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

CSE 234: Data Systems for Machine Learning
Winter 2025

LLMSys

Optimizations and Parallelization

MLSys Basics

Recap: Last Lecture

- GPU Matmul
- Operator compiler
- Triton

What is a "kernel" in the context of GPUs?

A. A specific section of the CPU used for memory operations.

B. A specific section of the GPU used for memory operations.

C. A type of thread that operates on the GPU.

D. A function that is executed simultaneously by tens of thousands of threads on GPU cores.

What is the function of shared memory in the context of GPU execution?

A. It’s HBM
B. It’s used to store all the threads in a block.
C. It can be used to “cache” data that is used by more than one thread, avoiding multiple reads from the global memory.
D. It’s used to store all the CUDA cores.

What is the significance of over-subscribing the GPU?

A. It reduces the overall performance of the GPU.

B. It ensures that there are more blocks than SMPs present on the device, helping to hide latencies and ensure high occupancy of the GPU.

C. It leads to a memory overflow in the GPU.

D. It ensures that there are more SMPs than blocks present on the device.

Which of the following is True about GPU Memory

A. On H100, a CPU process can access an array stored on H100 GPU memory

B. A thread in a threadblock can access its threadblock-level shared memory

C. Pinned memory is a part of memory allocated on GPU

D. print(a) function in C++ can print an array allocated via a = cudaMalloc(..)

Which of the following operations is most likely to be limited by arithmetic operations?

A. ReLU Activation
B. Linear layer (8192 outputs, 2048 inputs, batch size 1)
C. Batch normalization
D. Max pooling (3x3 window and unit stride)
E. Layer normalization
F. Linear layer (2048 outputs, 1024 inputs, batch size 512)

When picking a tile size for GEMM, why not always pick the biggest tile size?

A. The tile might not fit on the GPU HBM for some GEMM sizes

B. The bigger size could result in low parallelism for some GEMM sizes

C. Larger tiles have lower data reuse

D. Larger tiles means more data is read, lowering arithmetic intensity.

Today's Learning Goal

- **High-level DSL for CUDA: Triton**
- Graph Optimization
  - Manual
  - Automatic

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)
```

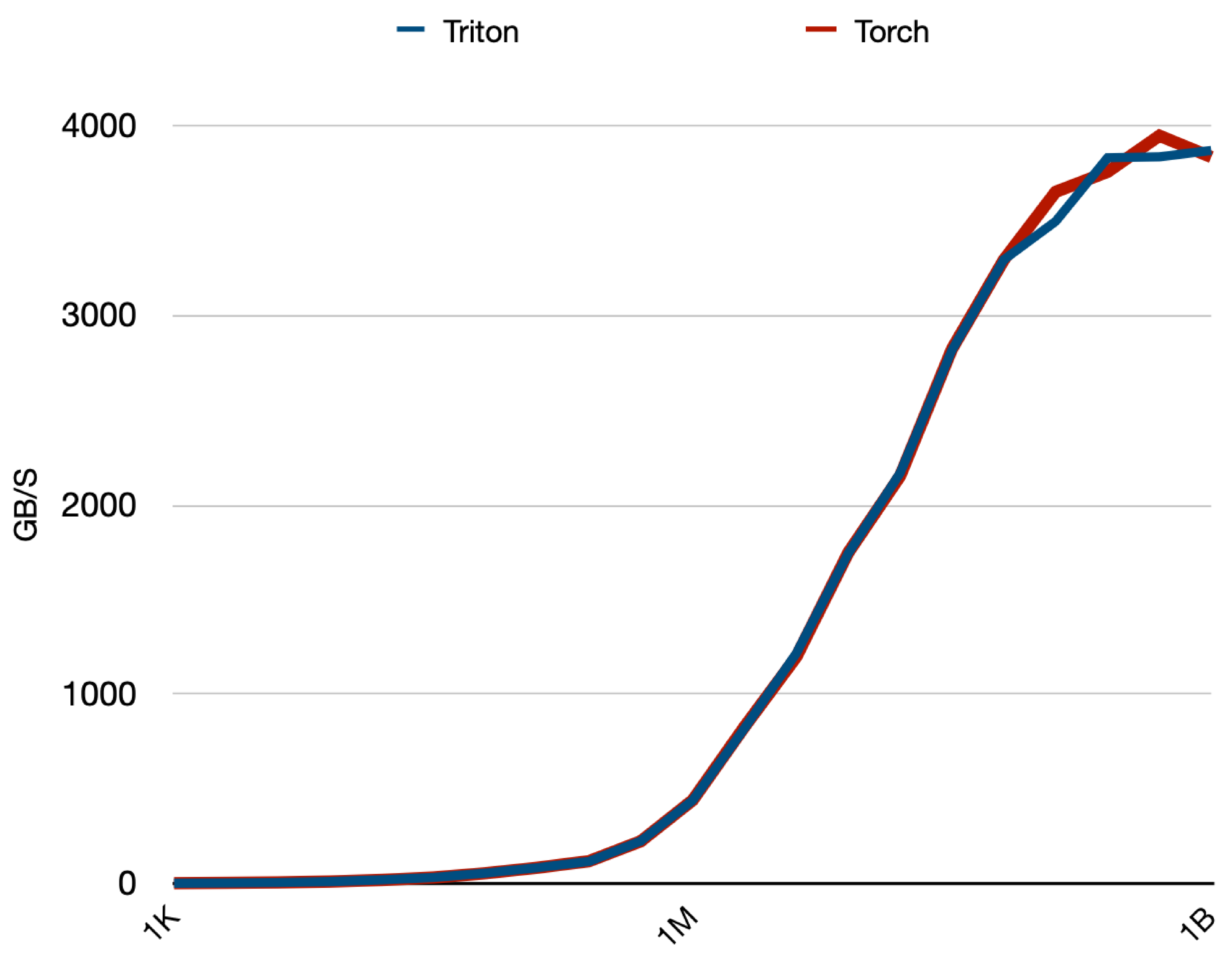
Example: elementwise add v2 (z = x + y)

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)
```

Elementwise Add Performance

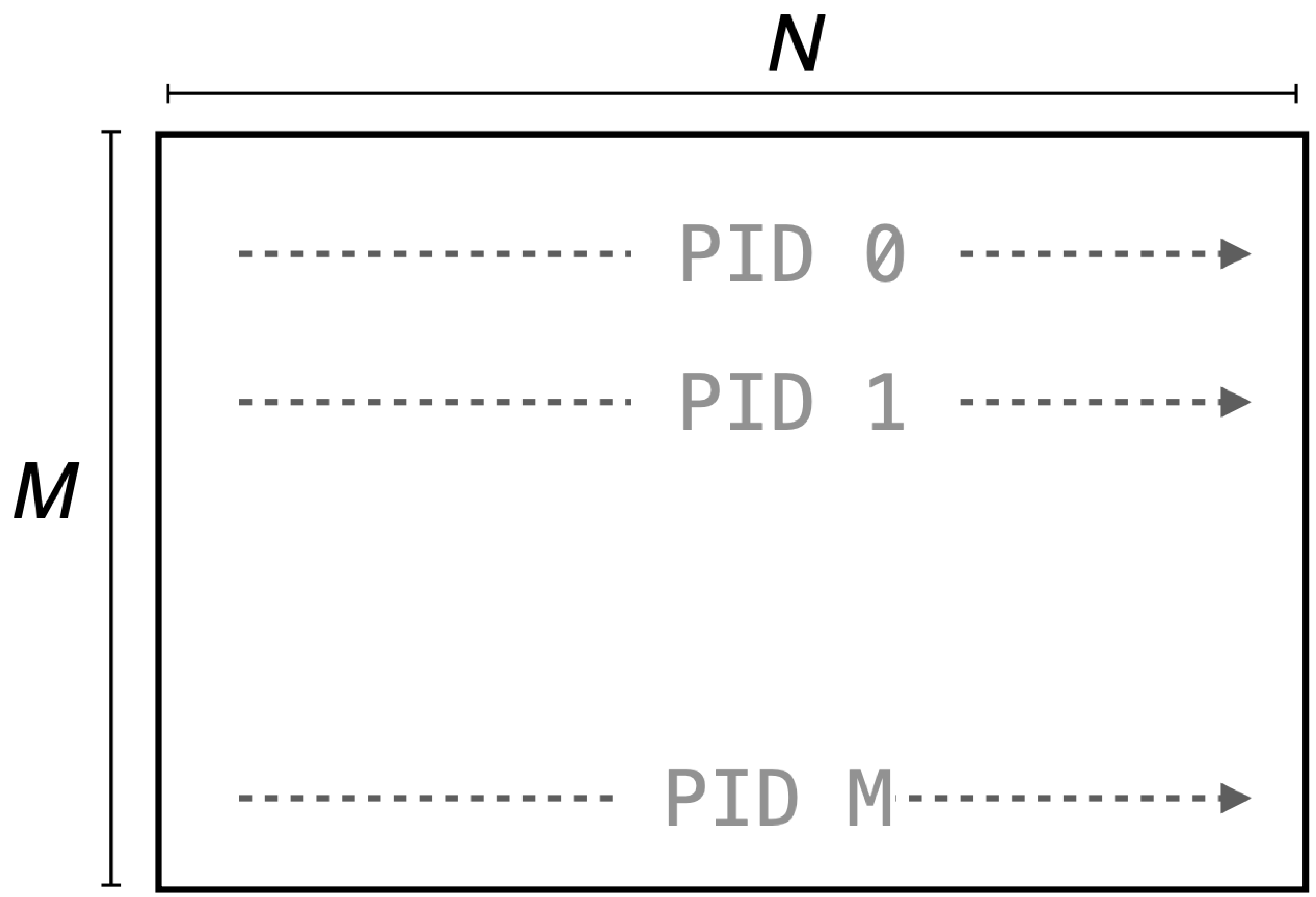


Another Example: Softmax

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

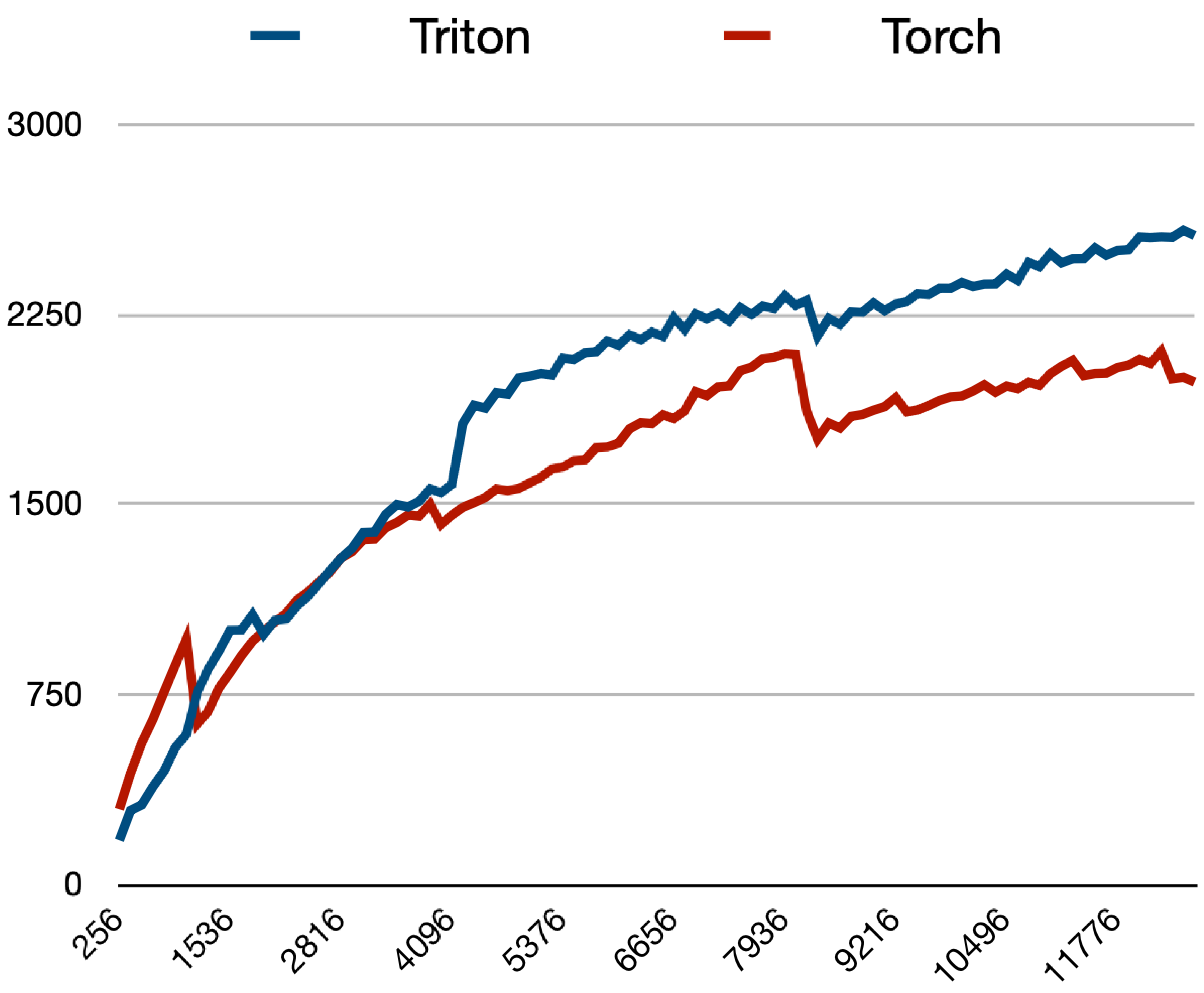
- How did you implement this in PA1?
  - Think about the potential overhead when compose softmax from primitives
- Performant option: implementing an end-to-end softmax kernel
  - Think about the complexity of implementing in CUDA

Triton Example: softmax

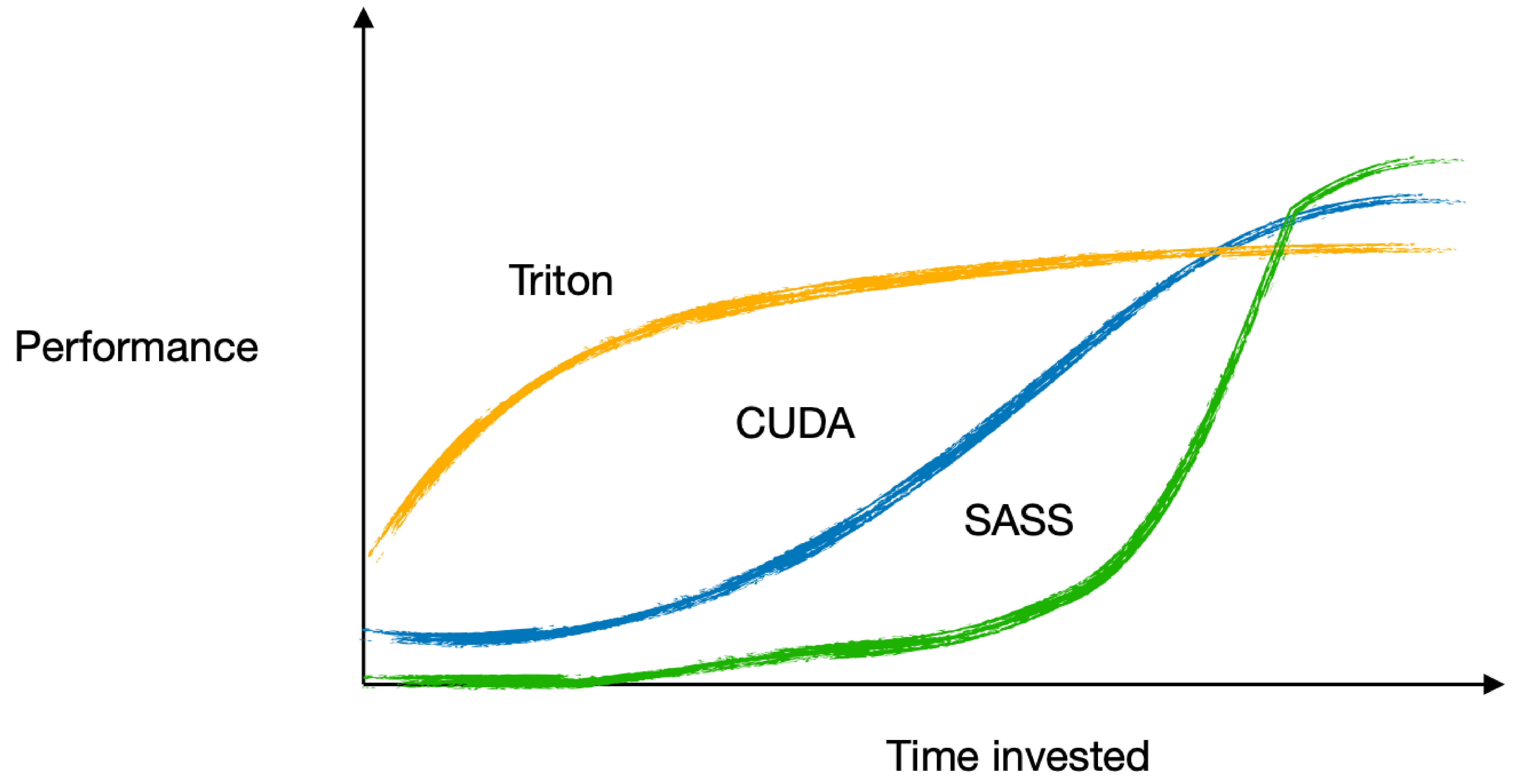



```
import triton.language as tl
Import triton

@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



Why Triton (seemingly) Succeeds



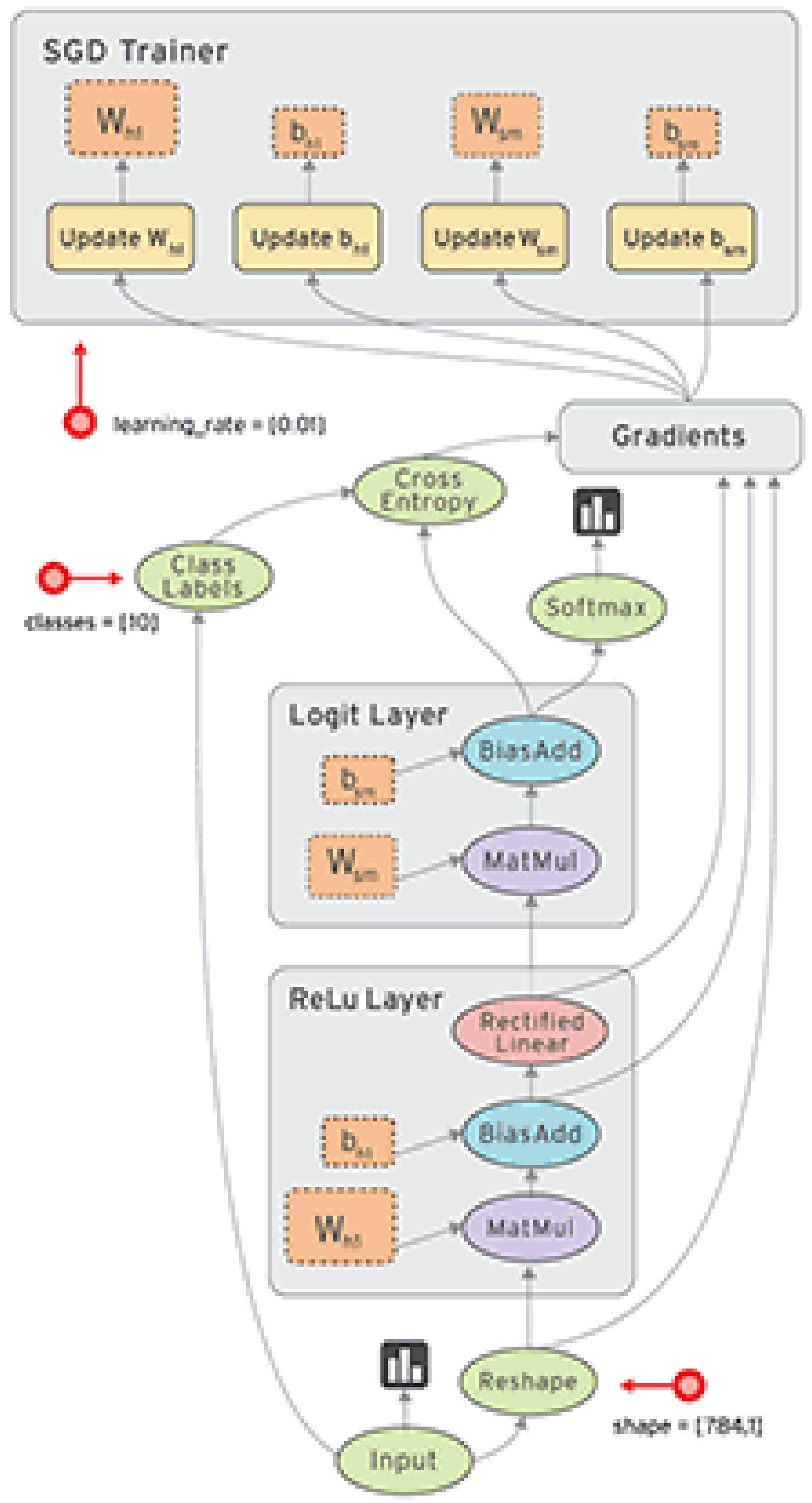
SASS = streaming assembly

Summary: Operator Optimization

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

Next: Graph Optimization



Dataflow Graph

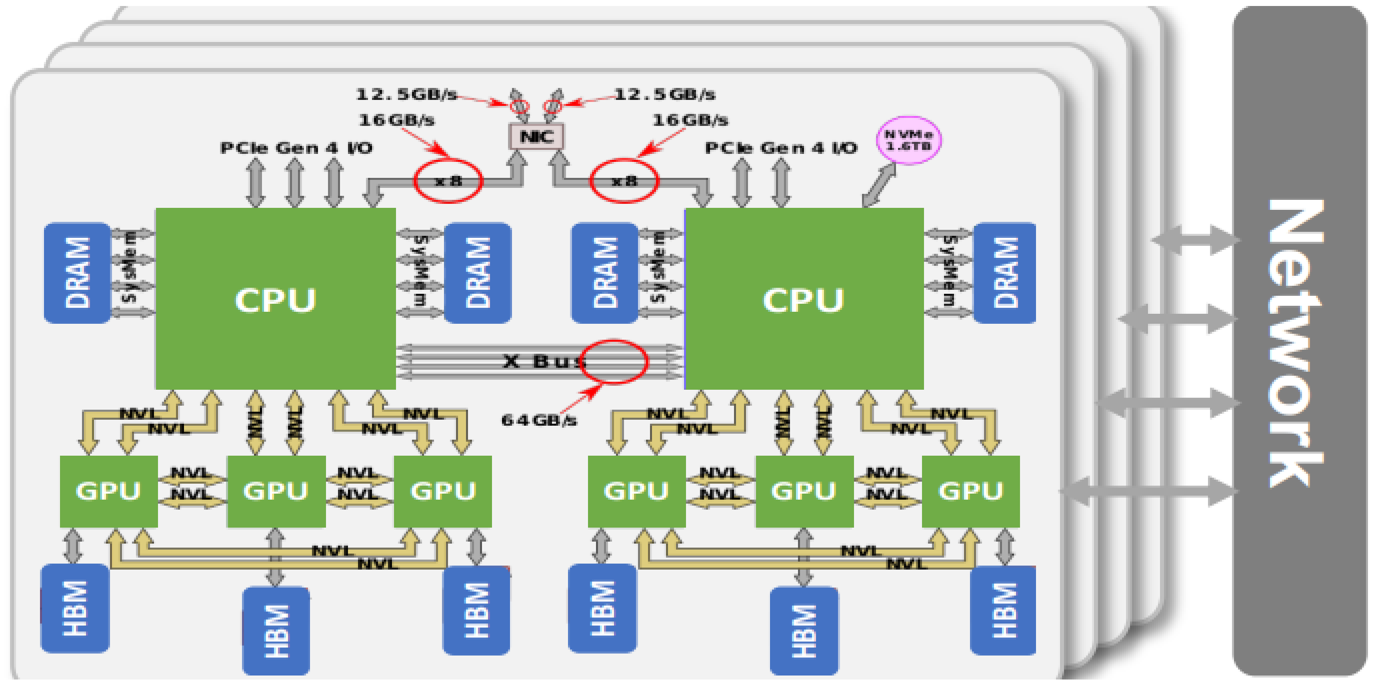
Autodiff

Graph Optimization

Parallelization

Runtime

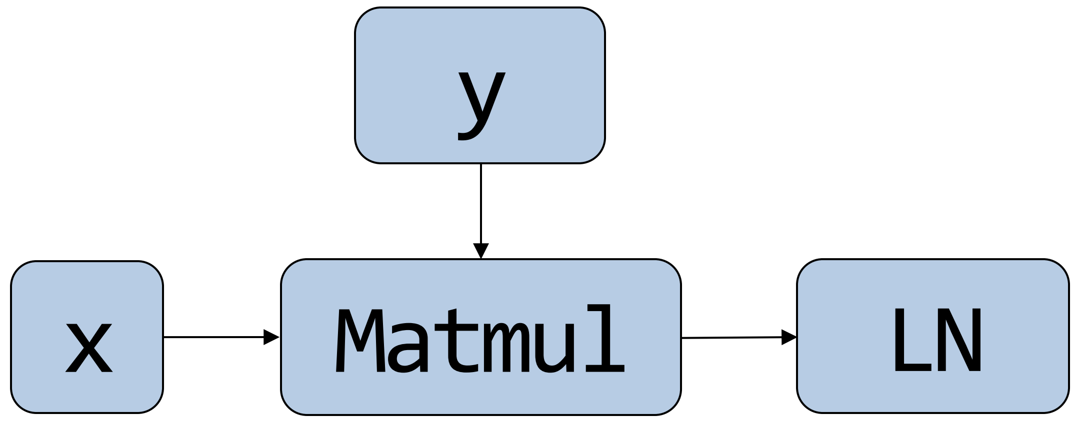
Operator optimization/compilation

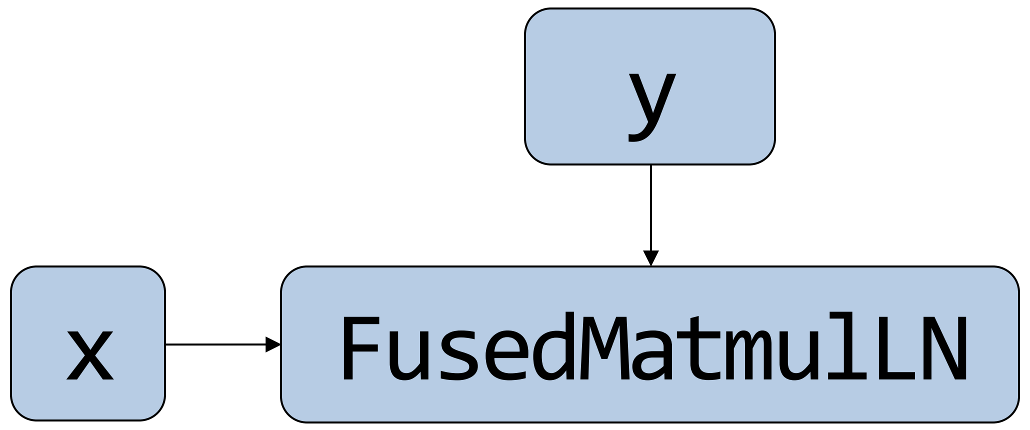


Recall Our Goal

- 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

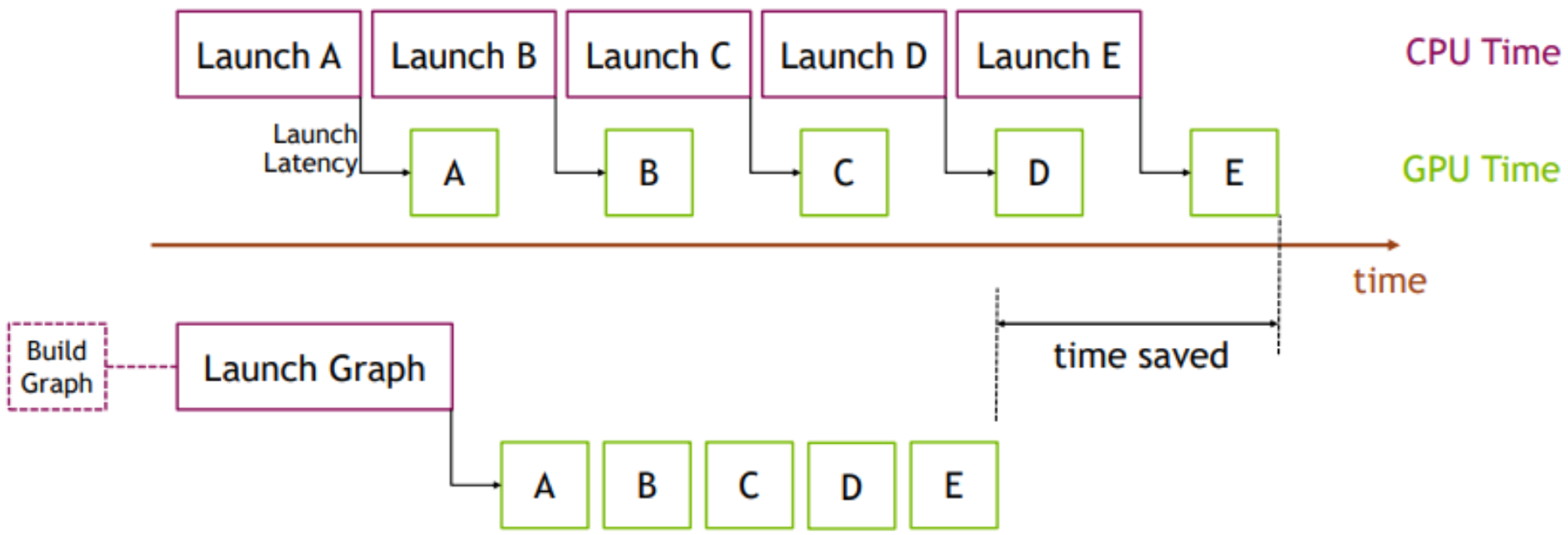




- Why operator fusion improves performance?
  - Reduce I/O
  - Reduce kernel launching
- Cons:
  - Requiring many fused ops: FusedABCOp
  - At some point, codebase becomes unmanageable

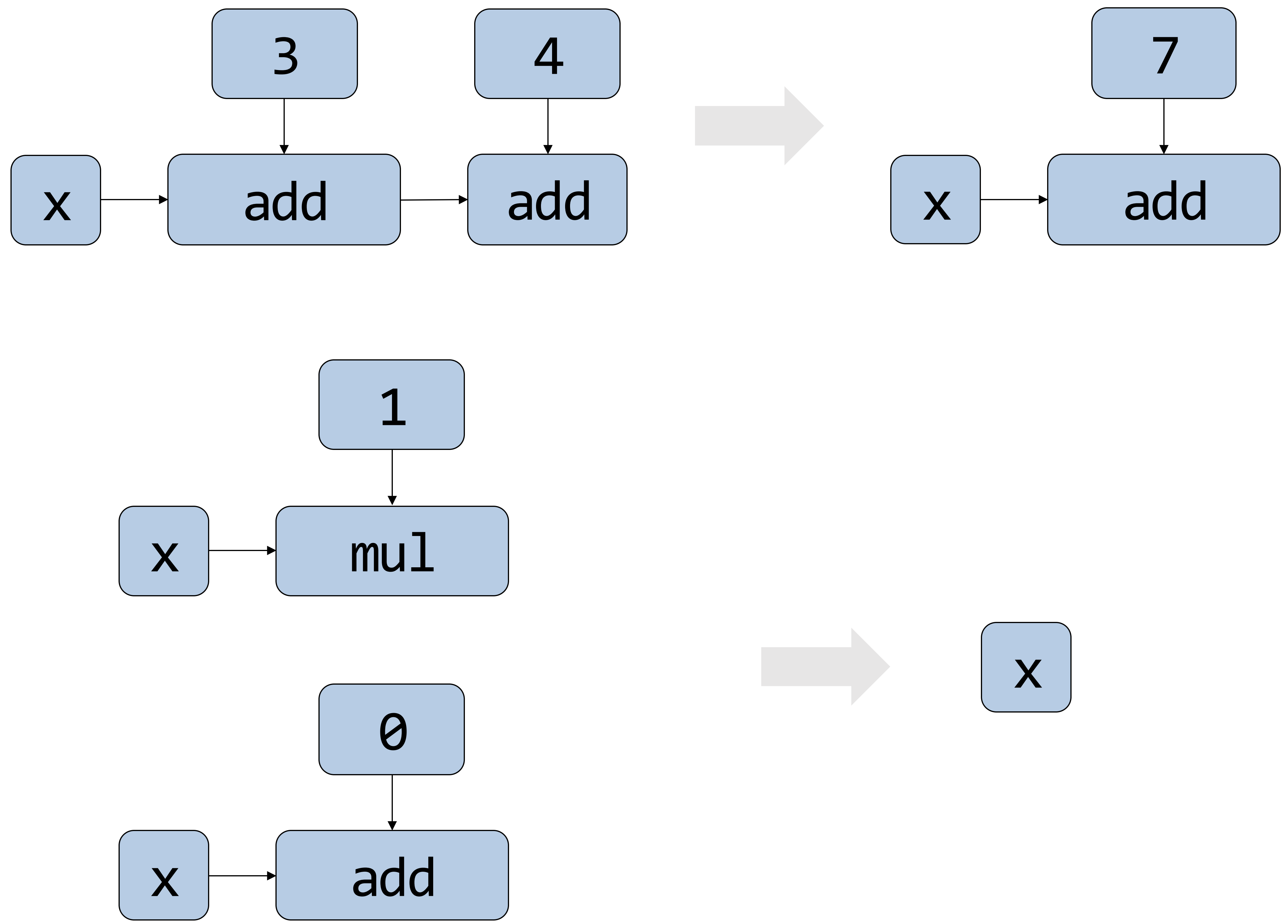
One trade-off in Practice results in "CUDA Graph"

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



More reading: https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/

Graph Optimization Templates: Constant Folding



Common Subexpression Elimination (CSE)

$$\begin{aligned} &\ldots \\ c &= a + b \\ d &= a \\ e &= b \\ f &= d + e \\ d &= x \\ &\ldots. \end{aligned}$$

$$\begin{aligned} &\ldots \\ c^3 &= a^1 + b^2 \\ d^1 &= a^1 \\ e^2 &= b^2 \\ &\sout{f^3 = d^1 + e^2} \\ f^3 &= c^3 \\ d^4 &= x^4 \\ &\ldots. \end{aligned}$$

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

…

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

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

$e^2 = b^2$

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

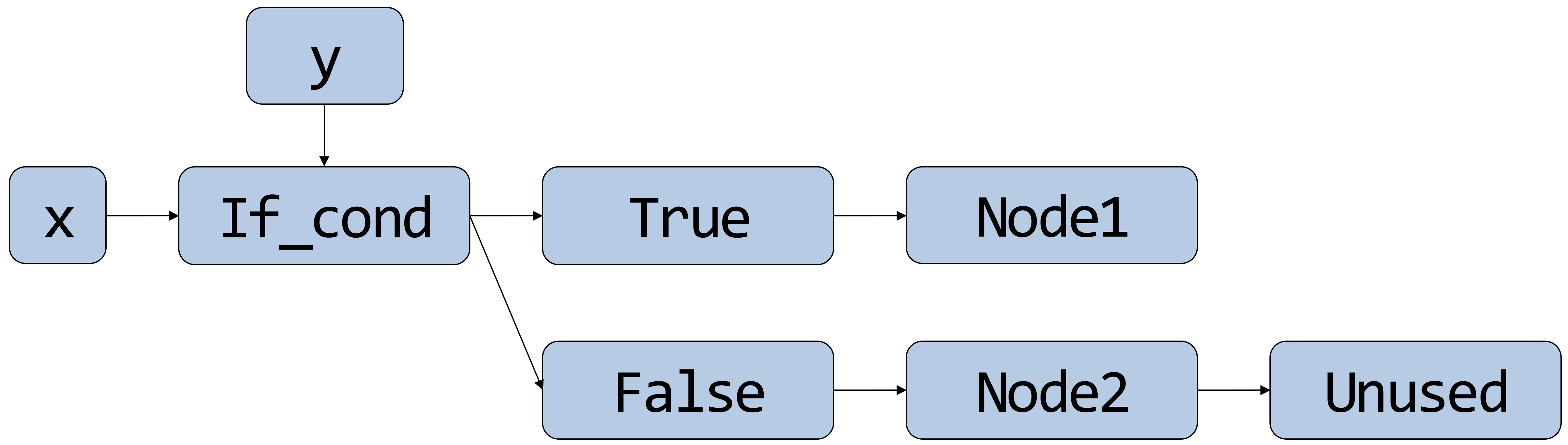
$f^3 = c^3$

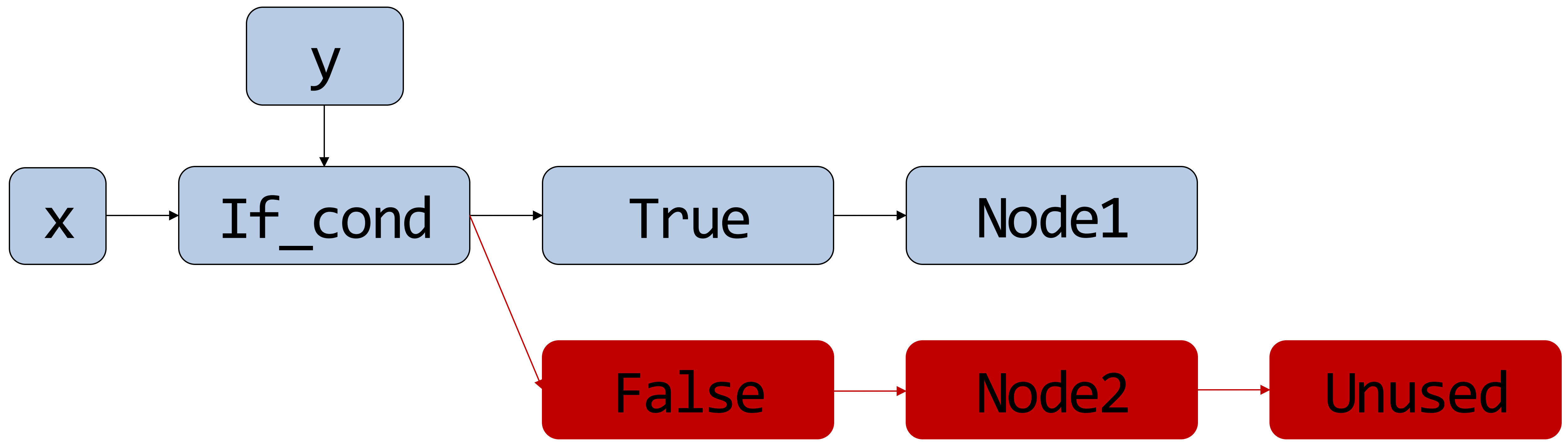
$d^4 = x^4$

….

DCE hit

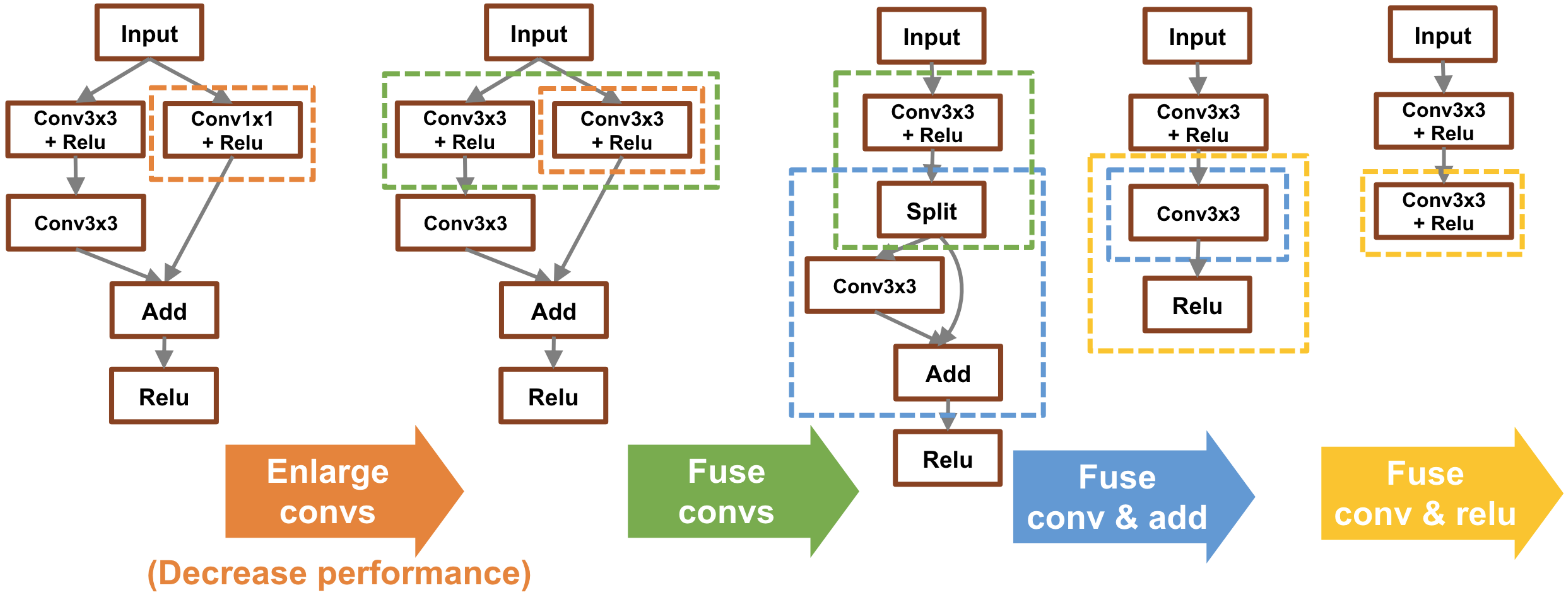
More templates for CSE and DCE





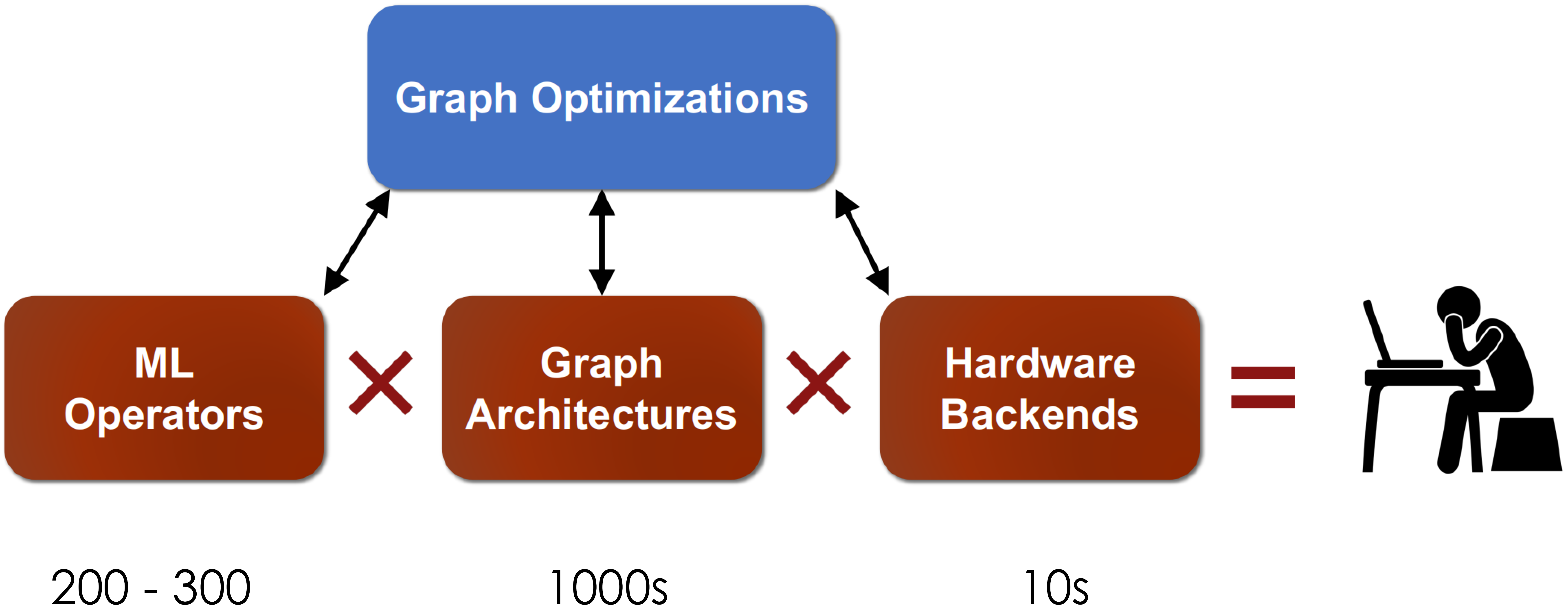
How to ensure performance gain?

- Greedily apply graph optimizations



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

Problems of Template-based Graph Optimizations



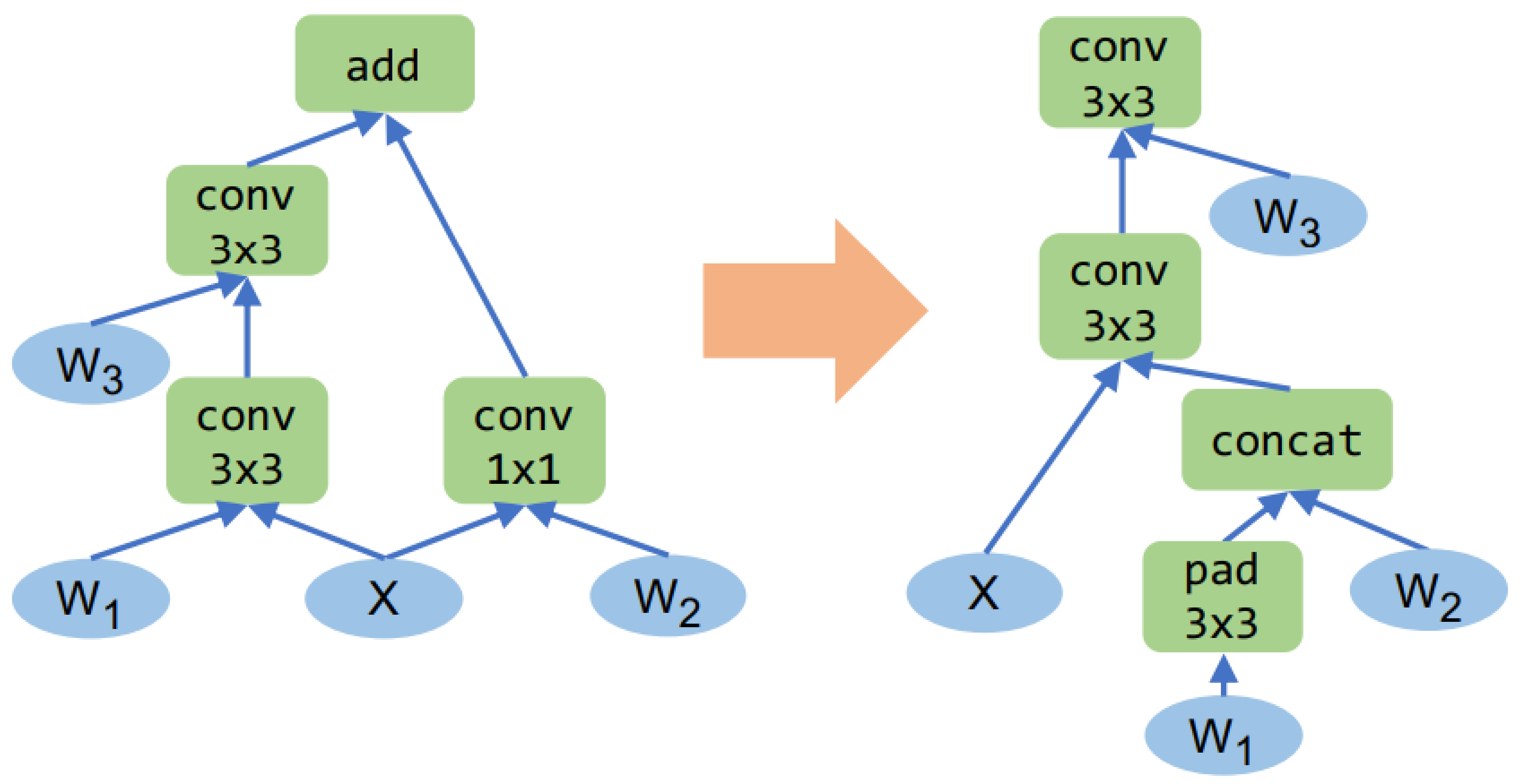
Problem: Infeasible to manually design graph optimizations for all cases

Problems of Template-based Graph Optimizations

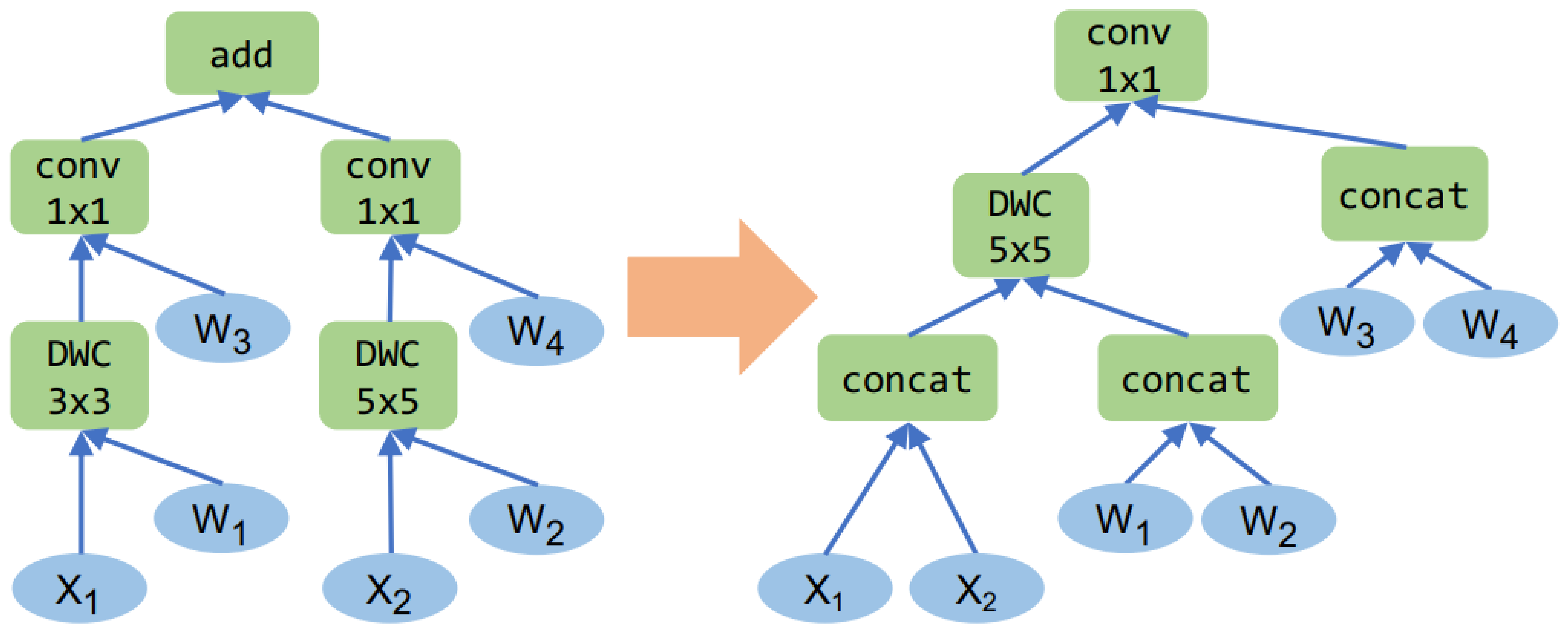
Robustness
Experts' heuristics do not apply to all DNNs/hardware

Scalability
New operators and graph structures require more rules

Performance
Miss subtle optimizations for specific DNNs/hardware



Only apply to **specific hardware**



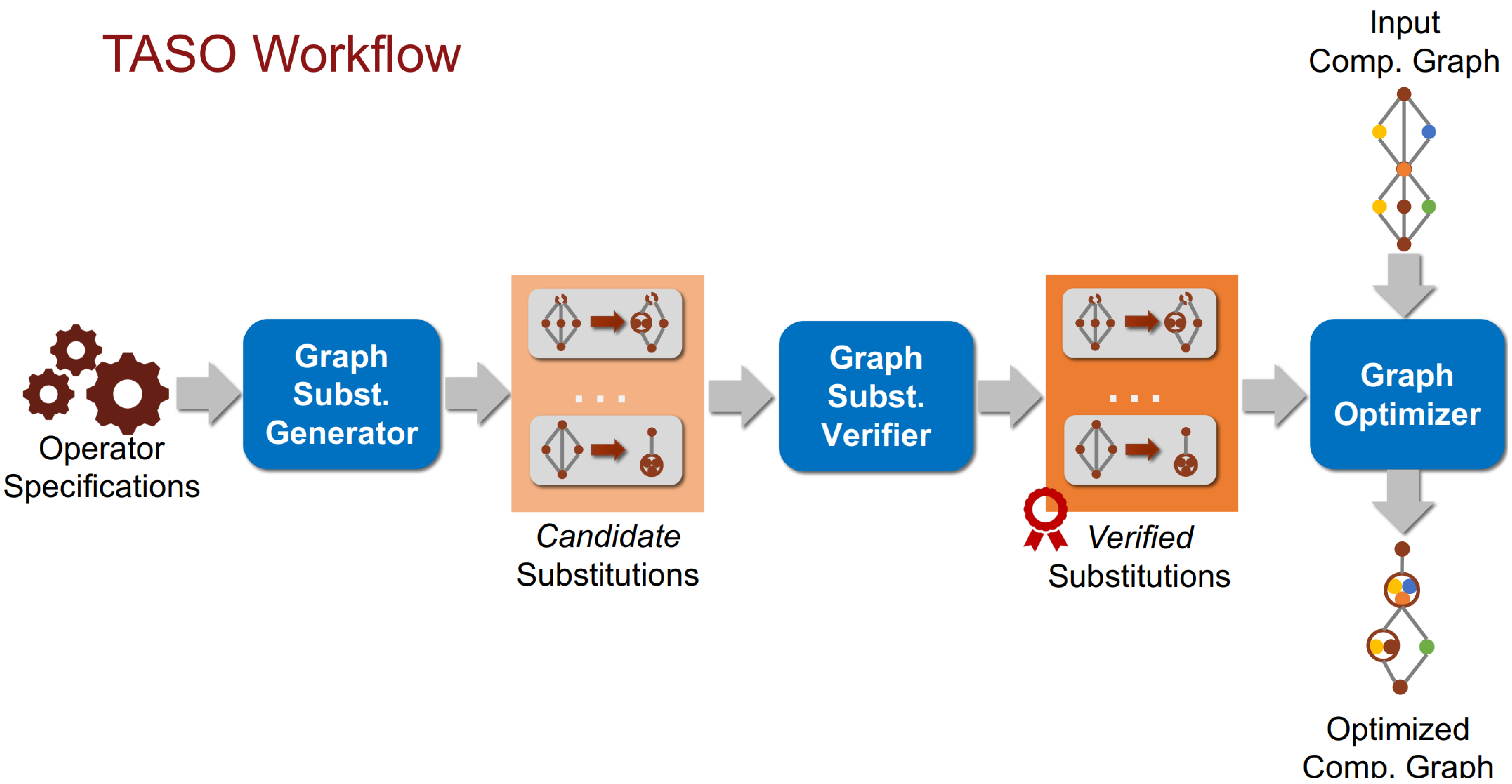
Only apply to **specialized graph structures**

Automate Graph Transformation

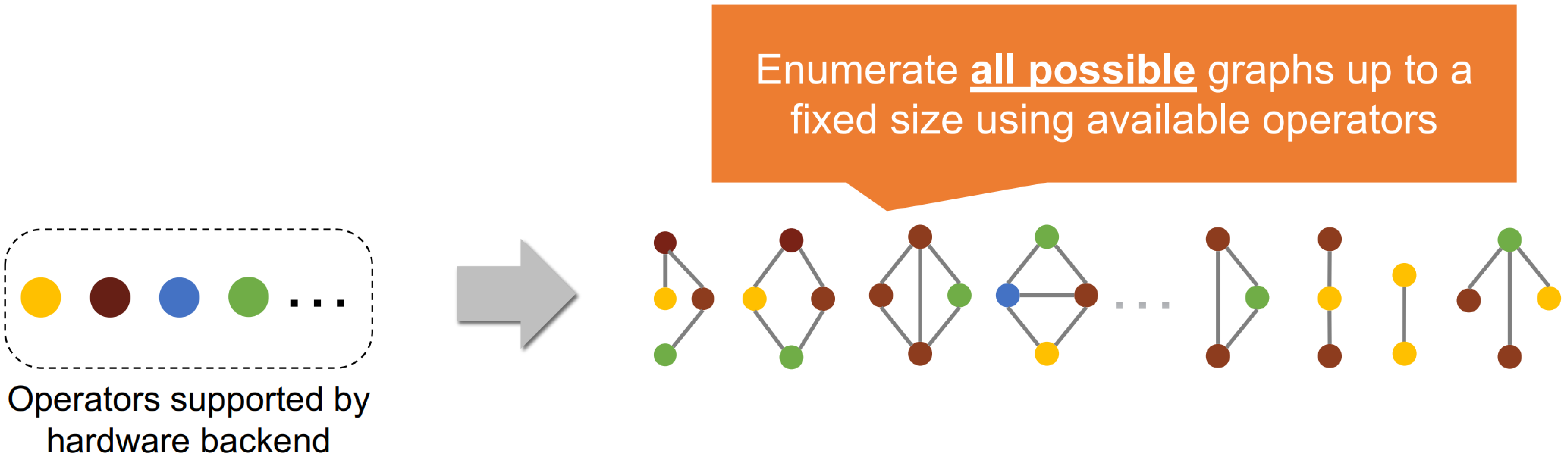
Key idea: replace manually-designed graph optimizations with automated generation and verification of graph substitutions for tensor algebra

Enumerate and Verify ALL possible graph

TASO Workflow

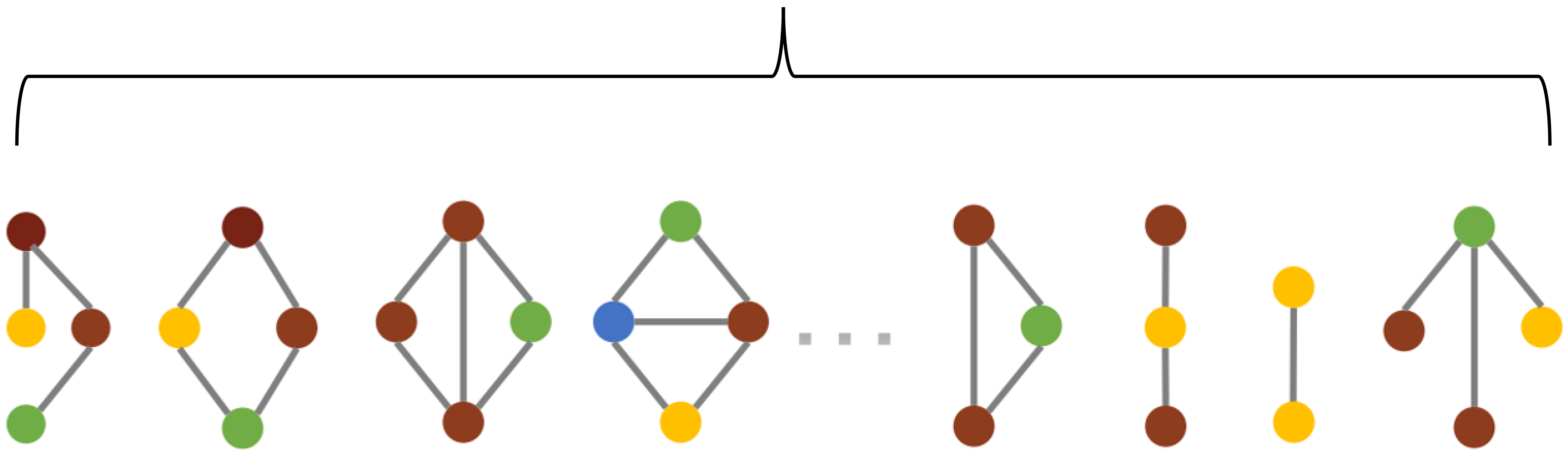


Graph Substitution Generator



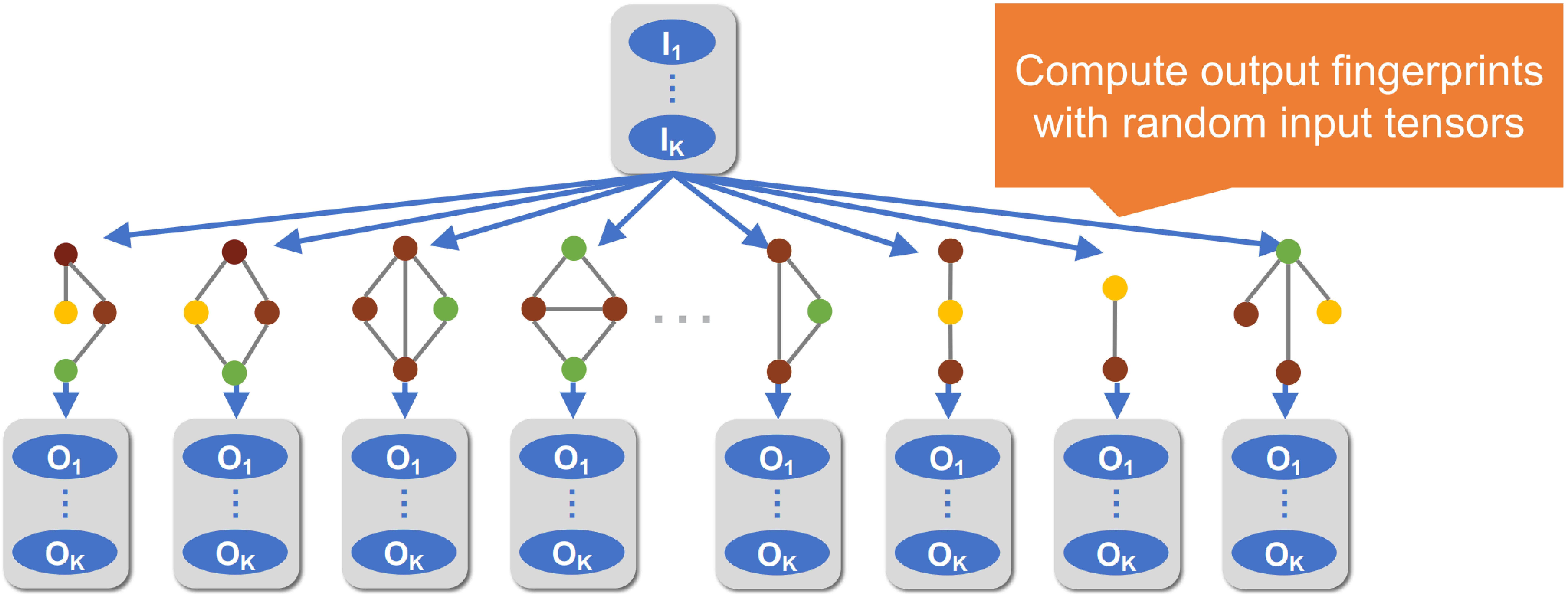
There are many subgraphs even only given 4 Ops

66M graphs with up to 4 operators



A substitution = a pair of equivalent graphs

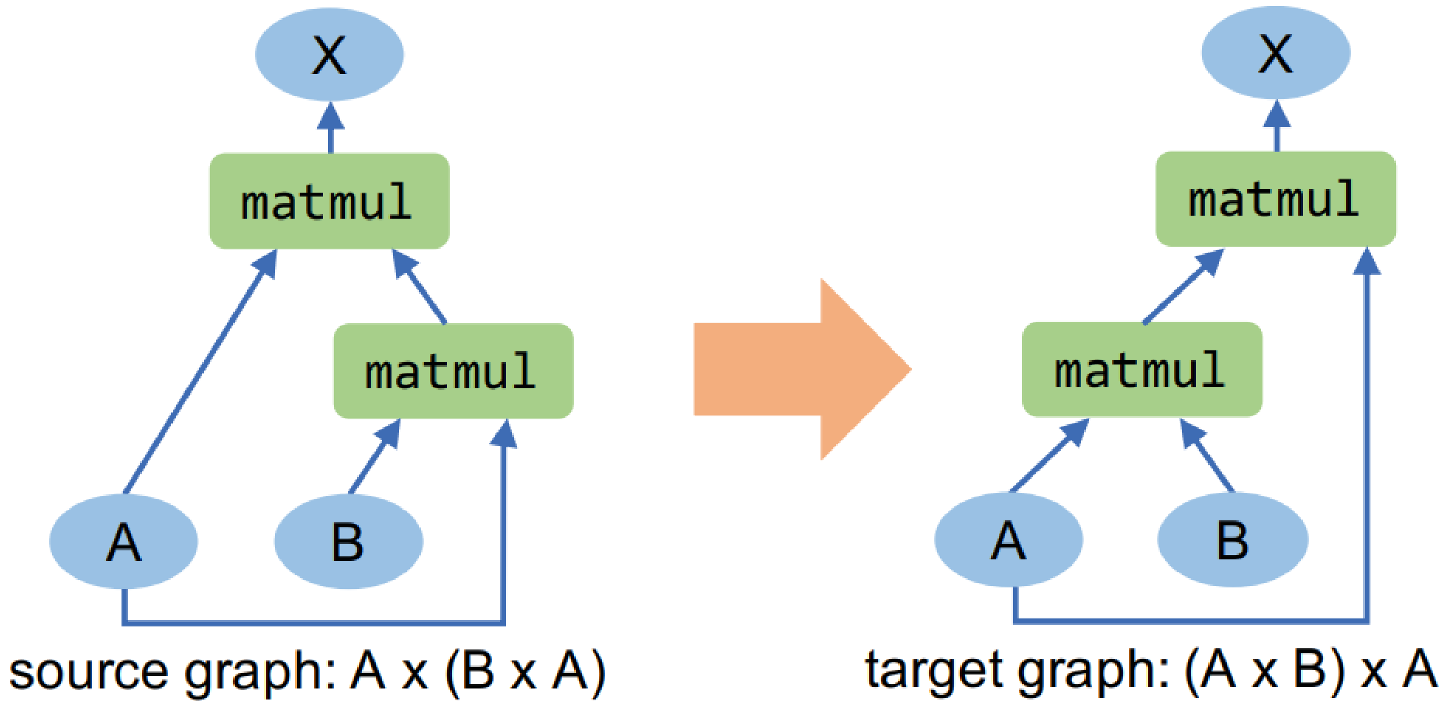
Graph Substitution Generator



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

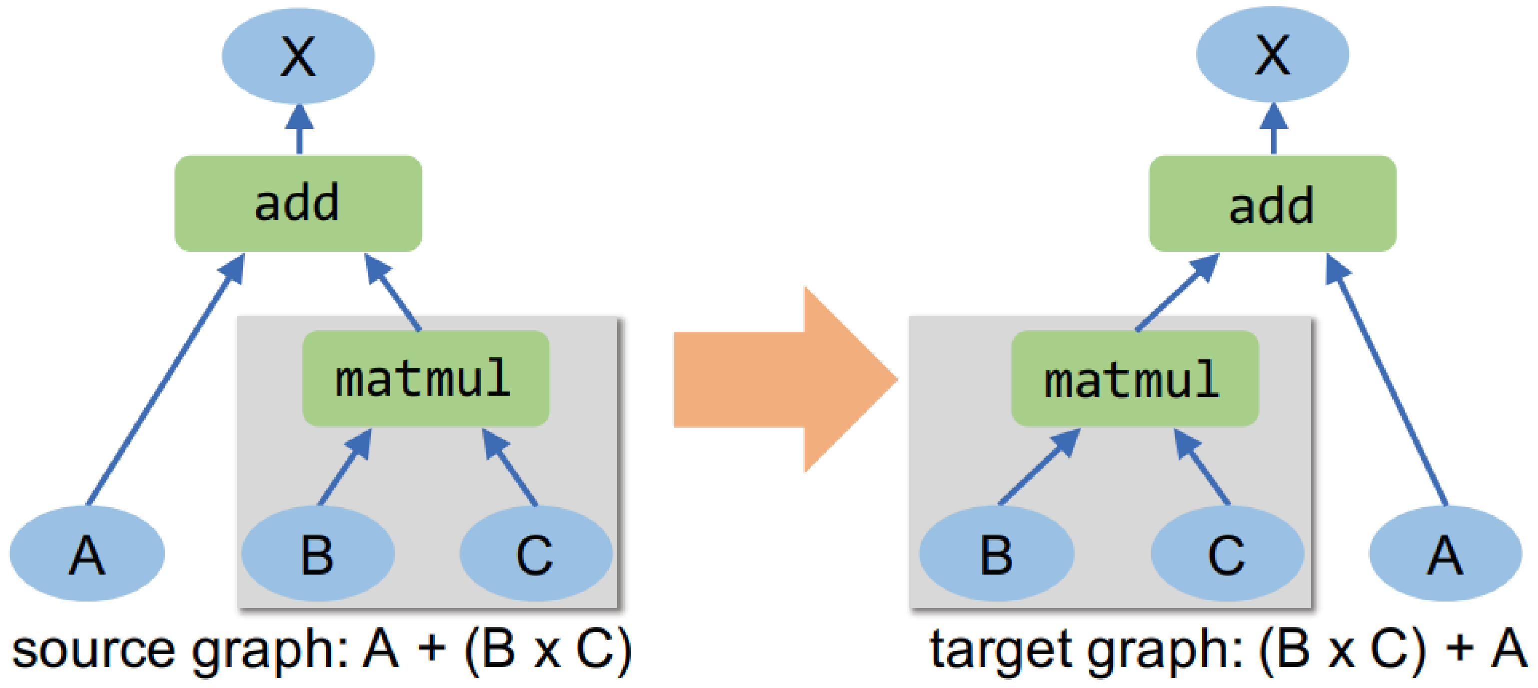
Pruning repeated graphs

28744 substitutions →



Variable renaming

→



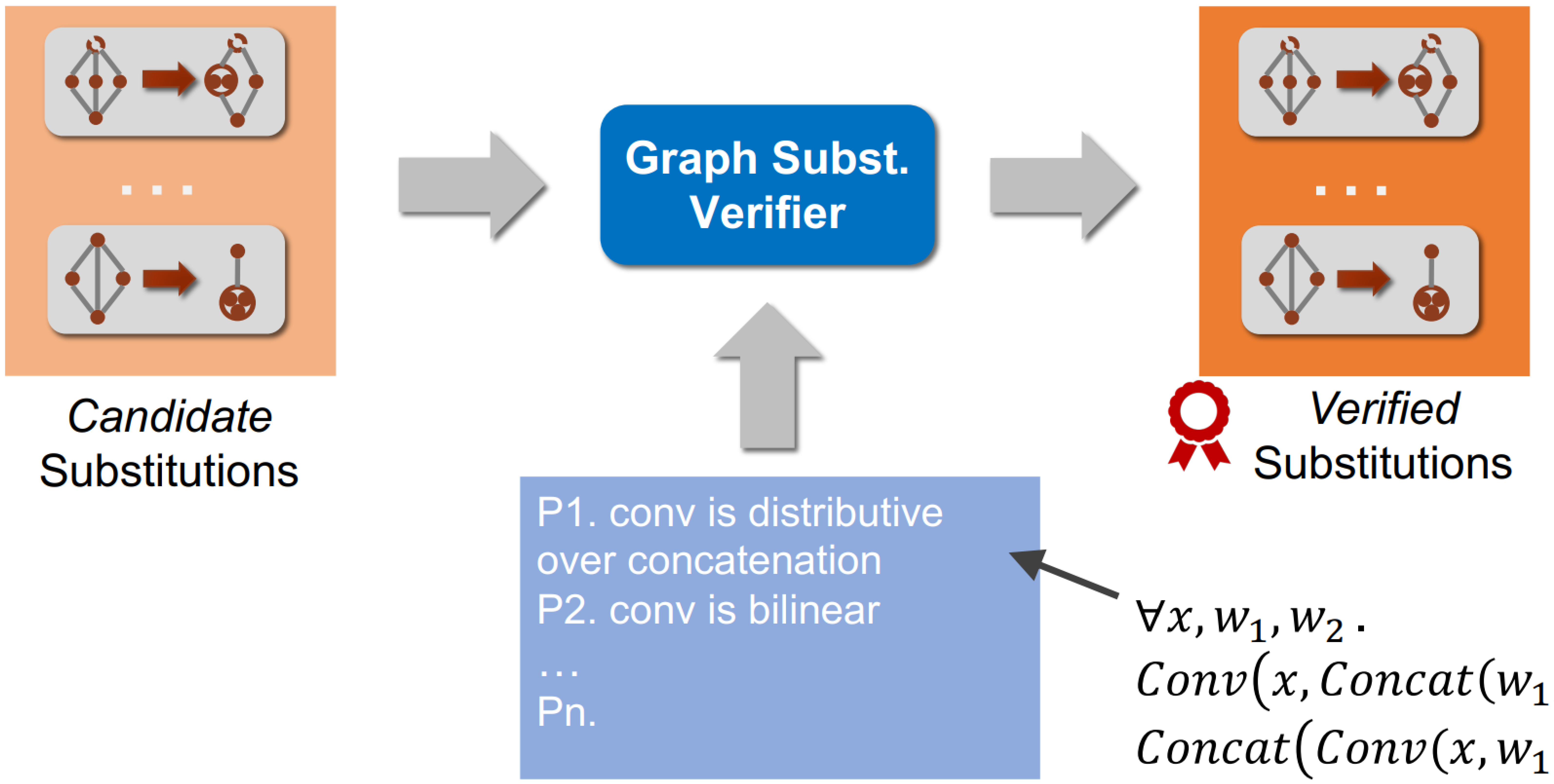
Common subgraph

→ 734 substitutio

Can we trust graph substitutions?

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

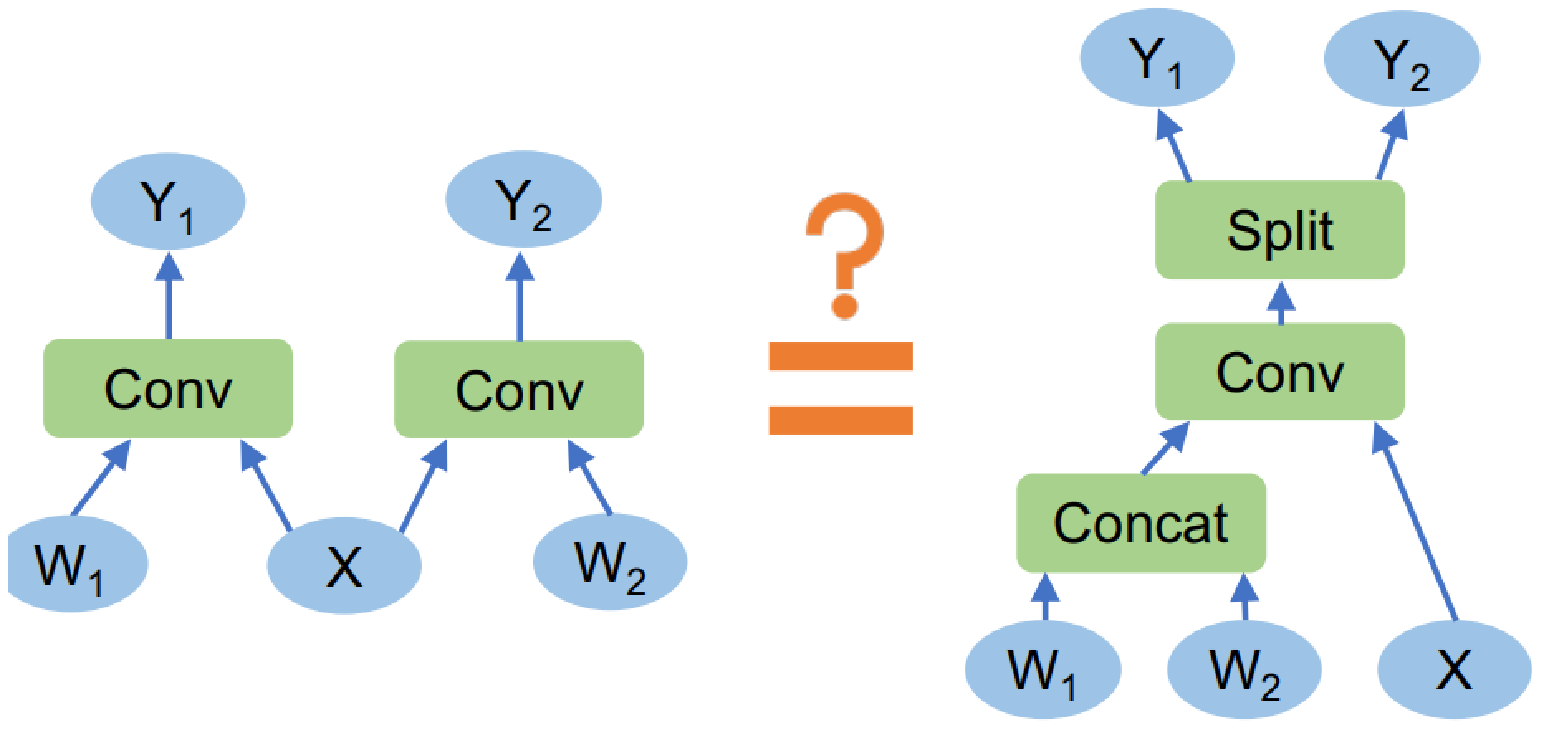
Substitution Verifier



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

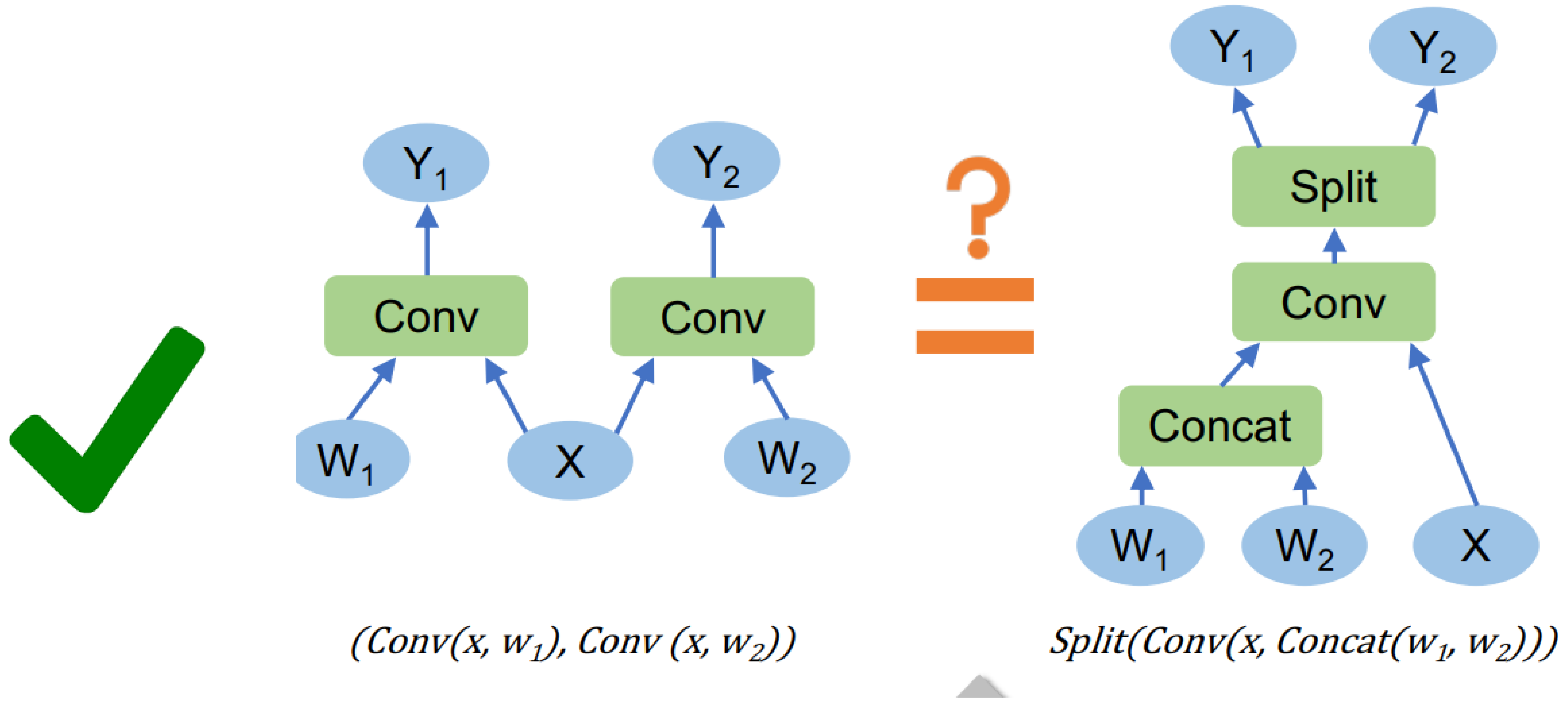
Idea: writing specifications are easier than actually, conducting the optimizations

How to Verify



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

Automated theorem prover



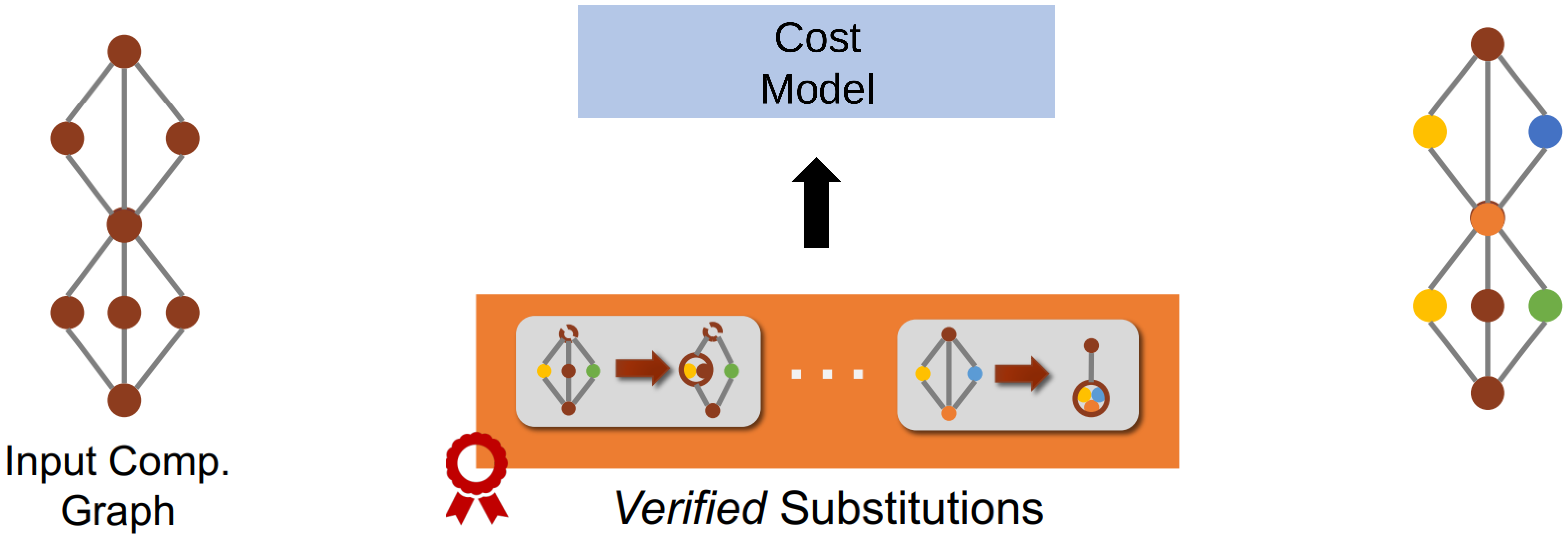
$\forall x, w_1, w_2 .$
$(Conv(x, w_1), Conv(x, w_2))$
$= Split\left(Conv(x, Concat(w_1, w_2))\right)$

P1. $\forall x, w_1, w_2 .$
$Conv(x, Concat(w_1, w_2)) =$
$Concat(Conv(x, w_1), Conv(x, w_2))$
P2. …

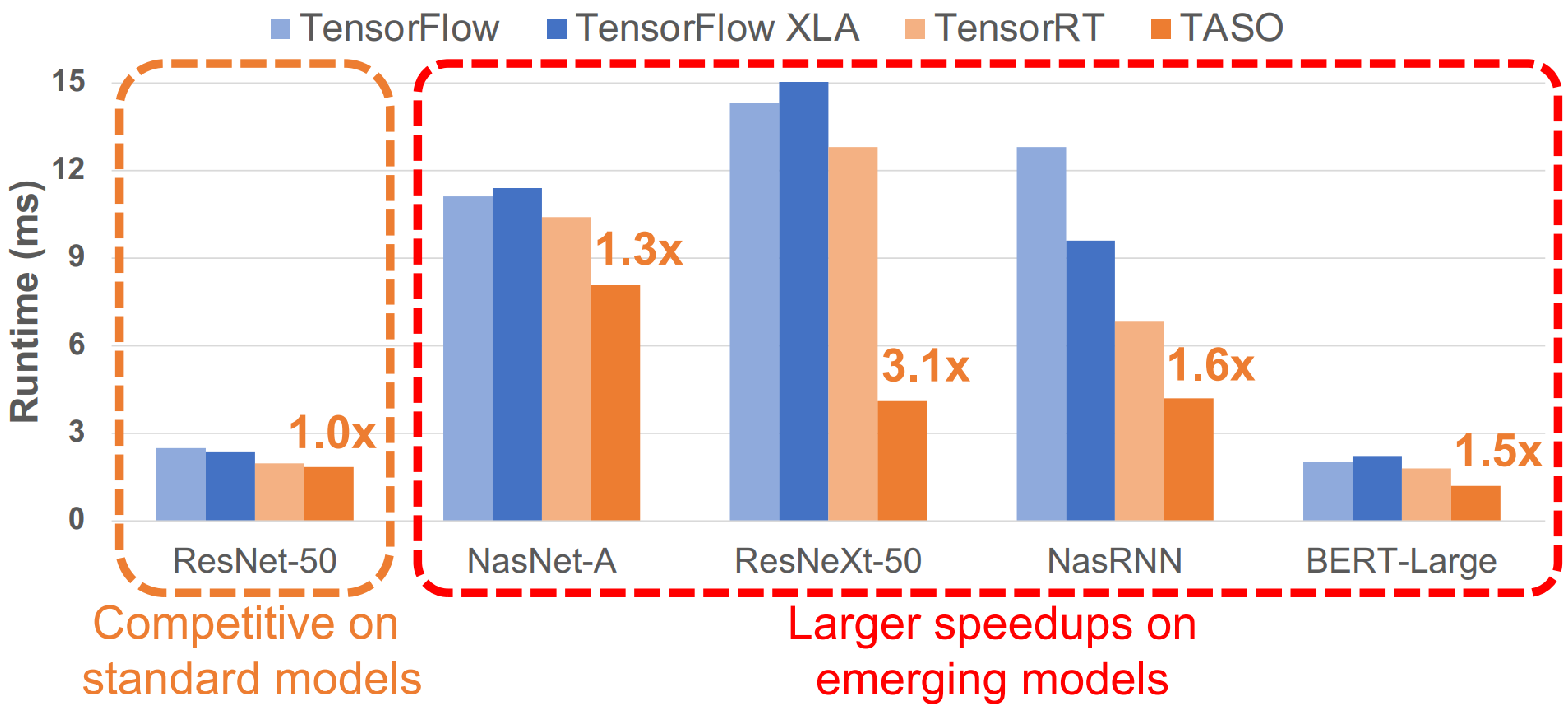
- Generating 743 substitutions = 5 mins
- Verify against 43 op specs = 10 mins
- Supporting a new op requires experts to write specs = 1400 LoC
  - vs. 53K LoC of manual optimization in TF

Incorporating substitutions

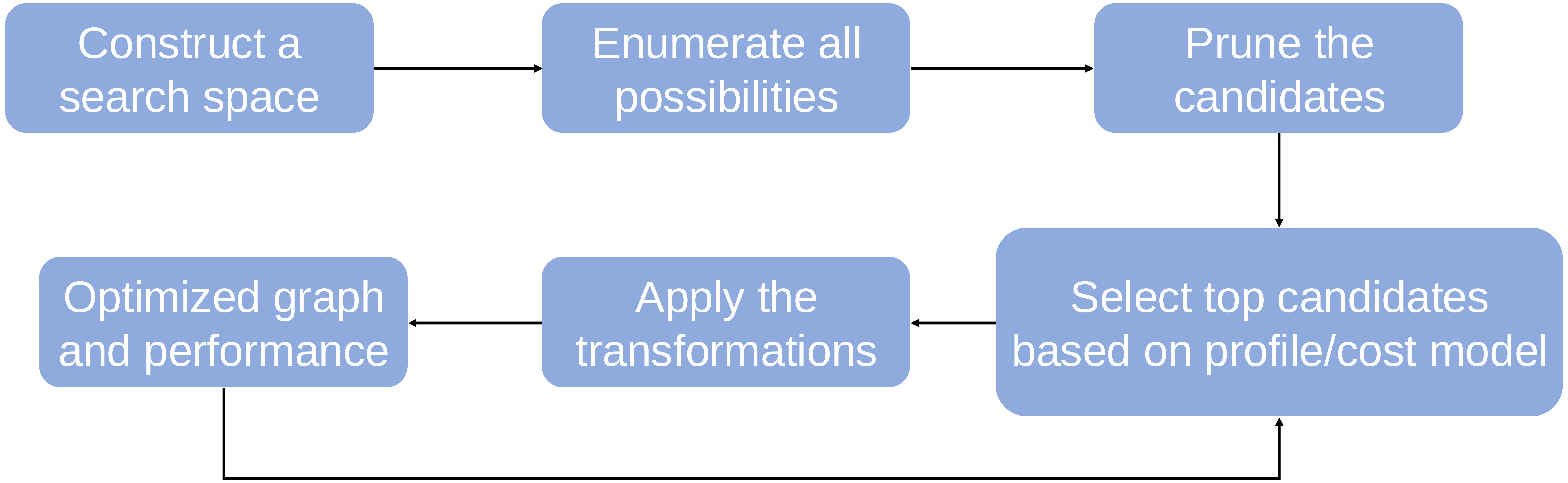
- Goal: apply verified substitutions to obtain an optimized graph
- Cost Model
  - Based on the sum of individual operator's cost
  - Profile each operator's cost on the target hardware
- Traverse the graph, apply substitutions, calculate cost, use backtracking



Performance (as of 2019)



Summary of Graph Optimization

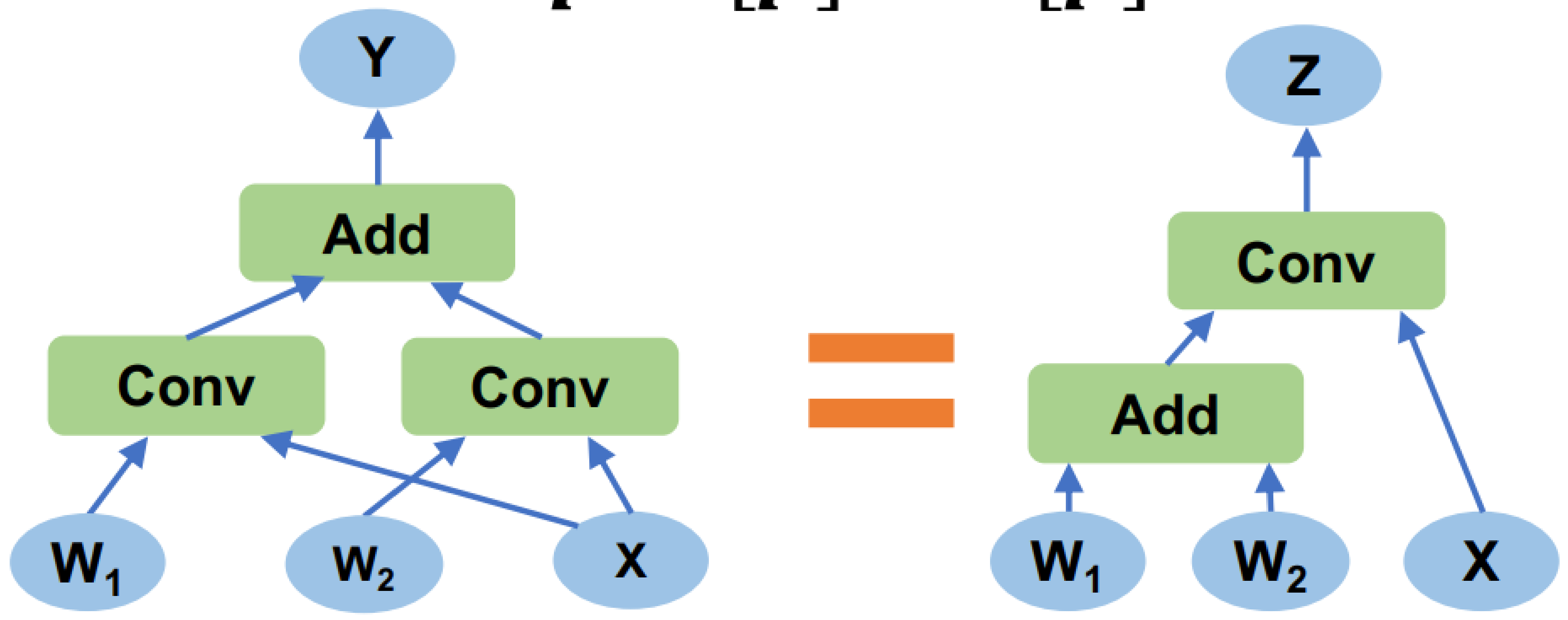


Limitations

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

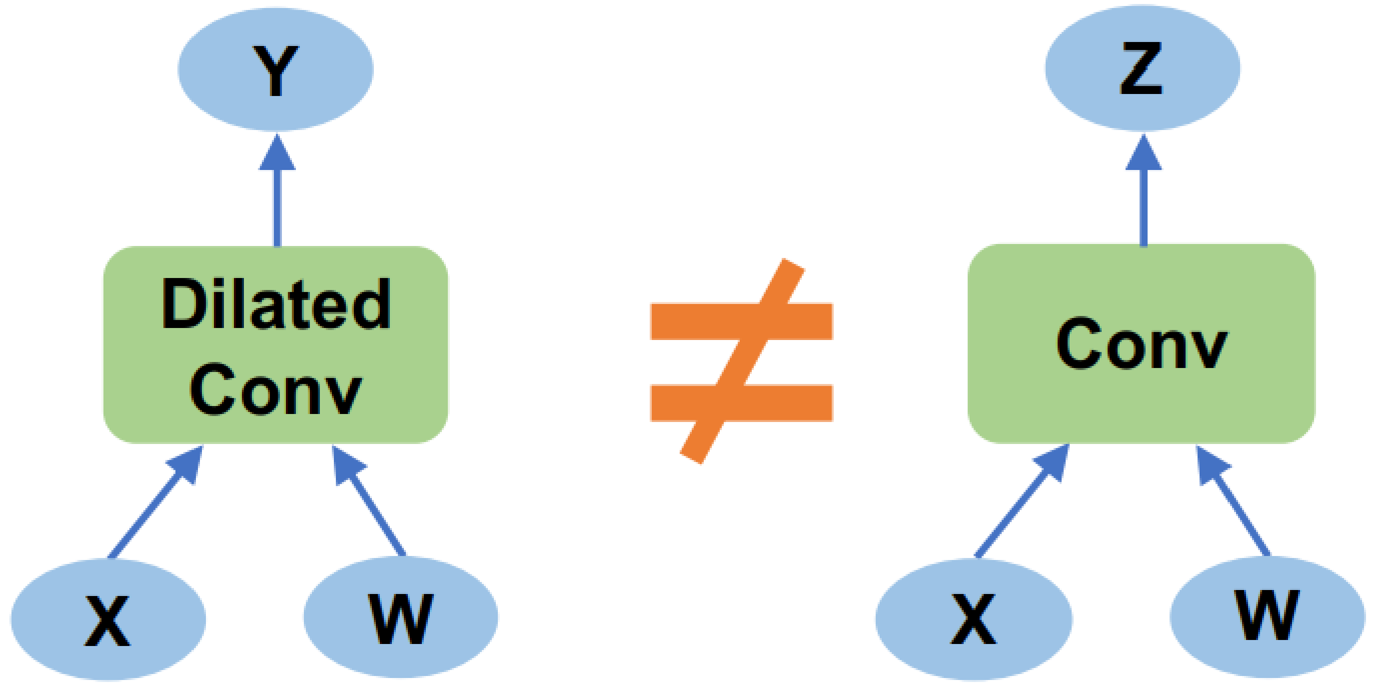
A Failure Example

$$\forall p.\ \boldsymbol{Y}[\boldsymbol{p}] = \boldsymbol{Z}[\boldsymbol{p}]$$



Fully Equivalent Transformations

$$\exists p.\ \boldsymbol{Y}[\boldsymbol{p}] \neq \boldsymbol{Z}[\boldsymbol{p}]$$



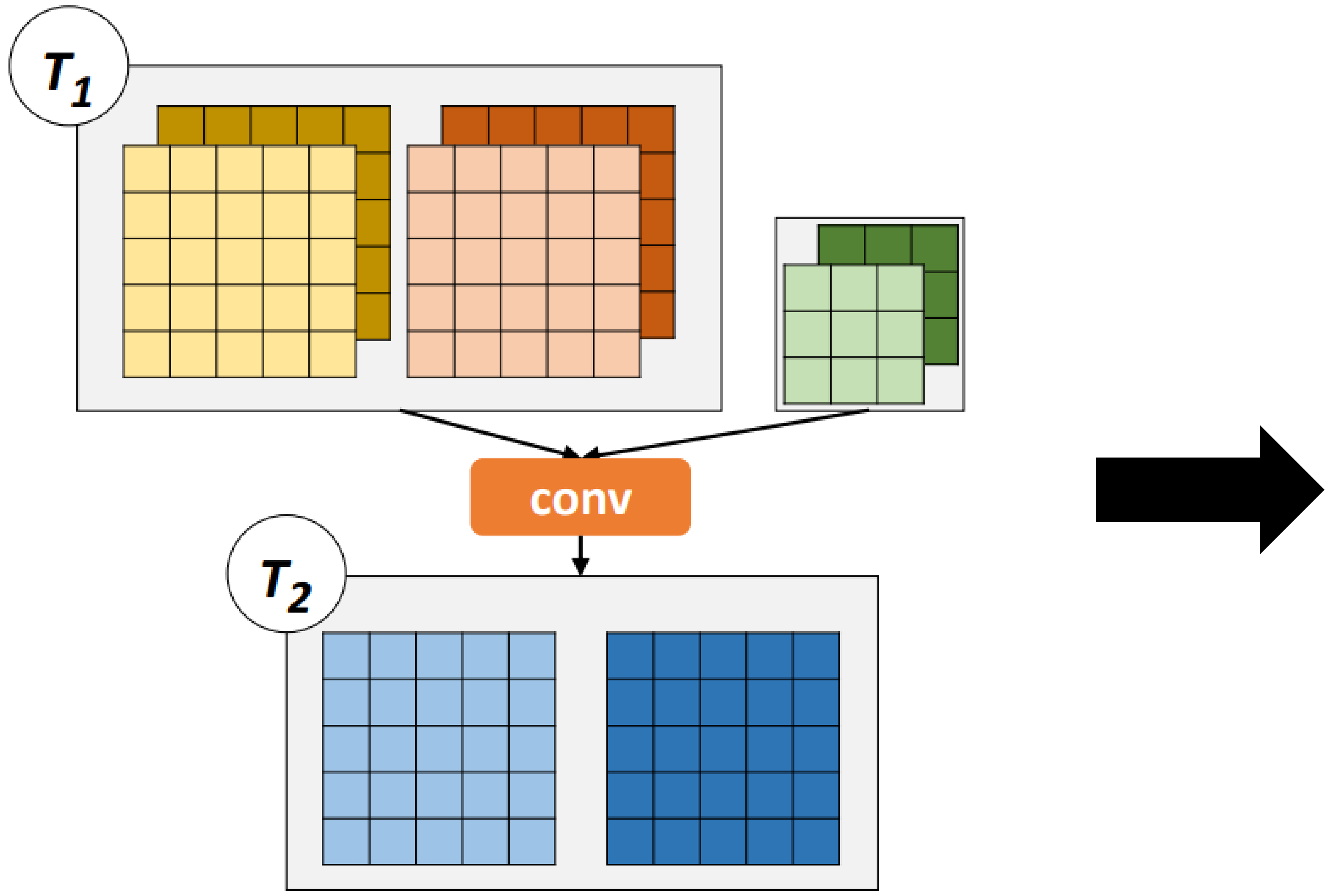
Partially Equivalent Transformations

- Math-equivalent
- Missing some optimization opportunities

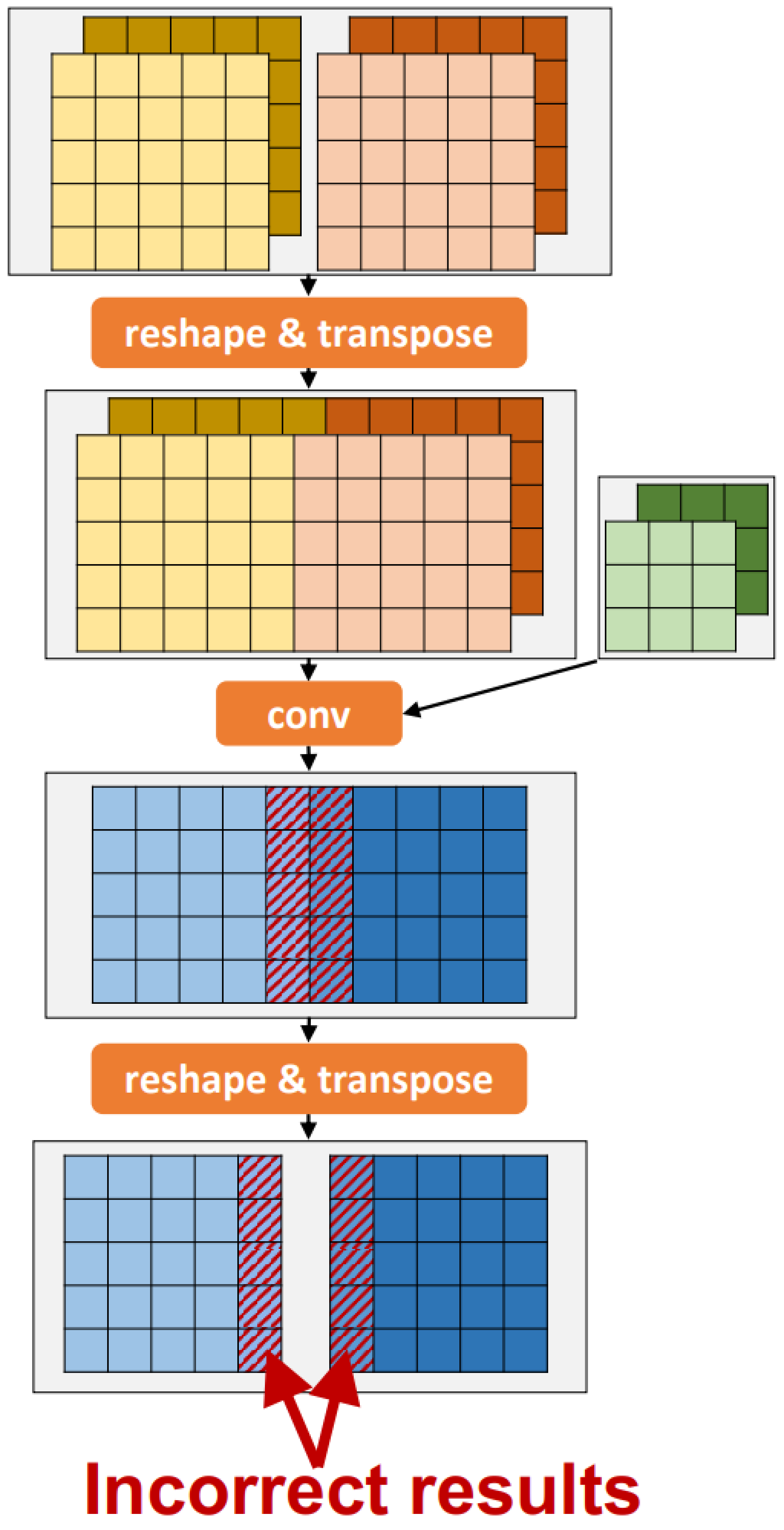
- Better performance
- Not fully equivalent -> accuracy loss

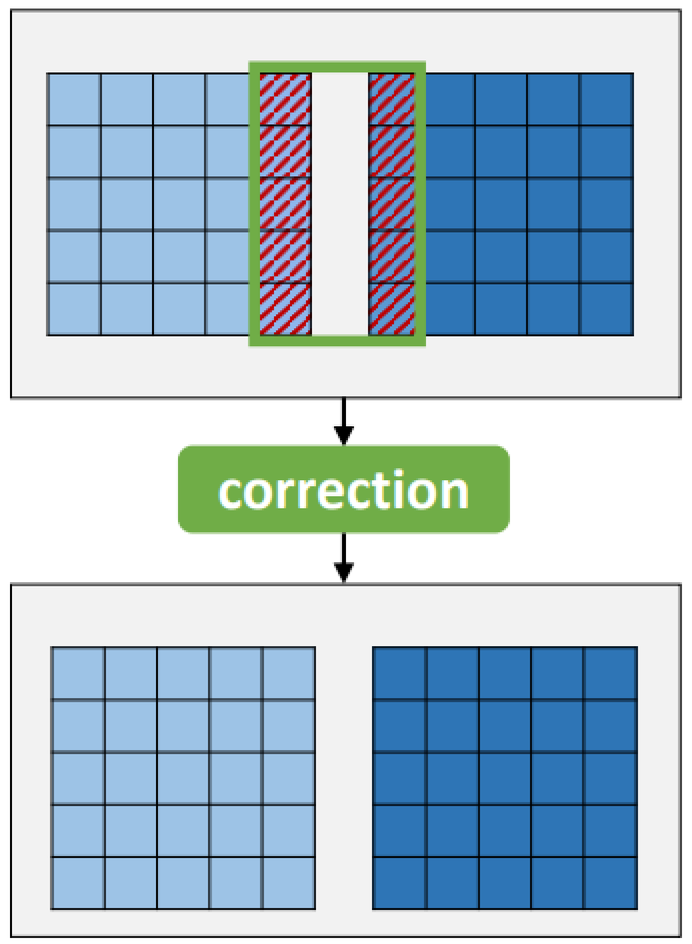
How about: exploit the larger space partially equivalent transformations for performance while still preserve correctness?

Motivating Example



Input Program

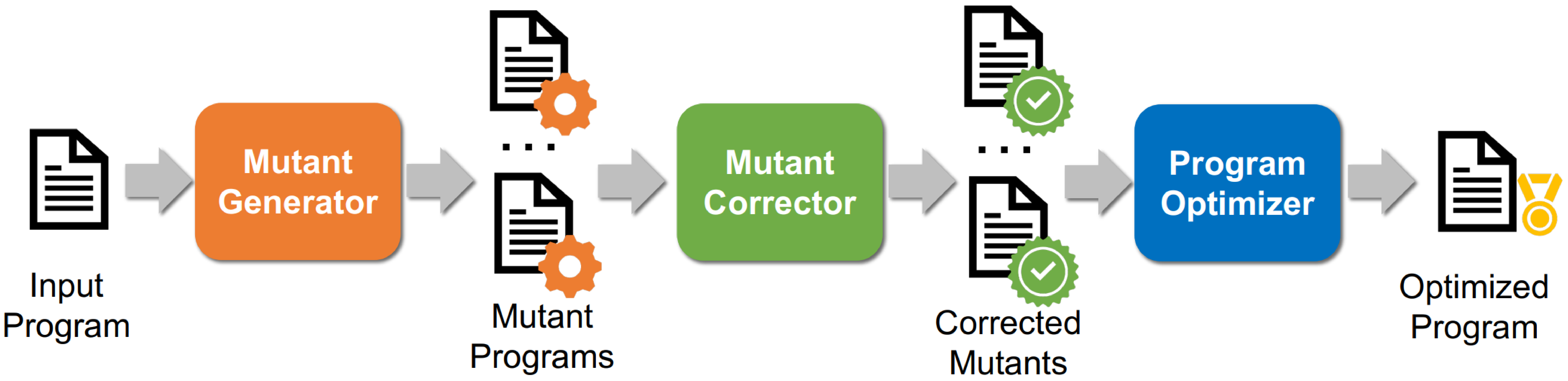




+ Correction

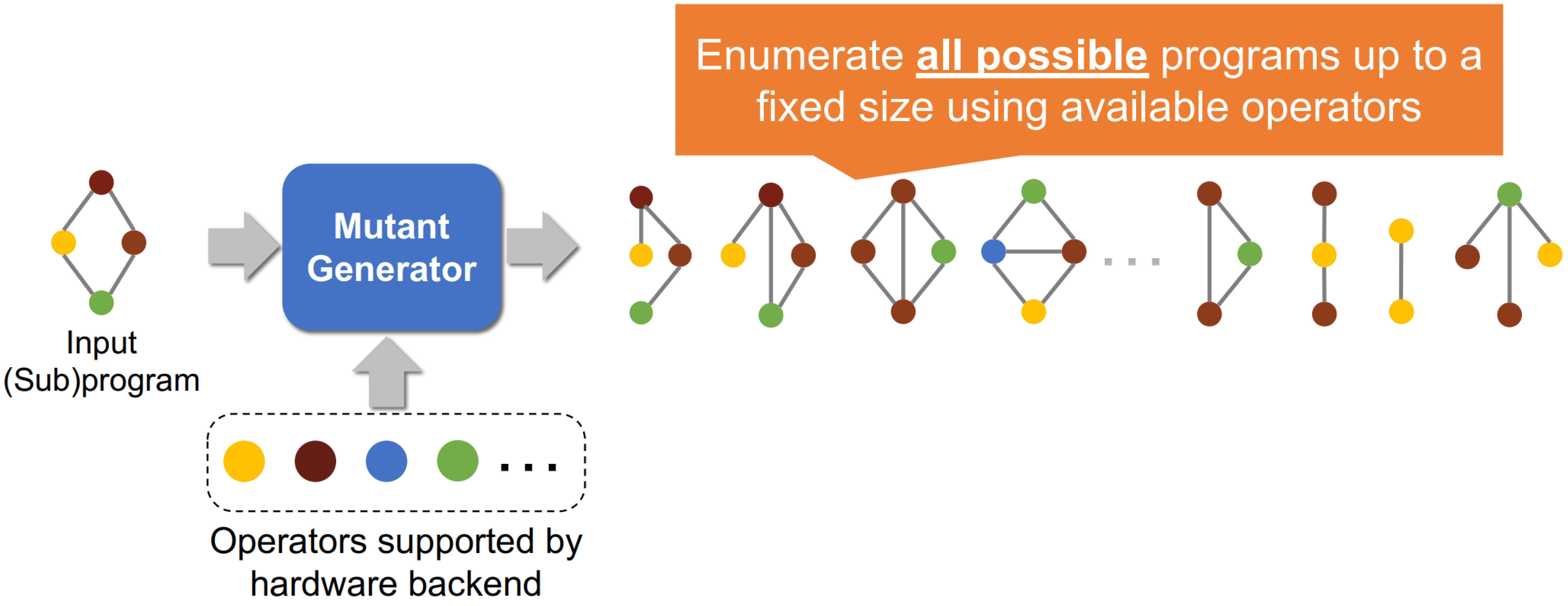
- Partial equivalent transformations + correction yield 1.2x speedup
- Which would otherwise be impossible in fully equivalent transformations space

Partially Equivalent Transformations

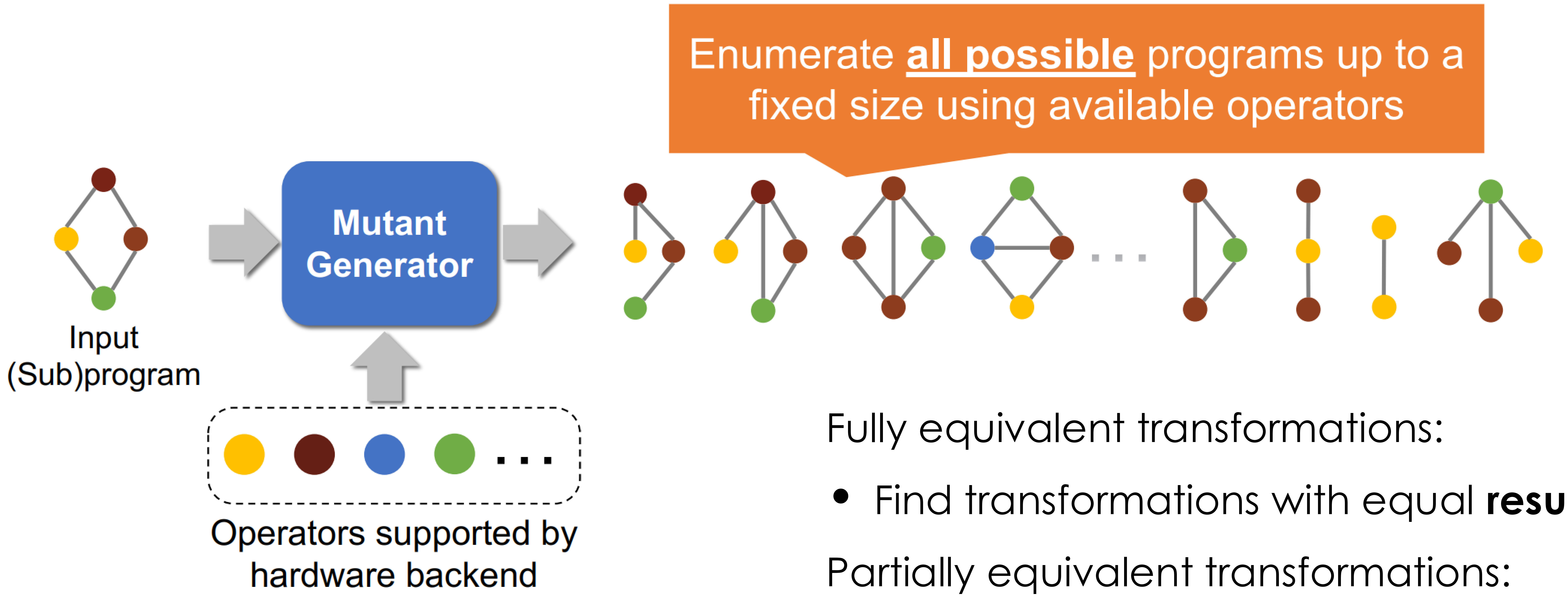


- How to mutate?
- How to correct?

Mutant Generator: Step 1



Mutant Generator: Step 2



Fully equivalent transformations:

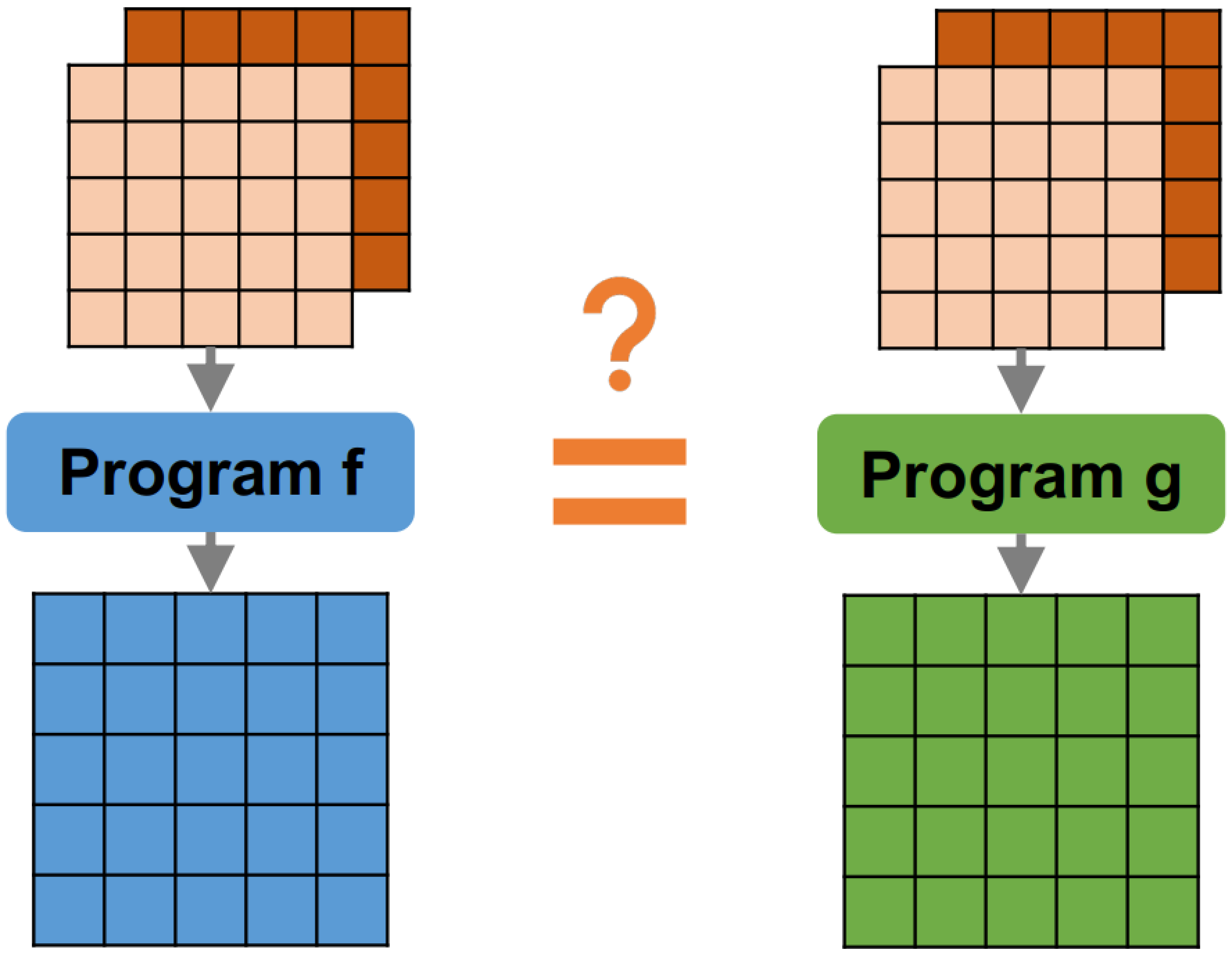
- Find transformations with equal **results**

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?

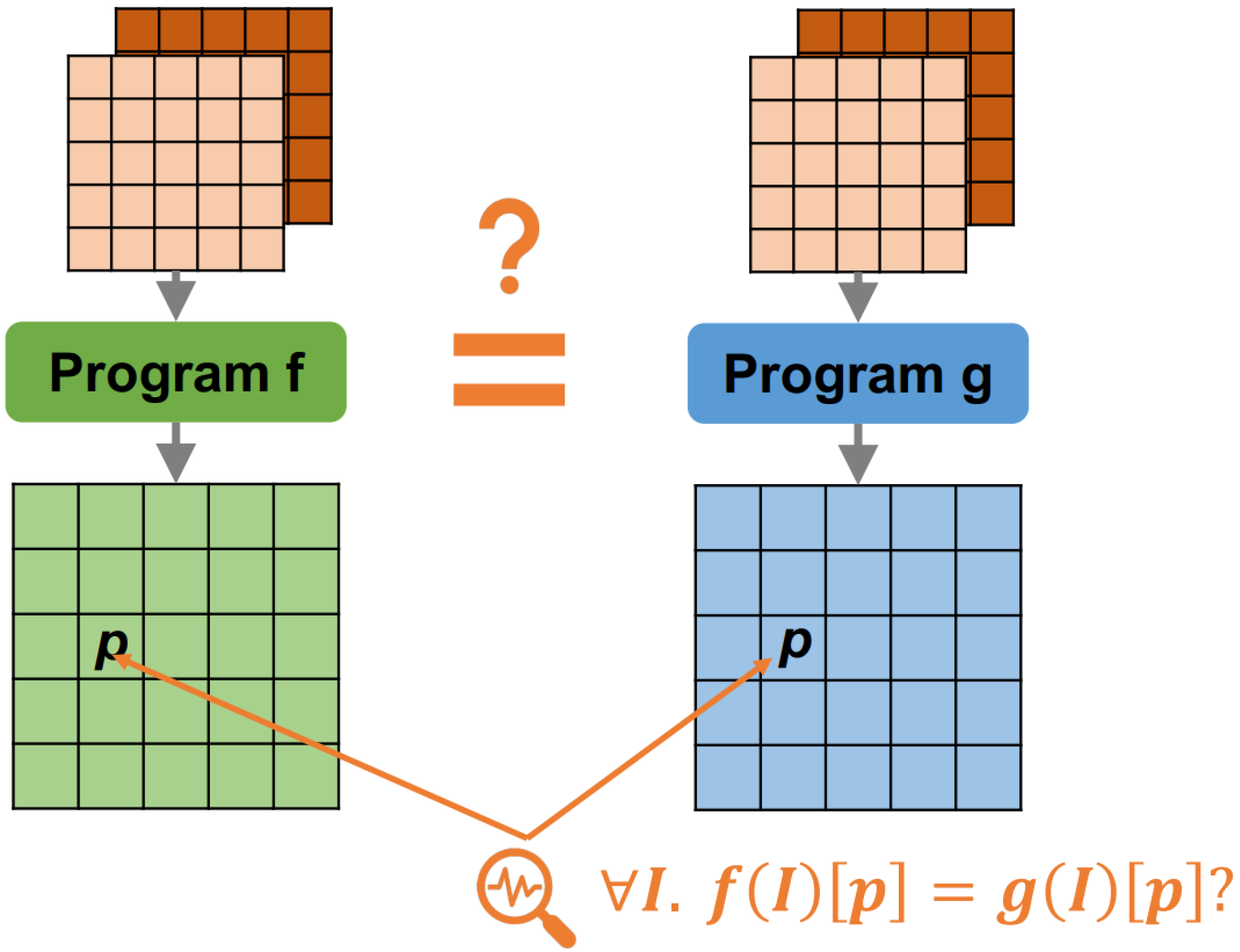


By Enumeration

- 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?
  - Reduce m and n



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

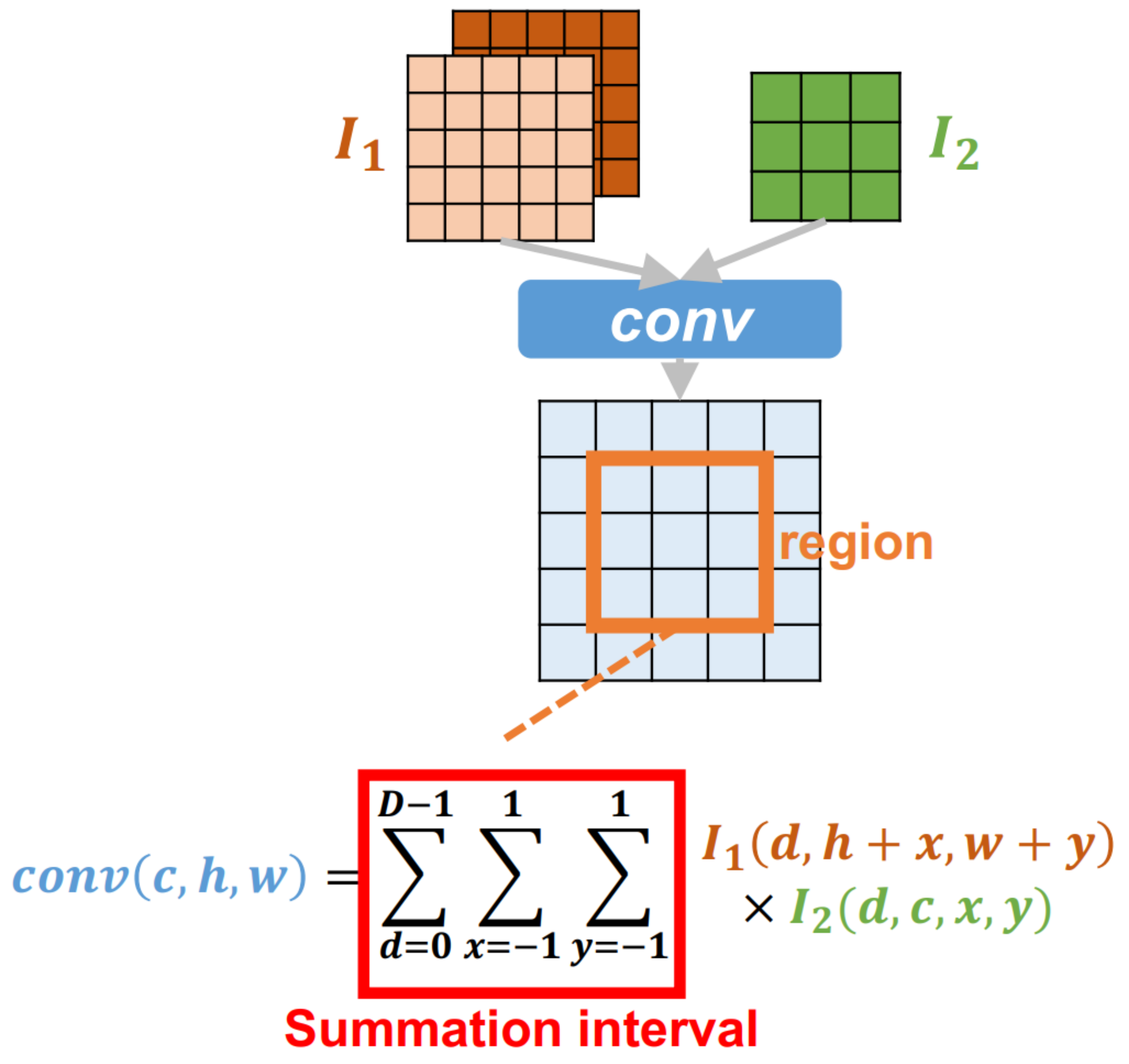
- Reason: Neural nets computation are mostly Multi-Linear
- Define Multi-linear: f is multi-linear if the output is linear to all inputs
  - $f(I_1, \dots, X, \dots, I_n) + f(I_1, \dots, Y, \dots, I_n) = f(I_1, \dots, X + Y, \dots, I_n)$
  - $\alpha f(I_1, \dots, X, \dots, I_n) = f(I_1, \dots, \alpha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

Important ML Operators are multi-linear

Operator	**Description**
`add`	Element-wise addition
`mul`	Element-wise multiplication
`conv`	Convolution
`groupconv`	Grouped convolution
`dilatedconv`	Dilated convolution
`batchnorm`	Batch normalization
`avgpool`	Average pooling
`matmul`	Matrix multiplication
`batchmatmul`	Batch matrix multiplication
`concat`	Concatenate multiple tensors
`split`	Split a tensor into multiple tensors
`transpose`	Transpose a tensor's dimensions
`reshape`	Decouple/combine a tensor's dimensions

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(mr)
    - r (# regions) <<< n

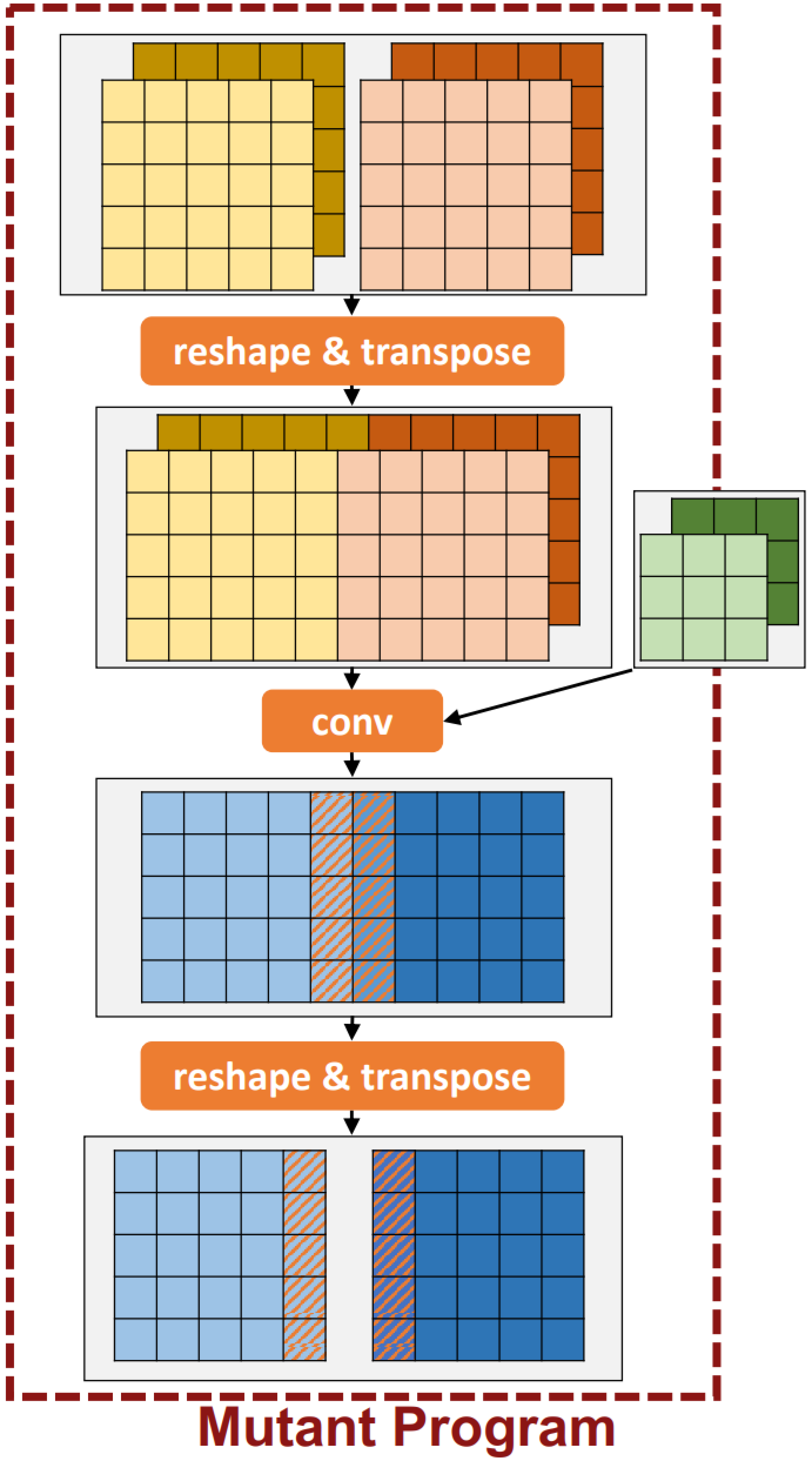


How to reduce m?

- 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(mr) -> O(tr) (t << m, r << n)

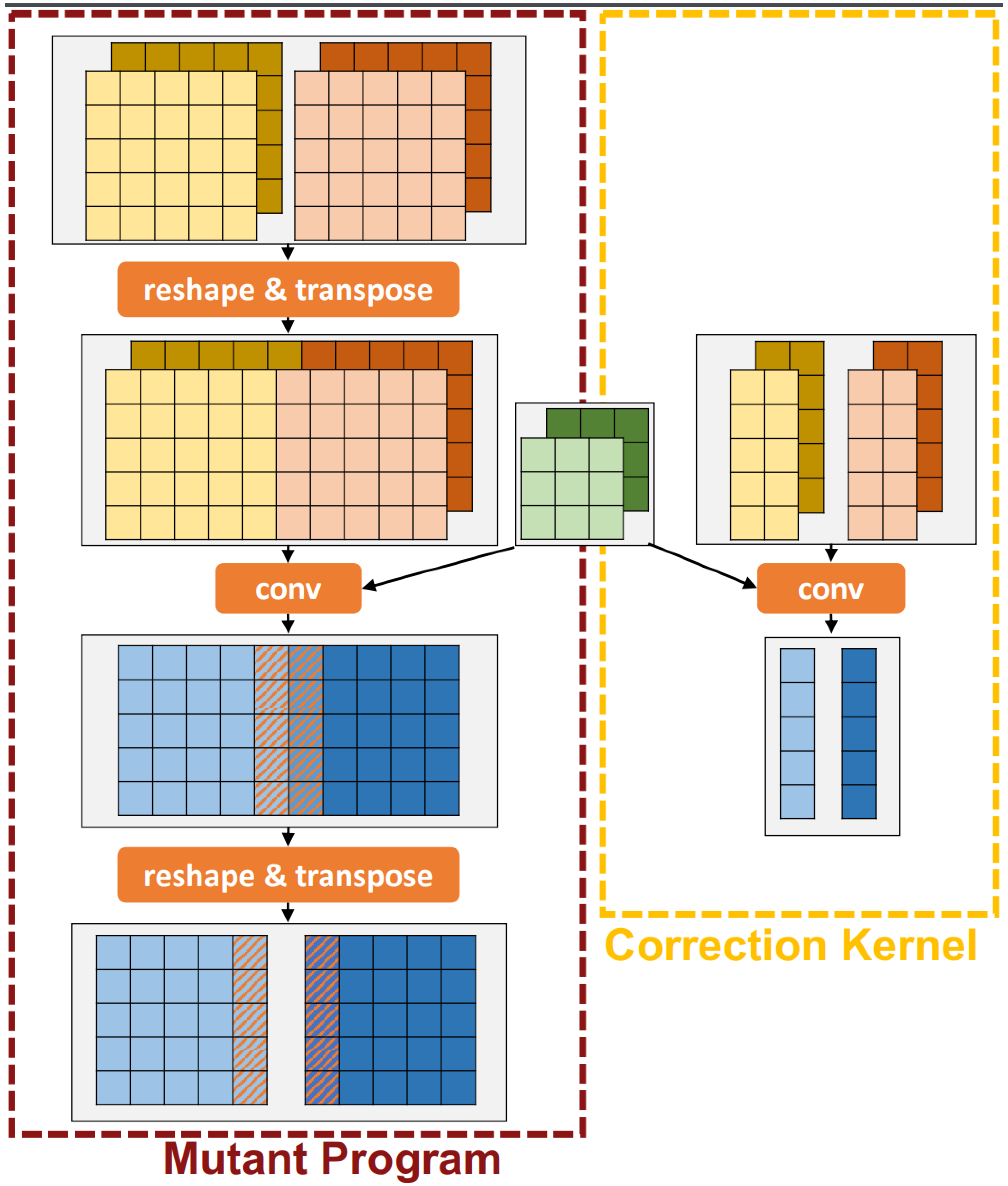
Correct the Mutant

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



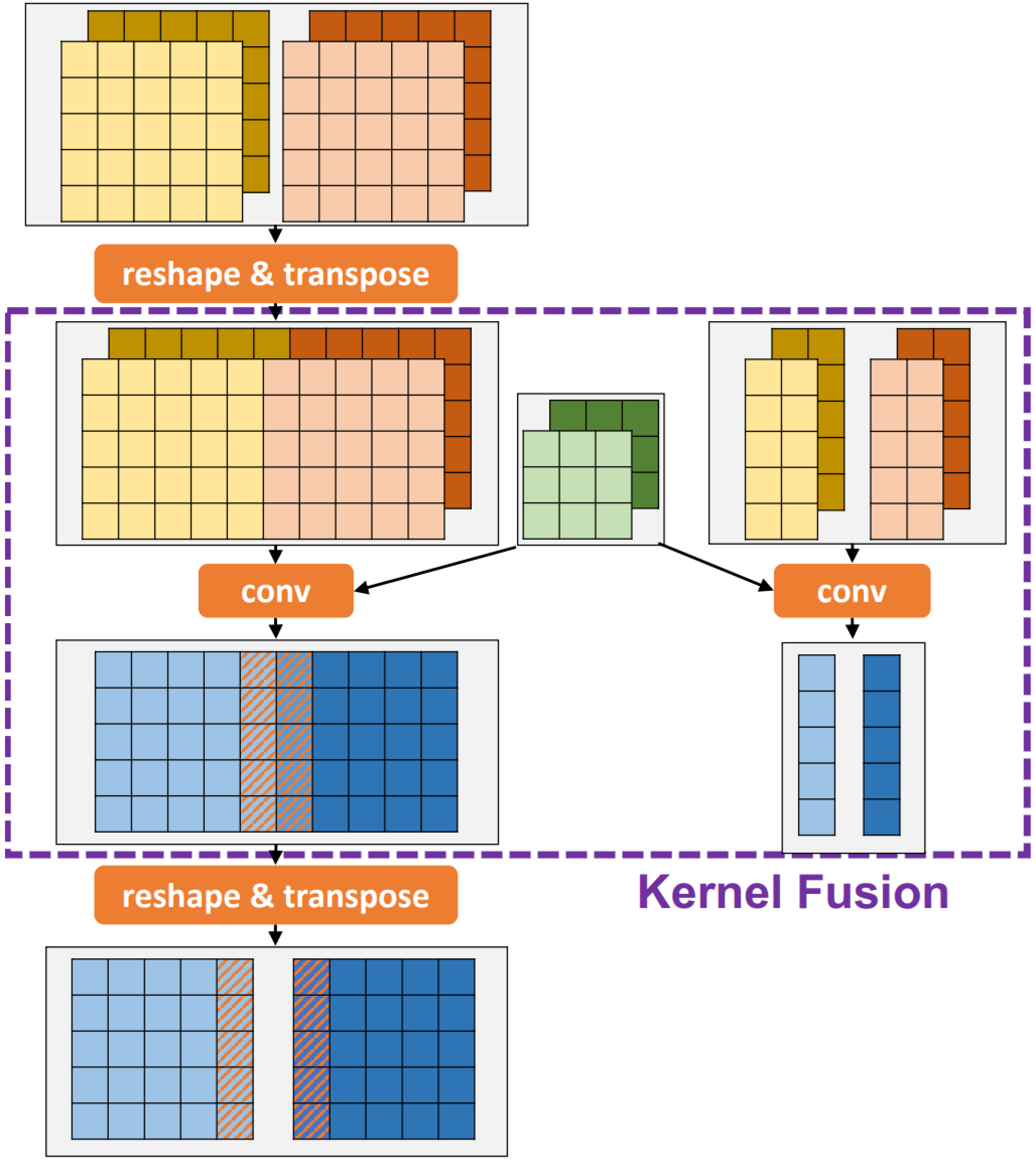
Correct the Mutant

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

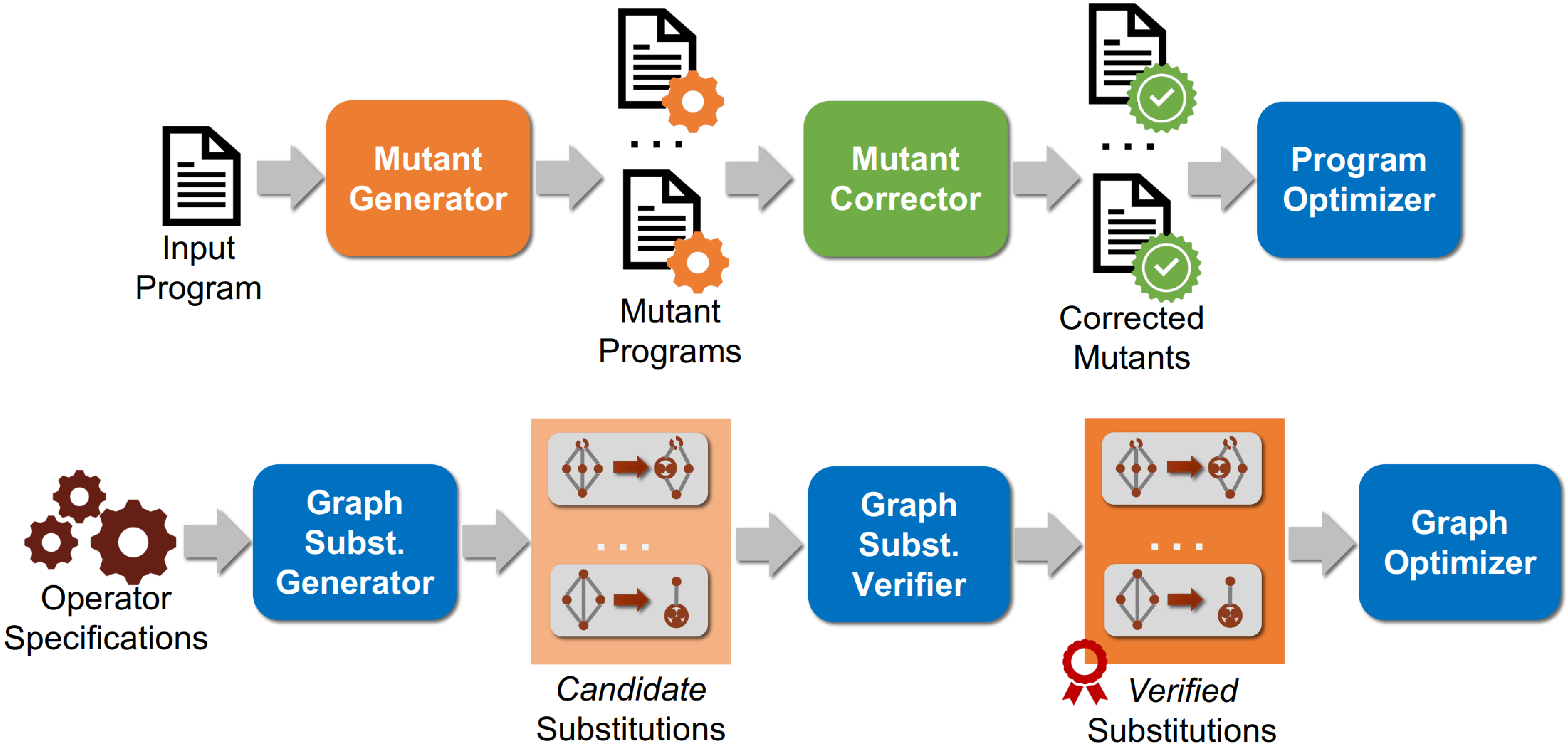


Correct the Mutant

- 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 — 2013

TensorRT
cuDNN
ONNX — 2016 ~ 2017

tvm — 2018

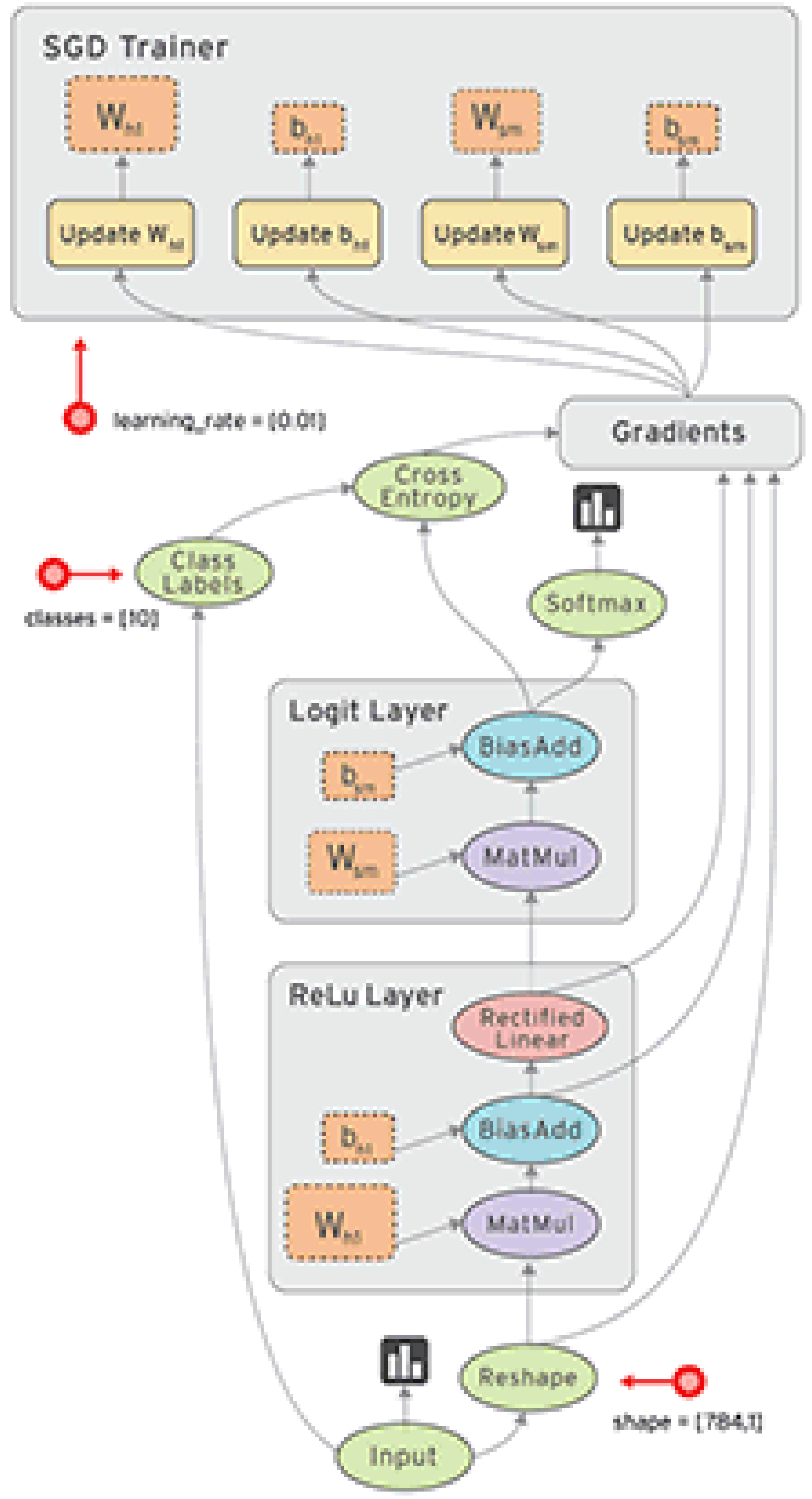
MLIR
Flexflow — 2019 ~2020

2.0
Torch Dynamo — Now

More Compiler in Guest Lecture

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

Big Picture: Where We Are



Dataflow Graph

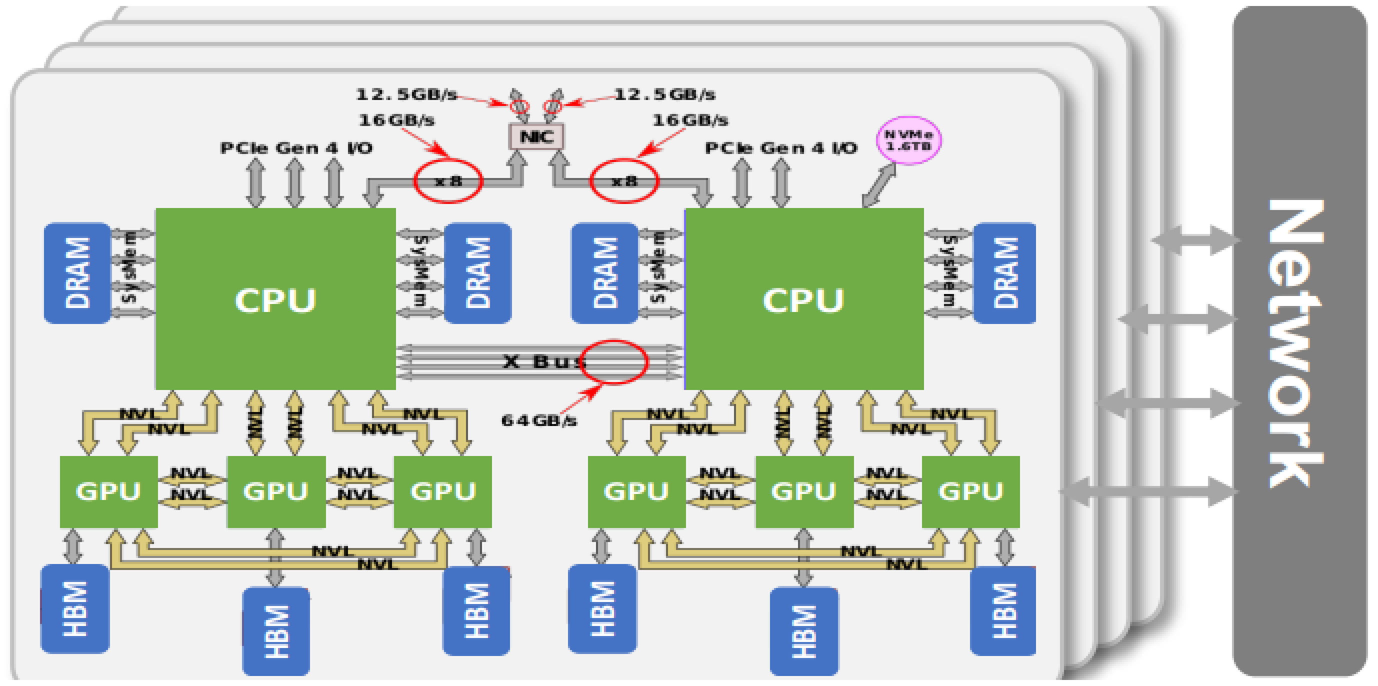
Autodiff

Graph Optimization

Parallelization

Runtime

Operator optimization/compilation



Next: Runtime

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