https://hao-ai-lab.github.io/cse234-w25/

CSE 234: Data Systems for Machine Learning
Winter 2025

LLMSys

Optimizations and Parallelization

MLSys Basics

Today’s Learning Goal

- Case study: Matmul on GPU
- Operator Compilation
- **High-level DSL for CUDA: Triton**
- Graph Optimization

Triton Programming Model

- Users define tensors in SARM, and modify them using torch-like primitives

Embedded in Python

Kernels are defined in Python using triton.jit

Pointer arithmetics

Users construct tensors of pointers and (de)reference them elementwise

Shape Constraints

Must have power-of-two number of elements along each dimension

Example: elementwise add v1 (z = x + y)

- Triton kernel will be mapped to a single block (SM) of threads
- Users will be responsible for mapping to multiple blocks

```
import triton.language as tl
Import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
  # same as torch.arrange
  offsets = tl.arange(0, 1024)
  # create 1024 pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load 1024 elements of X, Y, Z
  x = tl.load(x_ptrs)
  y = tl.load(y_ptrs)
  # do computations
  z = x + y
  # write-back 1024 elements of X, Y, Z
  tl.store(z_ptrs, z)


N = 1024
x = torch.randn(N, device='cuda’)
y = torch.randn(N, device='cuda’)
z = torch.randn(N, device='cuda’)
grid = (1, )
_add[grid](z, x, y, N)
```

Example: elementwise add v2 (z = x + y)

Use multiple blocks

- Index the block and apply offset
- Adds bound check

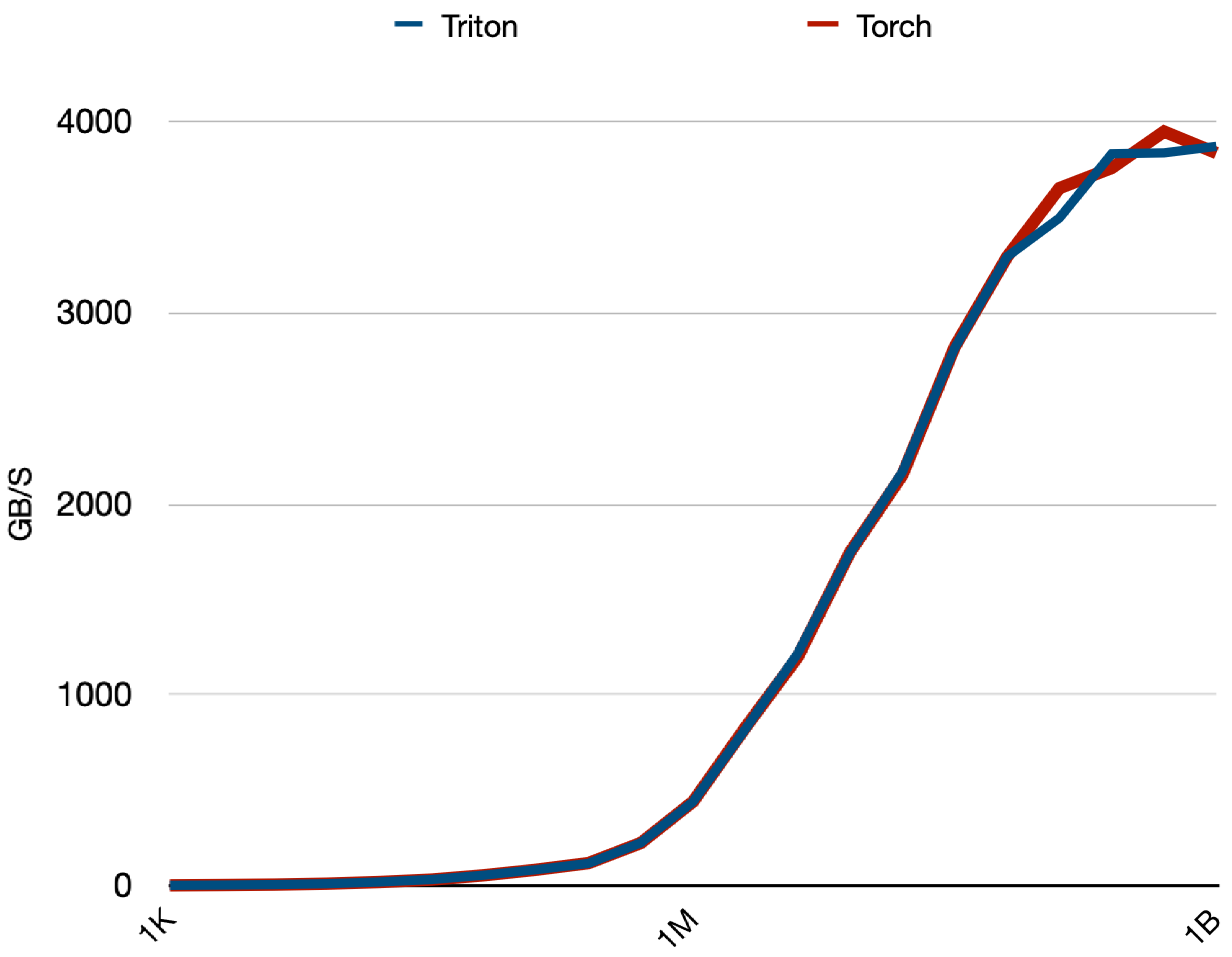
```
import triton.language as tl
Import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
  # same as torch.arrange
  offsets = tl.arange(0, 1024)
  offsets += tl.program_id(0)*1024
  # create 1024 pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load 1024 elements of X, Y, Z
  x = tl.load(x_ptrs, mask=offset<N)
  y = tl.load(y_ptrs, mask=offset<N)
  # do computations
  z = x + y
  # write-back 1024 elements of X, Y, Z
  tl.store(z_ptrs, z)


N = 192311
x = torch.randn(N, device='cuda’)
y = torch.randn(N, device='cuda’)
z = torch.randn(N, device='cuda’)
grid = (triton.cdiv(N, 1024), )
_add[grid](z, x, y, N)
```

Example: elementwise add v2 (z = x + y)

- Parametrize block size
- Why we do this?
  - Triton will do tiling for users
  - Avoid manipulating loops

```
import triton.language as tl
Import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N, BLOCK: tl.constexpr):
  # same as torch.arrange
  offsets = tl.arange(0, BLOCK)
  offsets += tl.program_id(0)*BLOCK
  # create 1024 pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load 1024 elements of X, Y, Z
  x = tl.load(x_ptrs, mask=offset<N)
  y = tl.load(y_ptrs, mask=offset<N)
  # do computations
  z = x + y
  # write-back 1024 elements of X, Y, Z
  tl.store(z_ptrs, z)


N = 192311
x = torch.randn(N, device='cuda’)
y = torch.randn(N, device='cuda’)
z = torch.randn(N, device='cuda’)
grid = lambda args: (triton.cdiv(N, args[‘BLOCK’]), )
_add[grid](z, x, y, N)
```

Elementwise Add Performance

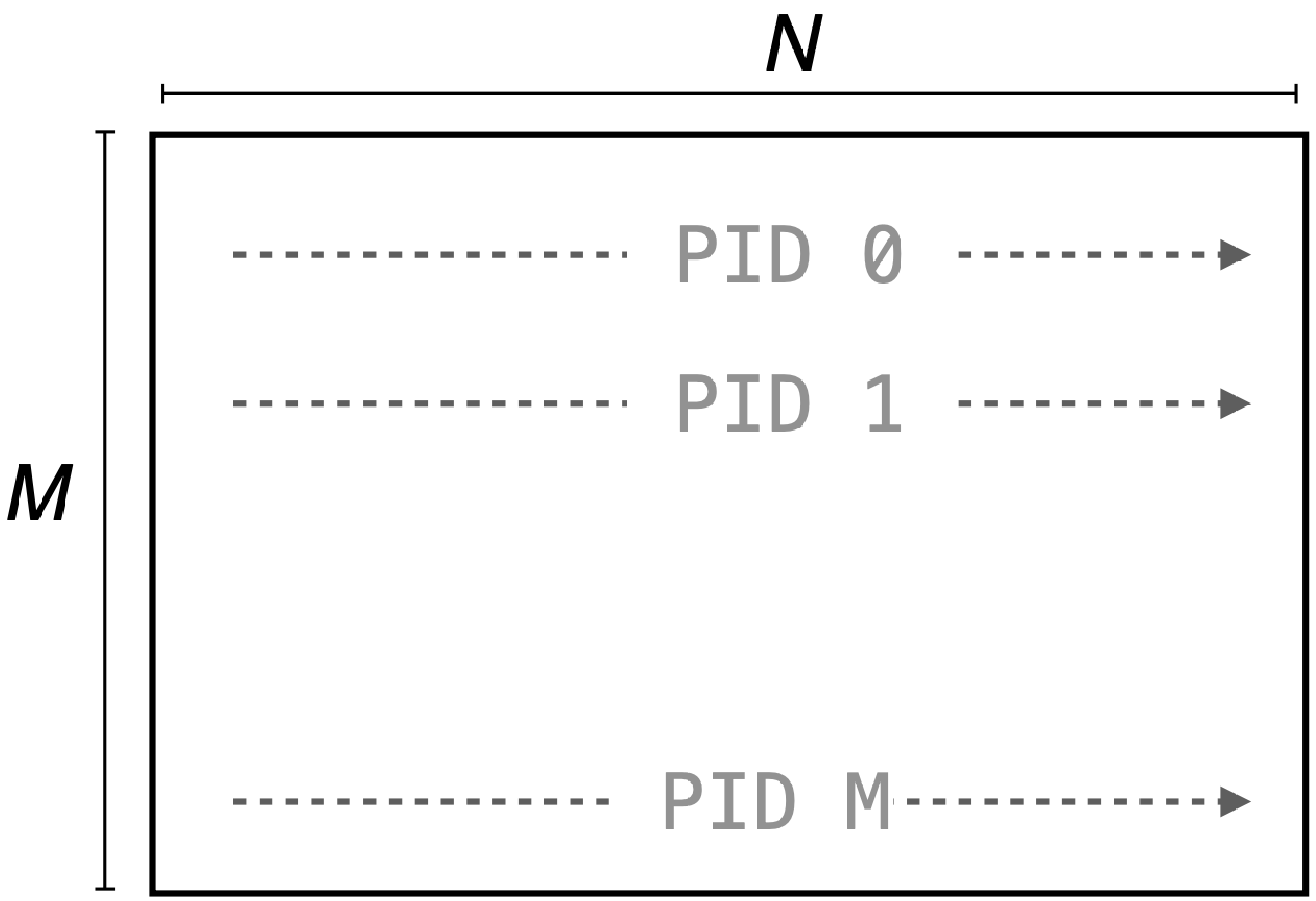


Another Example: Softmax

$$y_i = softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum e^{x_d}}$$

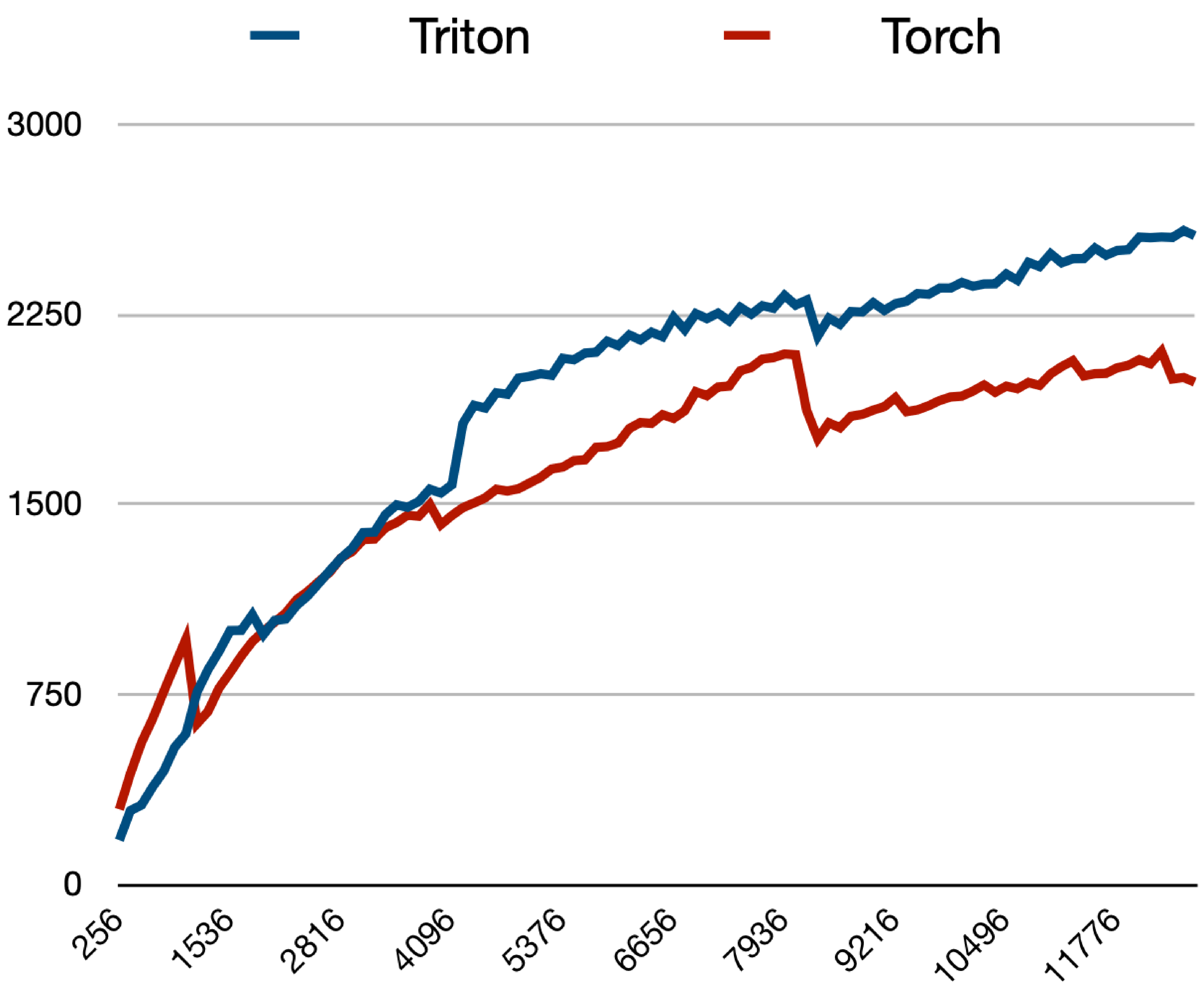
- How did you implement this in PA1?
  - Think about the potential overhead when compose softmax from primitives
- What if implementing an end-to-end softmax kernel
  - Think about the complexity of implementing in CUDA

Triton Example: softmax

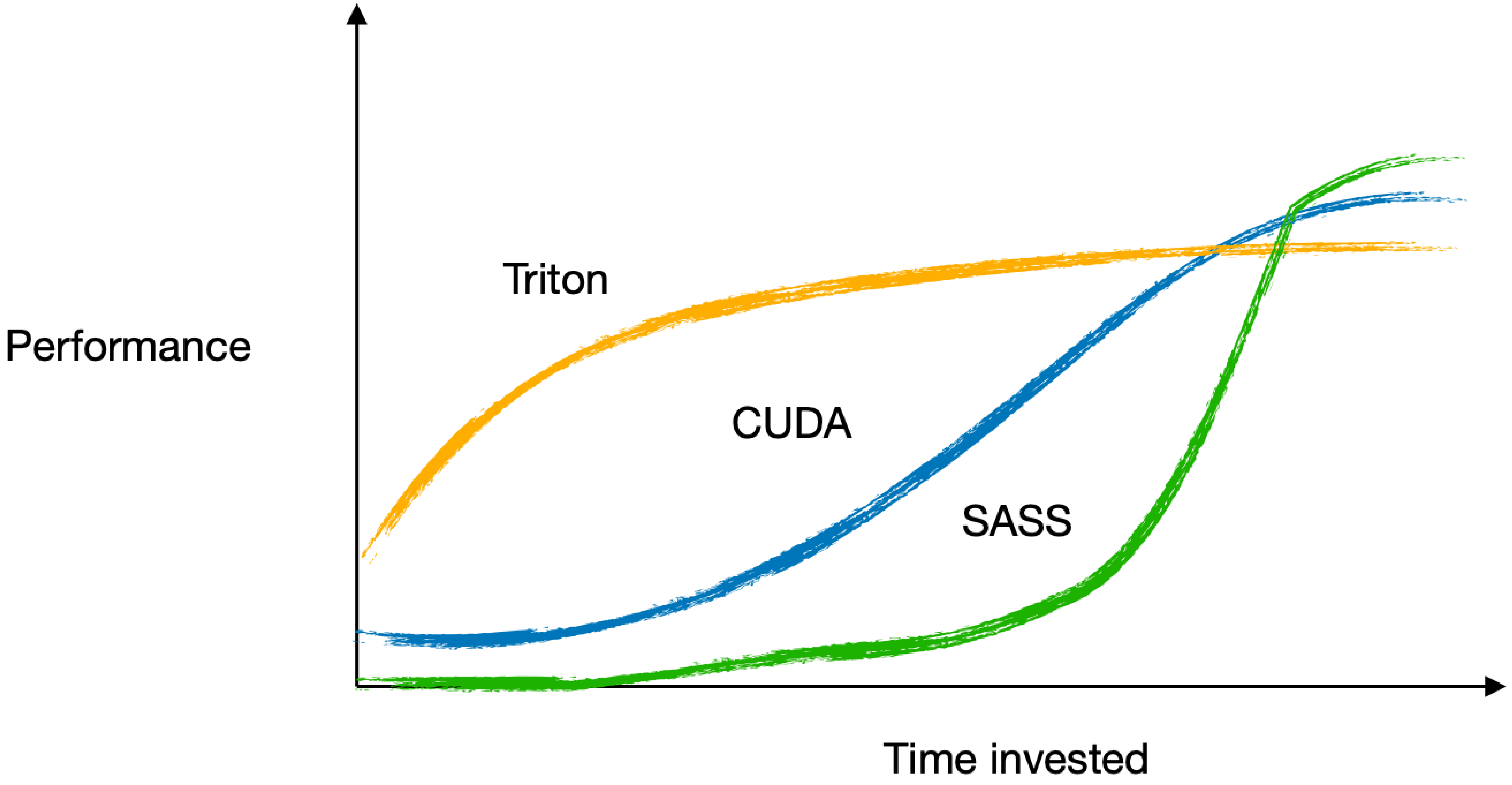


```
import triton.language as tl
Import triton

@triton.jit
def _softmax(z_ptr, x_ptr, stride, N, BLOCK: tl.constexpr):
  # Each program instance normalizes a row
  row = tl.program_id(0)
  cols = tl.arange(0, BLOCK)

  # Load a row of row-major X to SRAM
  x_ptrs = x_ptr + row*stride + cols
  x = tl.load(x_ptrs, mask = cols < N, other = float(‘-inf’))

  # Normalization in SRAM, in FP32
  x = x.to(tl.float32)
  x = x - tl.max(x, axis=0)
  num = tl.exp(x)
  den = tl.sum(num, axis=0)
  z = num / den;
  # Write-back to HBM
  tl.store(z_ptr + row*stride + cols, z, mask = cols < N)
```

Performance



Revisit Triton's Pitch

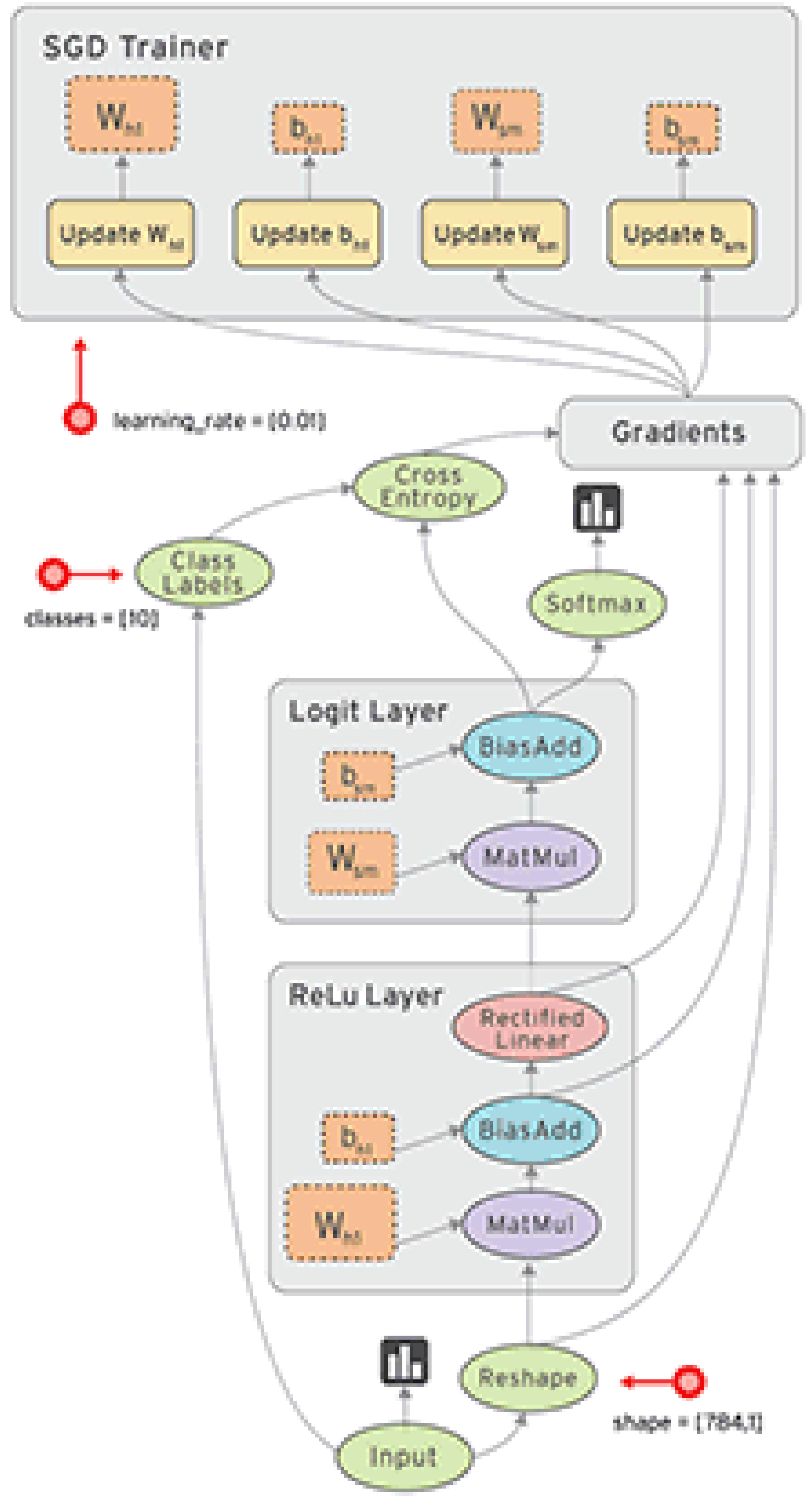


Operator Optimization: Wrapping Up

- Goal: to make individual operator run fast on diverse devices

1. General ways: vectorization, data layout, etc.
2. Matmul-specific: tiling to use fast memory
3. Parallelization SIMD using accelerators
4. Handcrafted operator kernels vs. automatically compile code
5. Triton to find the sweet spot

Wrapping Up Operator Optimization



Dataflow Graph

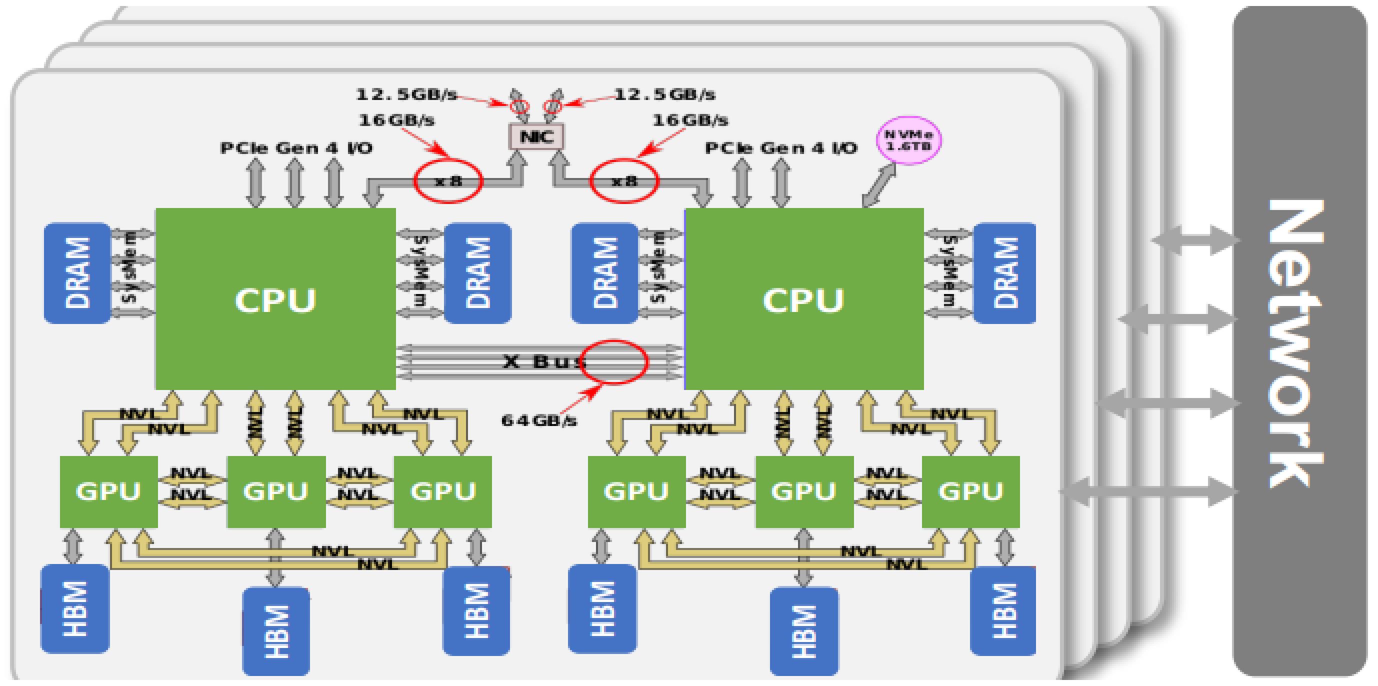
Autodiff

Graph Optimization

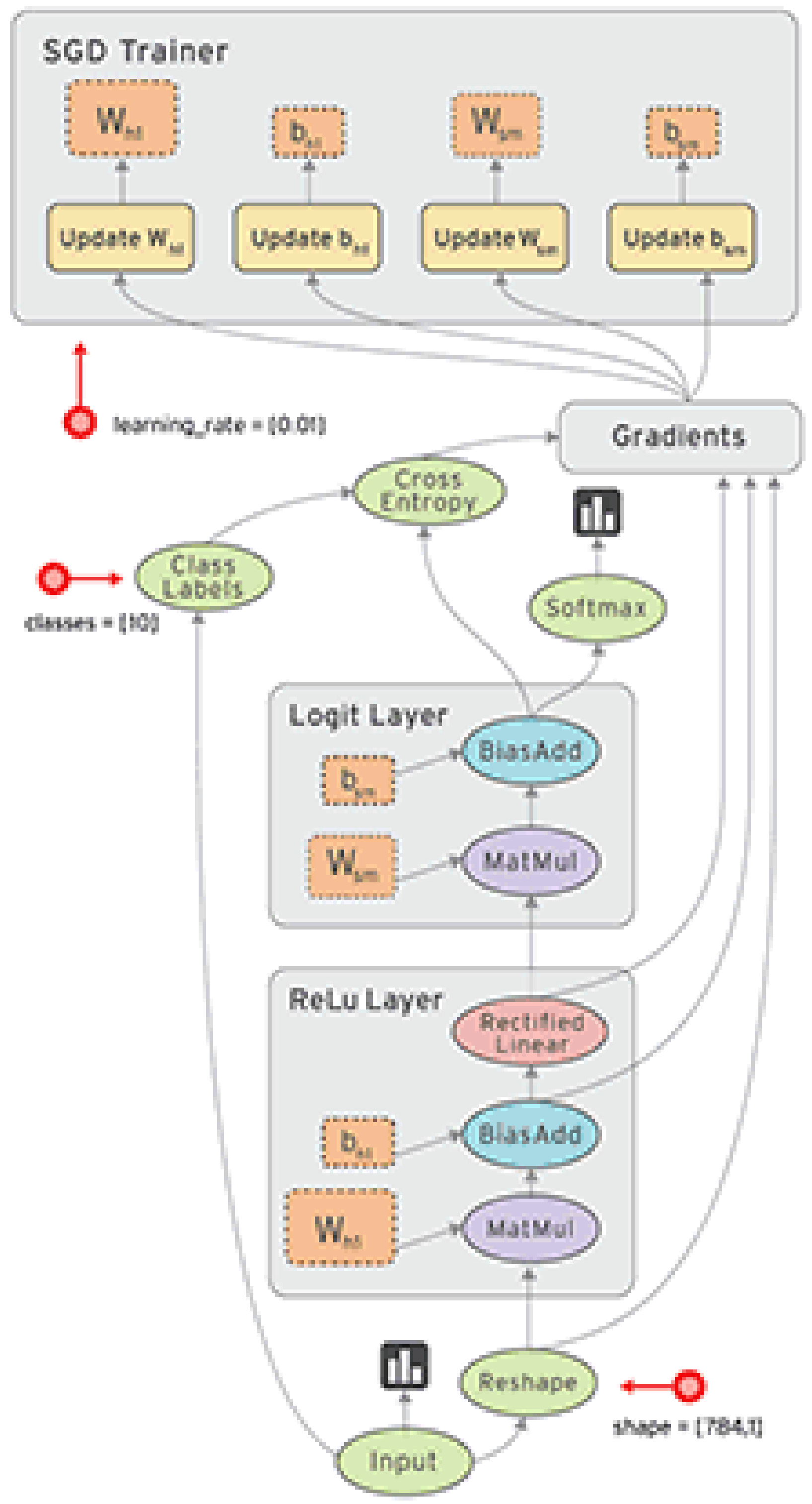
Parallelization

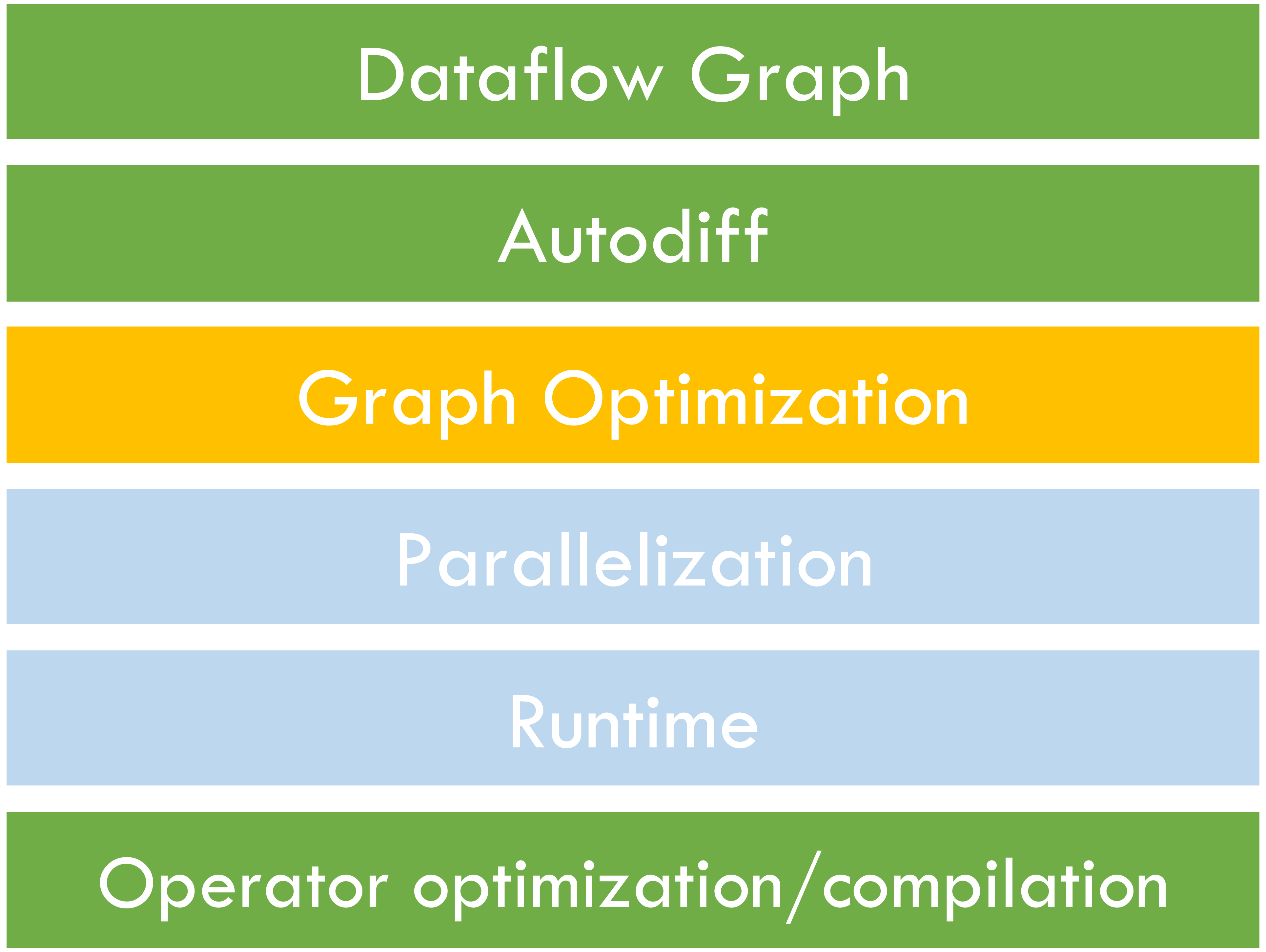
Runtime

Operator optimization/compilation



Next: Graph Optimization





Dataflow Graph

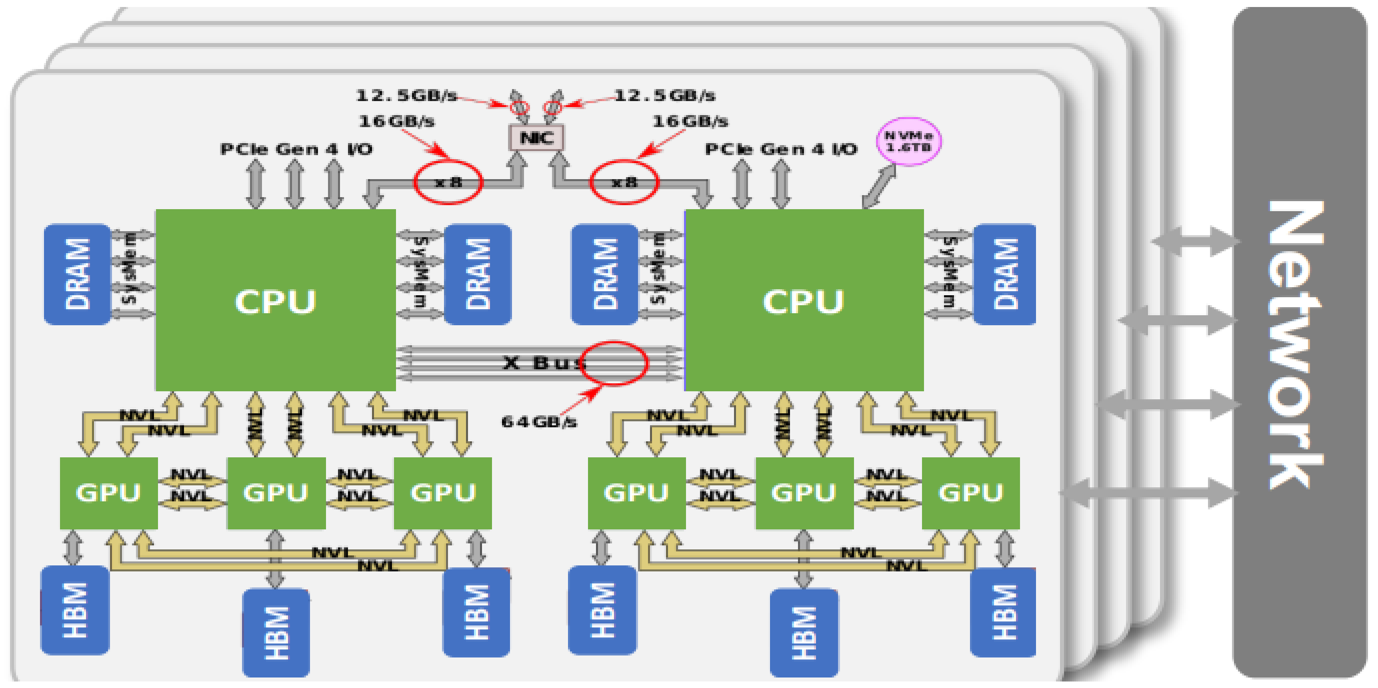
Autodiff

Graph Optimization

Parallelization

Runtime

Operator optimization/compilation

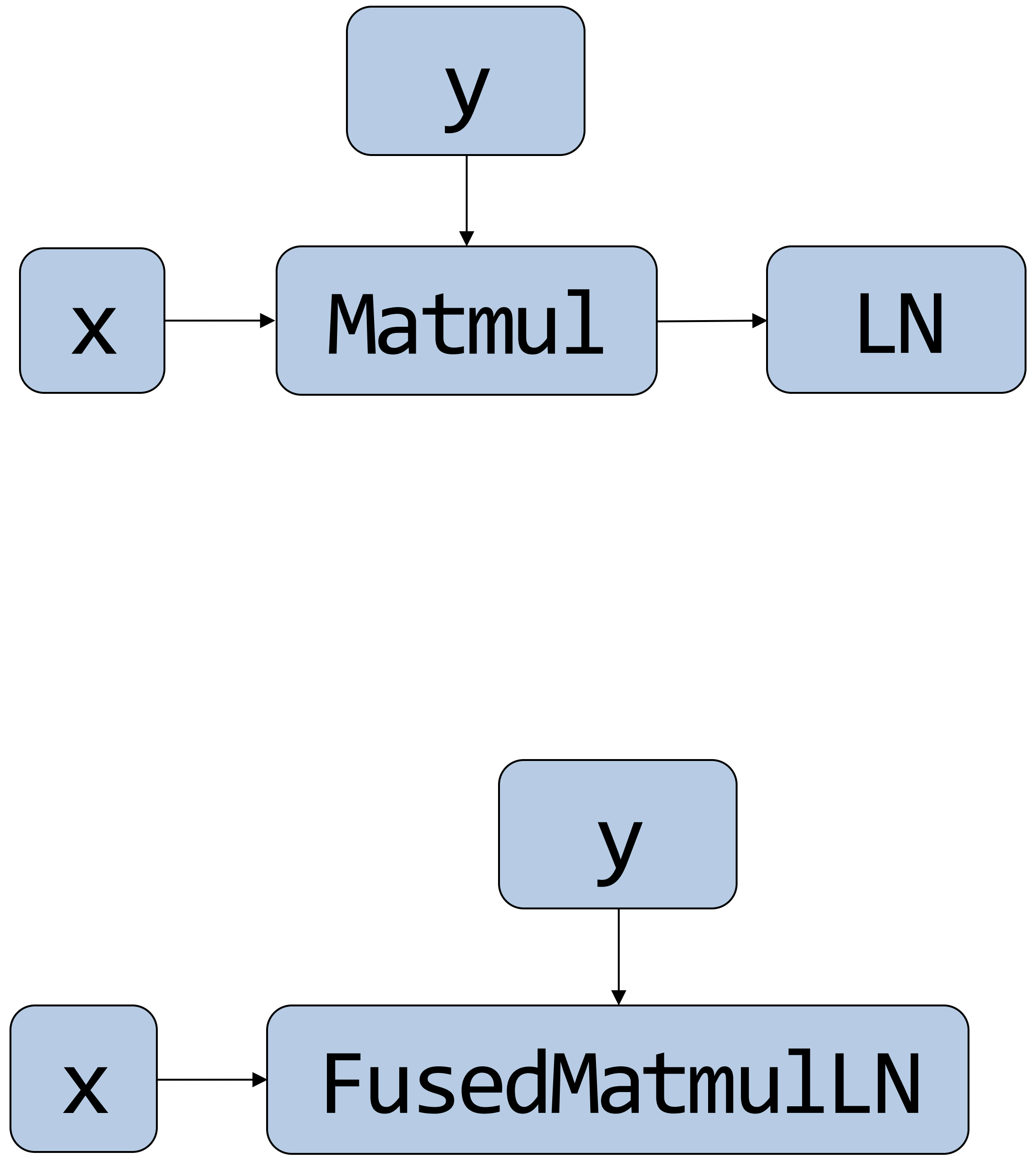


Recall Our Goal

- Goal: Rewrite the original Graph G to G';
  - G' runs faster than G
  - G' outputs equivalent results

- Straightforward solution: template
  - Human experts write (sub-)graph transformation templates
    - Guarantee correctness and performance gain
  - Run pattern matching over dataflow graph and replace

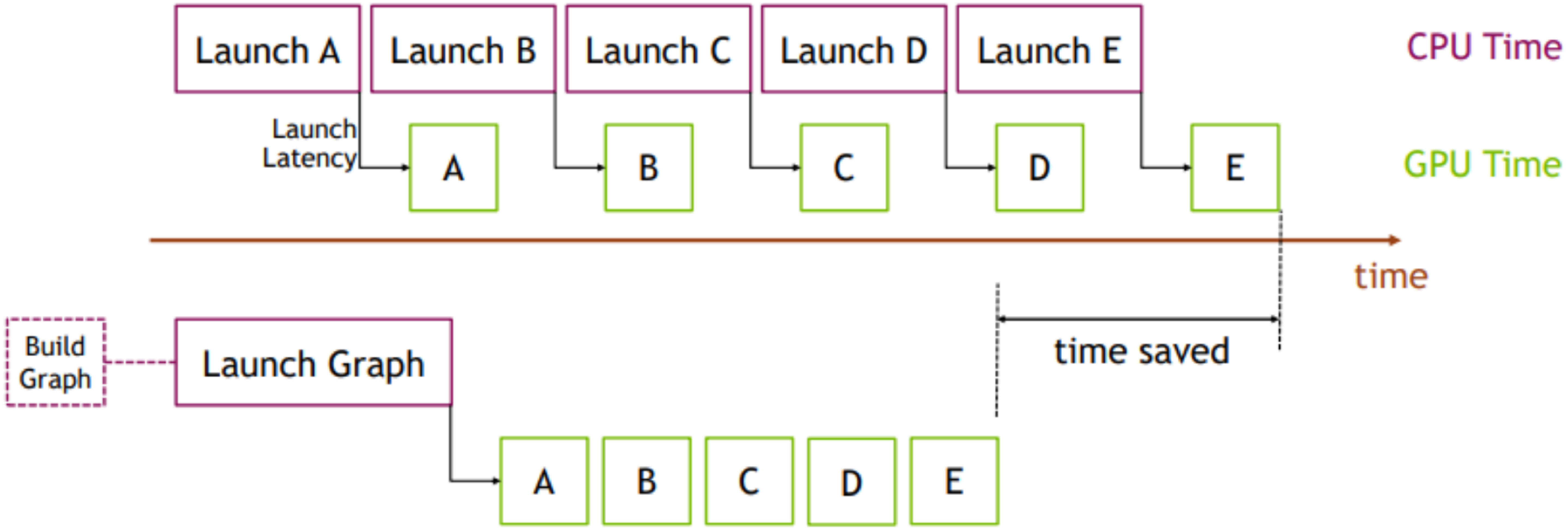
Graph Optimization Templates: Fusion



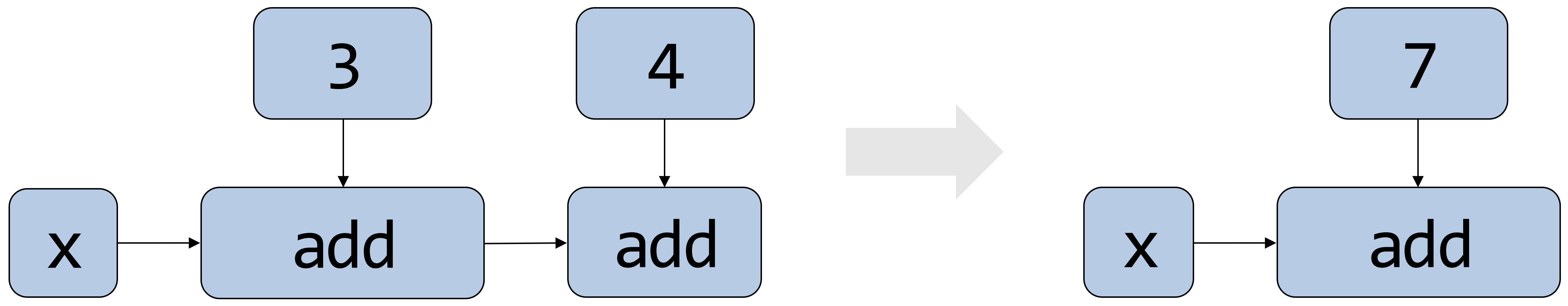
- Why operator fusion improves performance?
  - Reduce kernel launching
  - Reduce I/O
- Cons:
  - Requiring many fused ops: FusedABCOp
  - At some point, codebase becomes unmanageable

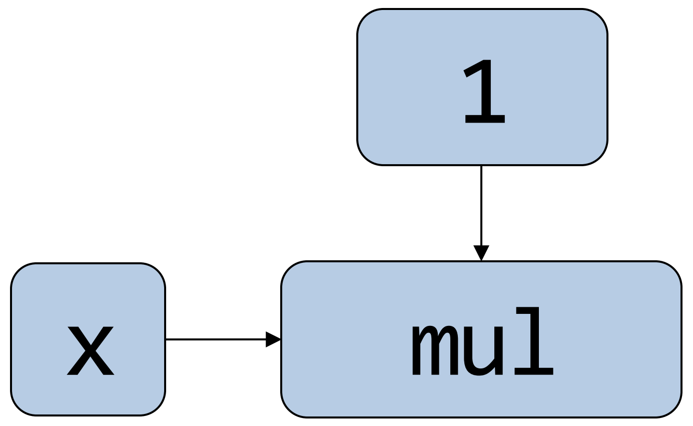
Operator Fusion in Practice: CUDA Graph

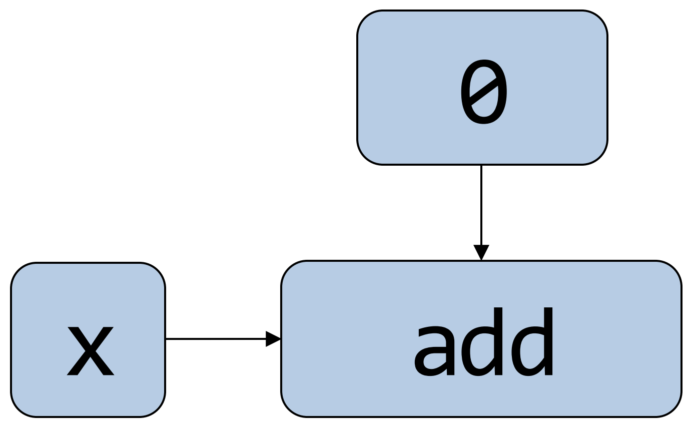
- Users are allowed to program using primitives with high-level APIs
- Graph is captured at CUDA level



Graph Optimization Templates: Constant Folding







A – (-B)	A + B
A + (A/C1)	

Common Subexpression Elimination (CSE)

$$\ldots$$
$$c = a + b$$
$$d = a$$
$$e = b$$
$$f = d + e$$
$$d = x$$
$$\ldots.$$

$$\ldots$$
$$c^3 = a^1 + b^2$$
$$d^1 = a^1$$
$$e^2 = b^2$$
~~$f^3 = d^1 + e^2$~~
$$f^3 = c^3$$
$$d^4 = x^4$$
$$\ldots.$$

CSE hit

Dead Code Elimination (DCE)

…

$c = a + b$

$d = a$

$e = b$

$f = d + e$

$d = x$

….

…

$c^3 = a^1 + b^2$

$d^1 = a^1$

$e^2 = b^2$

~~$f^3 = d^1 + e^2$~~

$f^3 = c^3$

$d^4 = x^4$

….

CSE hit

…

$c^3 = a^1 + b^2$

~~$d^1 = a^1$~~

$e^2 = b^2$

~~$f^3 = d^1 + e^2$~~

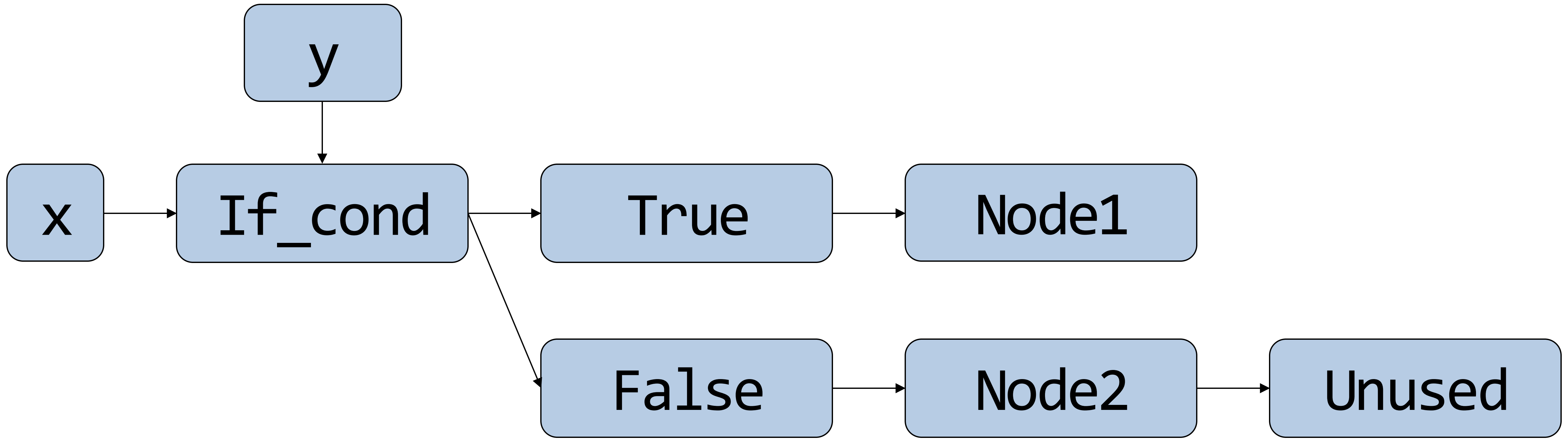
$f^3 = c^3$

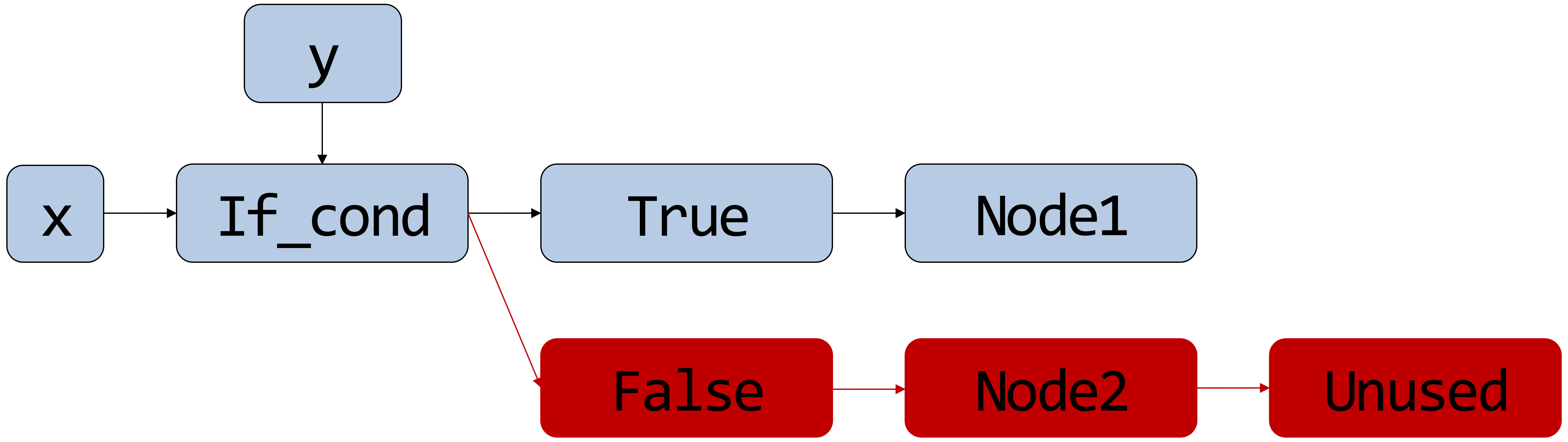
$d^4 = x^4$

….

DCE hit

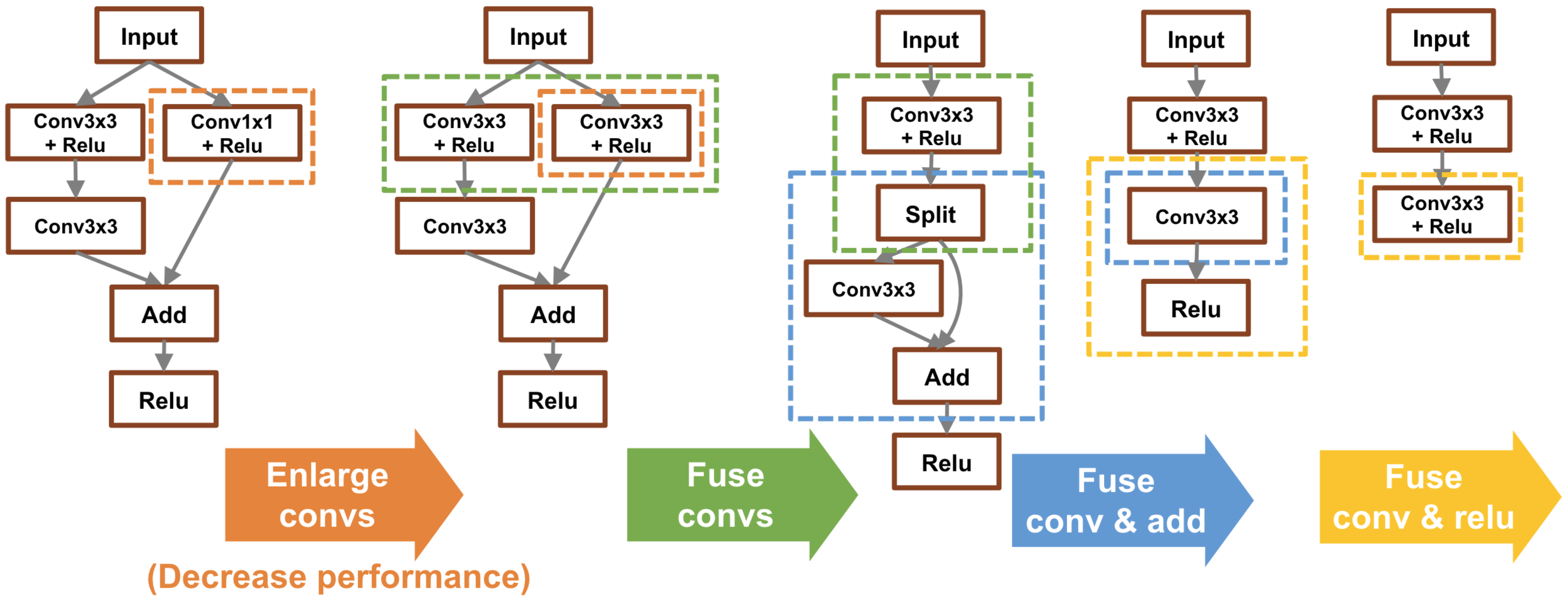
More templates for CSE and DCE





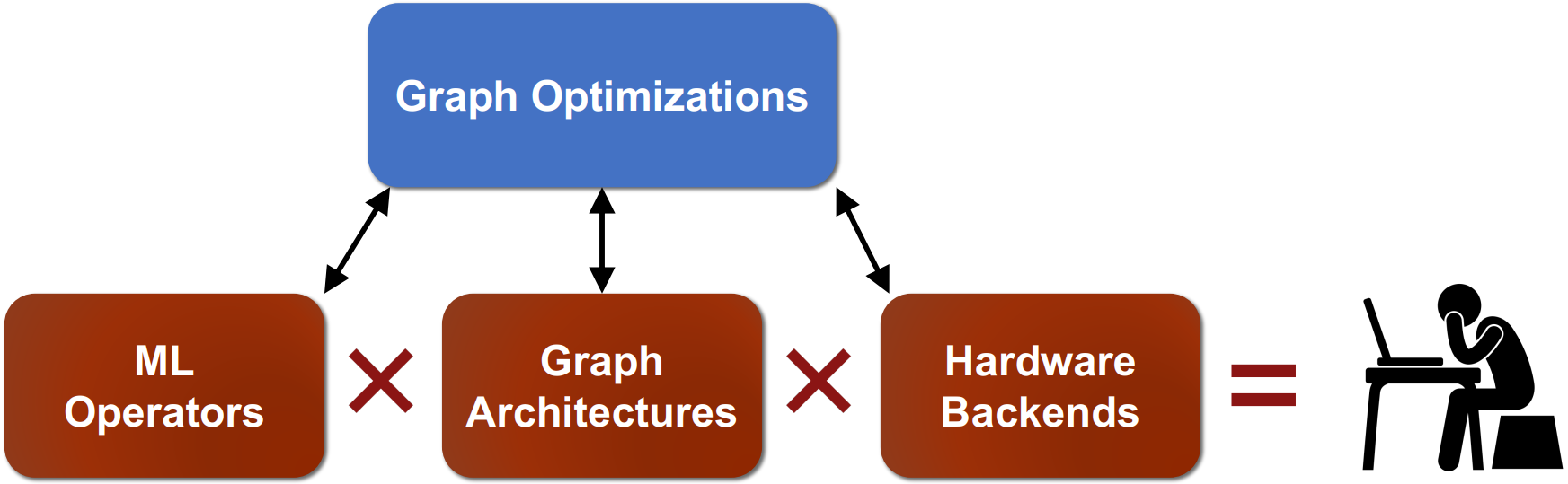
How to ensure performance gain?

- Greedily apply graph optimizations
- Recall the example below



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

Problems of Template-based Graph Optimizations



Graph Optimizations

ML Operators × Graph Architectures × Hardware Backends =

200 - 300

1000s

10s

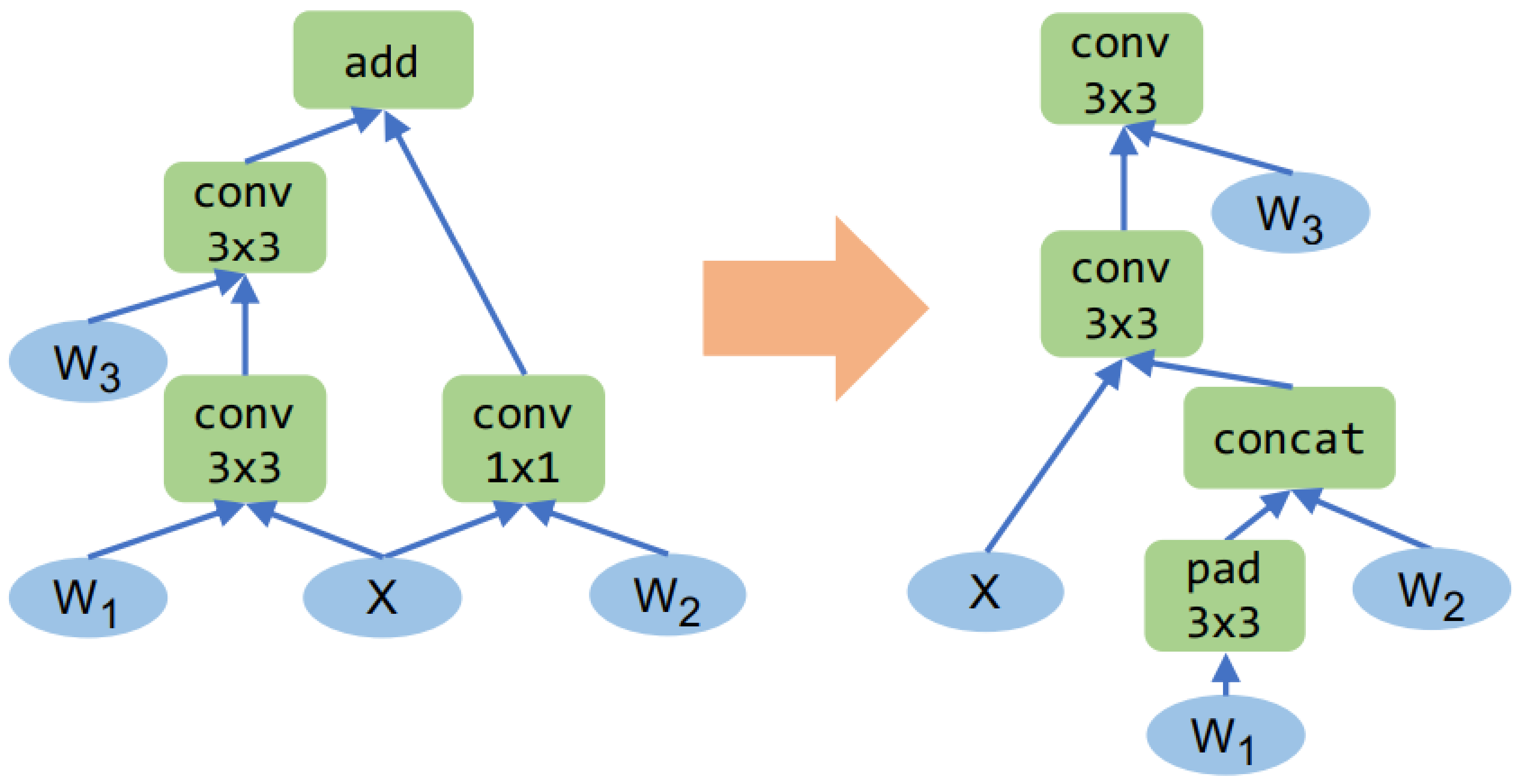
Problem: Infeasible to manually design graph optimizations for all cases

Problems of Template-based Graph Optimizations

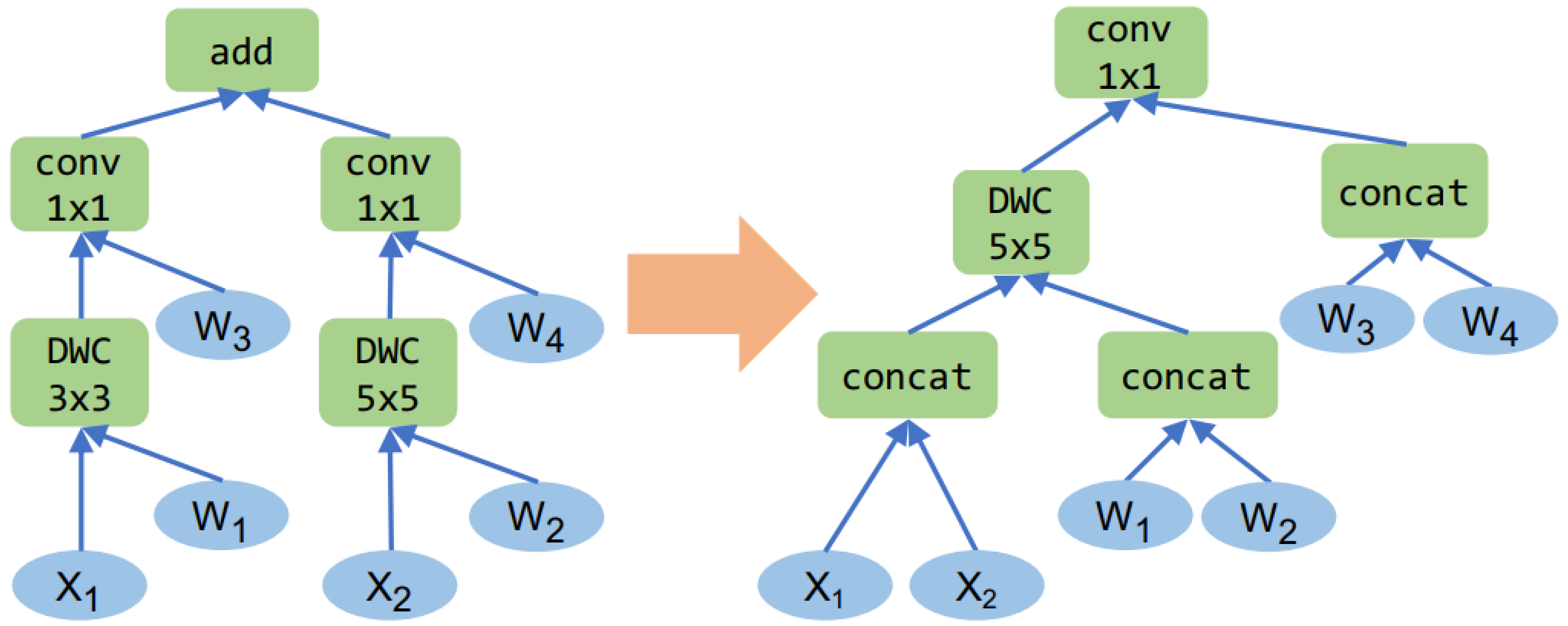
Robustness
Experts' heuristics do not apply to all DNNs/hardware

Scalability
New operators and graph structures require more rules

Performance
Miss subtle optimizations for specific DNNs/hardware



Only apply to **specific hardware**



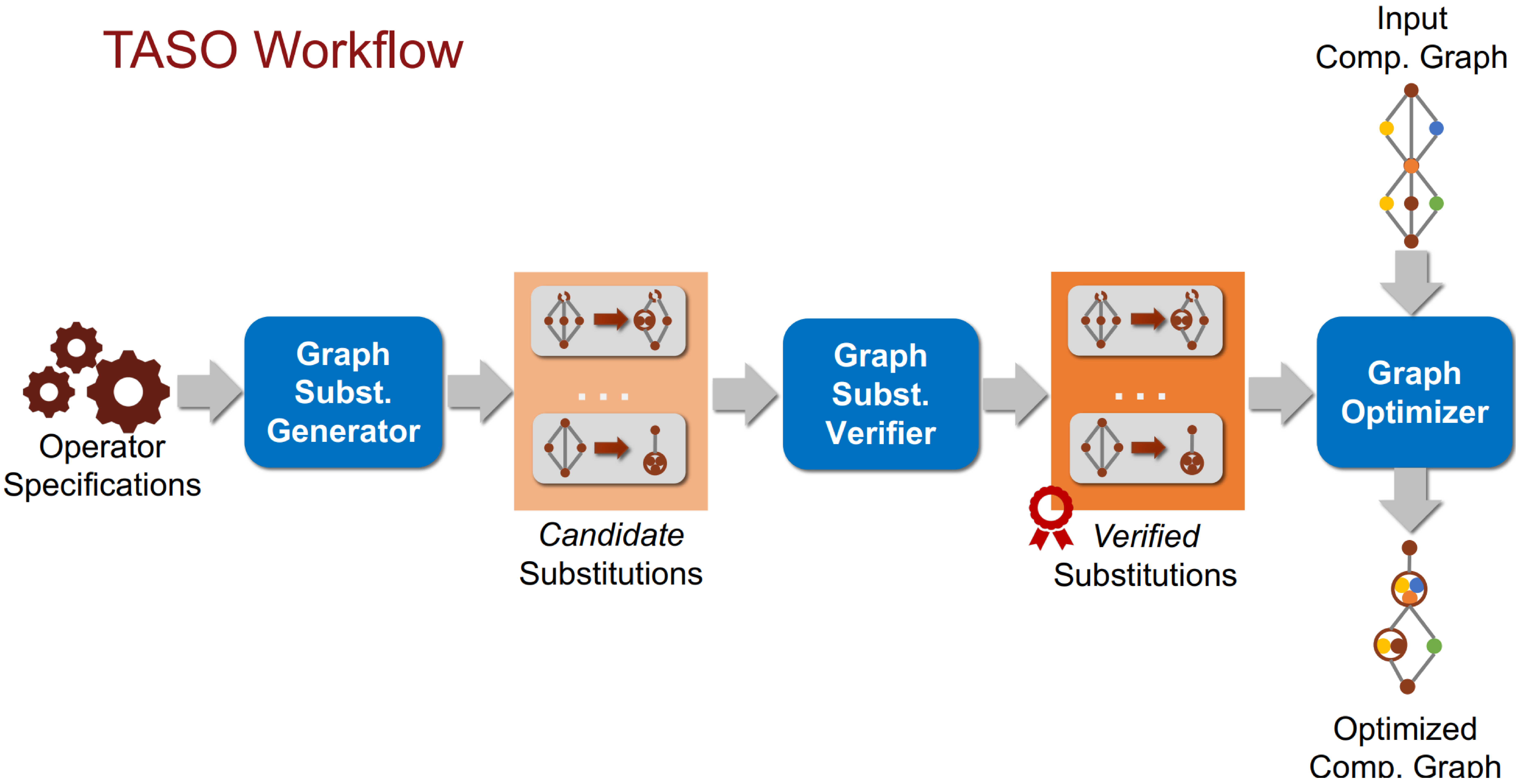
Only apply to **specialized graph structures**

Automate Graph Transformation

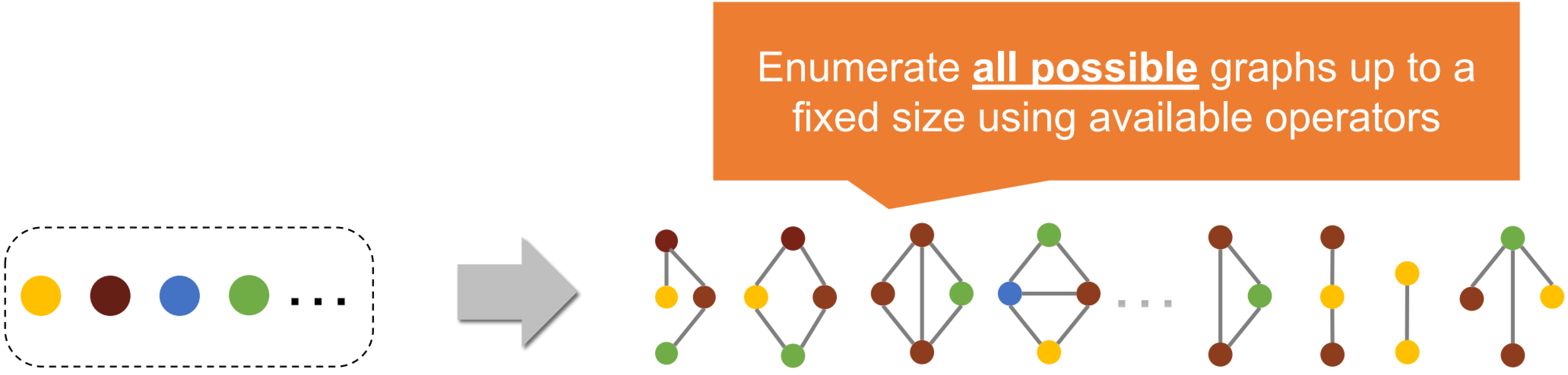
Key idea: replace manually-designed graph optimizations with automated generation and verification of graph substitutions for tensor algebra

Enumerate and Verify ALL possible graph

TASO Workflow



Graph Substitution Generator

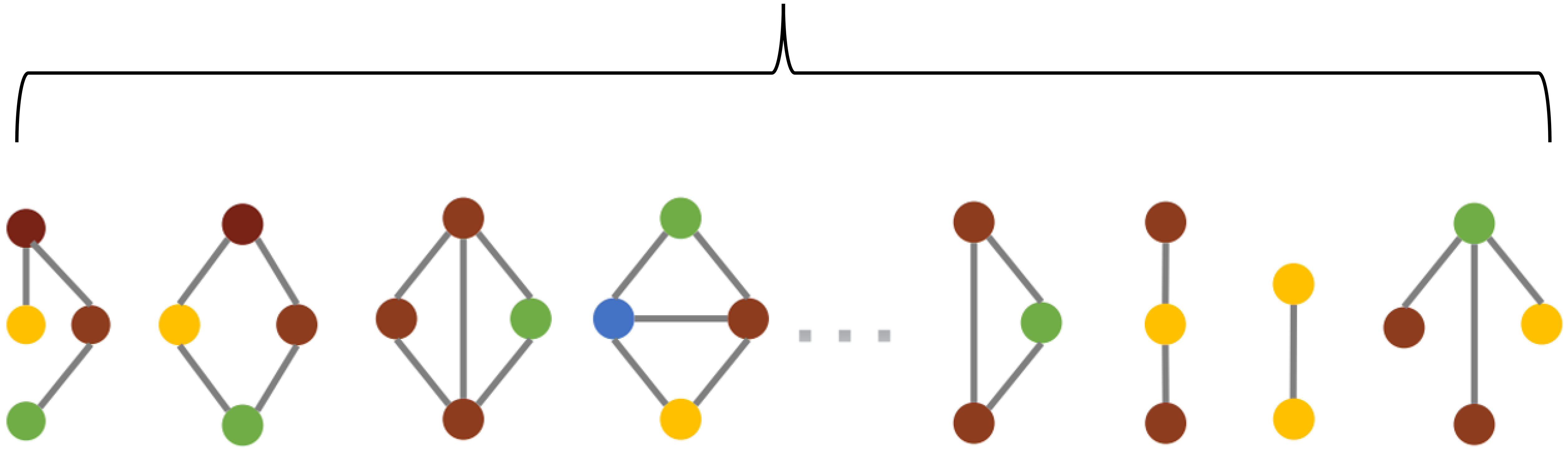


Enumerate **all possible** graphs up to a fixed size using available operators

Operators supported by hardware backend

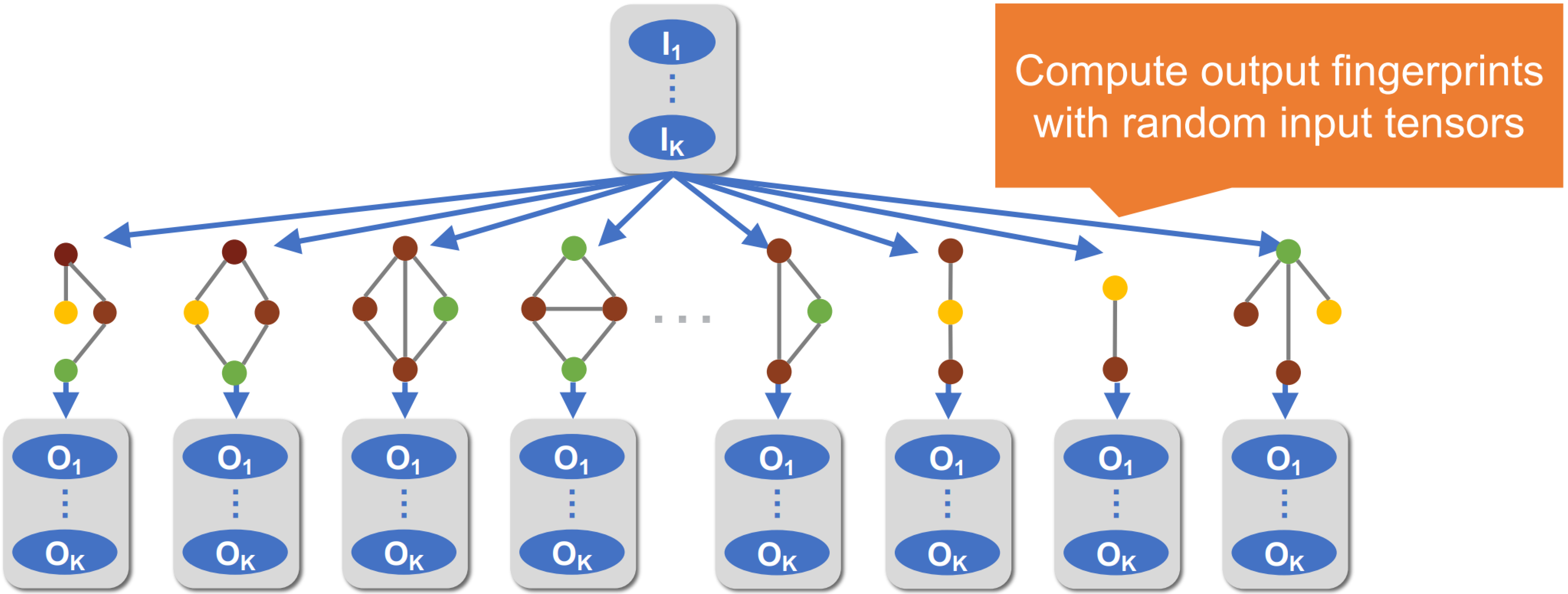
There are many subgraphs even only given 4 Ops

66M graphs with up to 4 operators



A substitution = a pair of equivalent graphs

Graph Substitution Generator

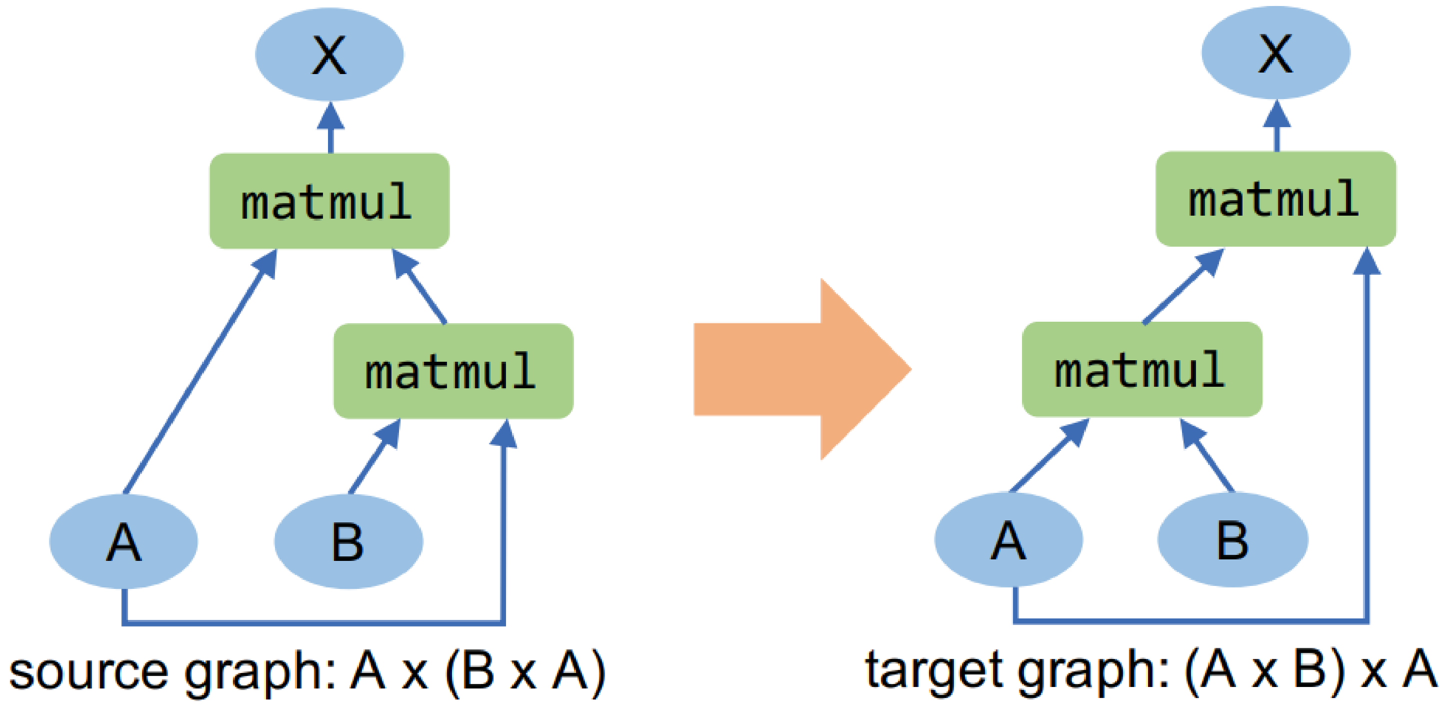


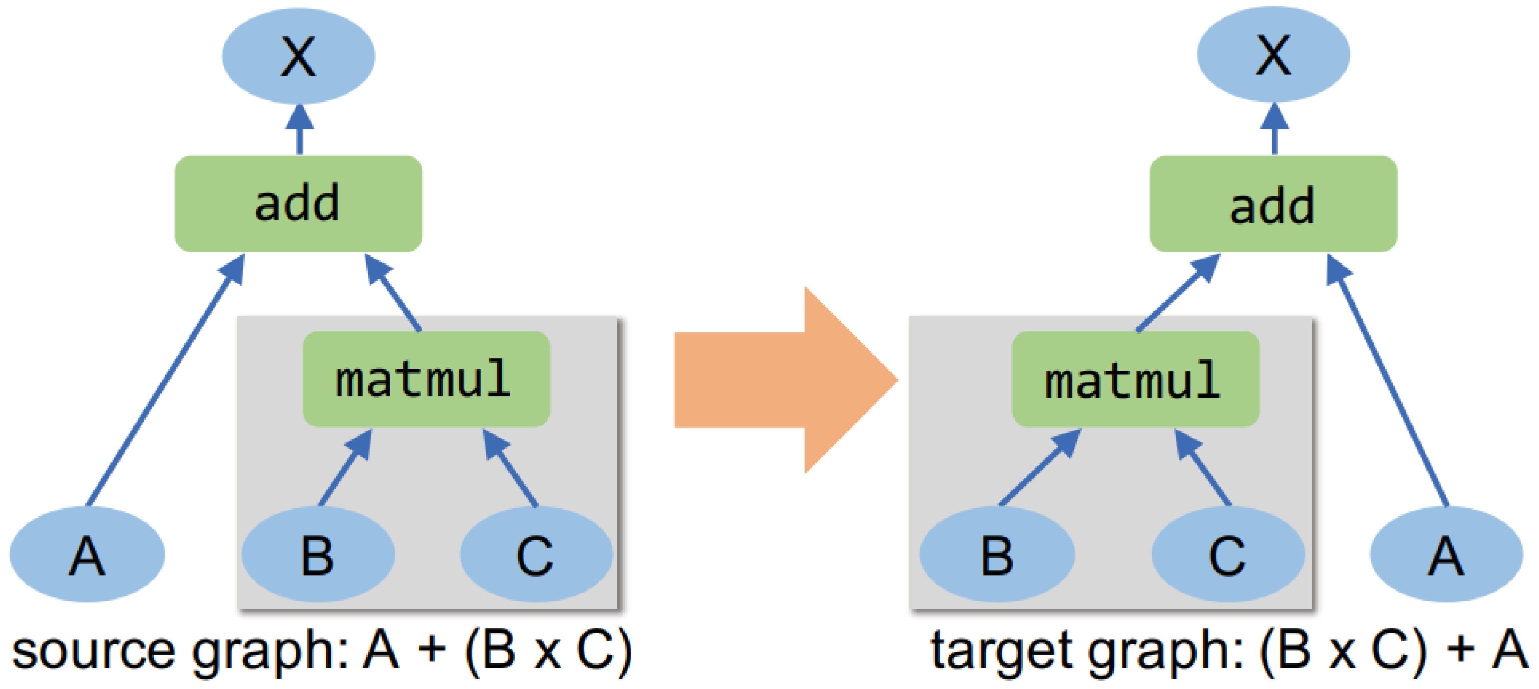
Compute output fingerprints with random input tensors

We can generate 28744 substitutions by enumerating graphs with up to 4 ops

Pruning repeated graphs

28744
substitutions





734
substitutio

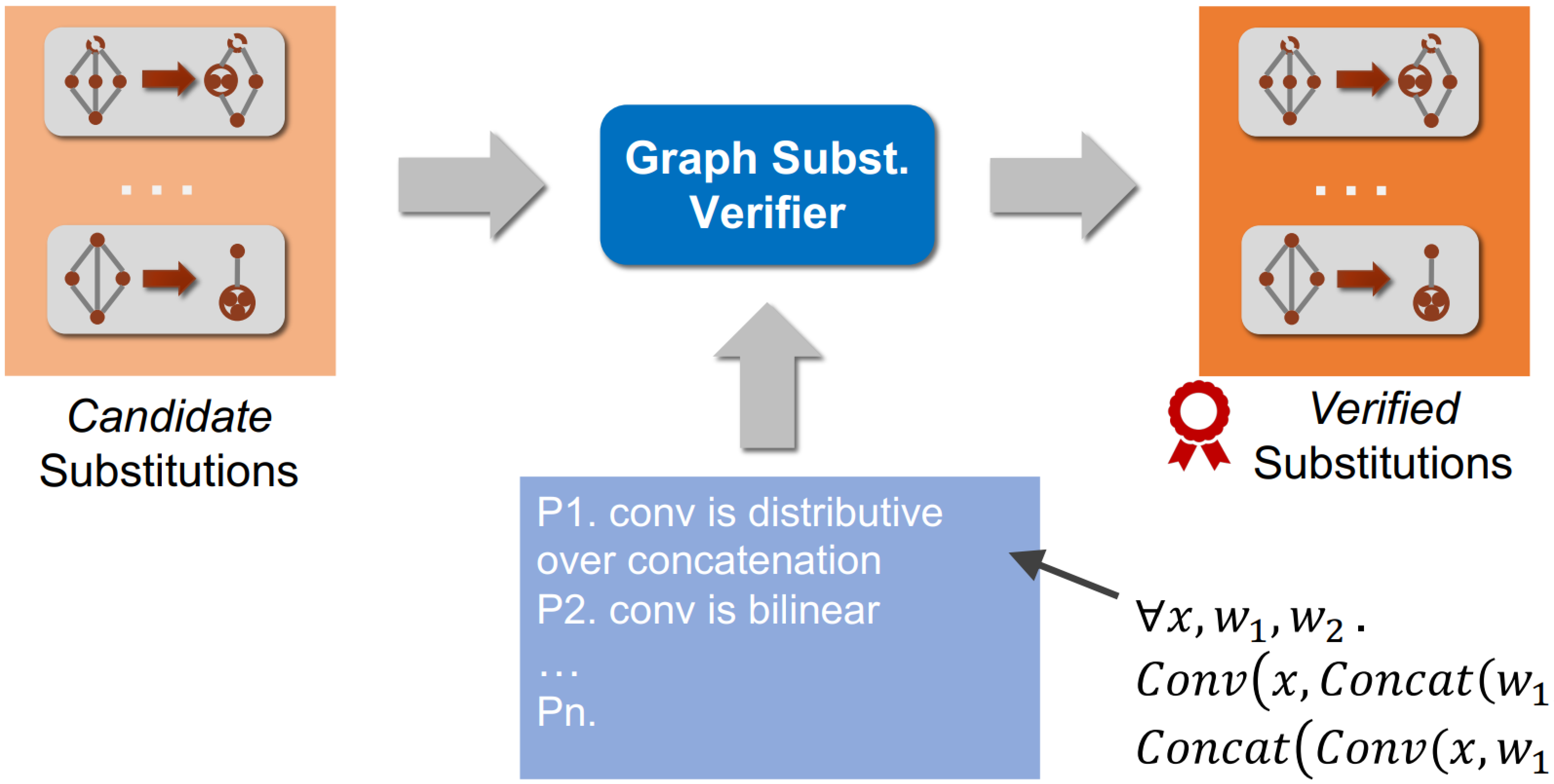
Variable renaming

Common subgraph

Can we trust graph substitutions?

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

Substitution Verifier



Candidate Substitutions

Graph Subst. Verifier

Verified Substitutions

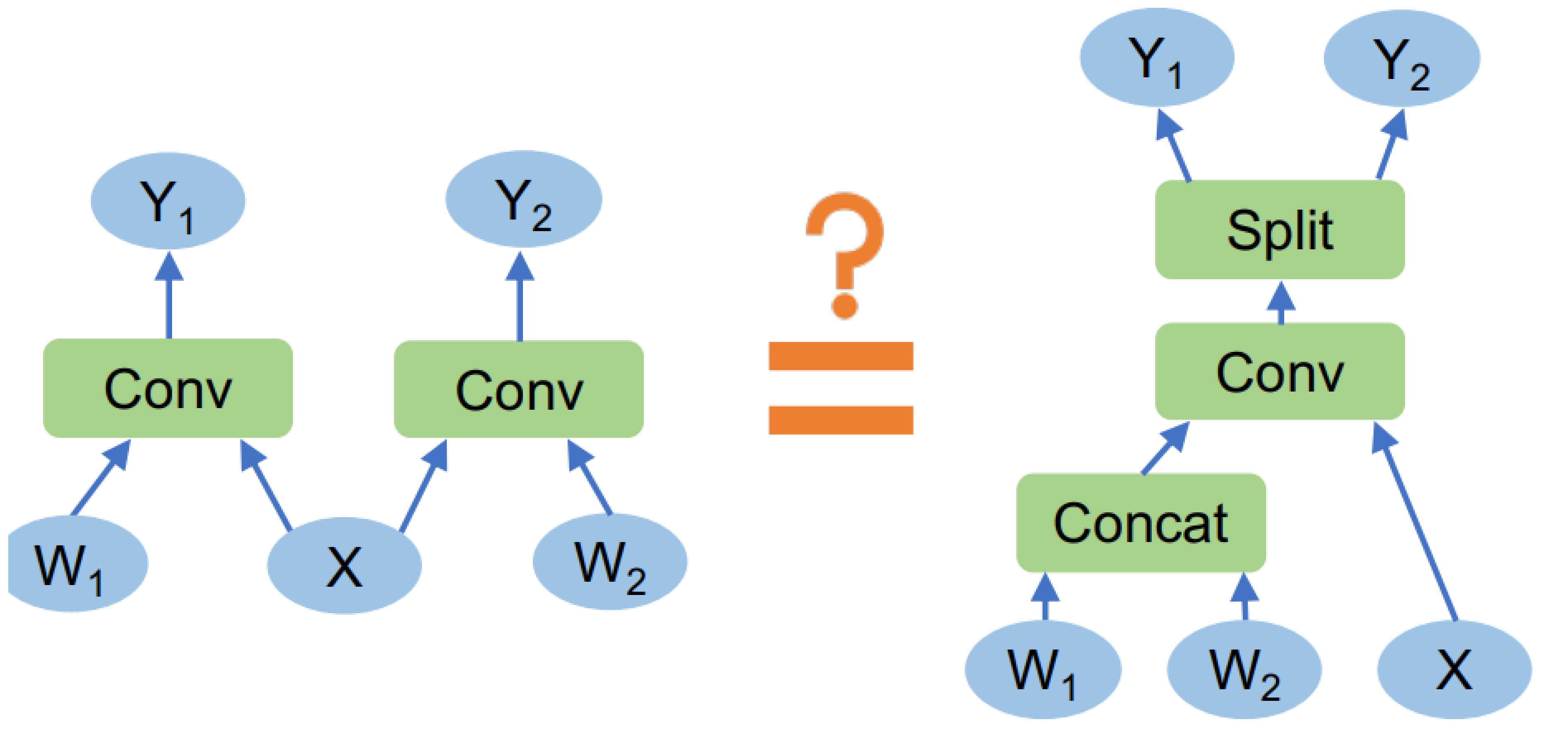
P1. conv is distributive over concatenation
P2. conv is bilinear
…
Pn.

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

DEV

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

How to Verify



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

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

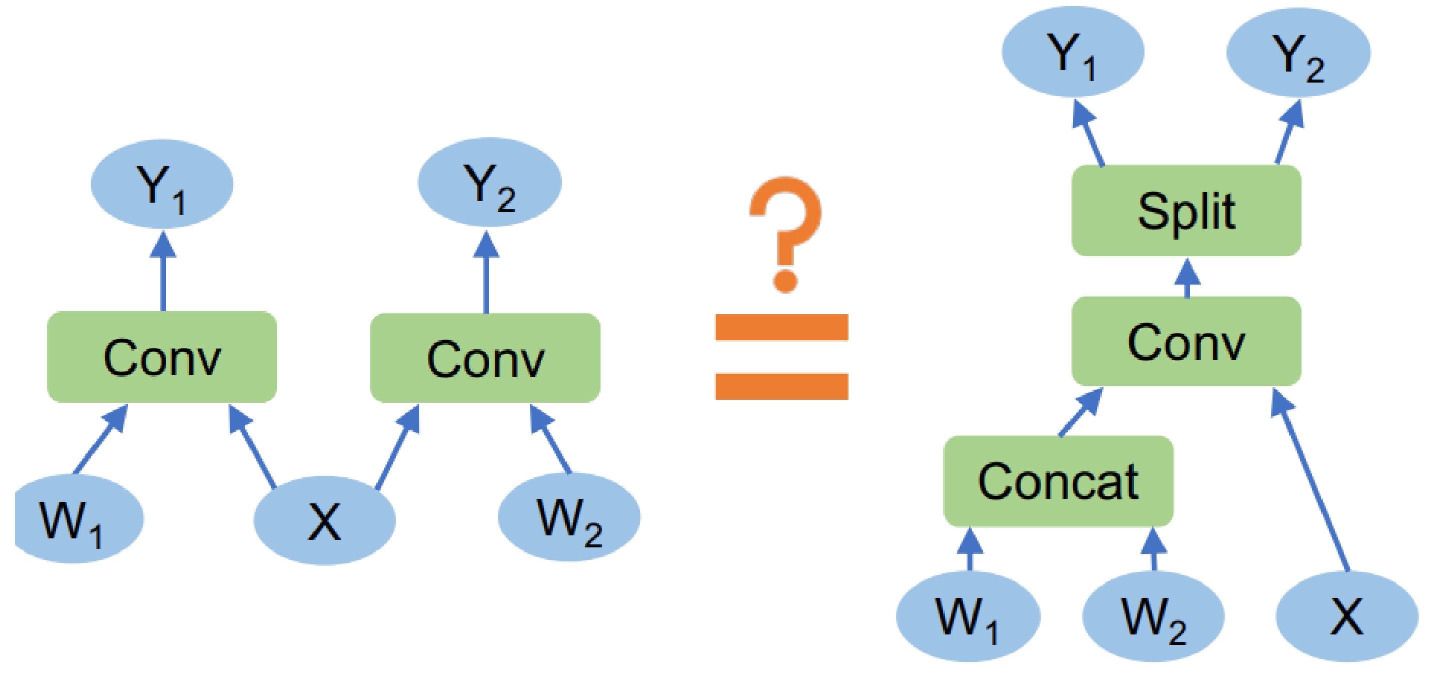
$$\forall x, w_1, w_2 .\ (Conv(x, w_1), Conv(x, w_2)) = Split\big(Conv(x, Concat(w_1, w_2))\big)$$

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

P2. …

Automated theorem prover

✓



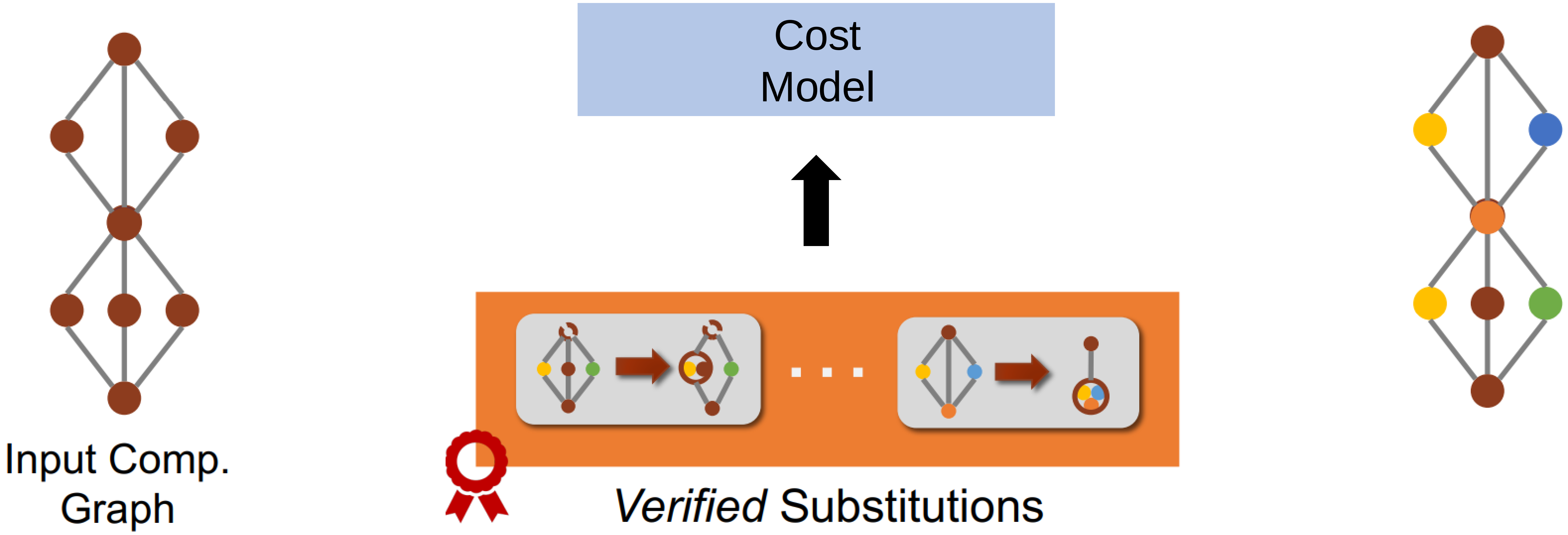
(Conv(x, w_1), Conv (x, w_2))

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

- 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

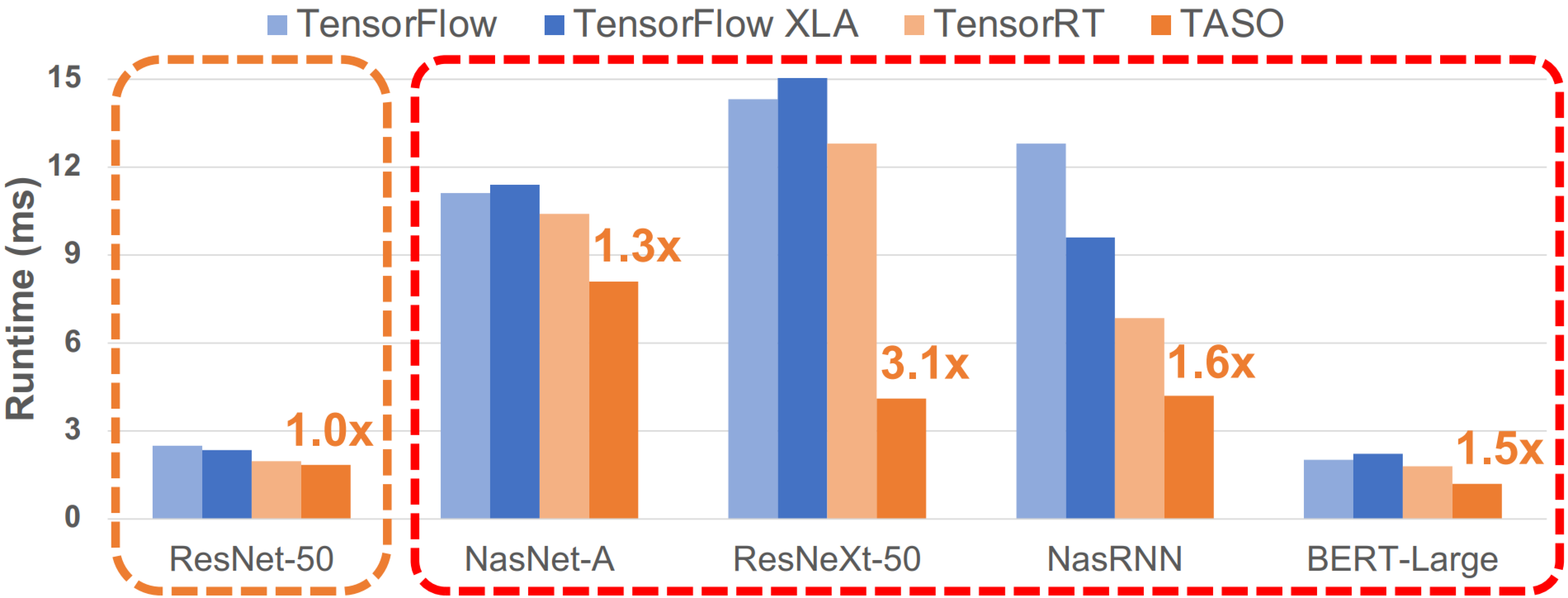


Cost Model

Input Comp. Graph

Verified Substitutions

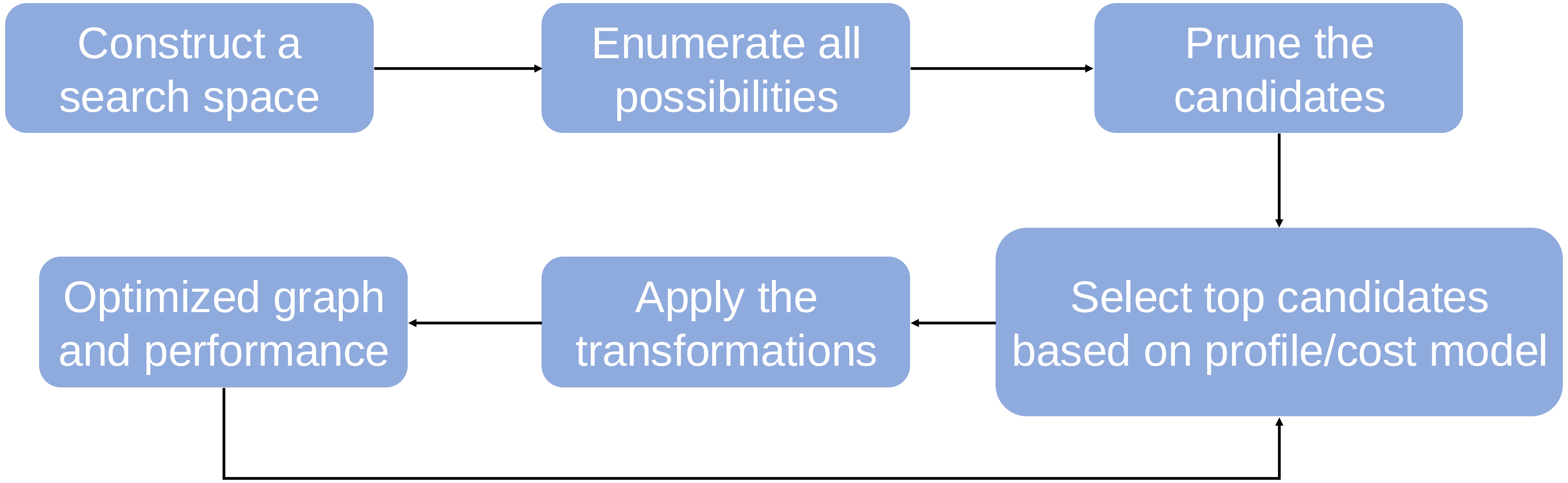
Performance (as of 2019)



Competitive on standard models

Larger speedups on emerging models

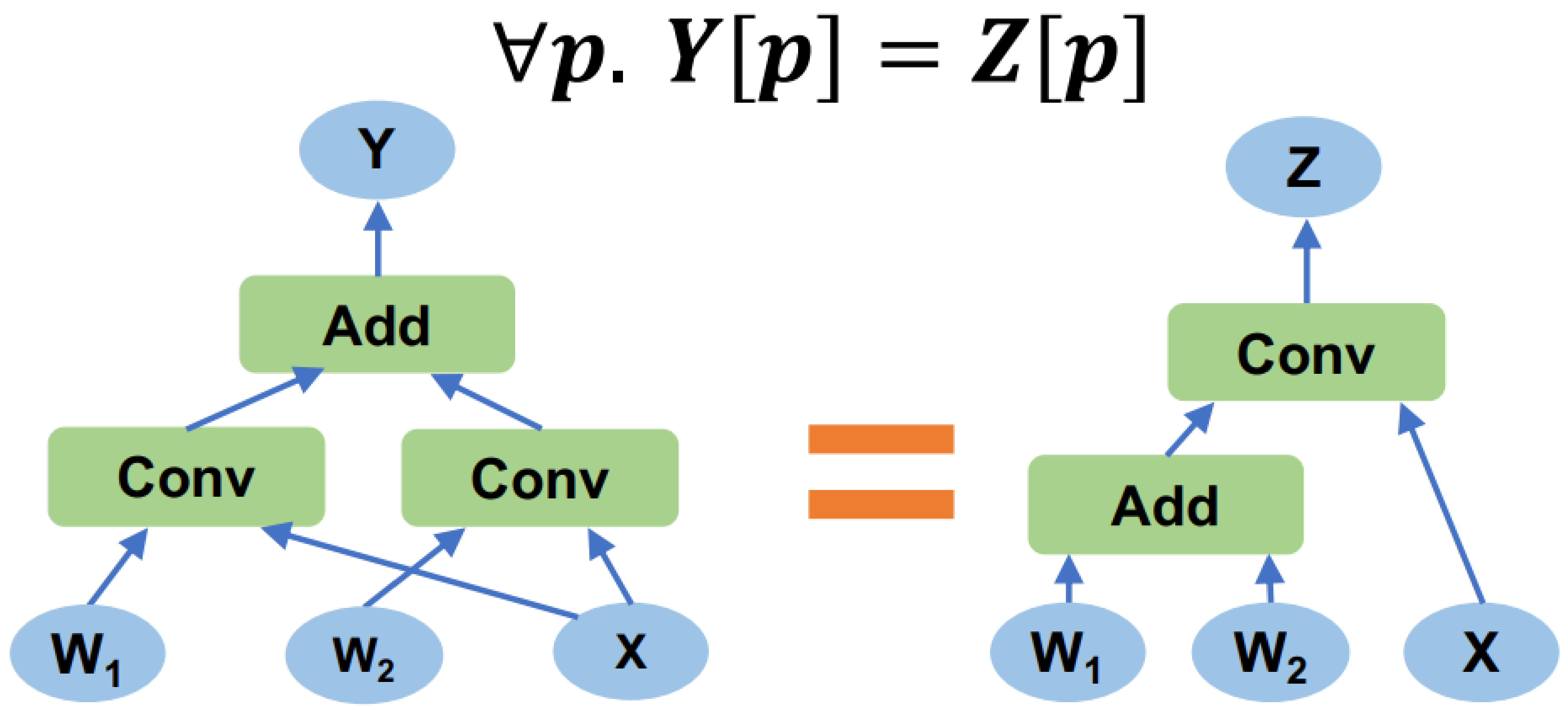
Summary of Graph Optimization


Construct a search space
Enumerate all possibilities
Prune the candidates
Select top candidates based on profile/cost model
Apply the transformations
Optimized graph and performance

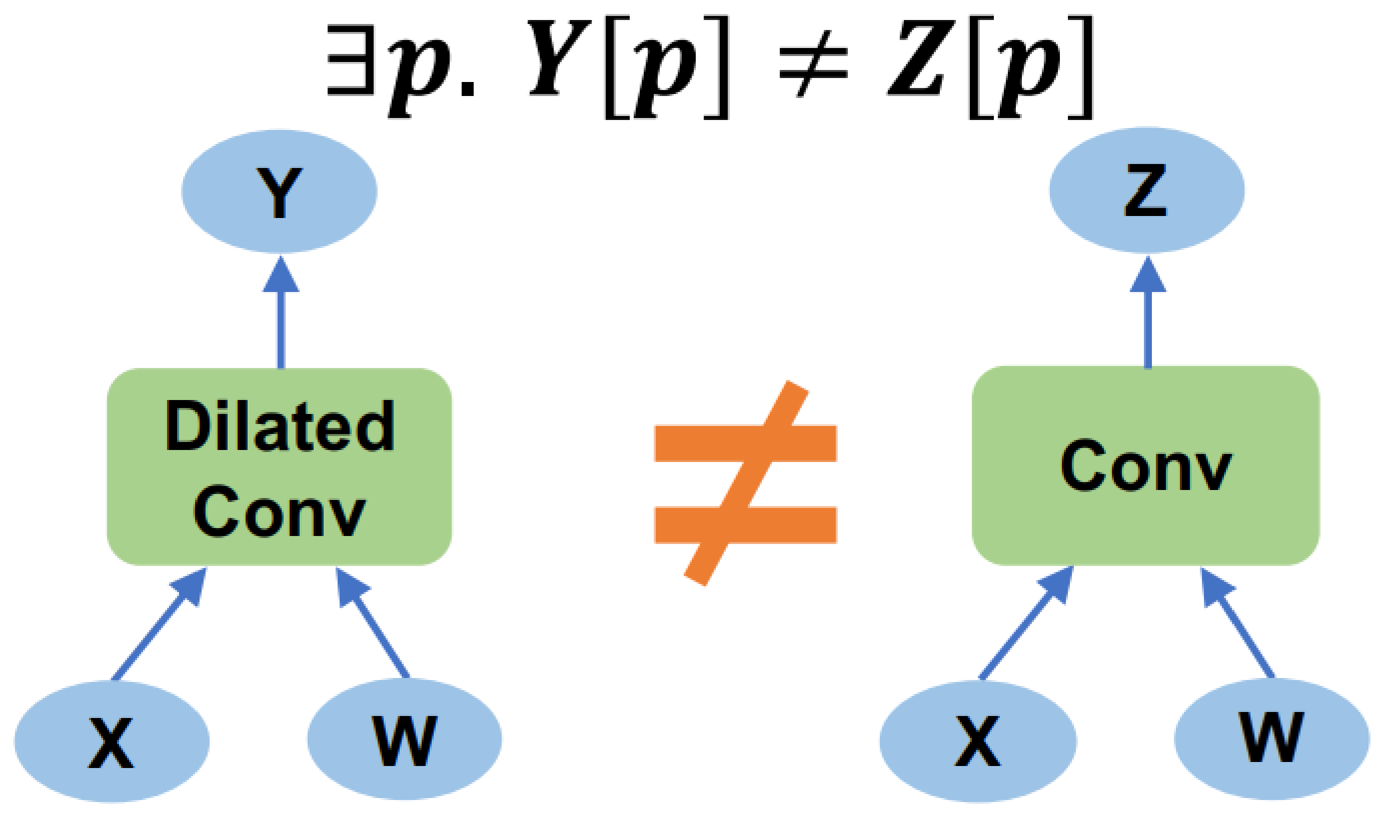
Limitations

- 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



Fully Equivalent Transformations



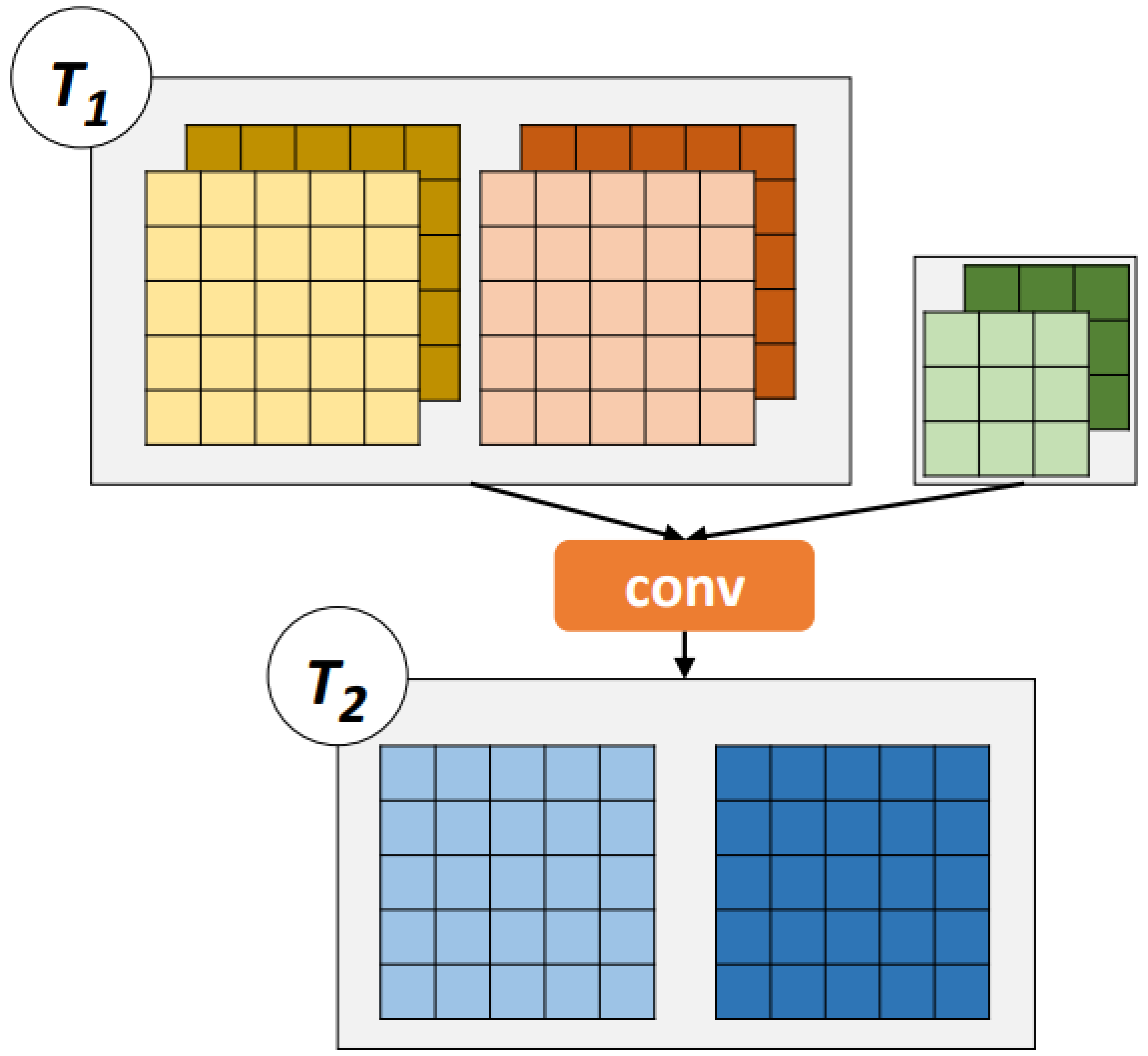
Partially Equivalent Transformations

- Math-equivalent
- Missing some optimization opportunities

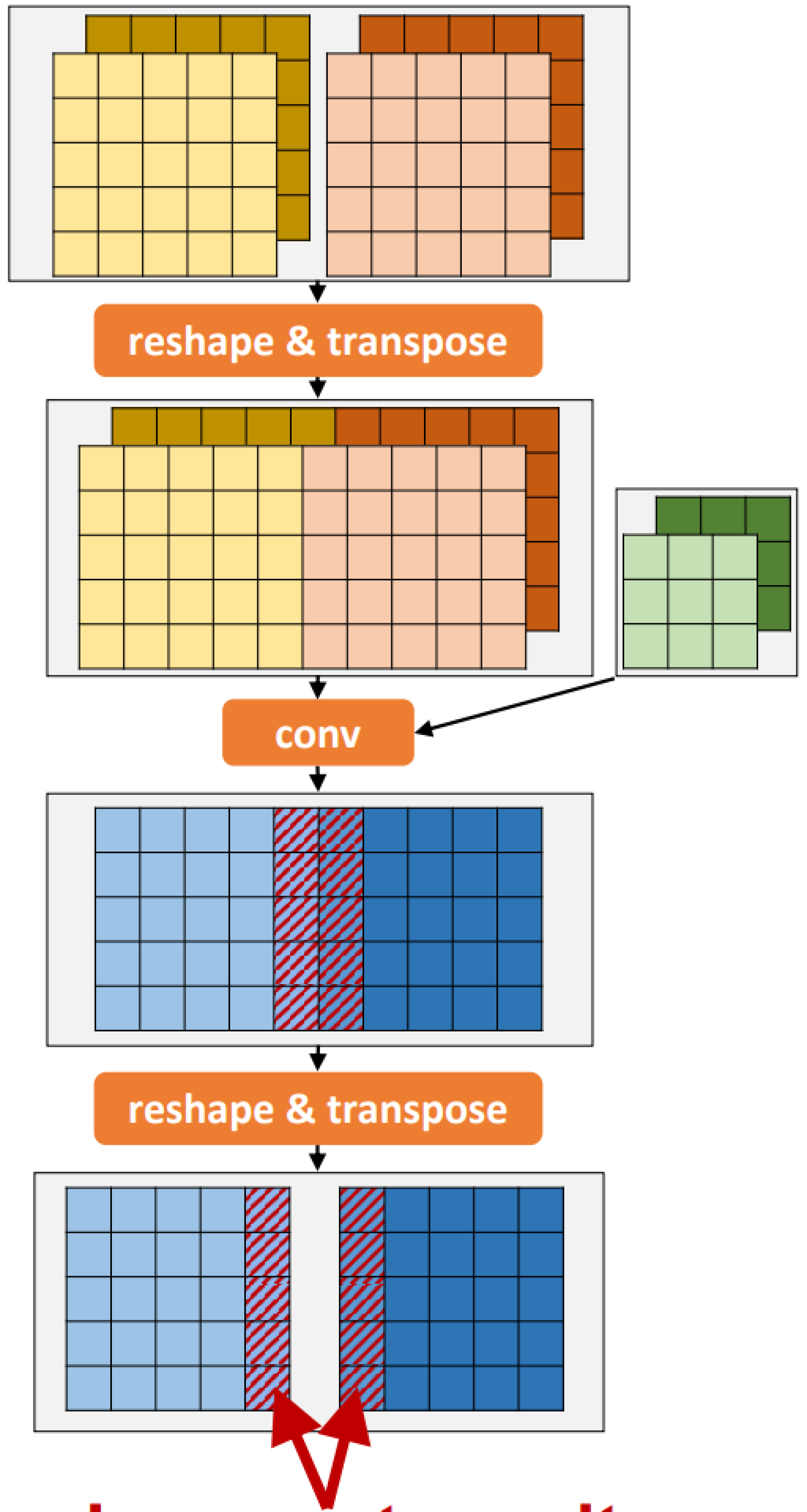
- Better performance
- Not fully equivalent -> accuracy loss

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

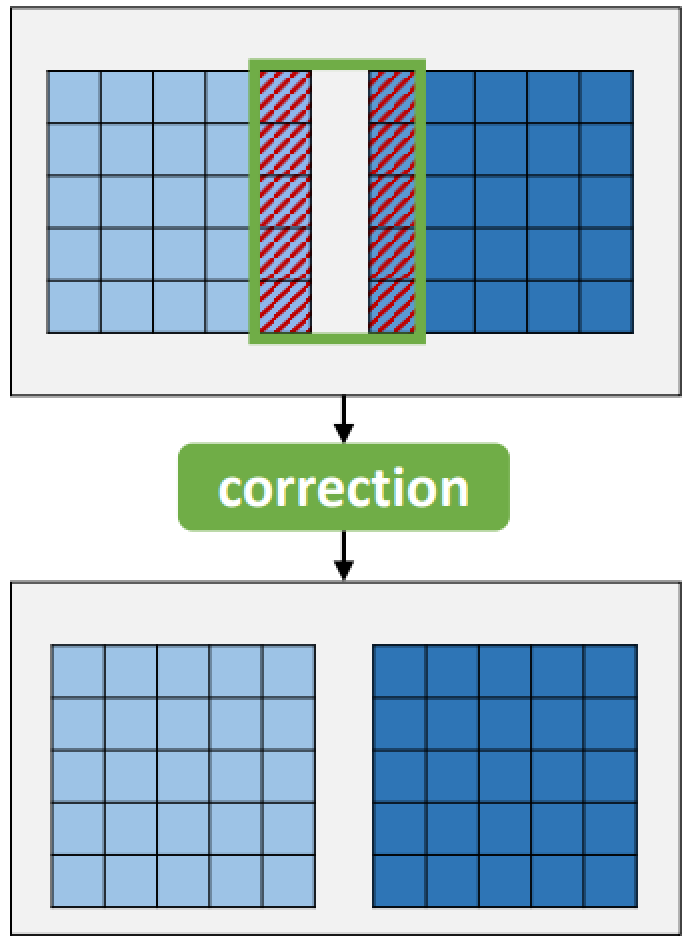
Motivating Example



Input Program



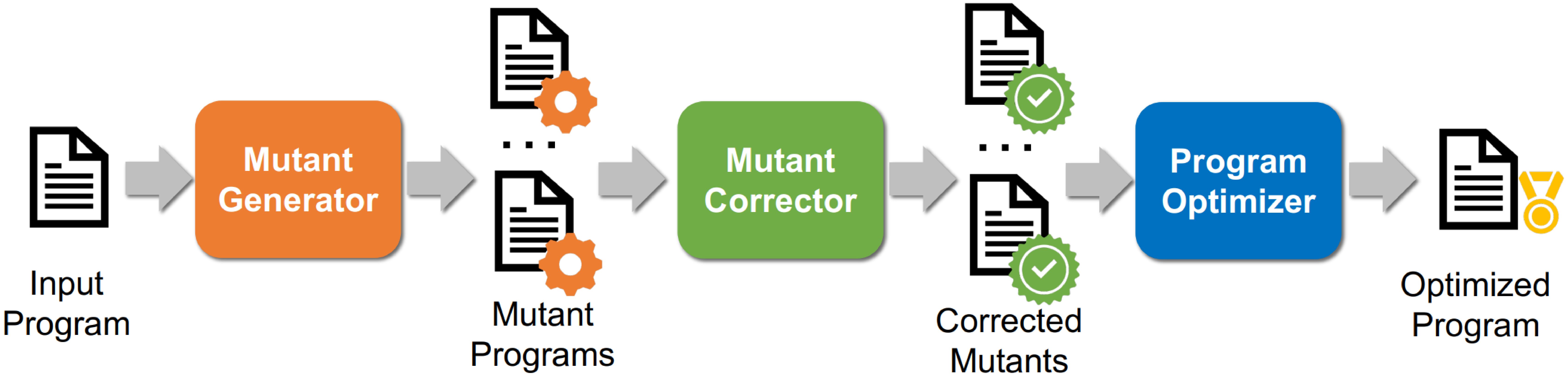
Incorrect results



+ Correction

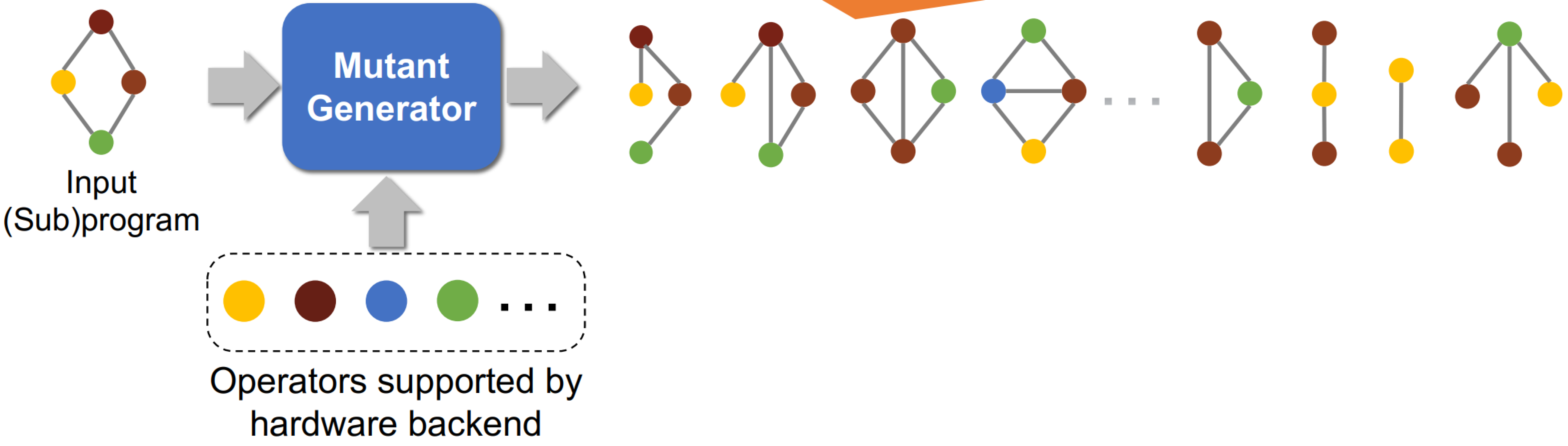
- Partial equivalent transformations + correction yield 1.2x speedup
- Which would otherwise be impossible in fully equivalent transformations space

Partially Equivalent Transformations



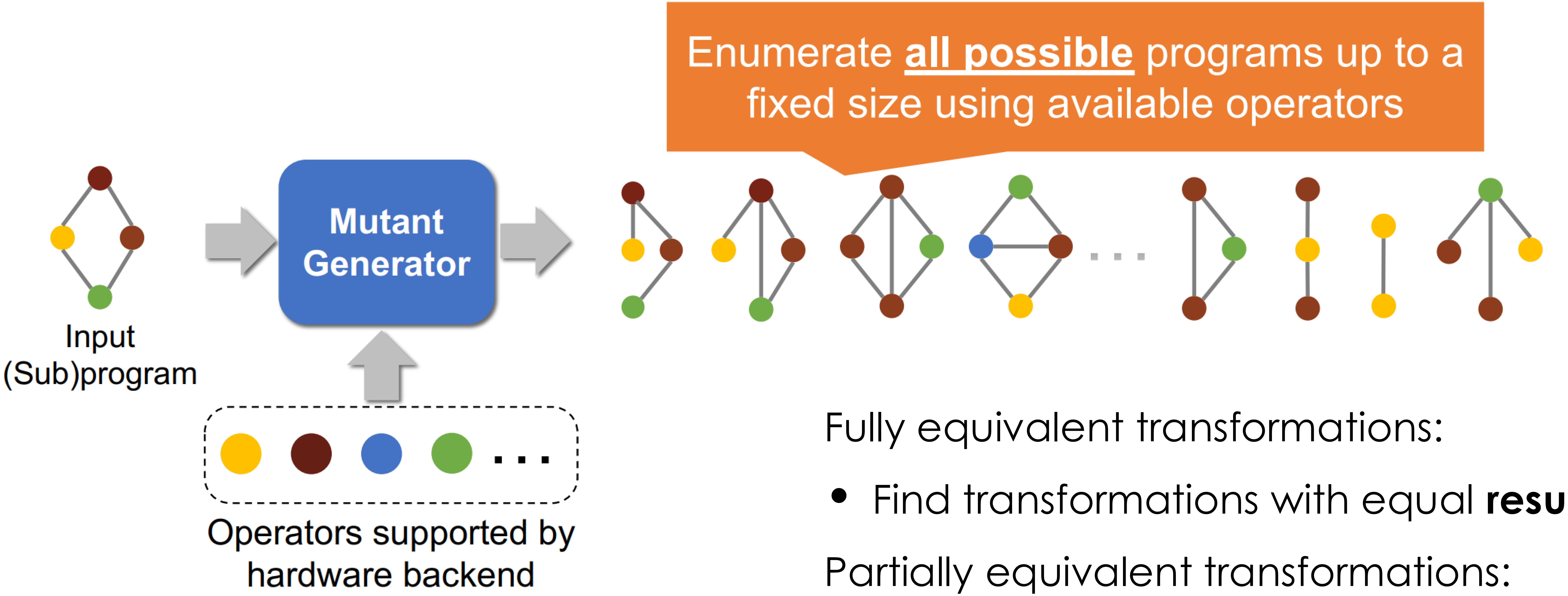
- How to mutate?
- How to correct?

Mutant Generator: Step 1



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

Mutant Generator: Step 2



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

Fully equivalent transformations:

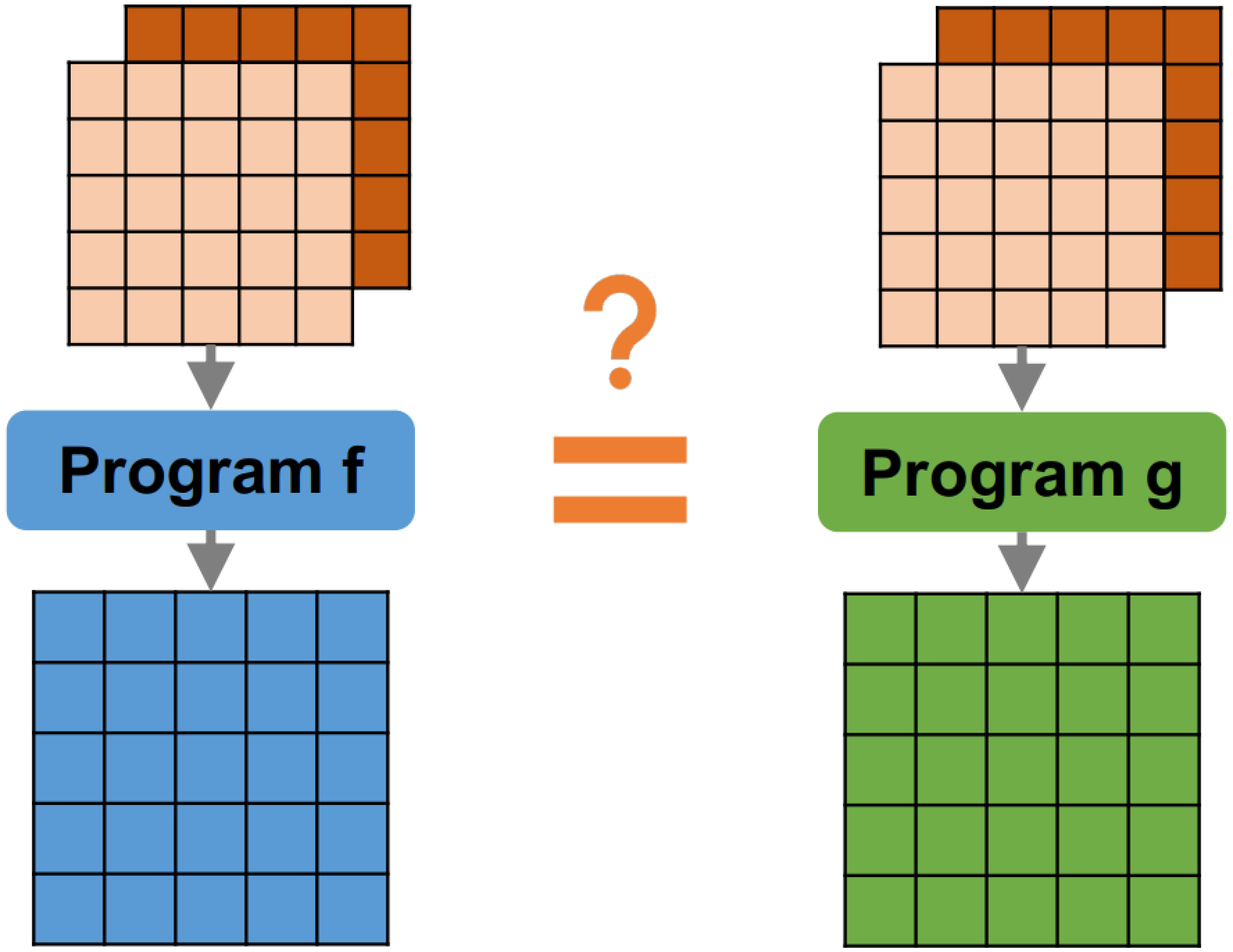
- Find transformations with equal **results**

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?

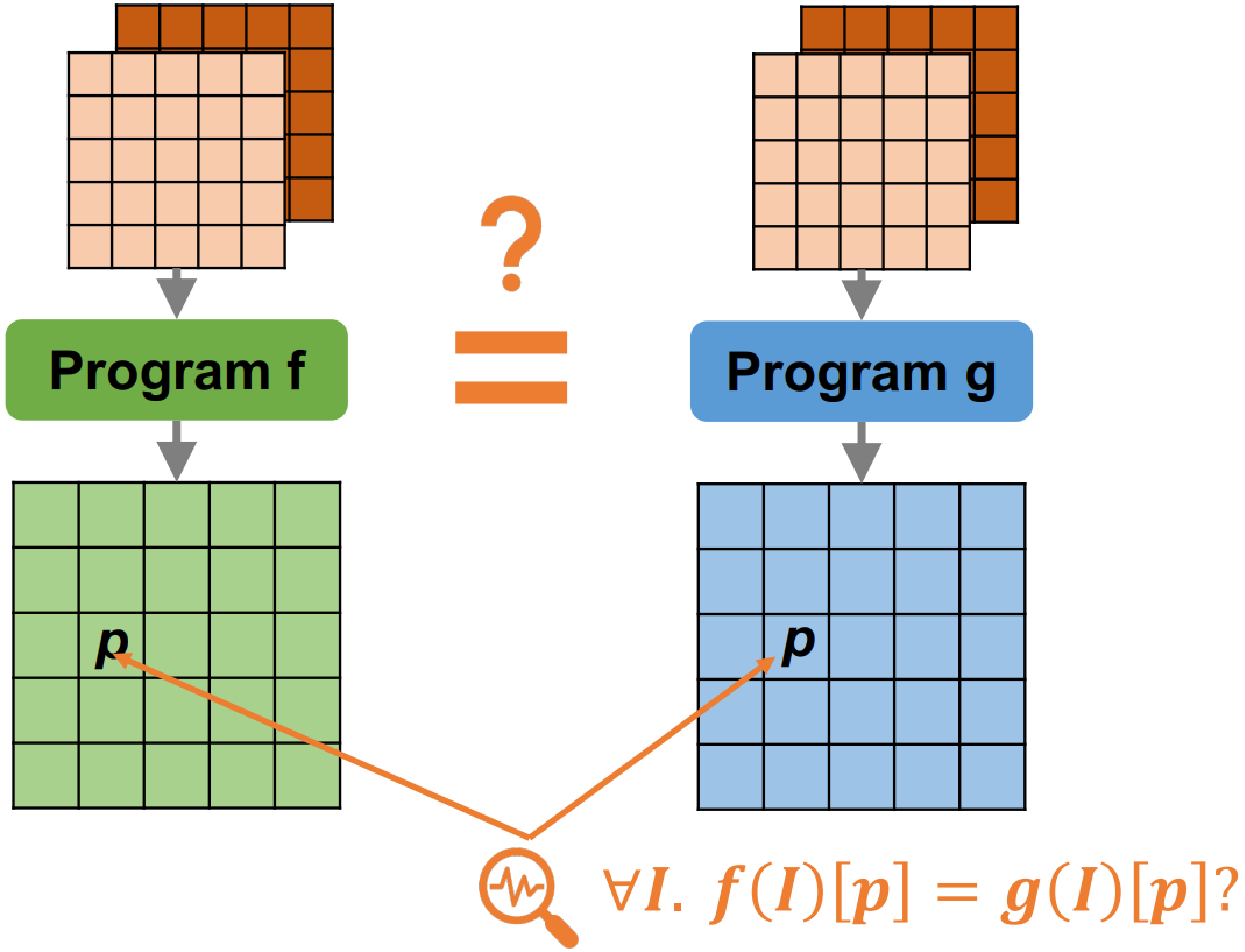


By Enumeration

- 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?
  - Reduce m and n



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

How to reduce n?

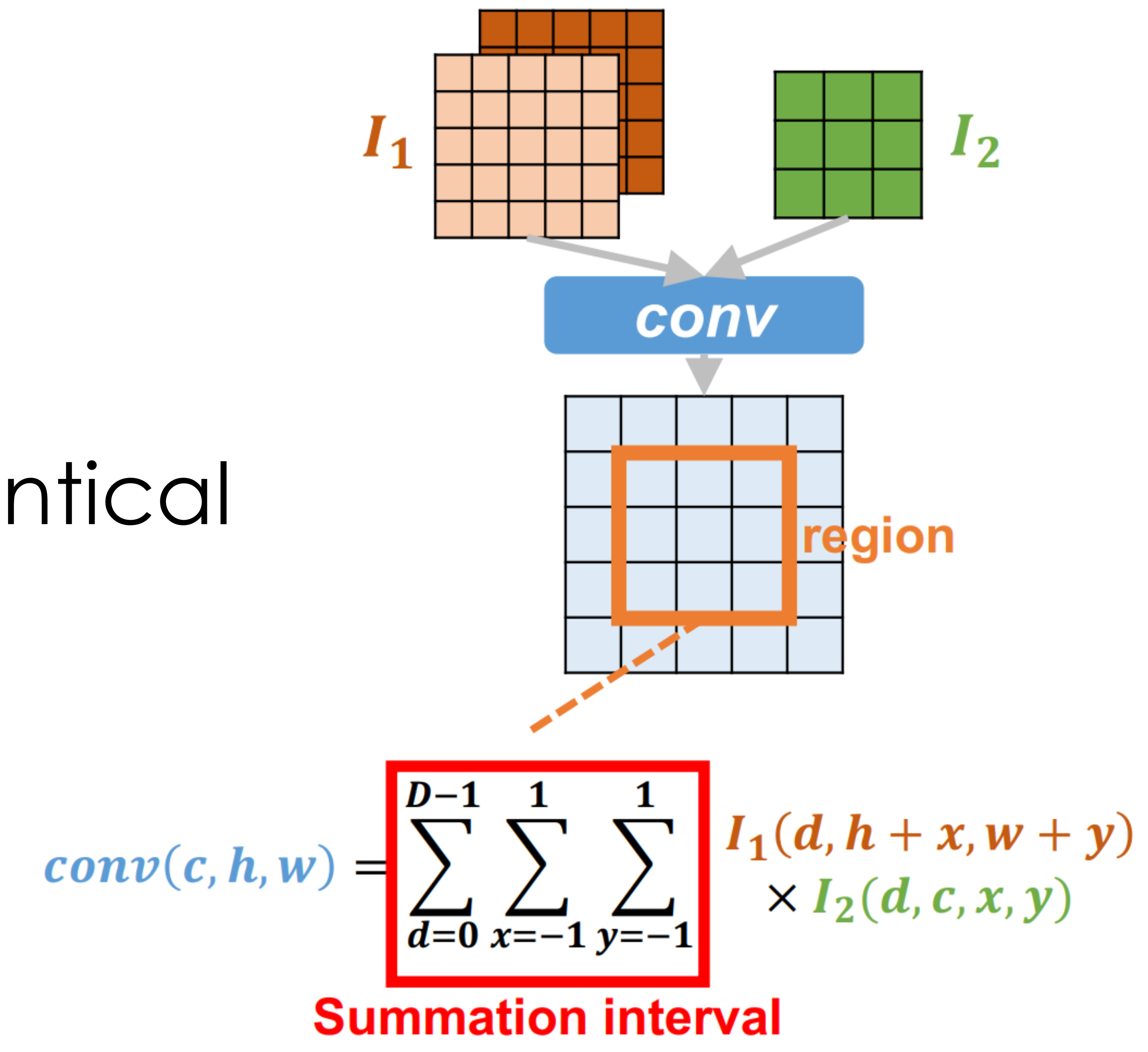
- Can we just check out a few (or even just one) position at f(I)[p] and assert the (in-)correctness?
- Answer: Yes for 80% of the computation

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

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



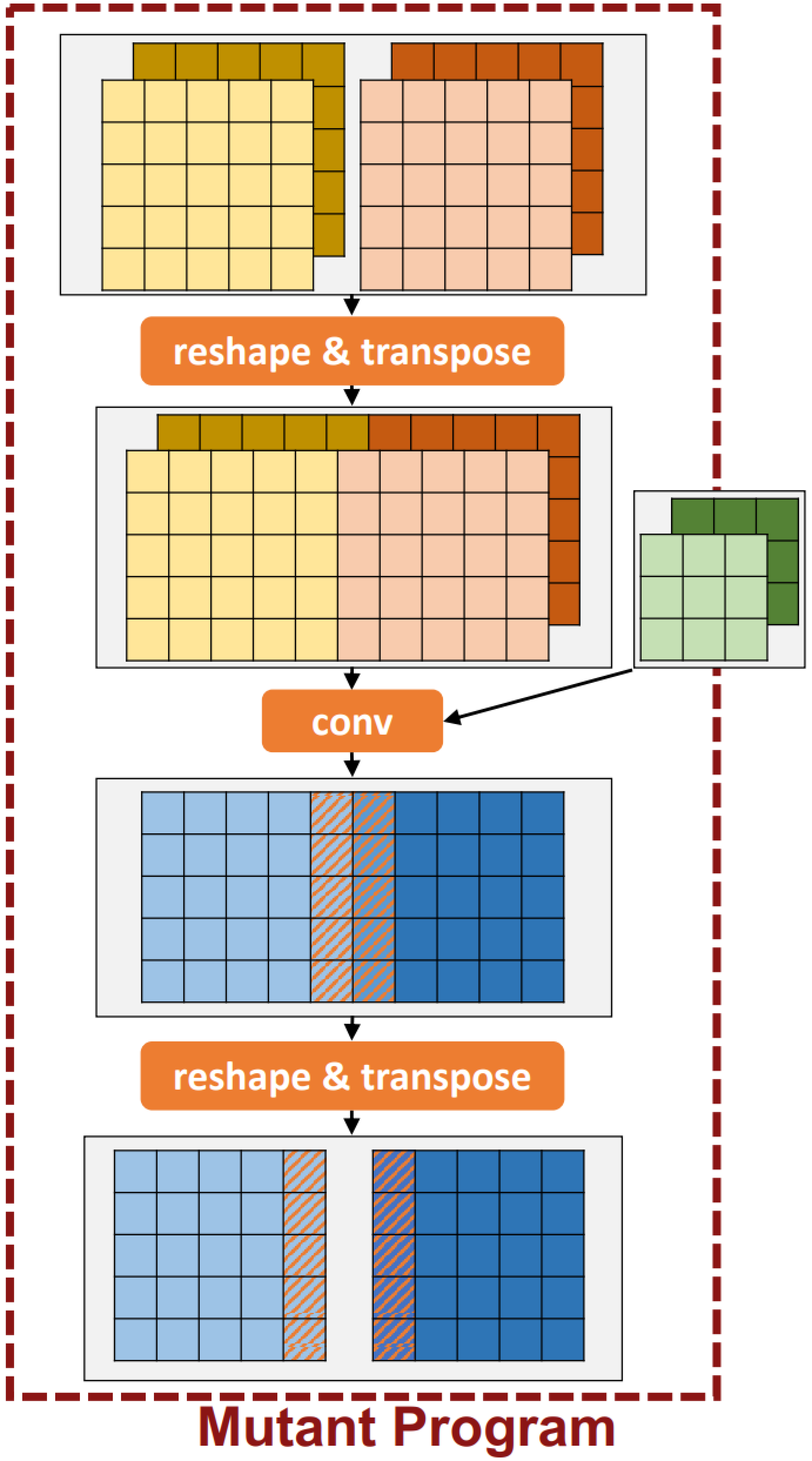
$$conv(c,h,w) = \boxed{\sum_{d=0}^{D-1}\sum_{x=-1}^{1}\sum_{y=-1}^{1}} I_1(d,h+x,w+y) \times I_2(d,c,x,y)$$

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)

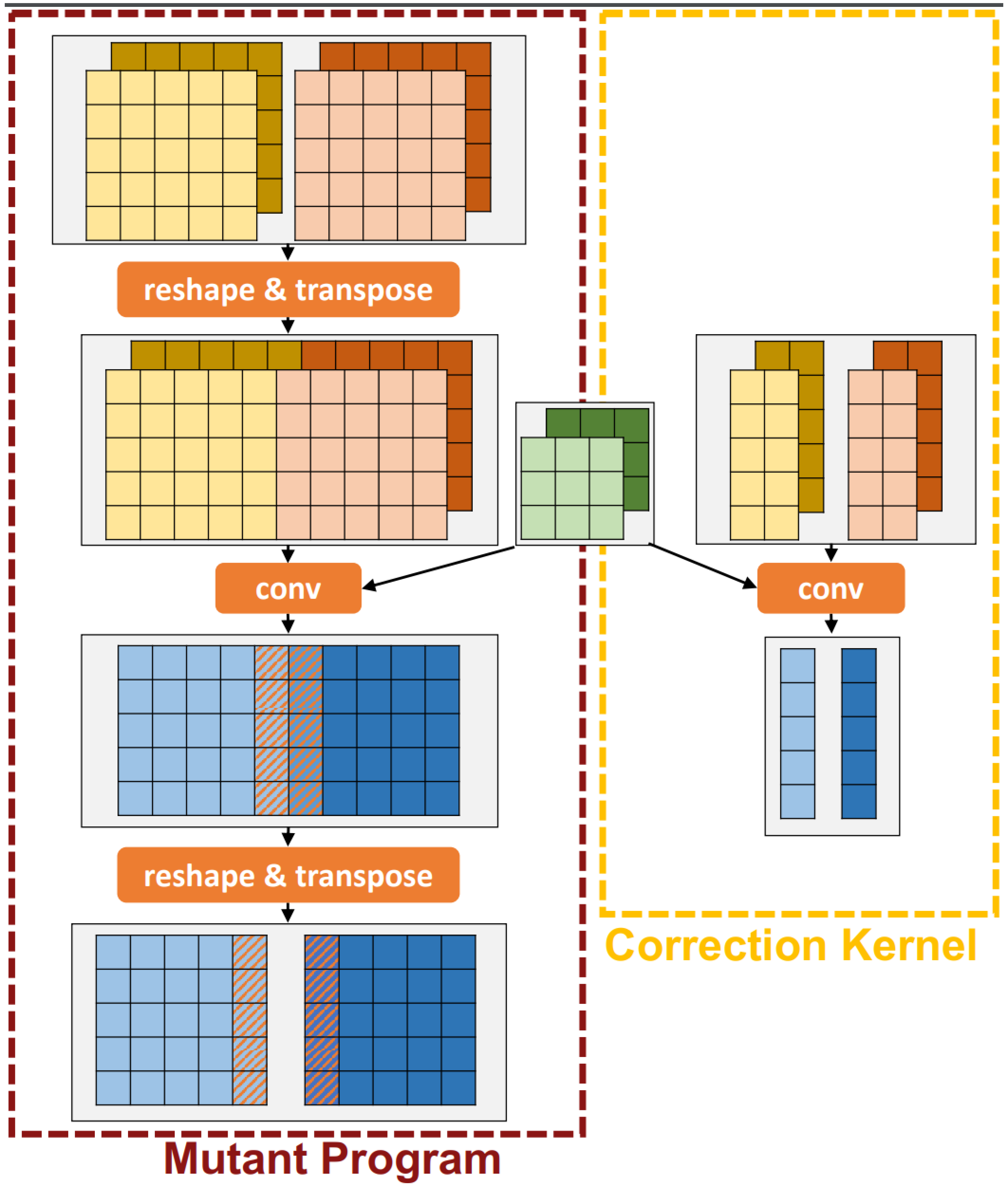
Correct the Mutant

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



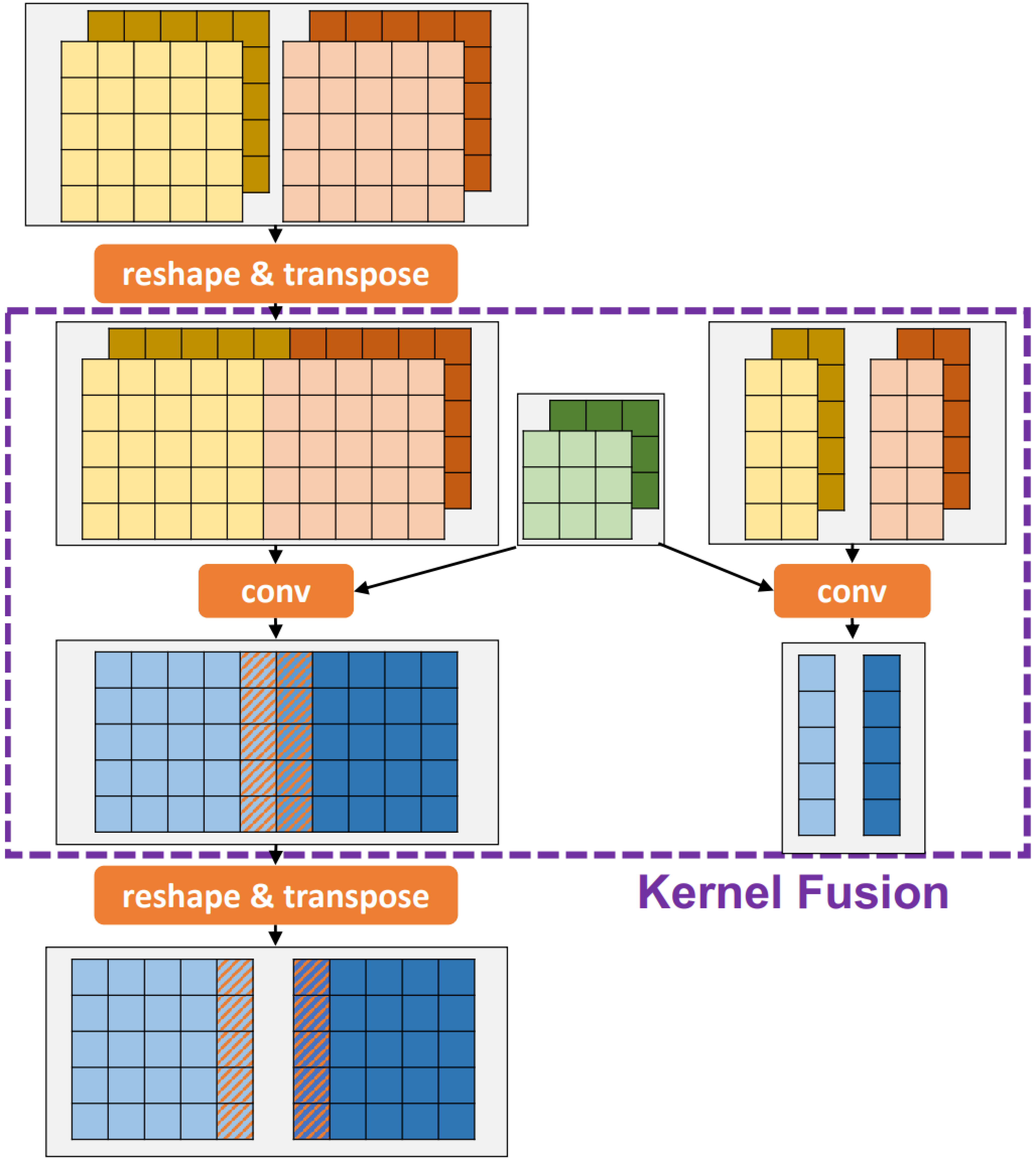
Correct the Mutant

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

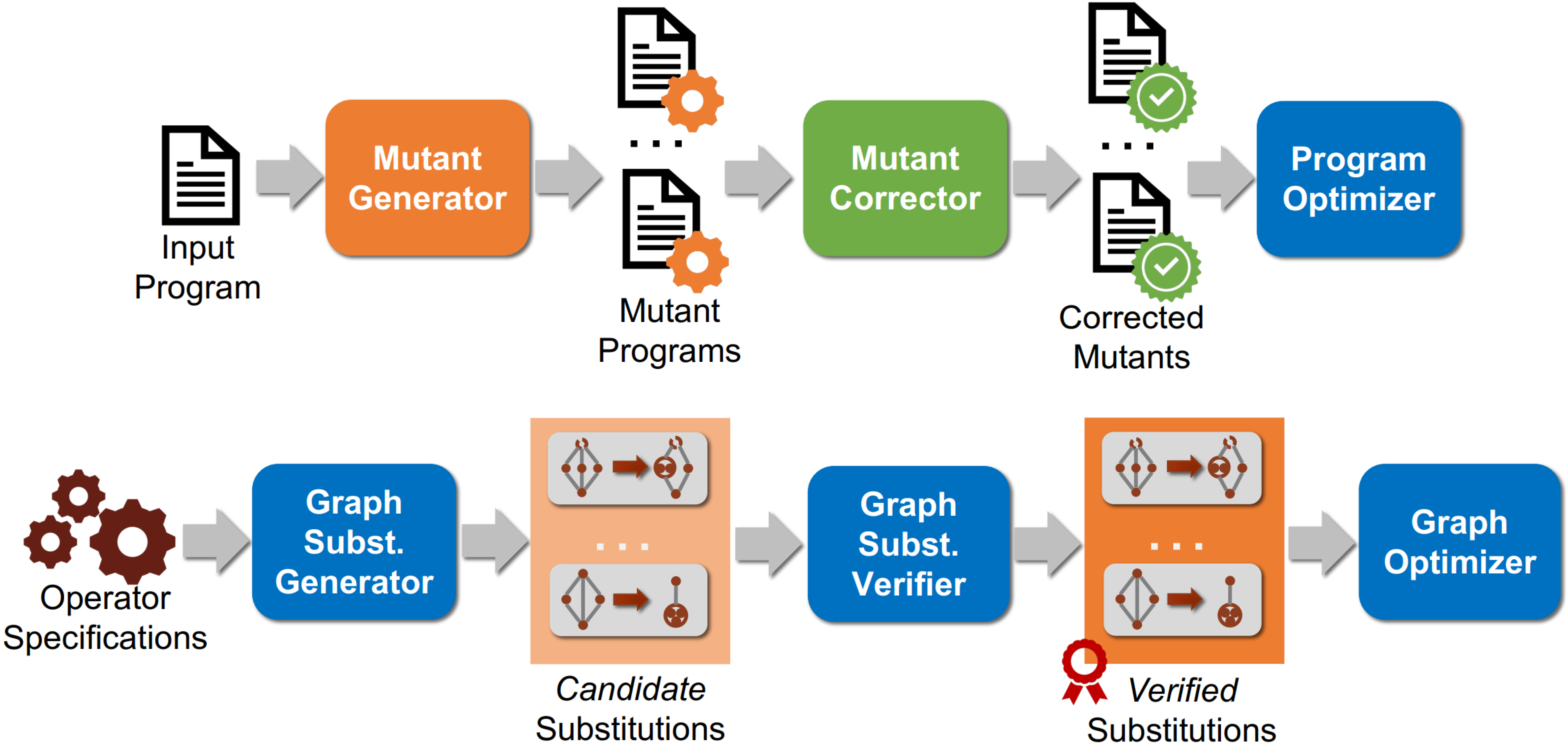


Correct the Mutant

- 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

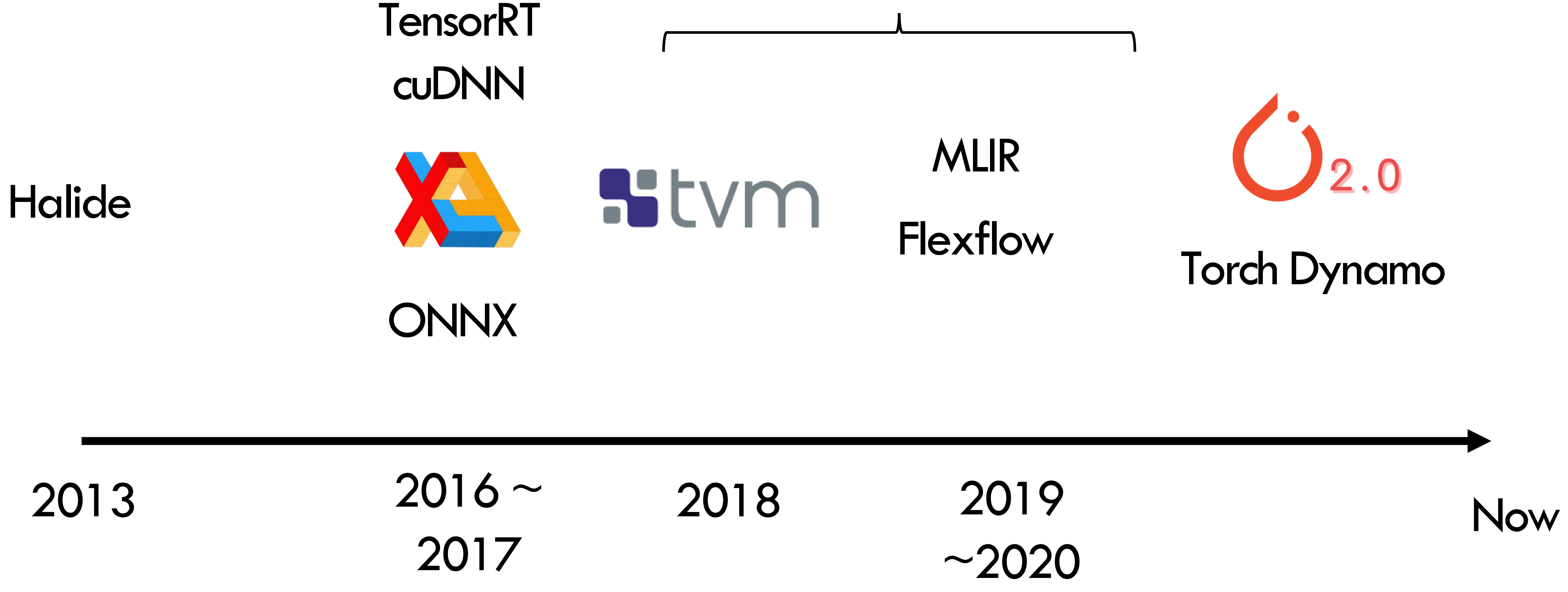
Summary & Questions to discuss

- 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

ML Compiler Retrospective

Q: why the community shifts away from compiler

500+ compiler papers are written during



Halide

TensorRT
cuDNN

ONNX

tvm

MLIR

Flexflow

2.0

Torch Dynamo

2013

2016 ~
2017

2018

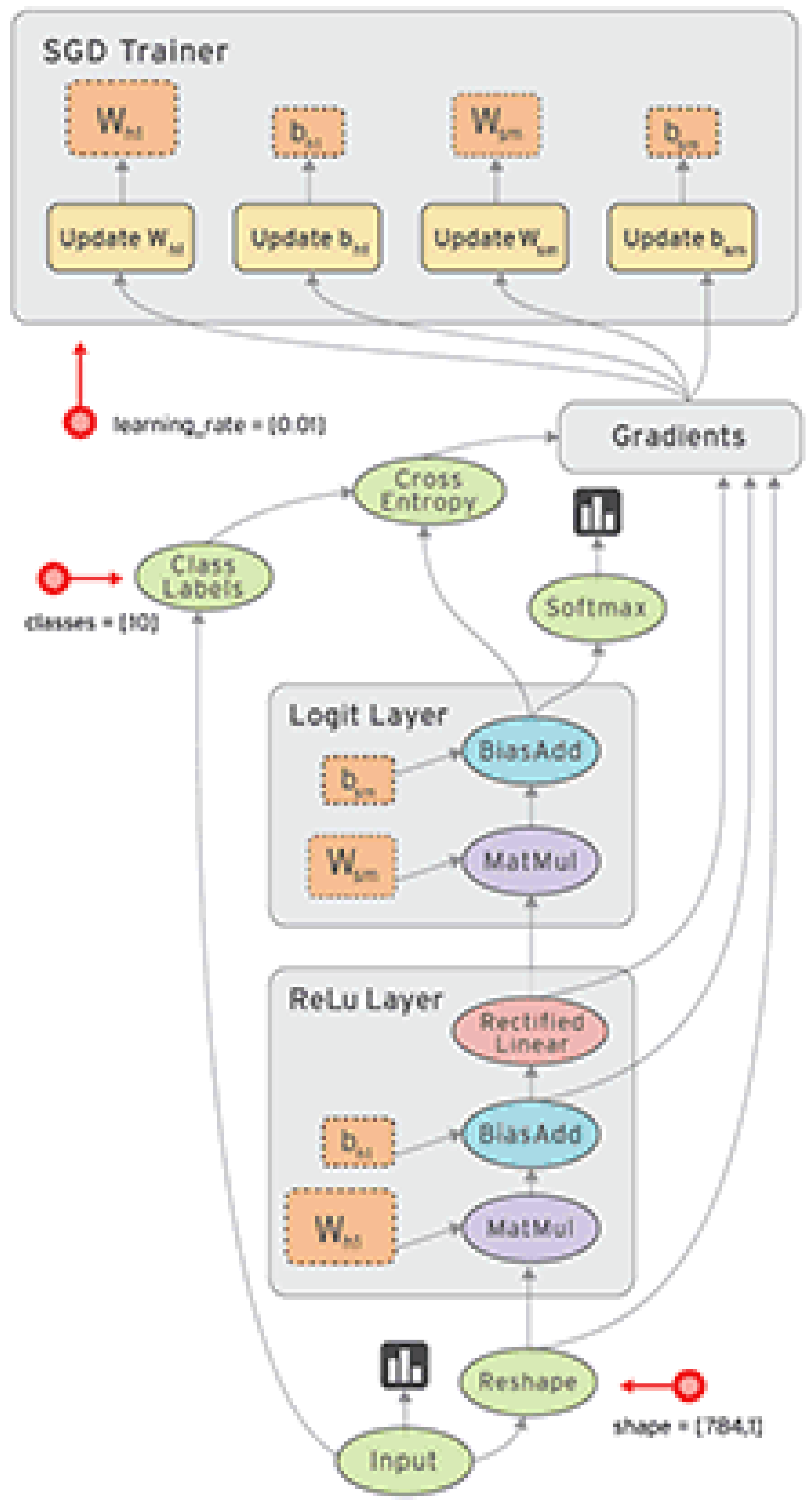
2019
~2020

Now

More Compiler / Graph Optimization

- Guest Speaker: Tianqi Chen
- A.k.a.: GOAT of MLSys
- Inventor of: XGBoost, TVM, MLC-LLM
- Date: Feb. 6
- Topic: Machine Learning Compilation

Big Picture: where are we



Dataflow Graph

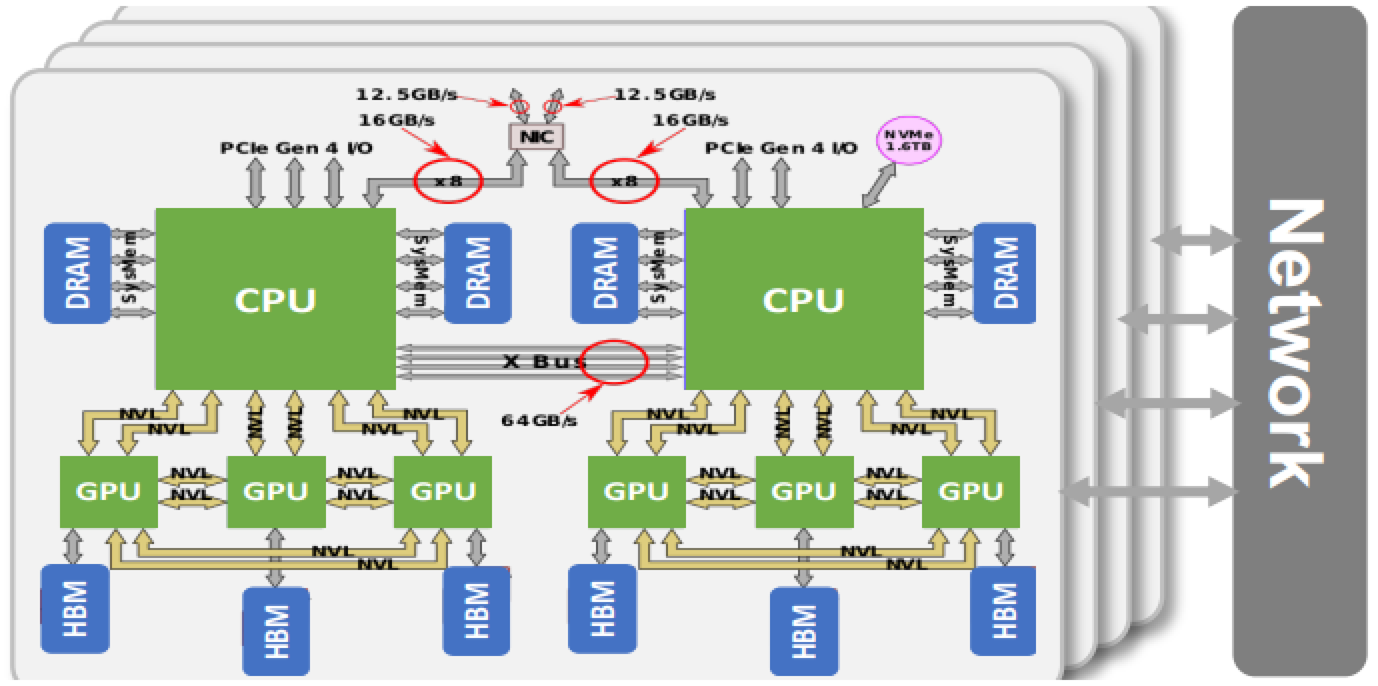
Autodiff

Graph Optimization

Parallelization

Runtime

Operator optimization/compilation



Next: Runtime

- “Batching”
- Checkpointing and rematerialization
- Swapping
- Quantization, Mixed precision, and Pruning