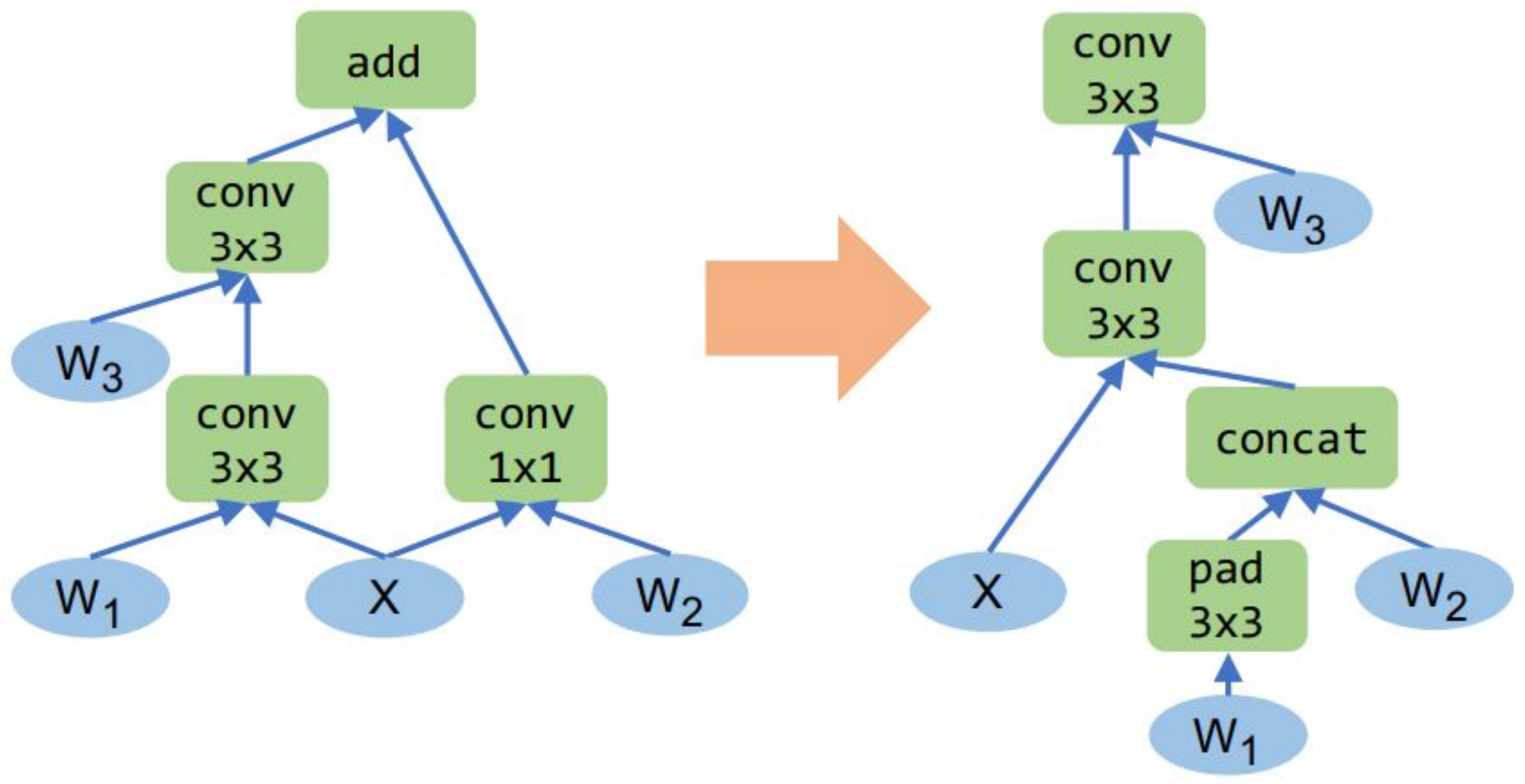
Problem: Infeasible to manually design graph optimizations
for all cases

Summary of Limitations

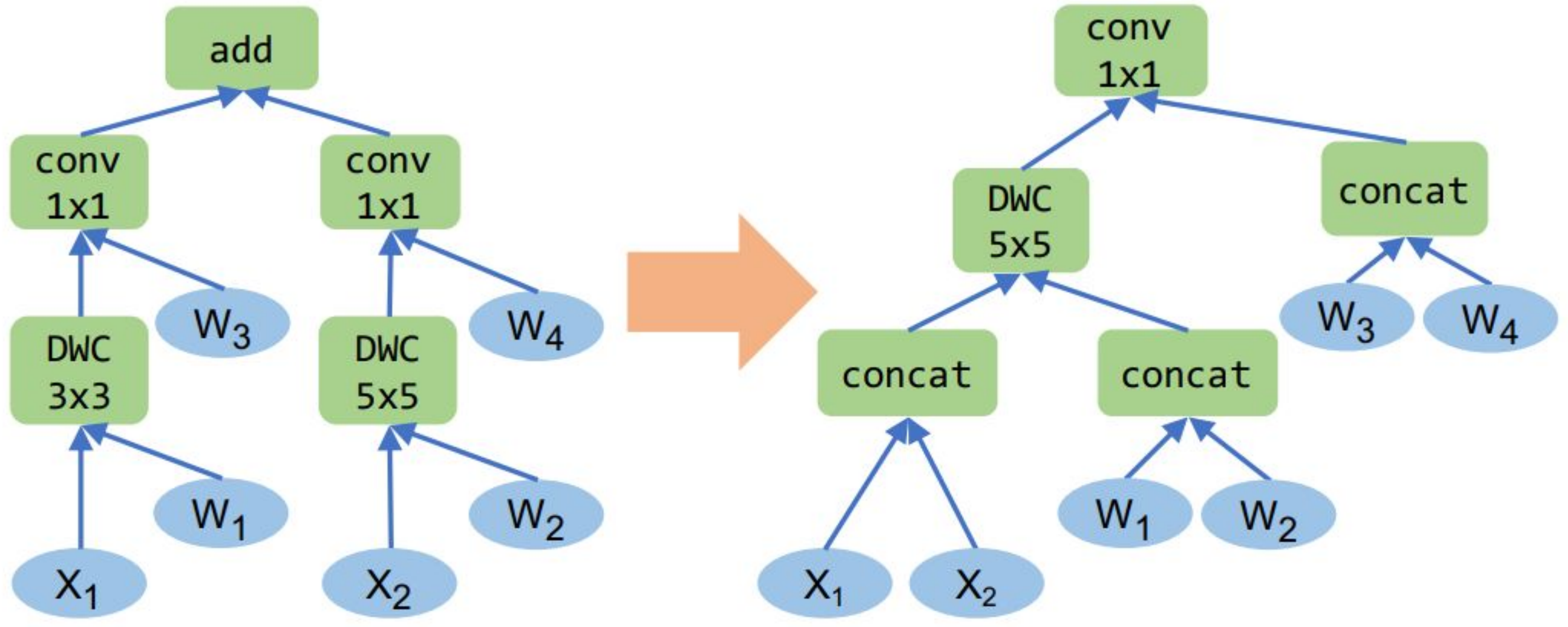
Robustness
Experts' heuristics do not apply to all DNNs/hardware

Scalability
New operators and graph structures require more rules

Performance
Miss subtle optimizations for specific DNNs/hardware



Only apply to **specific hardware**



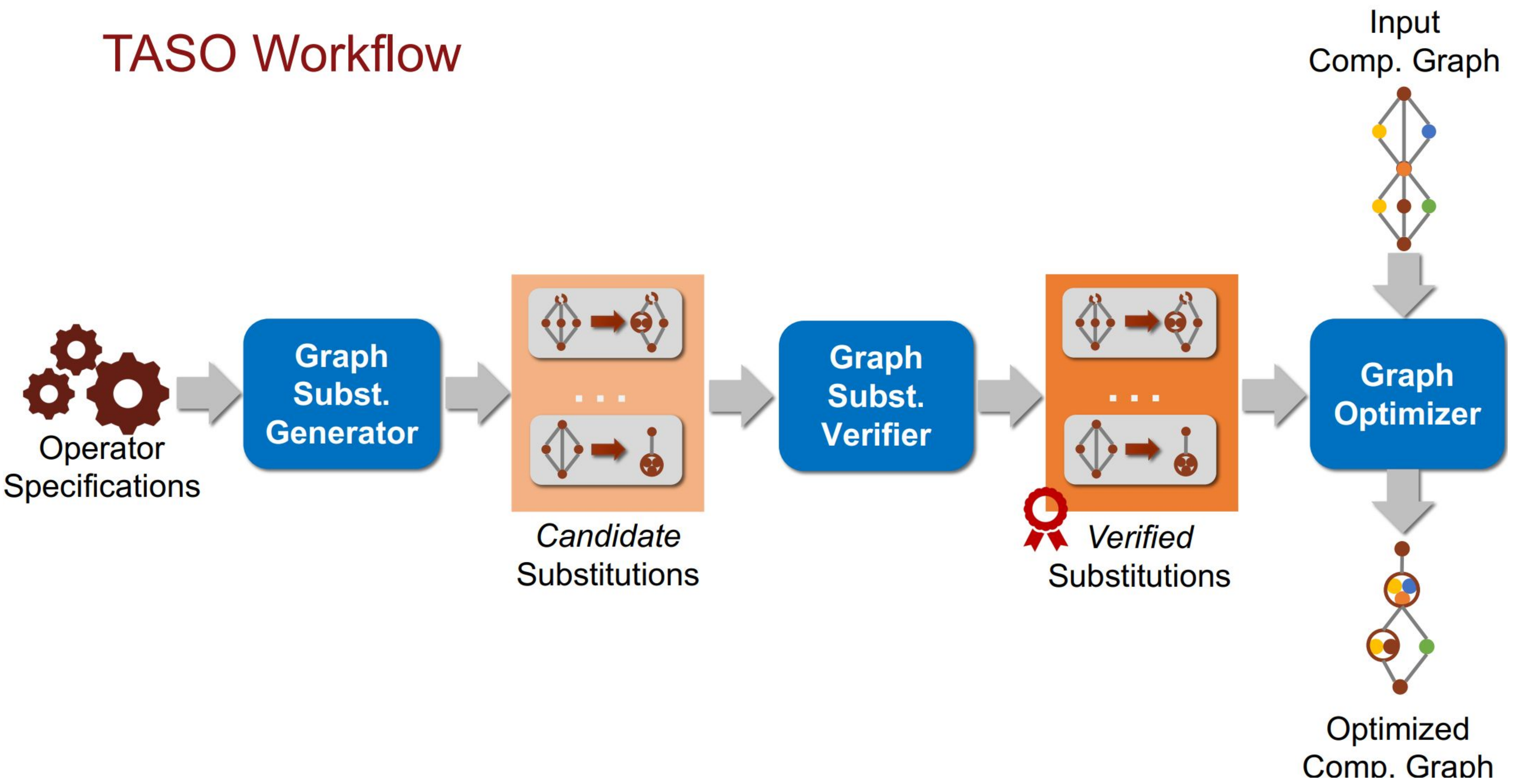
Only apply to **specialized graph structures**

Automate Graph Transformation Big Ideas

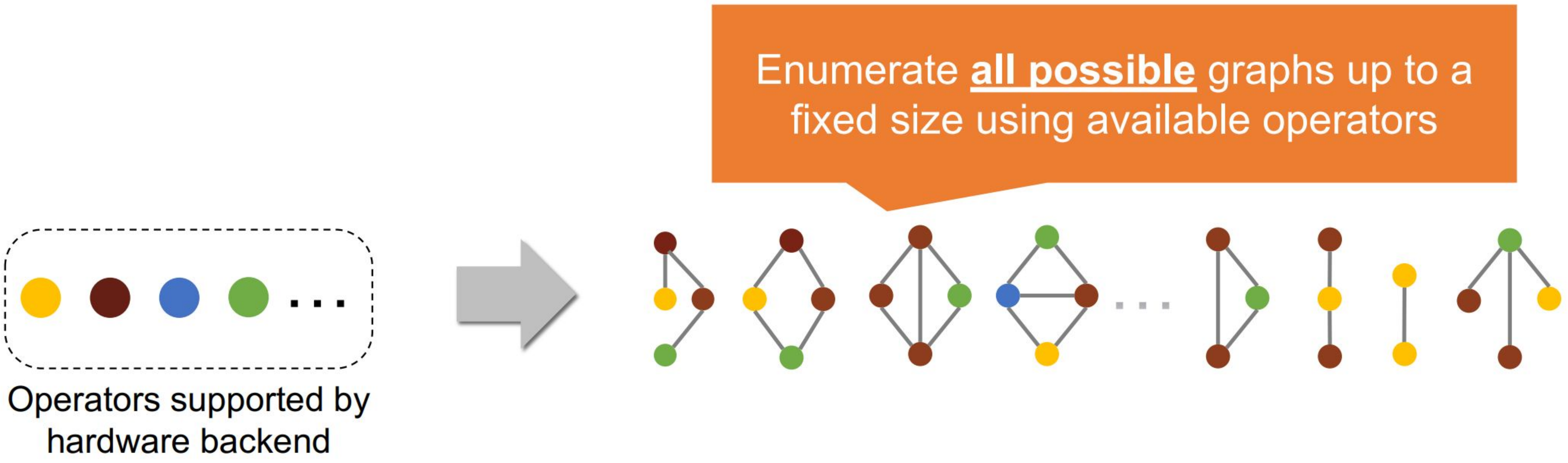
Key idea: replace manually-designed graph optimizations with automated generation and verification of graph substitutions for tensor algebra

- Less engineering effort: 53,000 LOC for manual graph optimizations in TensorFlow → 1,400 LOC
- Better performance: outperform existing optimizers by up to 3x
- Correctness: formally verified

TASO: Enumerate and Verify ALL possible graph

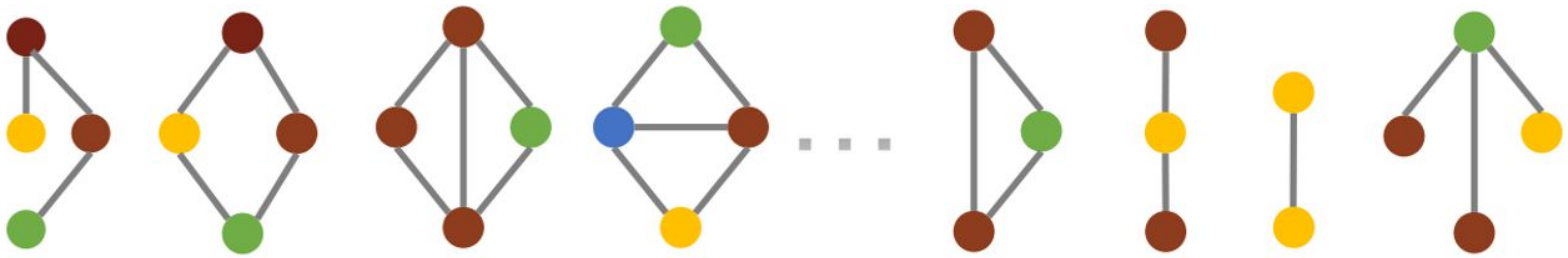


Graph Substitution Generator



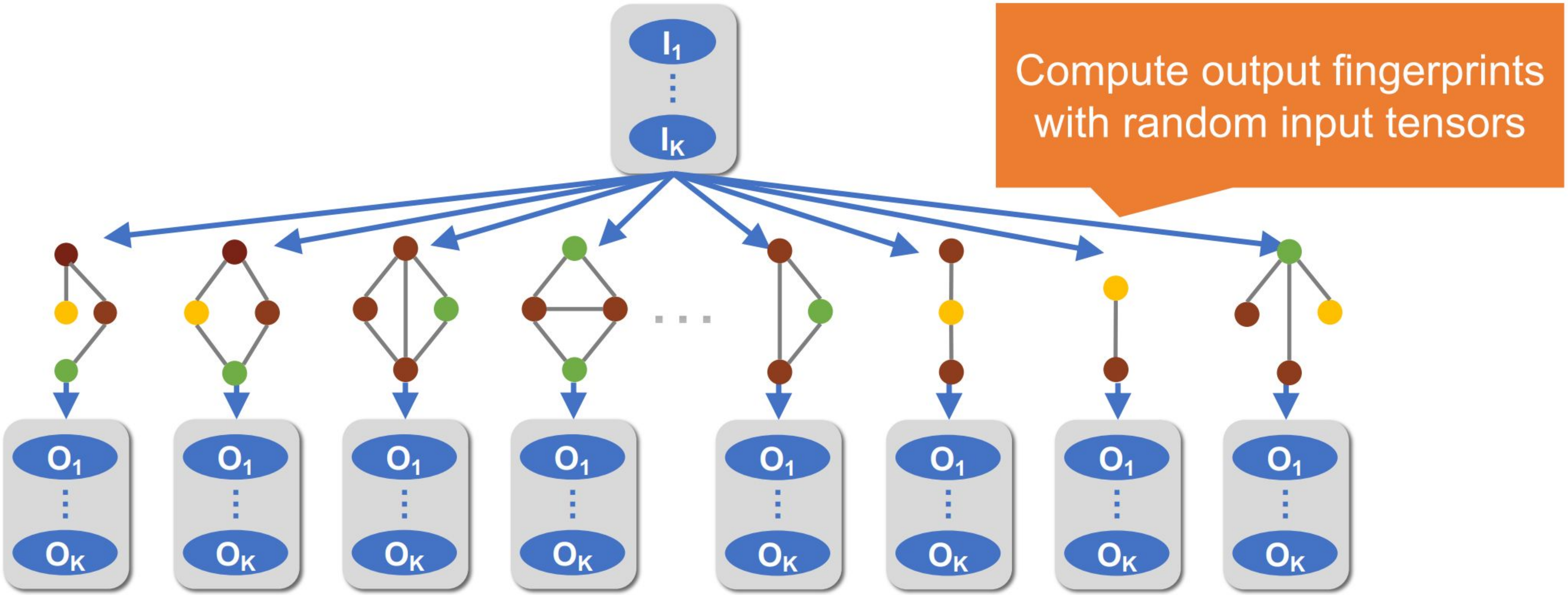
There are many subgraphs even only given 4 Ops

66M graphs with up to 4 operators



A substitution = a pair of equivalent graphs

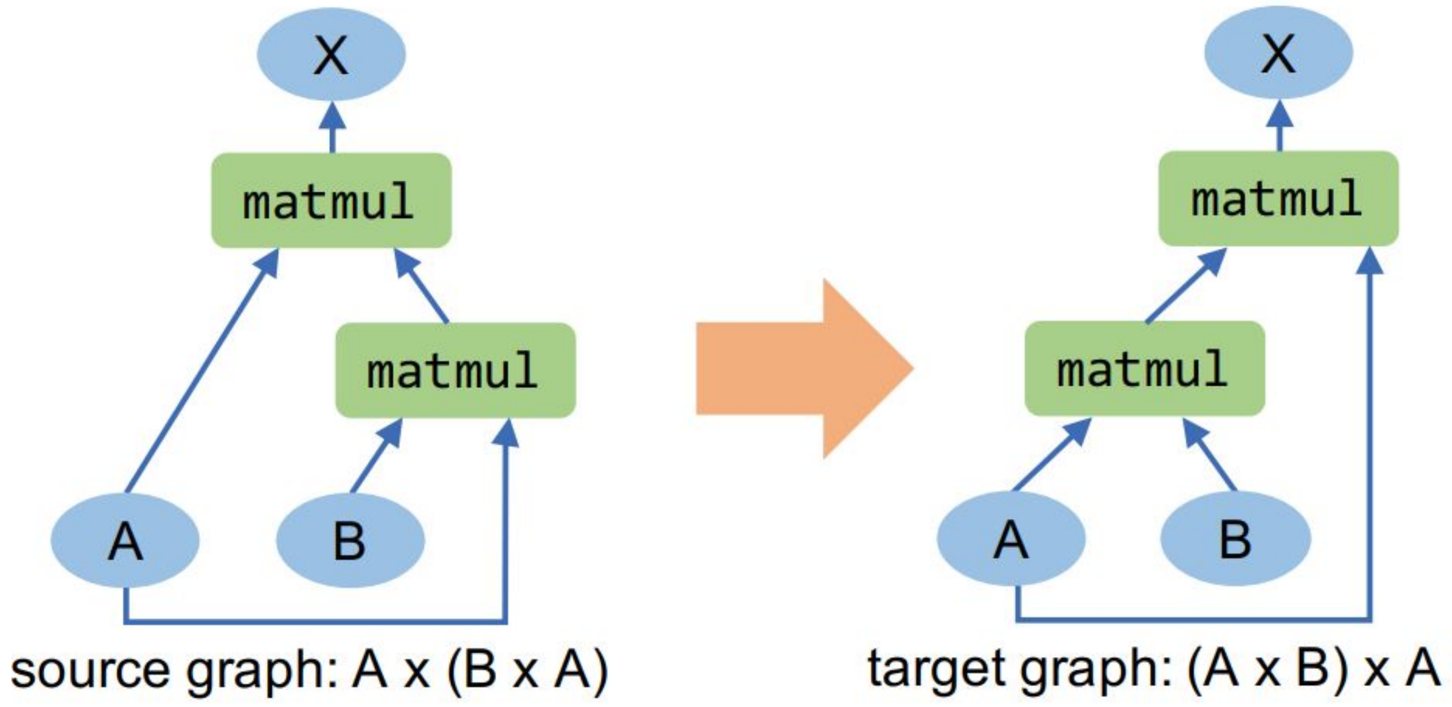
Graph Substitution Generator



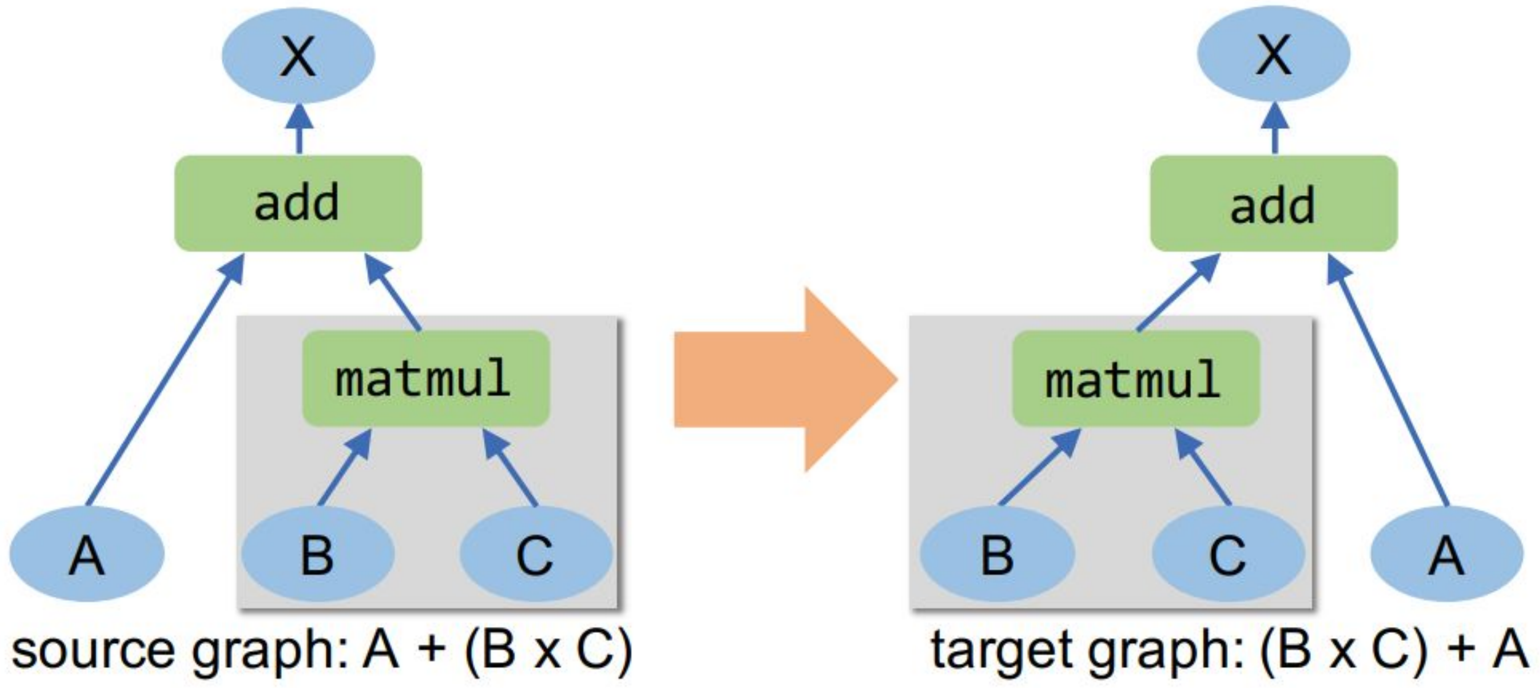
TASO generates 28744 substitutions by enumerating graphs with up to 4 ops

Pruning repeated graphs

28744 substitutions



Variable renaming



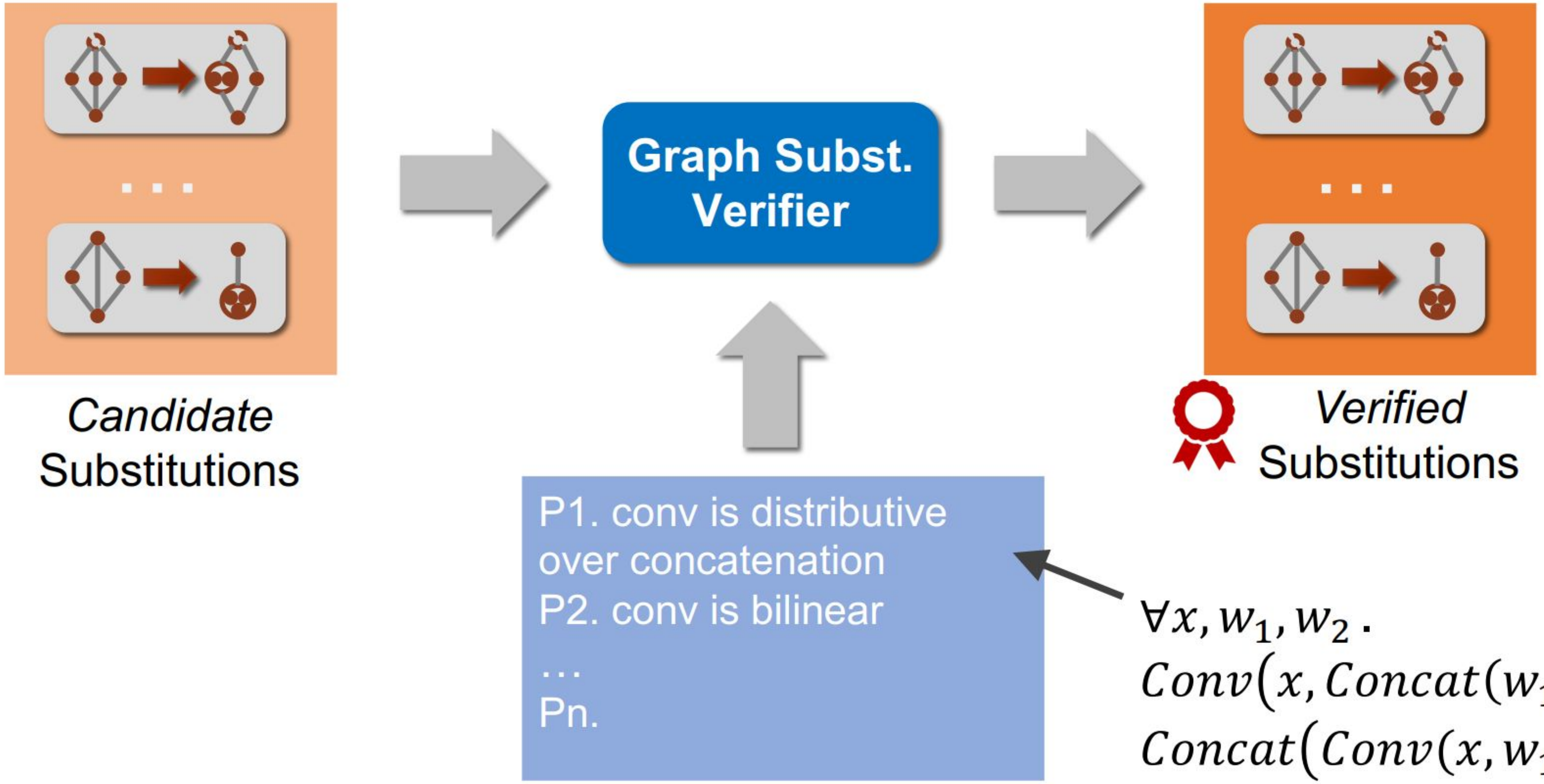
Common subgraph

734 substitutions

Can we trust graph substitutions?

- We have $f(a) = g(b)$, $f(b) = g(b)$
  - But can we say: $f(x) = g(x)$ for $\forall x$
- We need to verify formally.

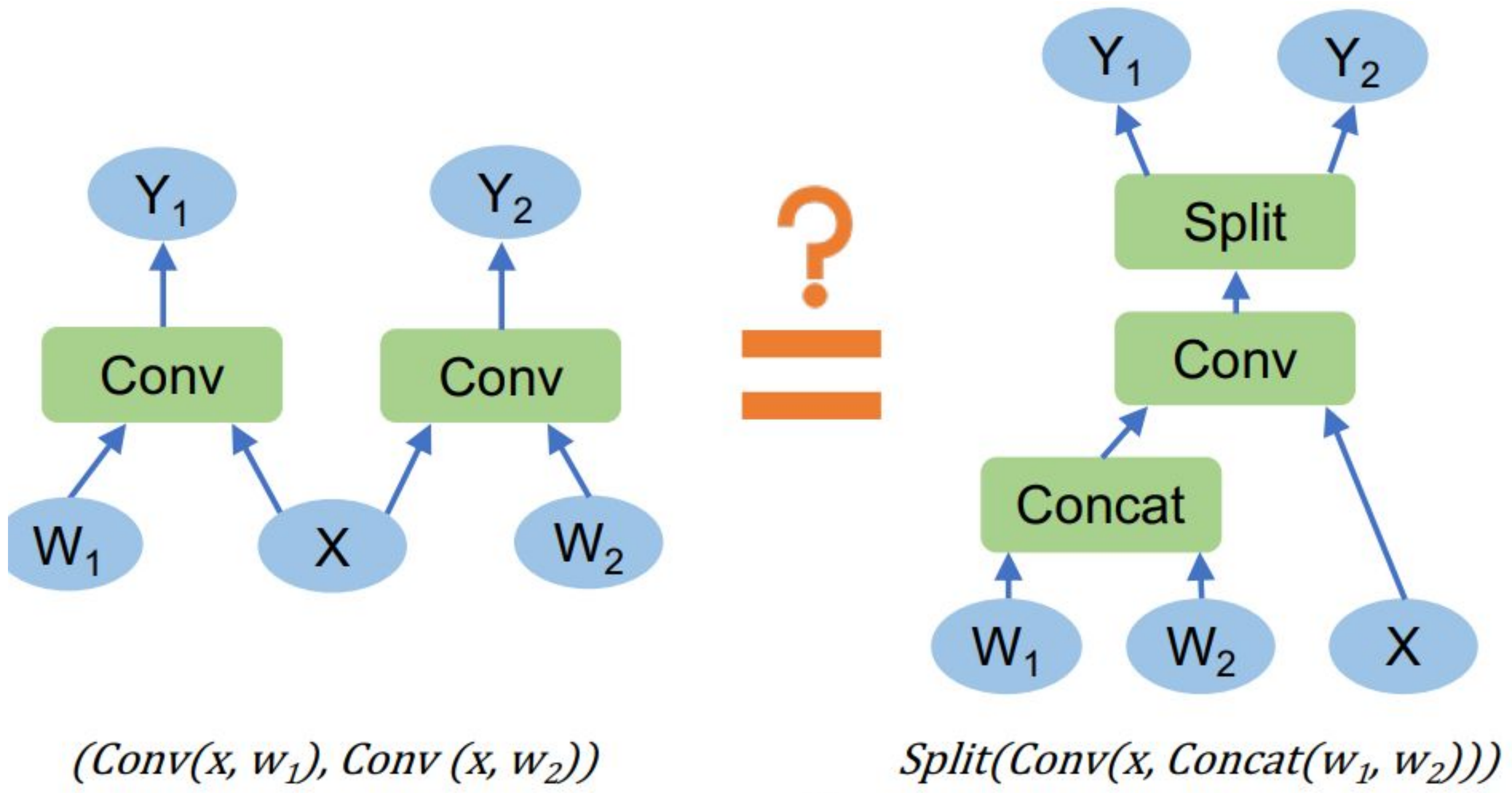
Substitution Verifier



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

Idea: writing specifications are easier than actually, conducting the optimizations

How to Verify

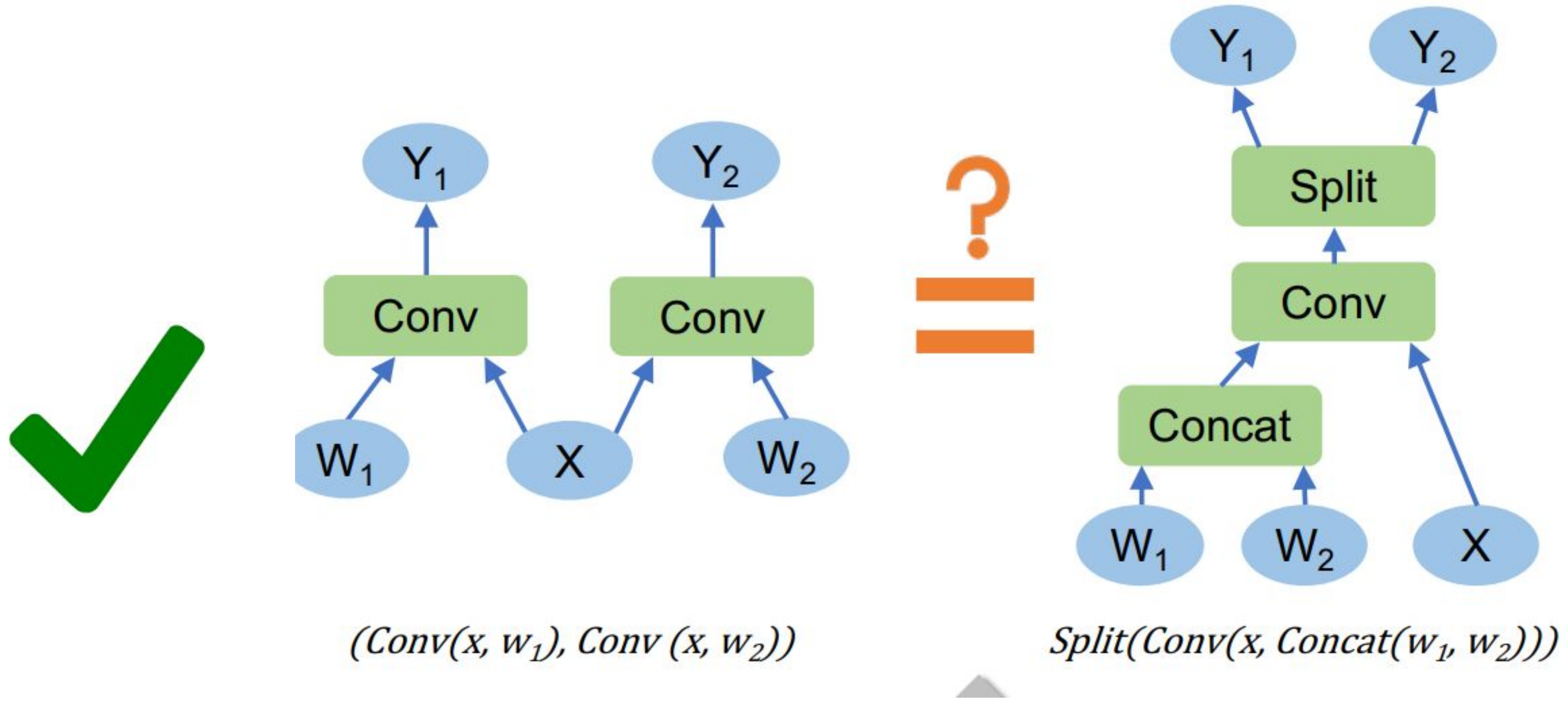


$$\forall x, w_1, w_2 .\ \left(Conv(x, w_1), Conv(x, w_2)\right) = Split\left(Conv\left(x, Concat(w_1, w_2)\right)\right)$$

P1. $\forall x, w_1, w_2 .\ Conv\left(x, Concat(w_1, w_2)\right) = Concat\left(Conv(x, w_1), Conv(x, w_2)\right)$

P2. …

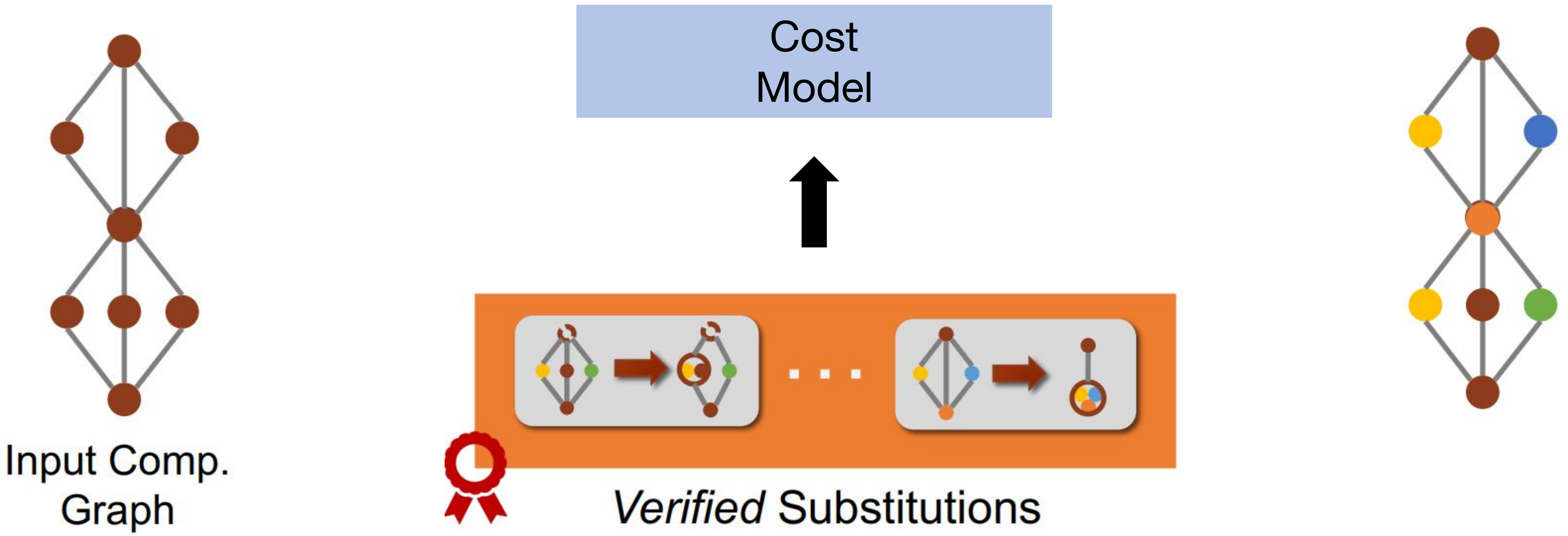
Automated theorem prover



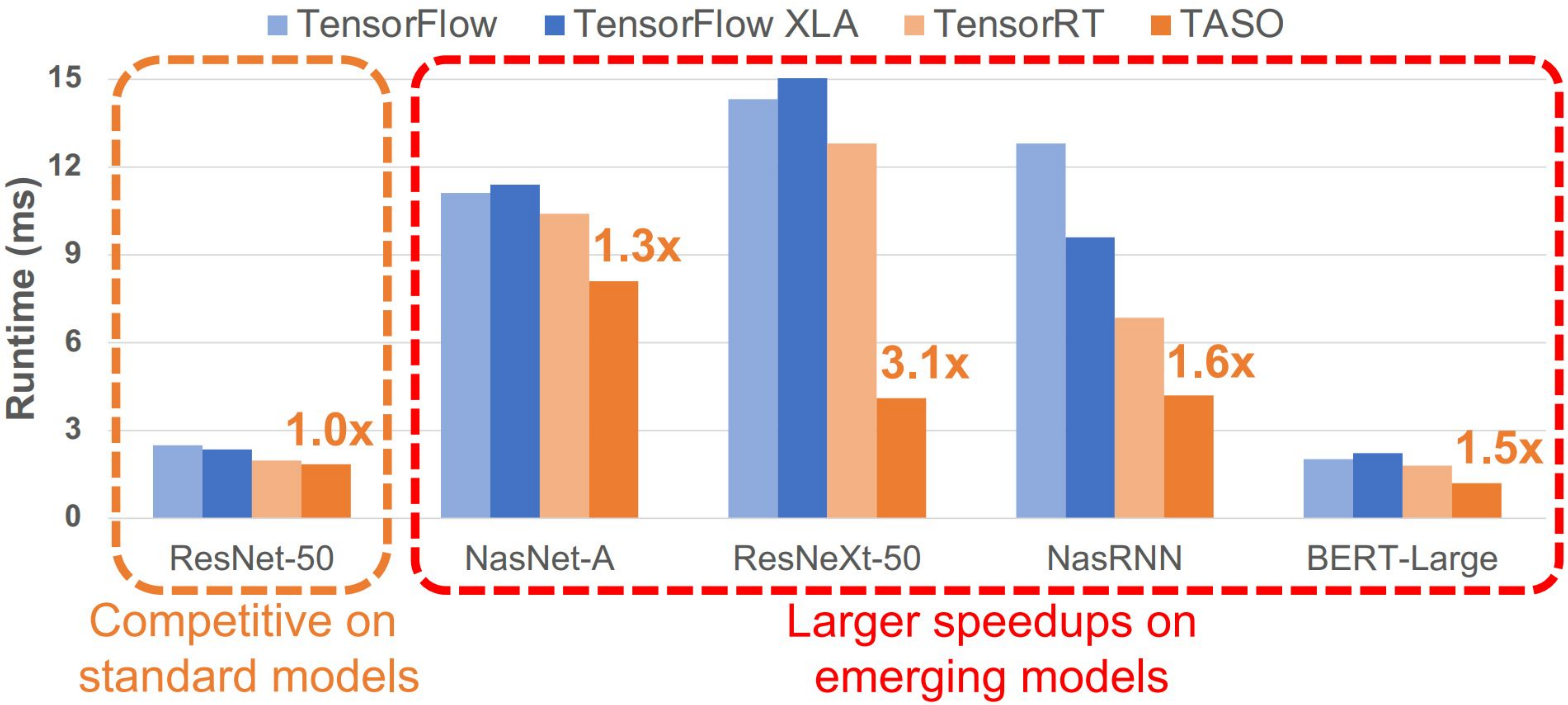
- Generating 743 substitutions = 5 mins
- Verify against 43 op specs = 10 mins
- Supporting a new op requires experts to write specs = 1400 LoC
  - vs. 53K LoC of manual optimization in TF

Incorporating substitutions

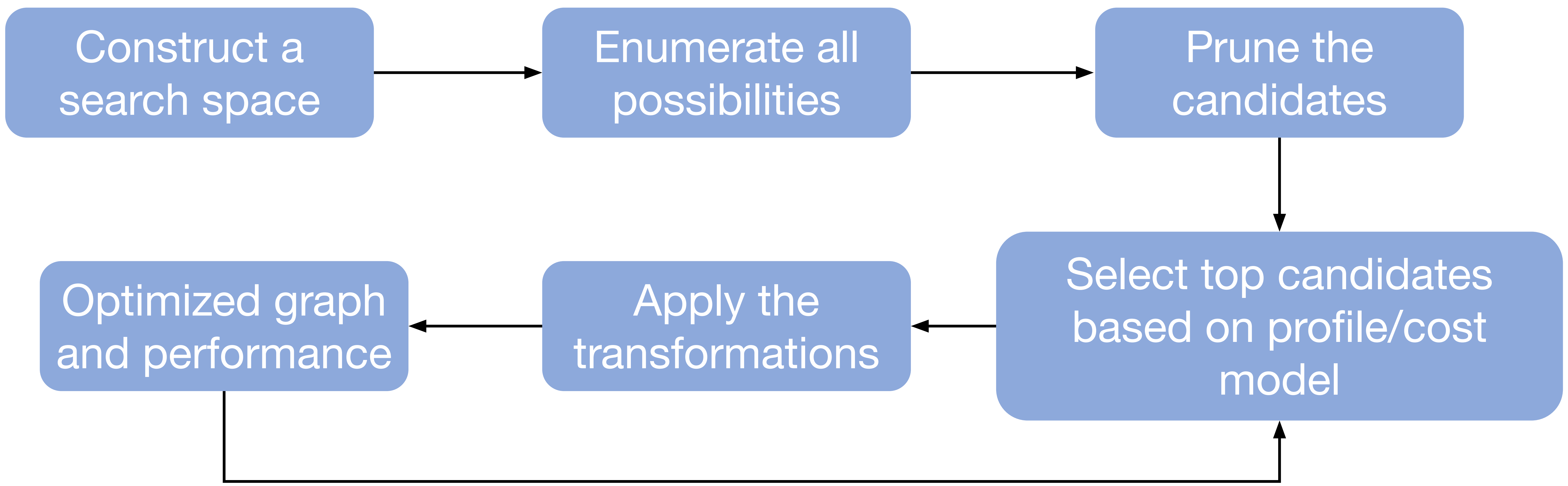
- Goal: apply verified substitutions to obtain an optimized graph
- Cost Model
  - Based on the sum of individual operator's cost
  - Profile each operator's cost on the target hardware
- Traverse the graph, apply substitutions, calculate cost, use backtracking



Performance (as of 2019)



Summary of Graph Optimization

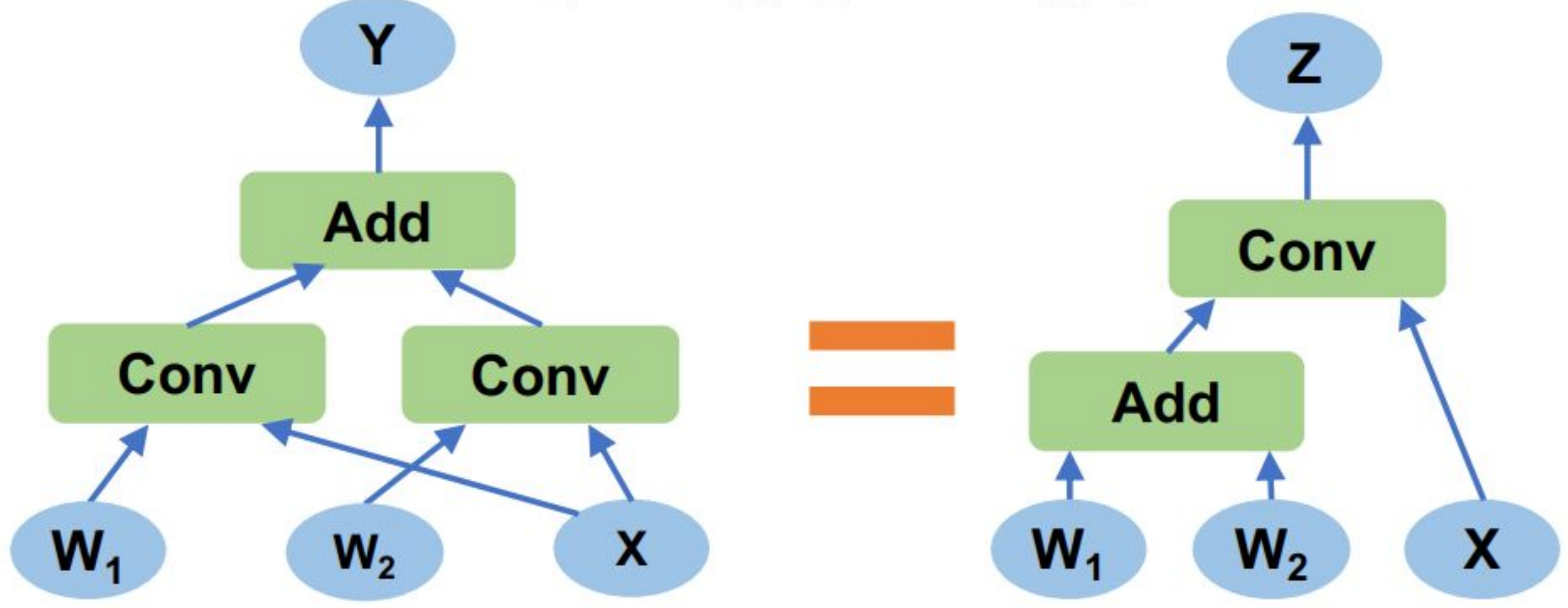


Limitations

- The best optimization is not covered by search space
- Search is too slow
- Evaluation of the resulting graph is too expensive
  - Limits your trial-and-error times

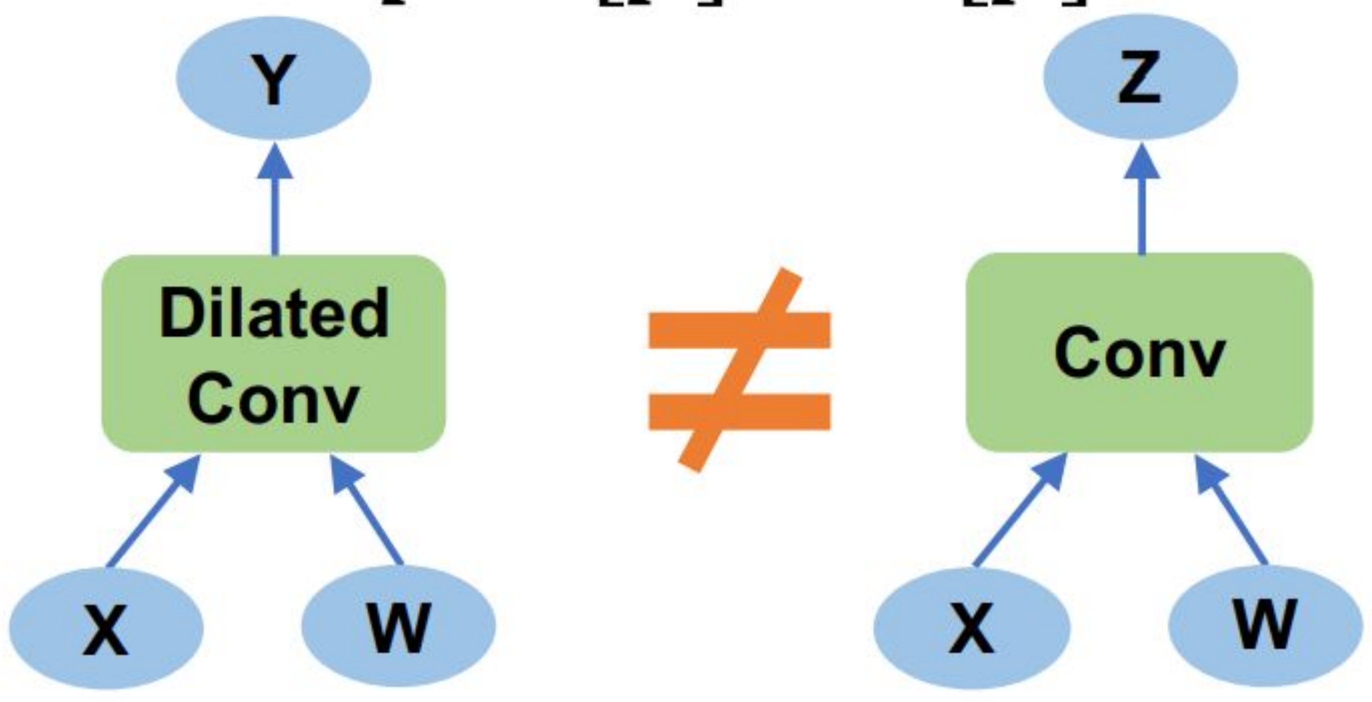
A Failure Example

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



Fully Equivalent Transformations

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



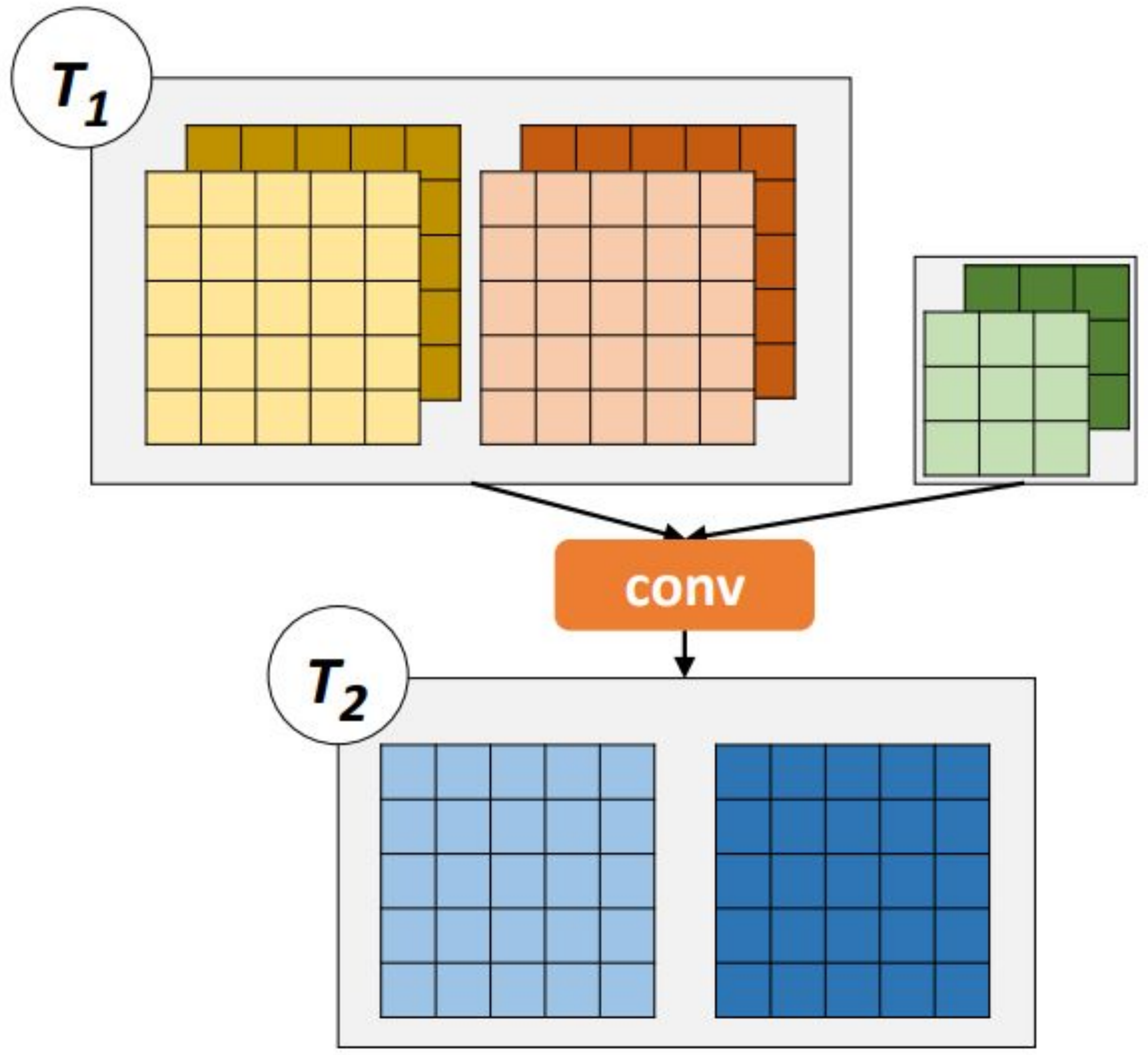
Partially Equivalent Transformations

- Math-equivalent
- Missing some optimization opportunities

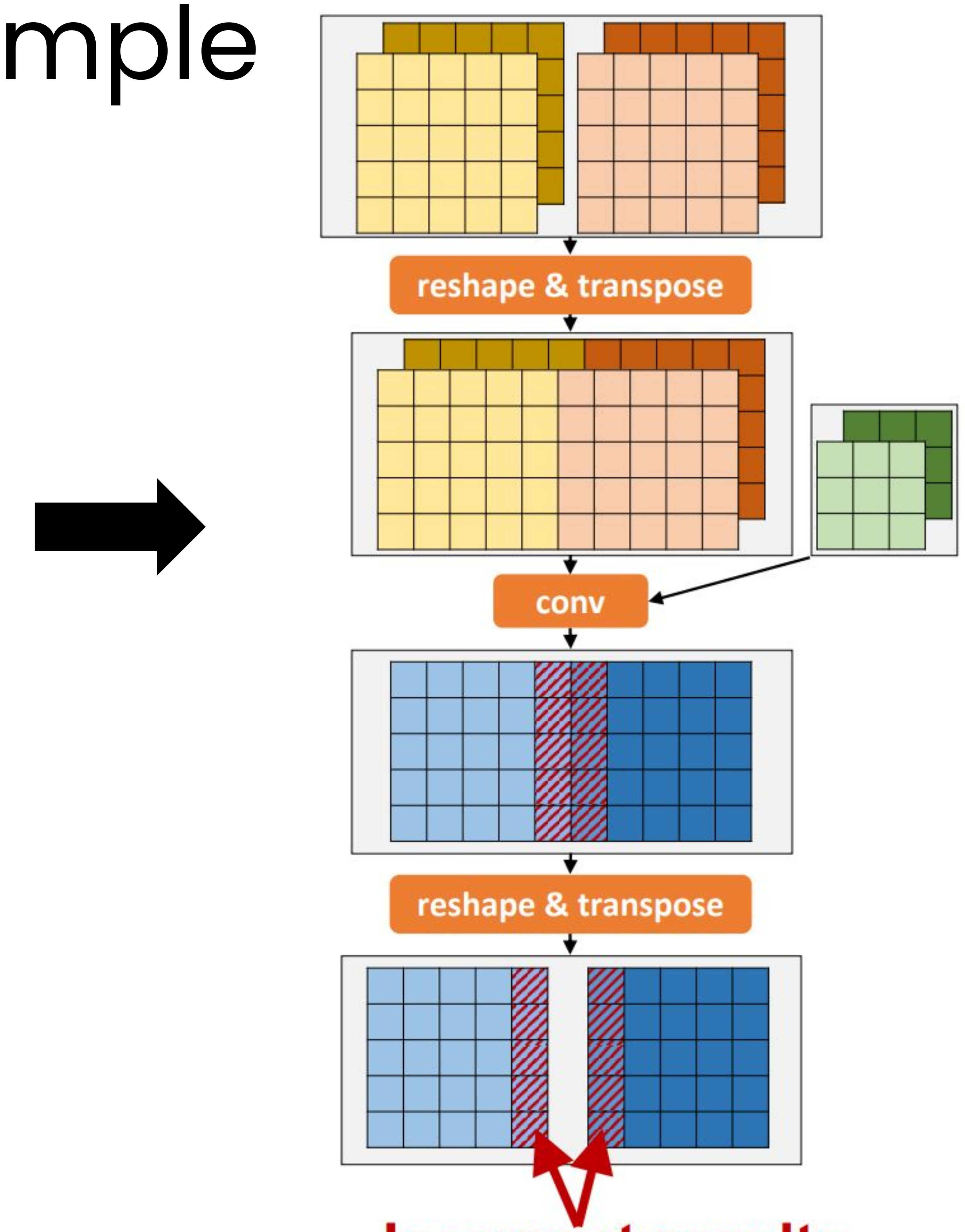
- Better performance
- Not fully equivalent -> accuracy loss

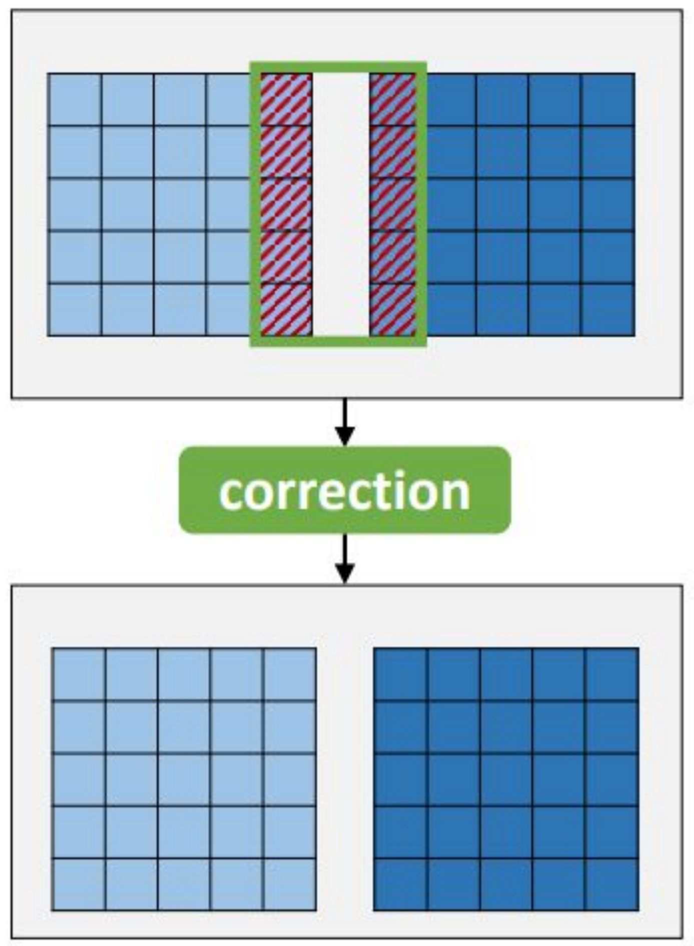
How about: exploit the larger space partially equivalent transformations for performance while still preserve correctness?

Motivating Example



Input Program

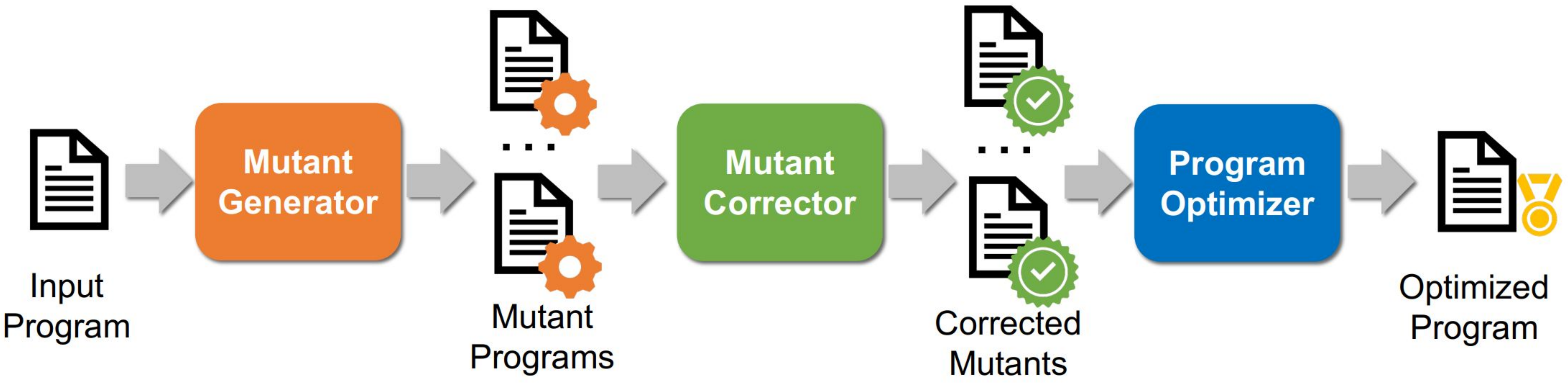




+ Correction

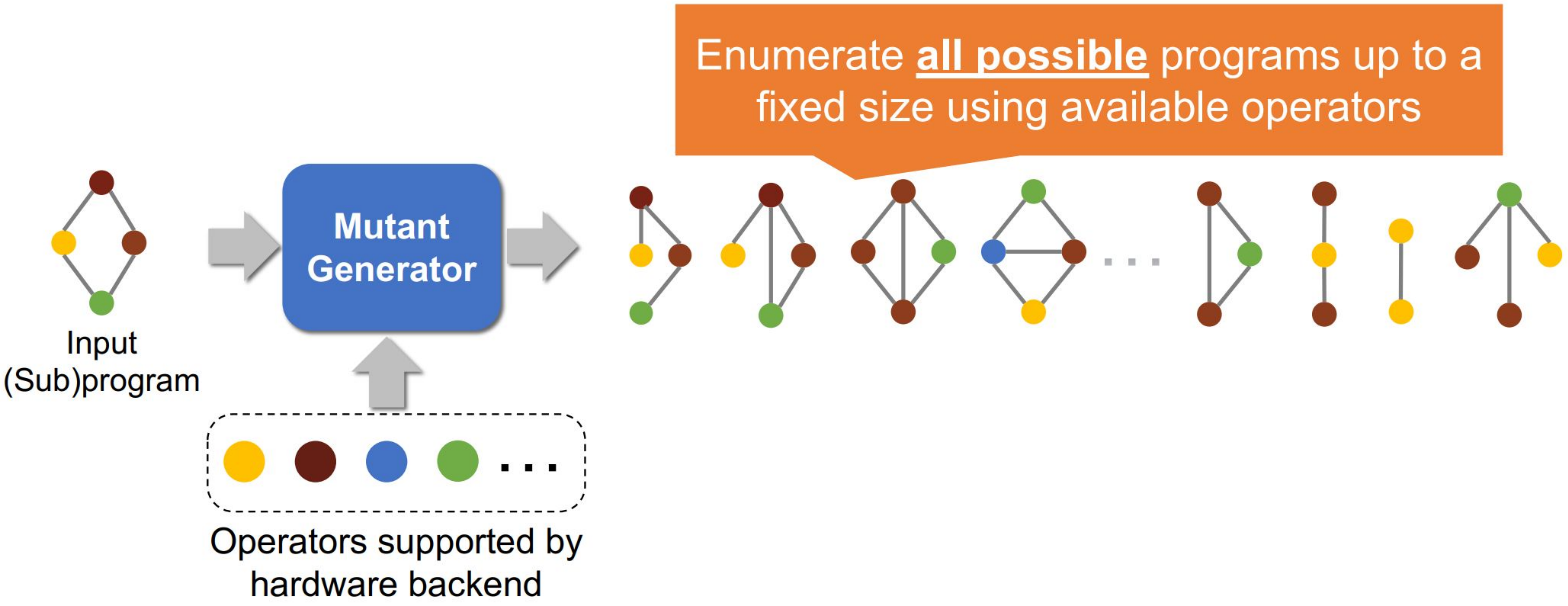
- Partial equivalent transformations + correction yield 1.2x speedup
- Which would otherwise be impossible in fully equivalent transformations

Partially Equivalent Transformations

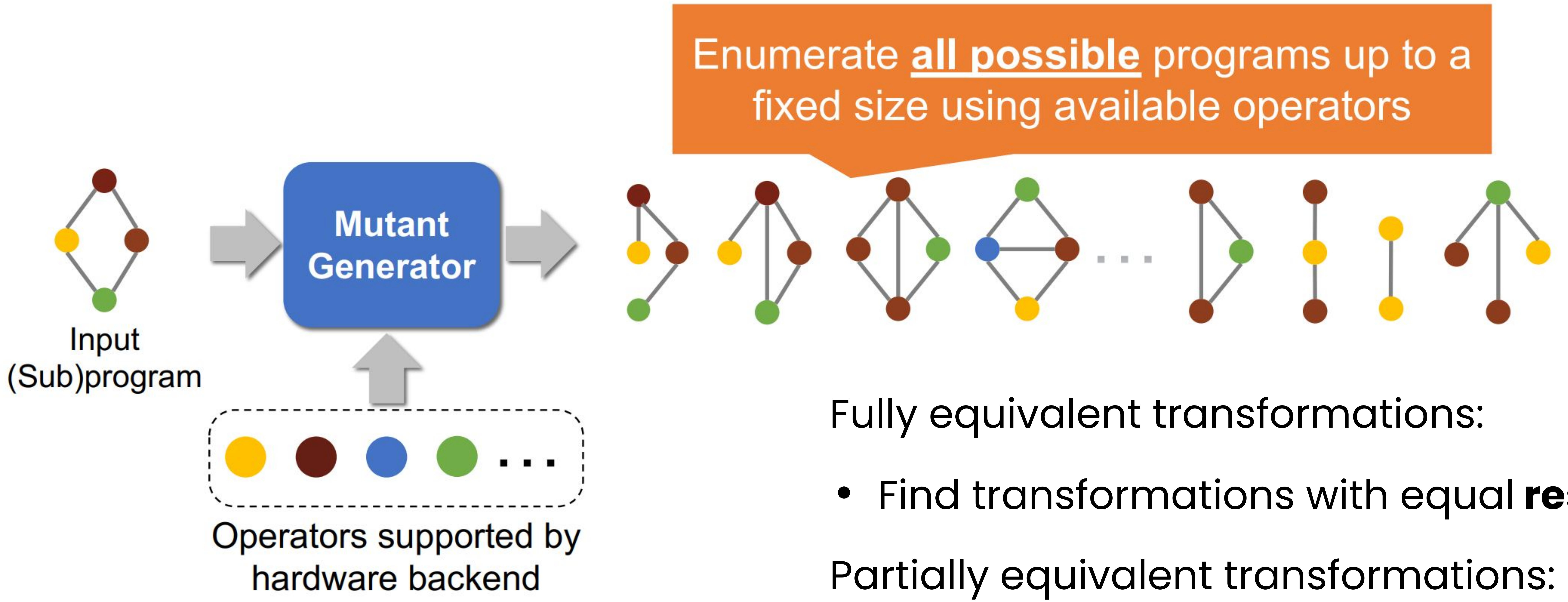


- How to mutate?
- How to correct?

Mutant Generator: Step 1



Mutant Generator: Step 2



Fully equivalent transformations:

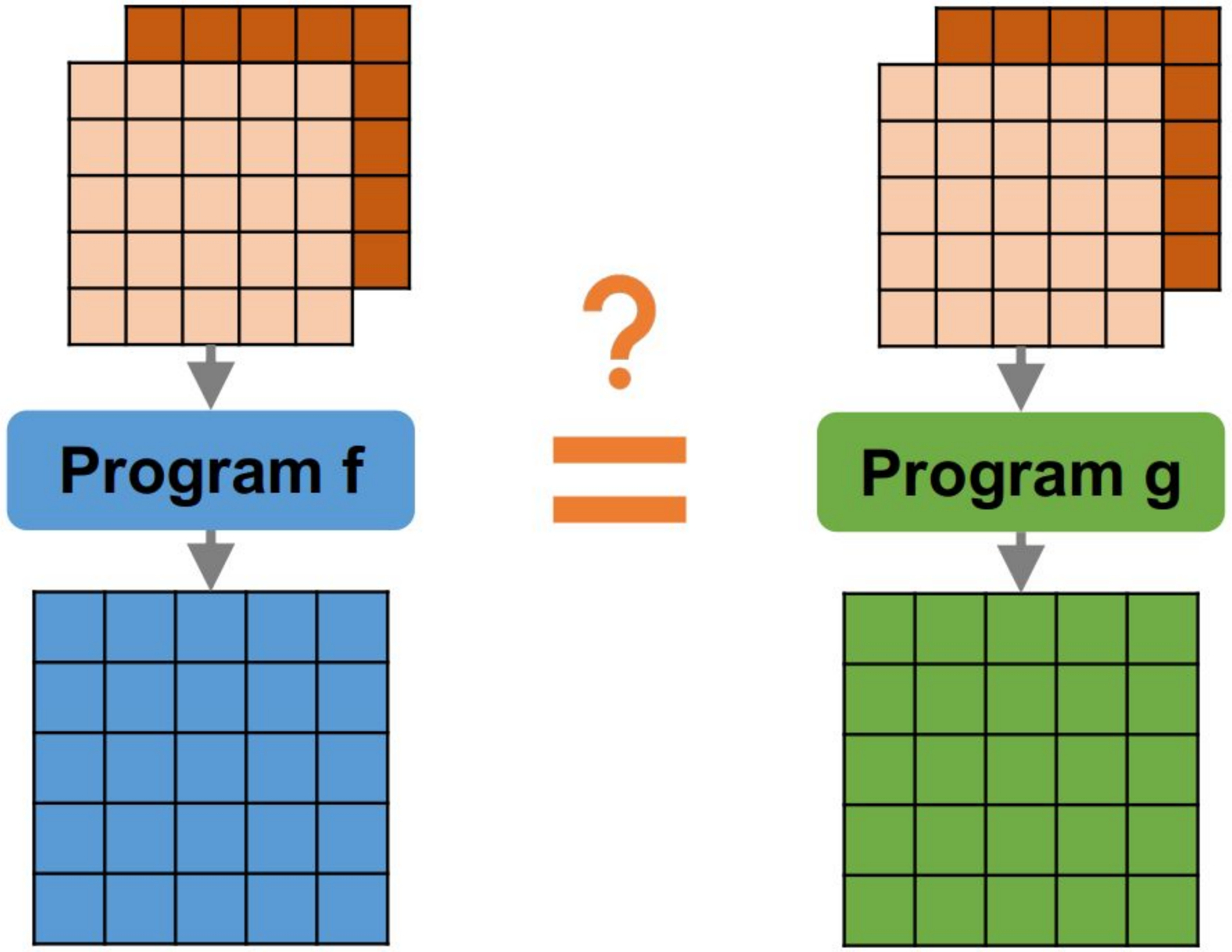
- Find transformations with equal **results**

Partially equivalent transformations:

- Find transformations with equal **shapes**

How to Detect and Correct?

- Which part of the computation is not equivalent?
- How to correct the results?

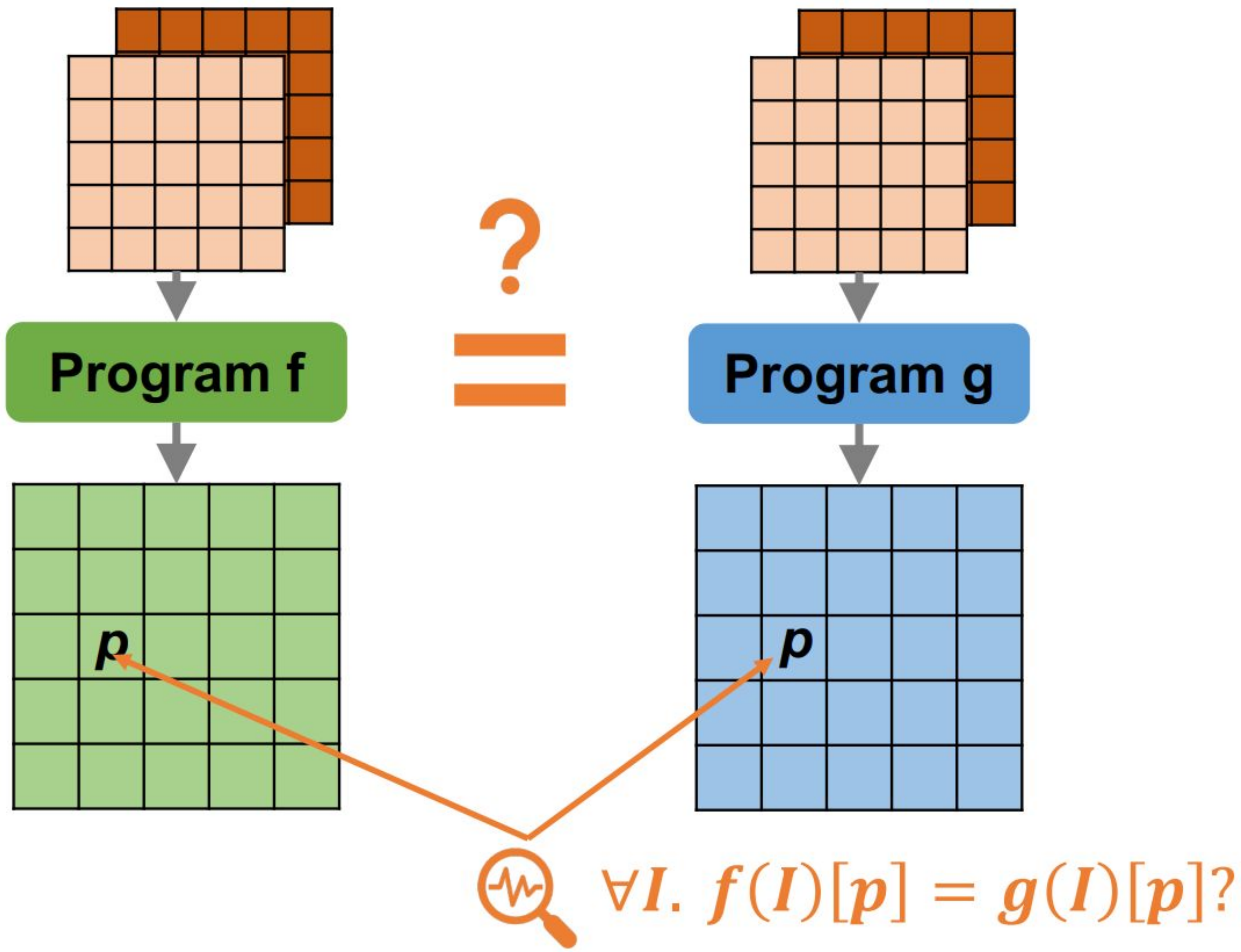


By Enumeration

- For each possible input I
  - For each position p
    - Check if f(I)[p] == g(I)[p]

- Complexity O(m x n):
  - m: possible inputs
  - n: output shape

- How to reduce enumeration effort?



How to reduce n?

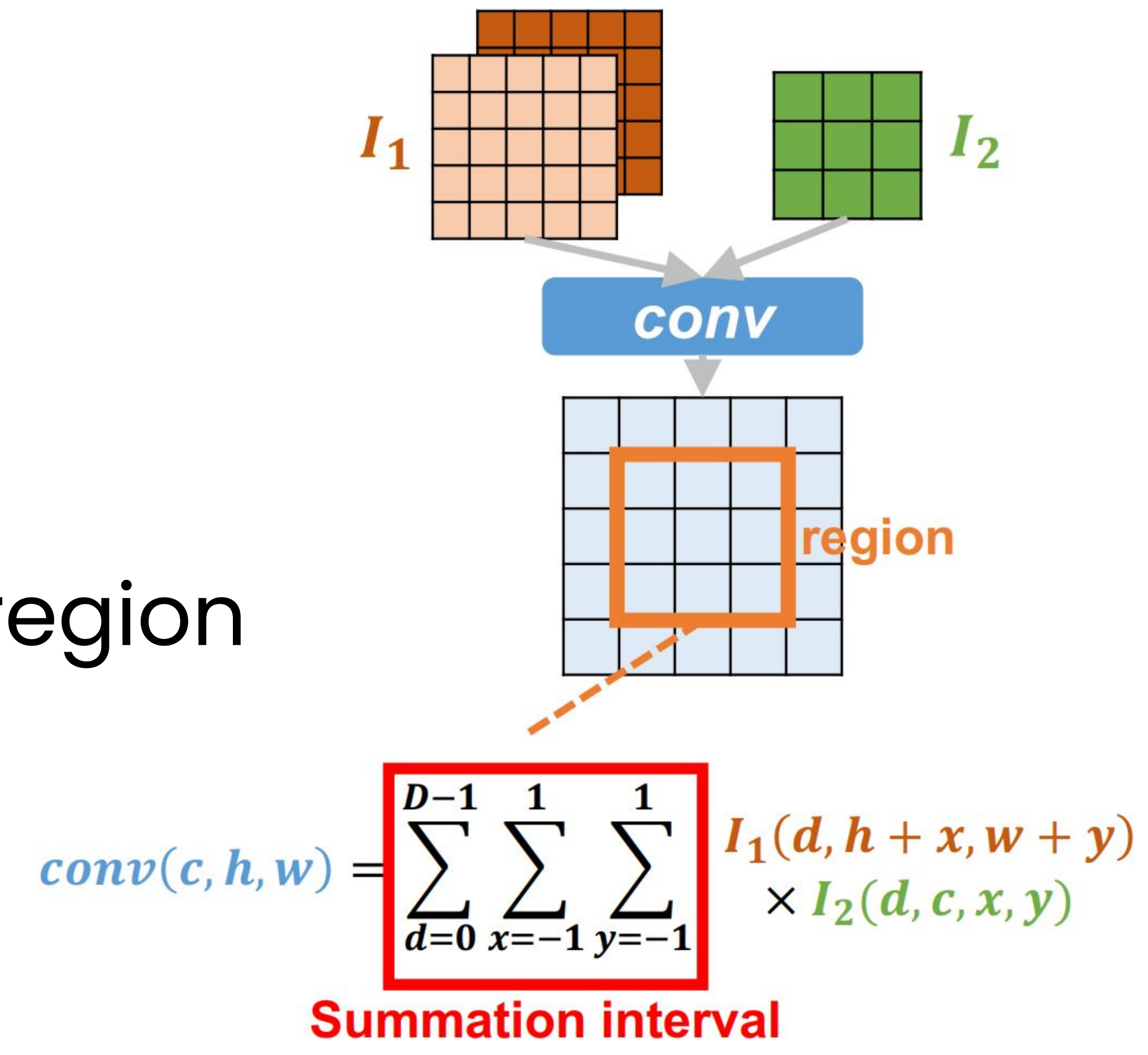
- Can we just check out a few (or even just one) position at f(I)[p] and assert the (in-)correctness?
- Answer: Yes for 80% of the computation

- Reason: Neural nets computation are mostly Multi-Linear
- Define Multi-linear: f is multi-linear if the output is linear to all inputs
  - $f(I_1, \dots, X, \dots, I_n) + f(I_1, \dots, Y, \dots, I_n) = f(I_1, \dots, X + Y, \dots, I_n)$
  - $\alpha f(I_1, \dots, X, \dots, I_n) = f(I_1, \dots, \alpha X, \dots, I_n)$

How to reduce n

- Theorem 1: For two Multi-linear functions f and g, if f=g for O(1) positions in a region, then f=g for all positions in the region
- Implications: only need to examine O(1) positions for each region
  - Reduce O(mn) -> O(m)

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

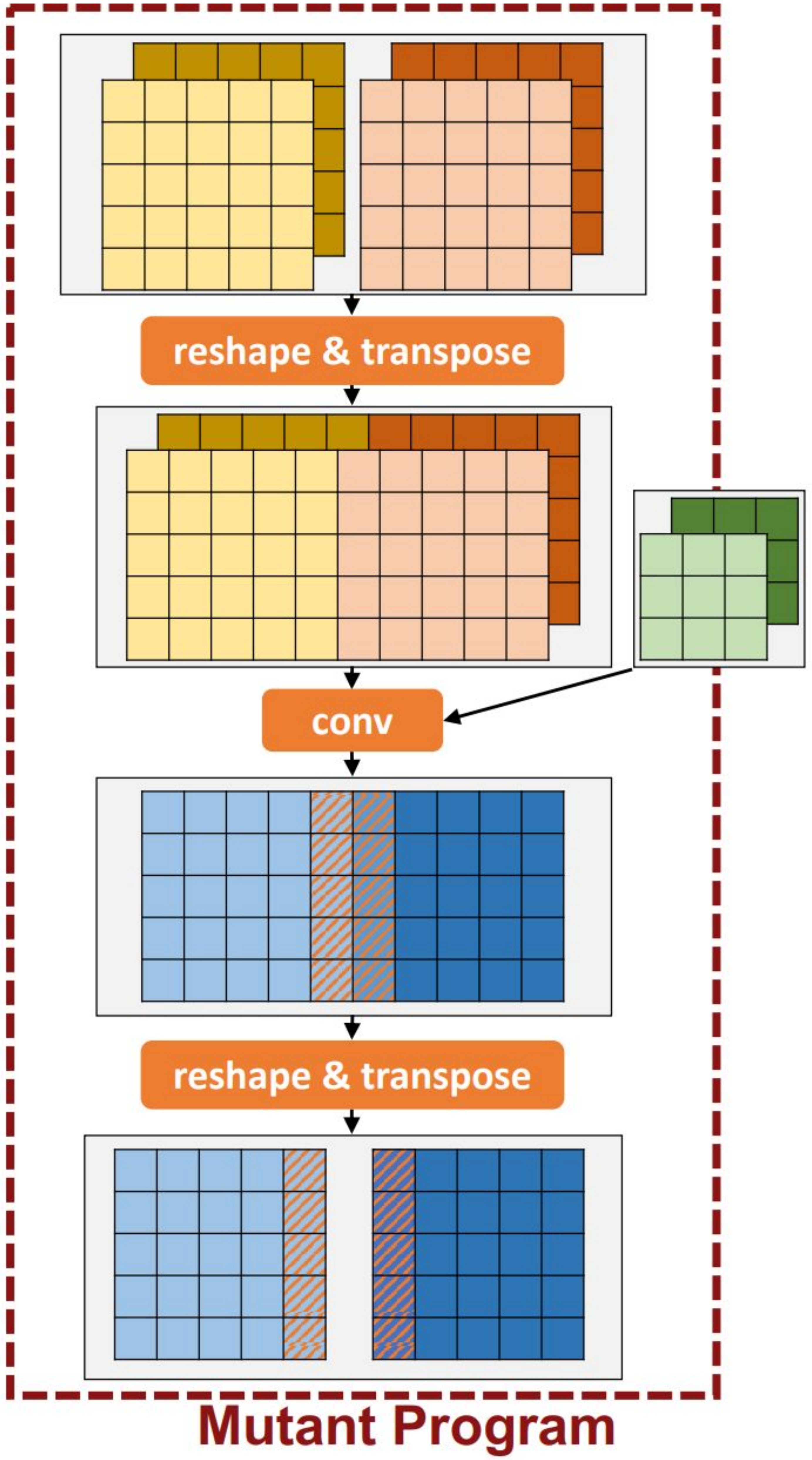


How to reduce m?

- Theorm 2: if $\exists I, f(I)[p] \neq g(I)[p]$, then the probability that f and g give identical results on t random inputs is $\left(\frac{1}{2^{31}}\right)^t$
- Implications: Run t random tests with random input, and if all t passed, it is very unlikely f and g are inequivalent
- O(mn) -> O(m) -> O(t) (t << m)

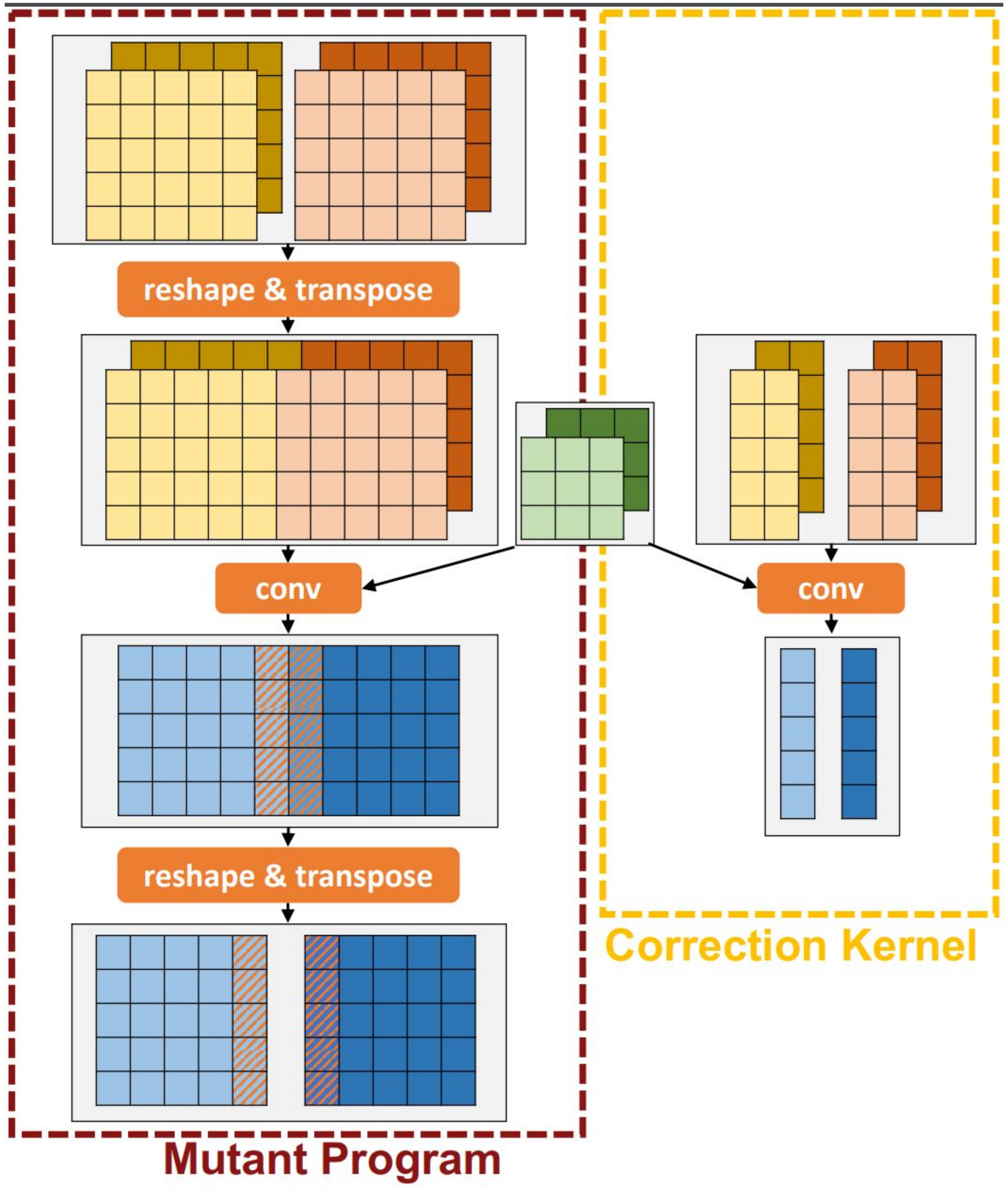
Correct the Mutant

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



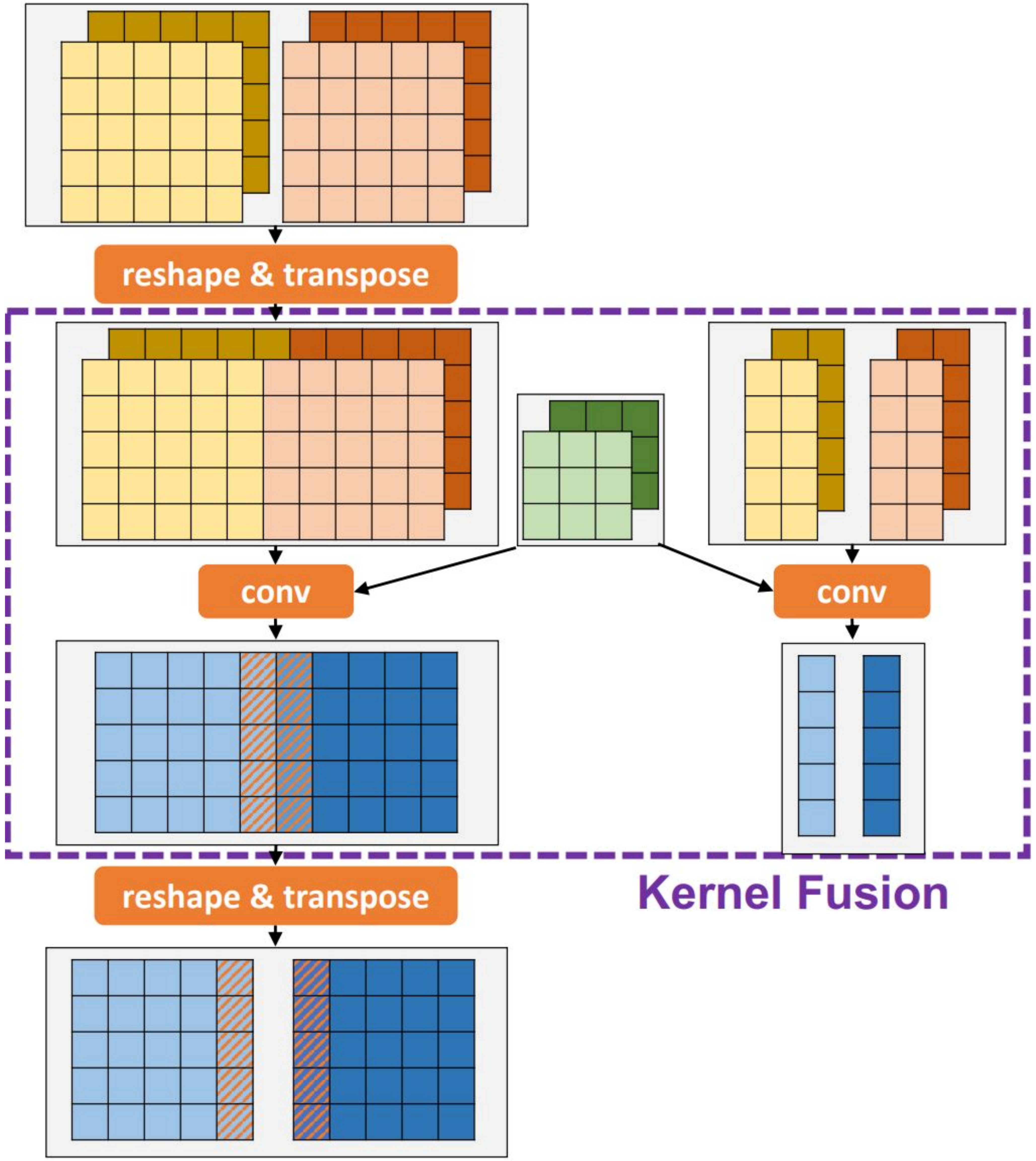
Correct the Mutant

- Goal: quickly and efficiently correcting the outputs of a mutant program
- Step 1: recompute the incorrect outputs using the original program

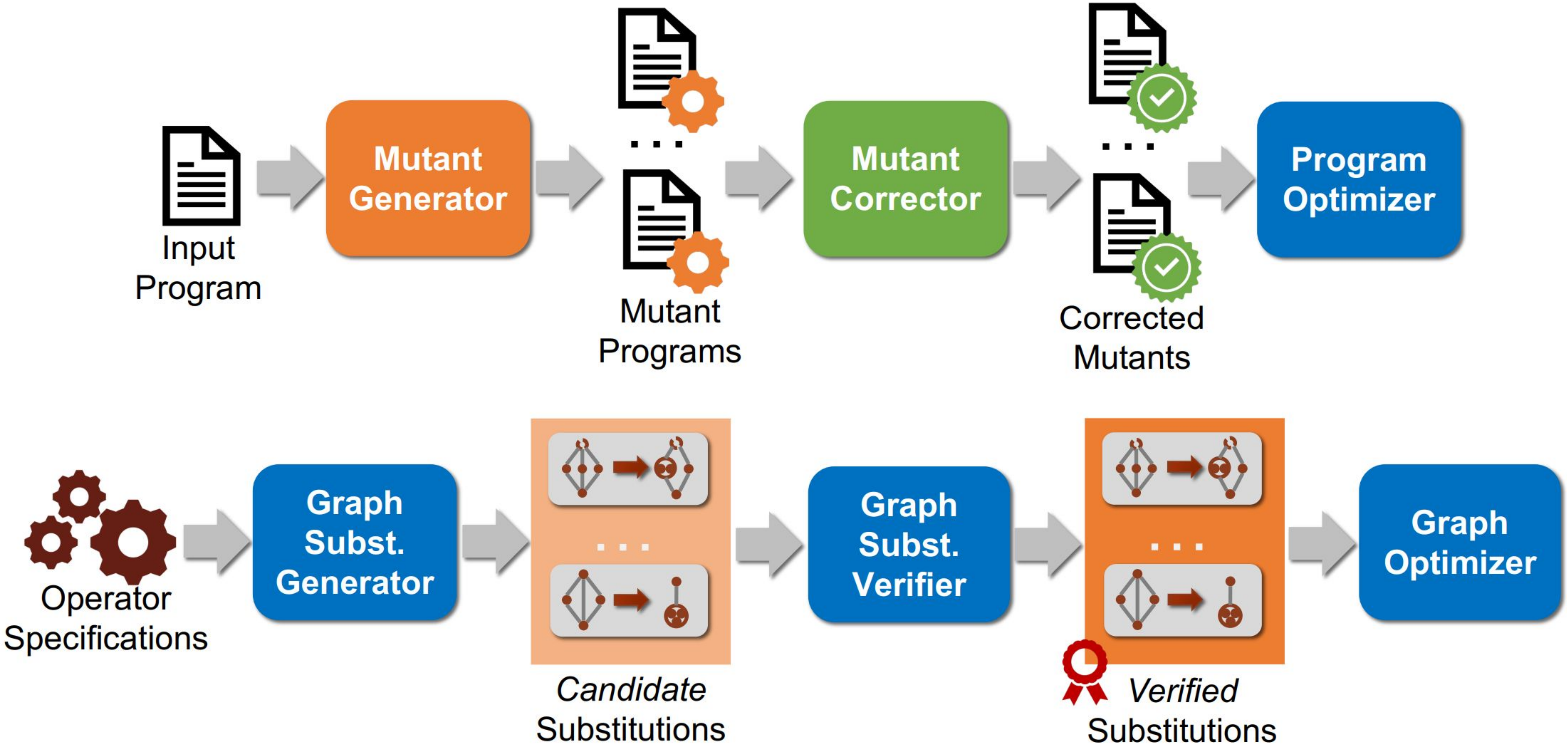


Correct the Mutant

- 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



Recap



Input
Program
Mutant
Generator
Mutant
Programs
Mutant
Corrector
Corrected
Mutants
Program
Optimizer
Operator
Specifications
Graph
Subst.
Generator
Candidate
Substitutions
Graph
Subst.
Verifier
Verified
Substitutions
Graph
Optimizer

Summary & Questions to discuss

- Fully equivalent transformations vs. Partial
  - How to define search space
  - How to prune search space
  - How to verify & correct
  - How to apply to the ML graph optimization

Compilation Process

ML Models

Transformer, ResNet, LSTM ...

Lowering

IR and transformers

Op-level optimizations

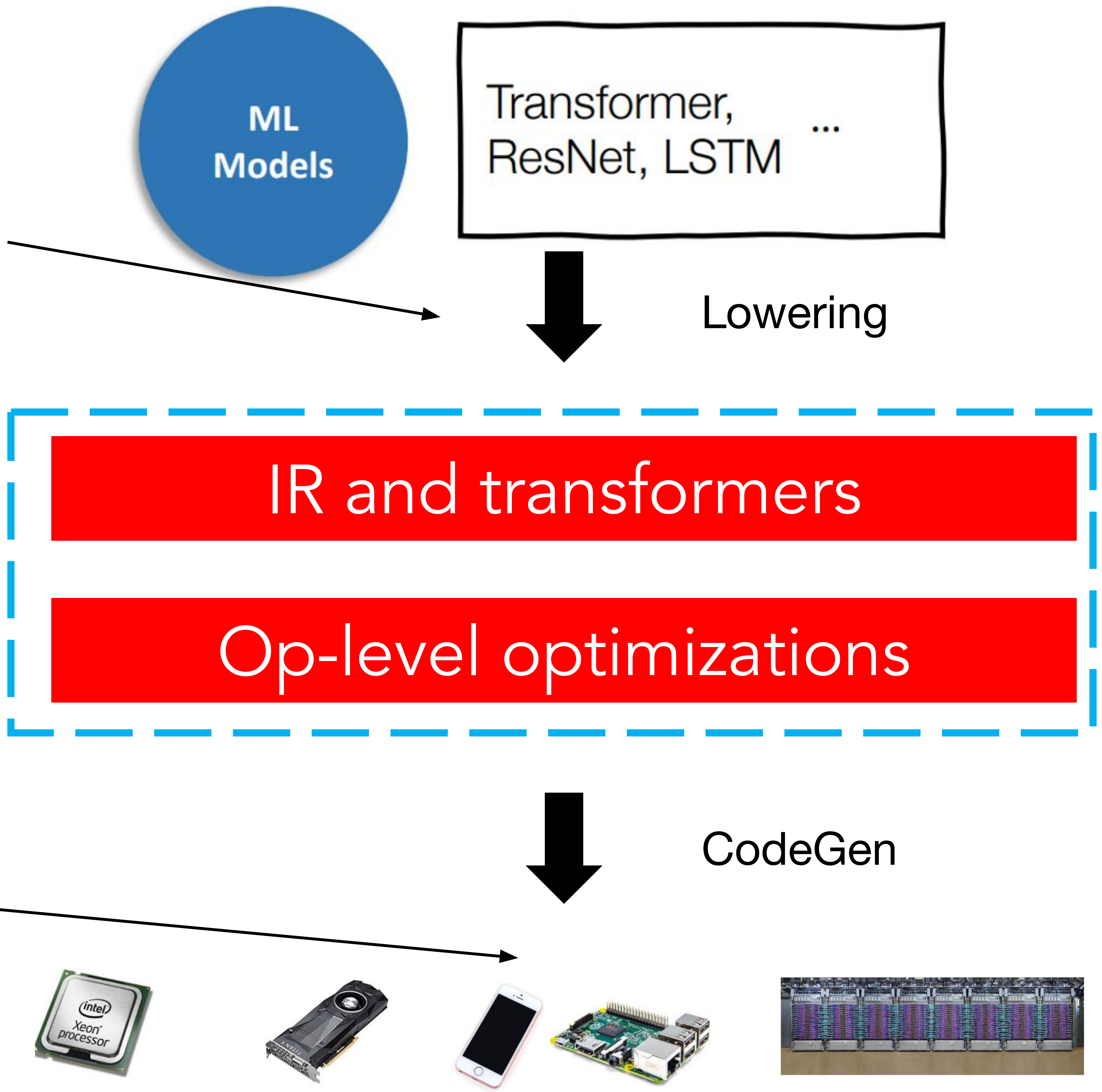
CodeGen

Ideally, they should be co-optimized? Guest lectures

Topics will be covered later by Guests

How to lower user program to IRs?

- The program is in PY
- Control flows
- Dynamism



How to co-optimize to maximize the potential

Emerging hardware: TPUs/LPUs

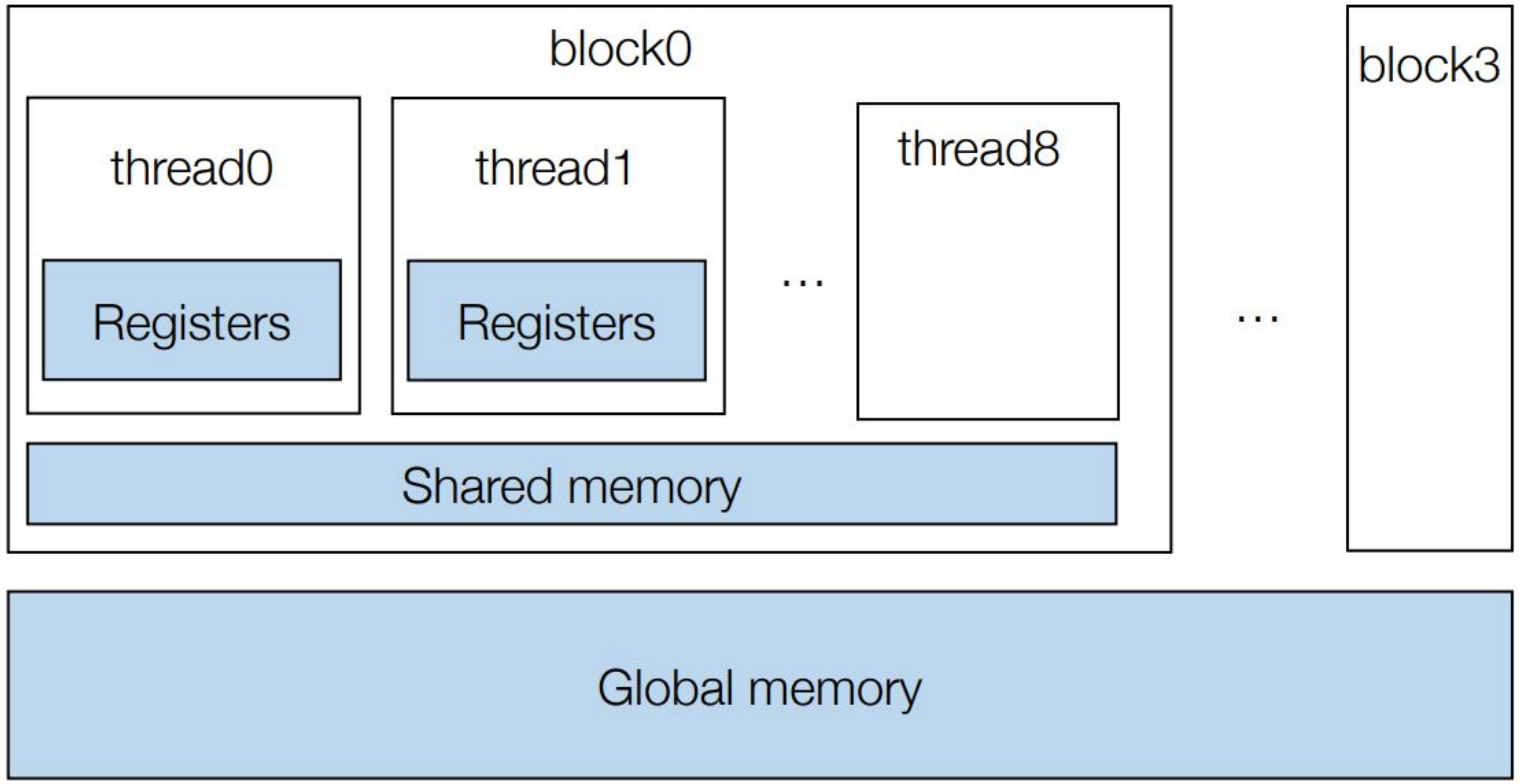
Agenda on this part

- ML Compilation Overview
  - Operator compilation
  - Graph optimization
- **Memory Optimization**
  - Activation checkpointing
  - Quantization and Mixed precision
- Two Guest Talks covering details in compilation, JIT, graph fusion, and beyond:
  - Meta PyTorch lead developer: Jason Ansel

Memory Optimization

- Checkpointing and rematerialization
- CPU Swapping
- Quantization and Mixed precision

Recap: Memory Hierarchy



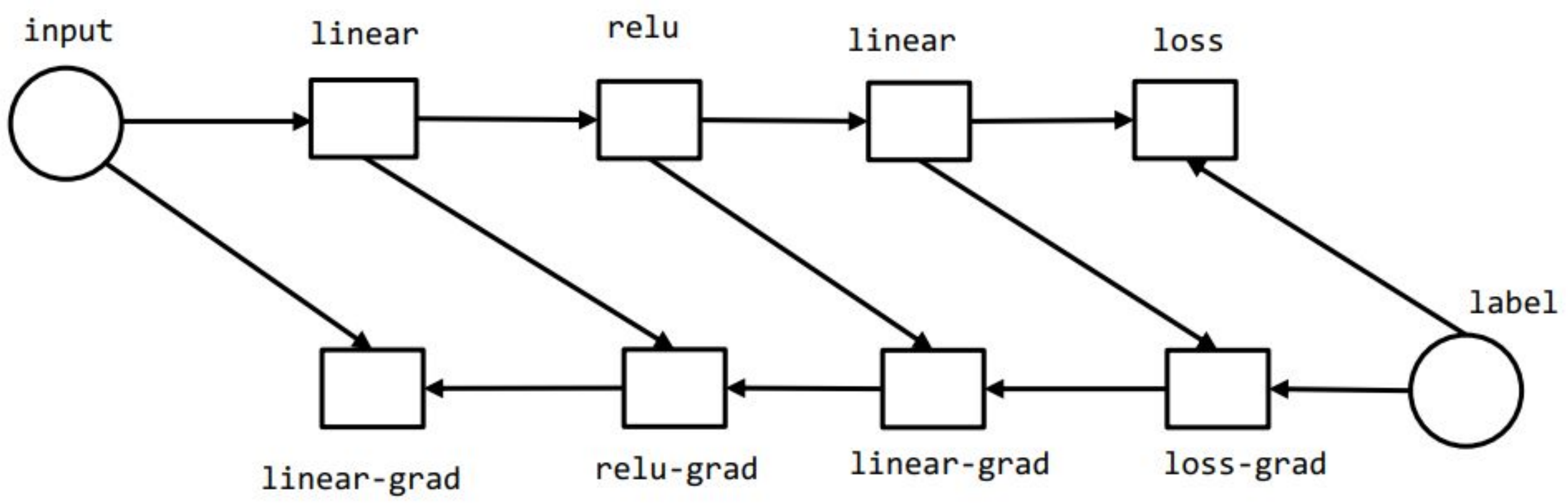
Shared memory: 64 KB per core

GPU memory(Global memory):

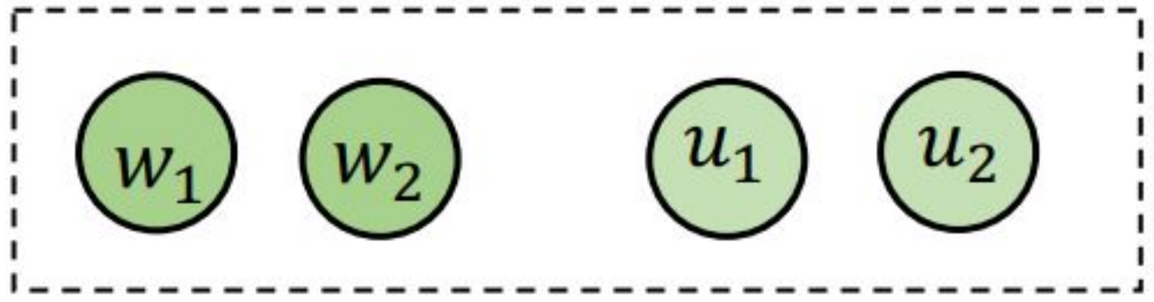
RTX3080 10GB
RTX3090 24GB
A100 40/80 GB

Source of Memory Consumption

A simplified view of a typical computational graph for training, weights are omitted and implied in the grad steps.



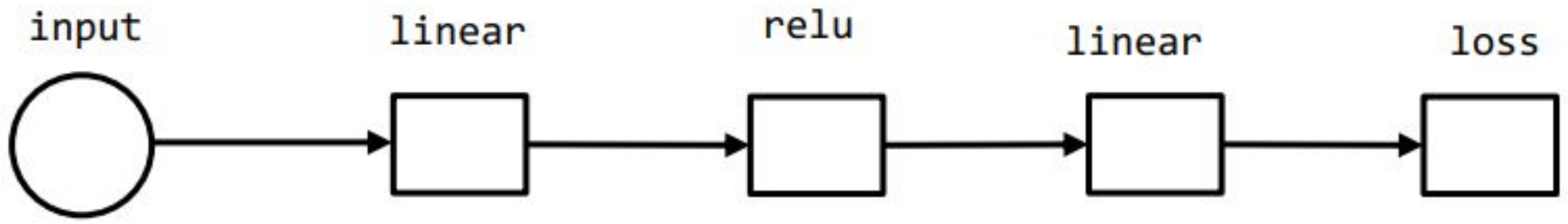
Optimizer states



Sources of memory consumption

- Model weights
- Optimizer states
- Intermediate activation values

At Inference



We only need O(1) memory for computing the final output of a N layer deep network by cycling through two buffers