https://haoailab.com/cse291-s26/

CSE/DSC 291: Deep Learning Systems
Spring 2026

LLM, diffusion, and case studies

Optimizations and Parallelization

Basics

MCQ Time

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?

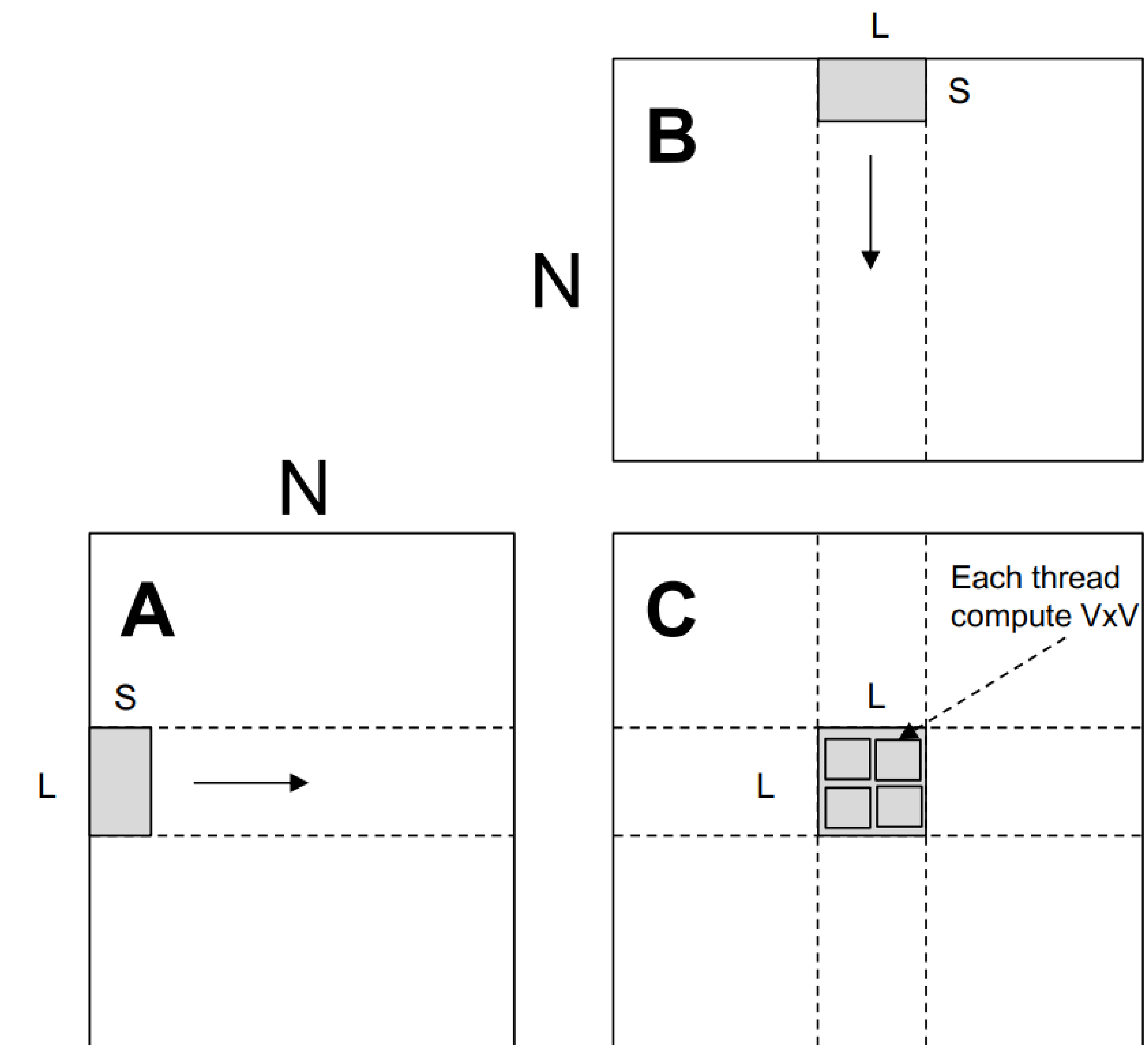
A. ReLU Activation
B. Linear layer (8192 outputs, 2048 inputs, batch size 1)
C. Batch normalization
D. Max pooling (3x3 window and unit stride)
E. Layer normalization
F. Linear layer (2048 outputs, 1024 inputs, batch size 512)

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.

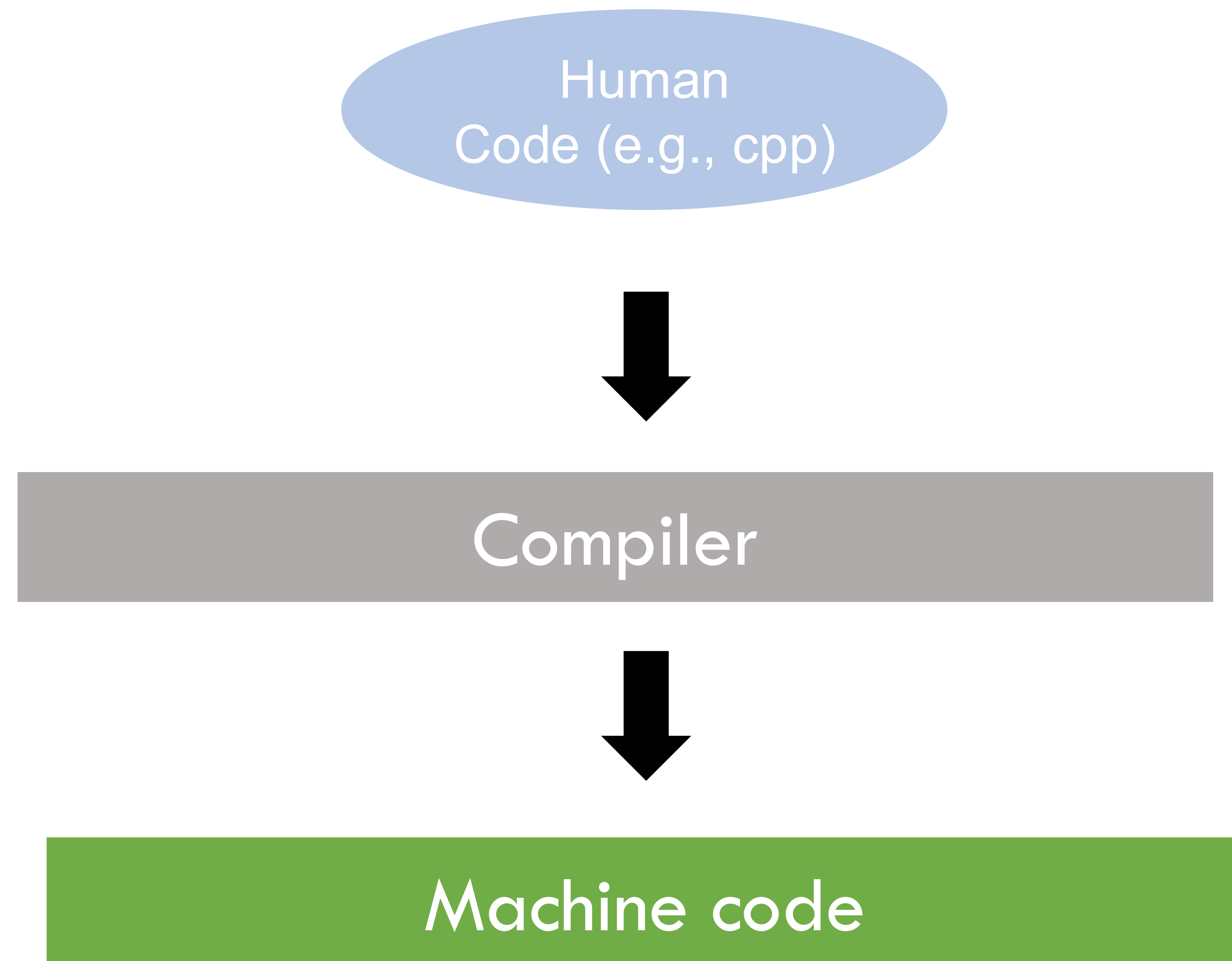
Recap of Matmul Problem

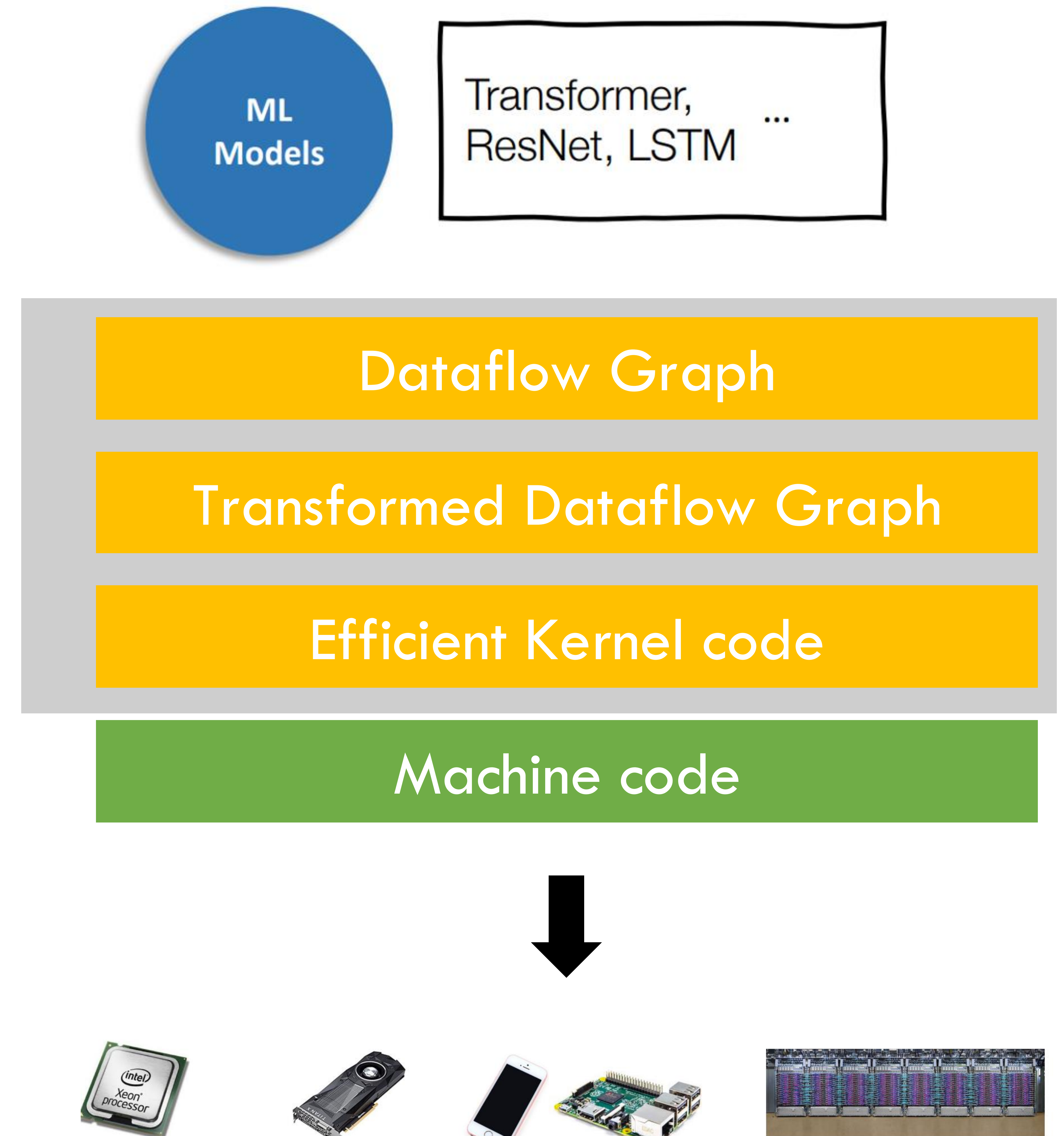
- How to implement an highly efficient kernel
- How to choose configs.
  - #threads
  - #registers
  - Amount of SRAM



- Solution 1:
  - expert-craft -> Enumerate configs -> profile
- Solution 2: Operator compilation

Traditional vs. ML Program





ML Compilation: Big Picture

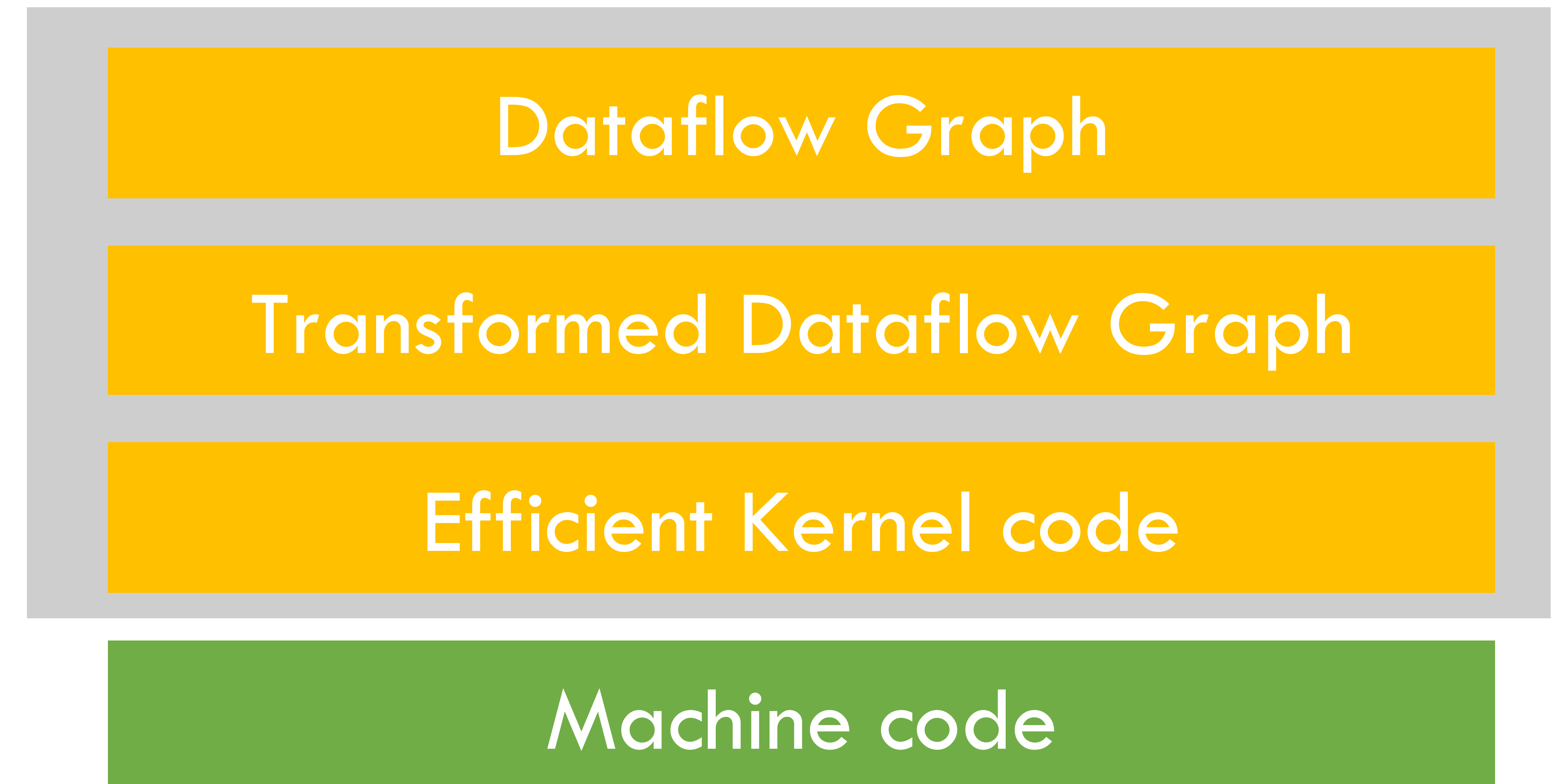
ML compilation's Goal:

Automatically generate *optimal* configurations and code

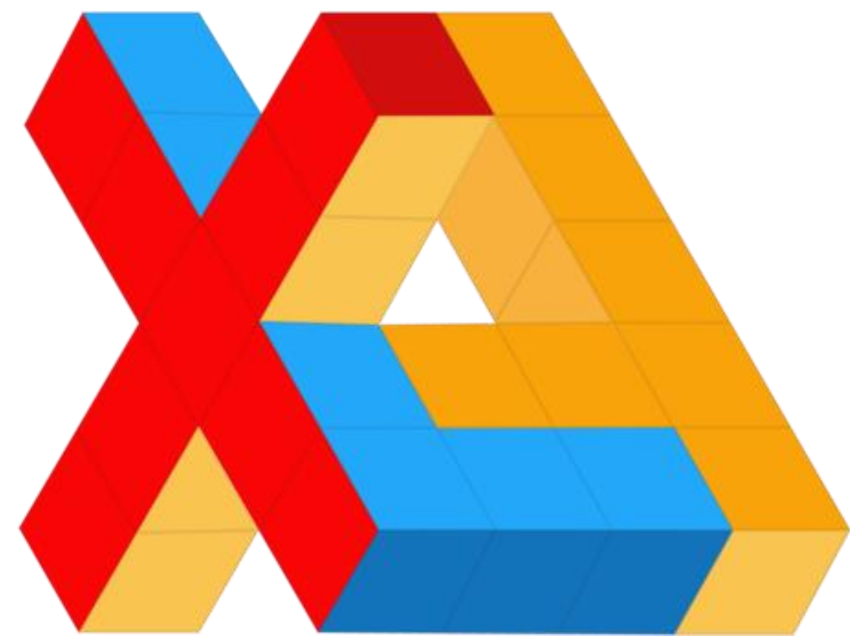
given users code and target hardware

Grand Problems to Build ML Compilers

- **Programming-level: program translation**
  - Automatically transform an arbitrary (usually imperative) code (by developers) into a compile-able code (e.g., static dataflow graph)?
- **Graph-level: graph compilation**
  - Automatic graph transformations to make it faster
- **Op-level: operation compilations**
  - How to make operator fast on different hardware?

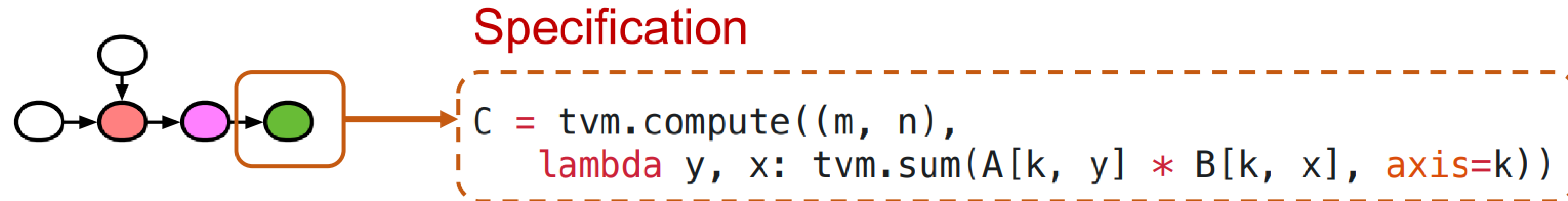


Notable Compilers



Modular

Example: Operator Compilation



Search Space of Possible Program Optimizations

Low-level Program Variants

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

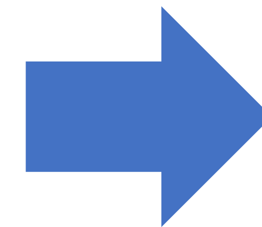
```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

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]
```



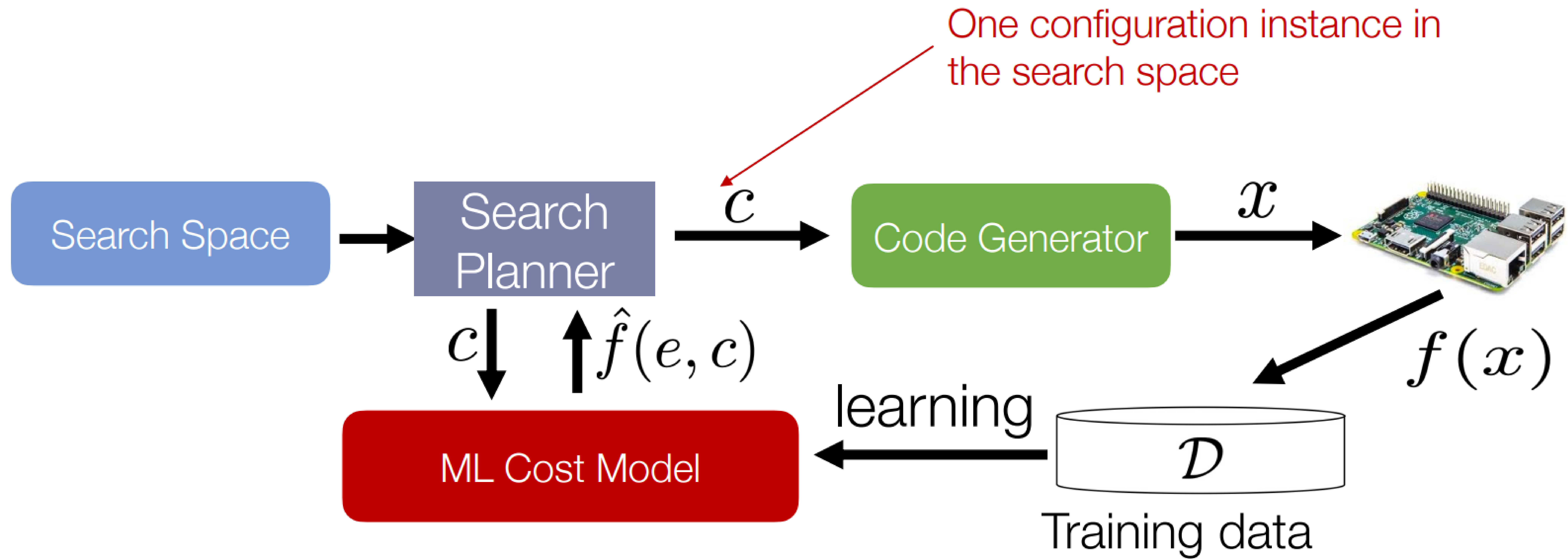
```
for xi in range(4):
  for xo in range(32):
    C[xo * 4 + xi]
     = A[xo * 4 + xi] + B[xo * 4 + xi]
```

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

Problems

- We need to enumerate many possibilities
  - How to represent all "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 (or LLM)



Discussion

ML compilation's Promise:

Automatically generate *optimal* configurations and code

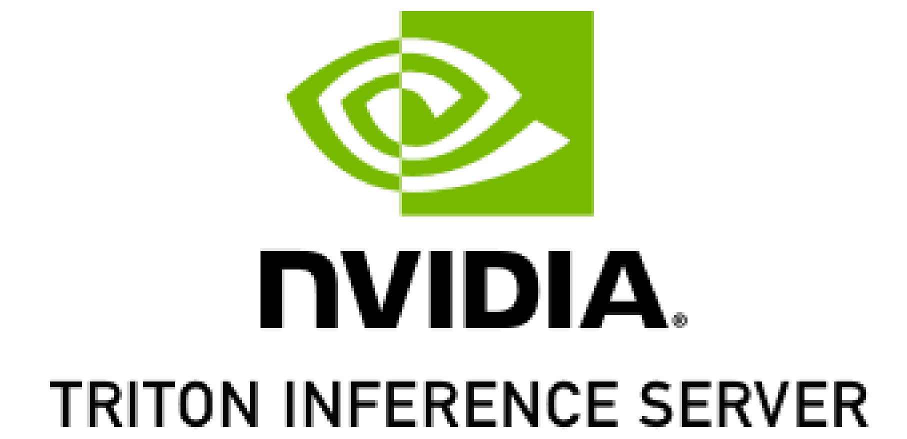
given users ML code on target hardware

Q: How well are we achieving our goals in ML compilers?

Next

- Operator Compilation
- **High-level DSL for CUDA: Triton**
- Graph Optimization

Which Triton?



Device-specific DSL (e.g., CUDA) vs. Compiler

+ developers can do whatever the heck they want:

- squeeze the last bits of performance
- use whatever data-structure you want

-- developers can do whatever the heck they want:

- Require deep expertise; performance optimization is very time-consuming
- Codebases are complex and hard to maintain

+ Very fast iteration speed for developers

- Can prototype ideas quickly and give it to compiler

-- Cannot represent certain types of ideas

- In-operator control flow
- Custom data structure

-- Code generation is a old difficult problem

- heavy use of templates and pattern-matching
- lots of performance cliffs

Triton's Pitch (Please think about why this makes sense)

+ simpler than CUDA; more expressive than graph compilers:

-- less expressive than CUDA; more complicated than graph compilers;

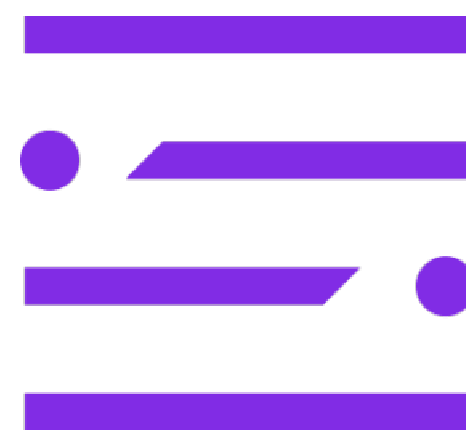
Triton Programming Model

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

Embedded in Python

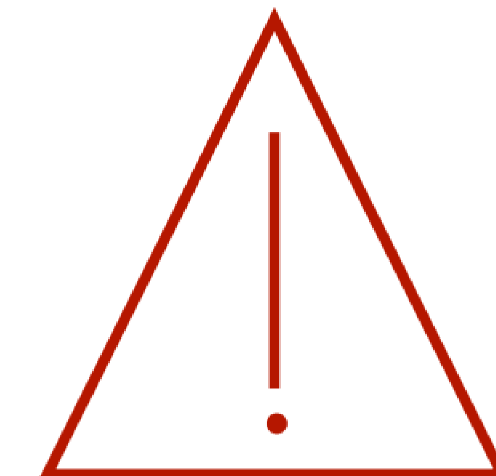
Kernels are defined in Python using triton.jit

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)
```

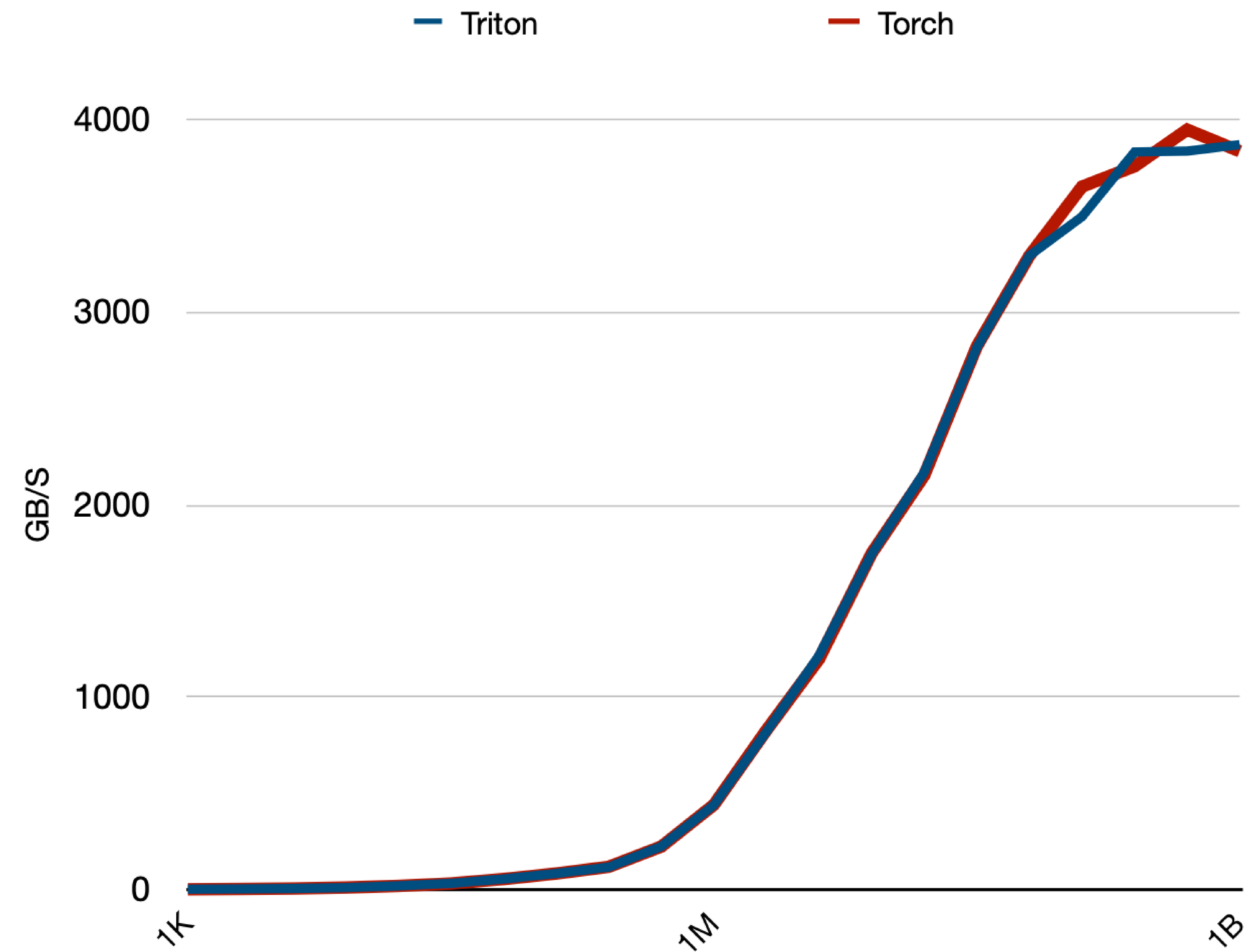
Example: elementwise add v2 (z = x + y)

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

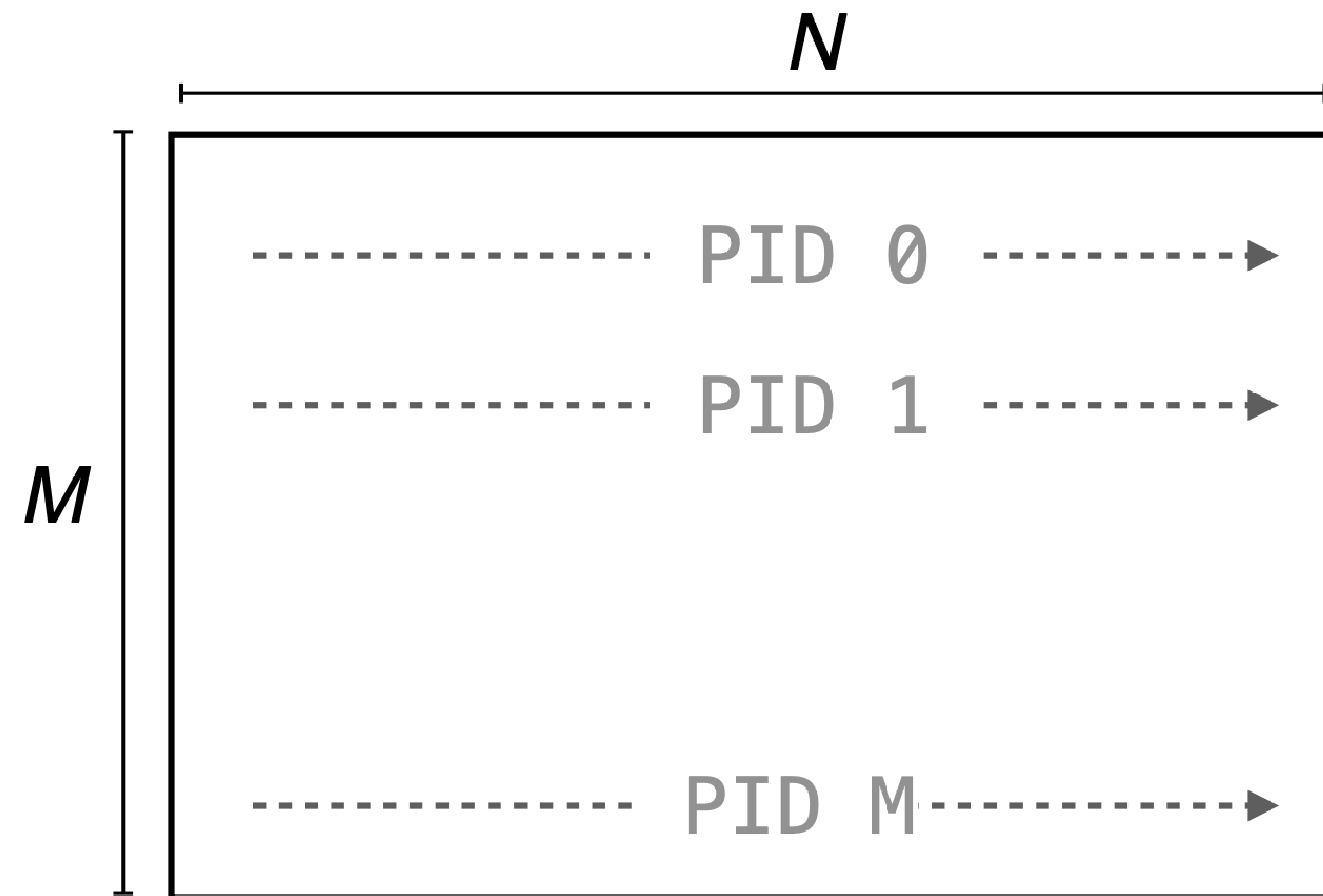


Another Example: Softmax

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

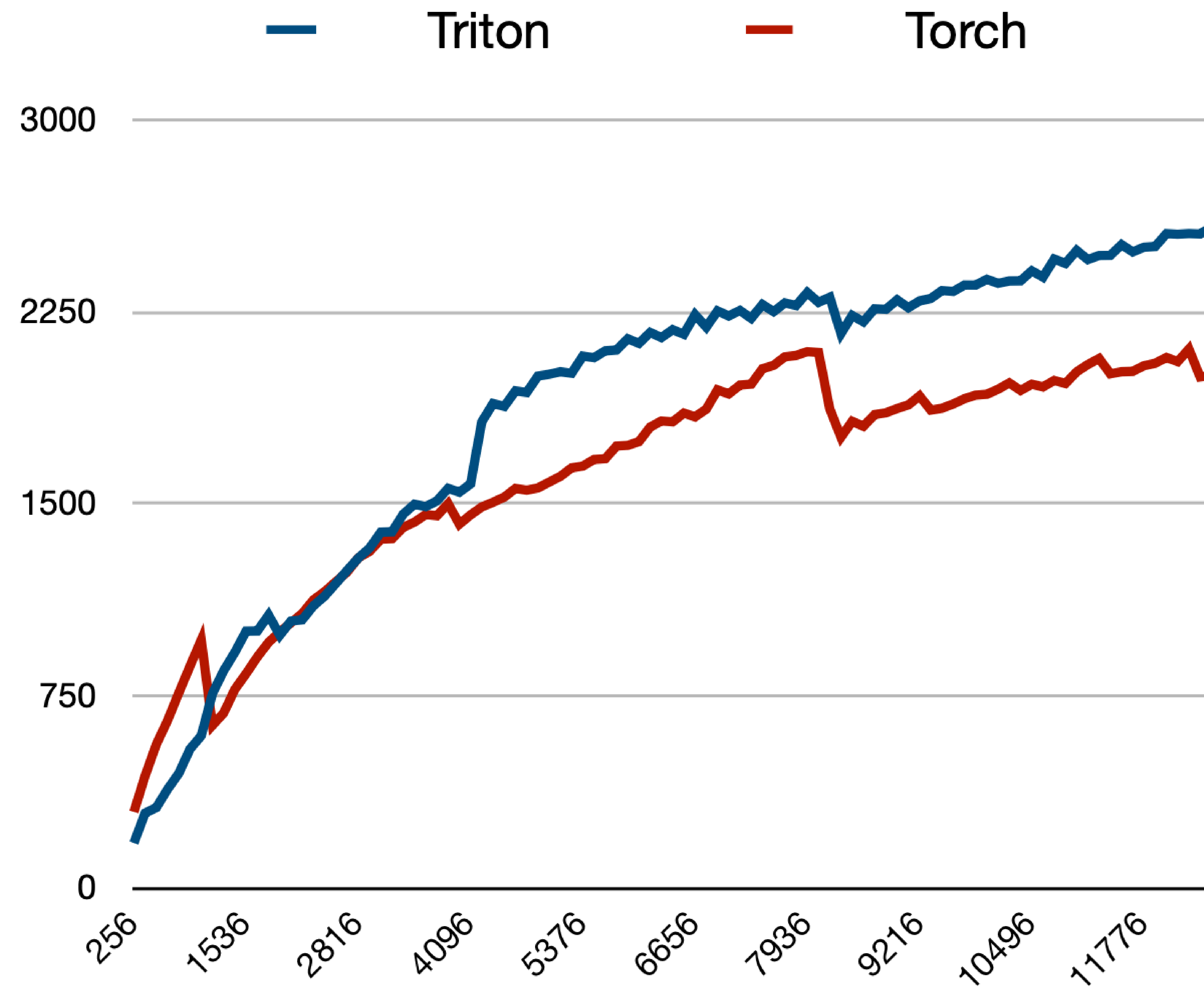
- How did you implement this in PA1?
  - Think about the potential overhead when compose softmax from primitives
- Performant option: implementing an end-to-end softmax kernel
  - Think about the complexity of implementing in CUDA

Triton Example: softmax

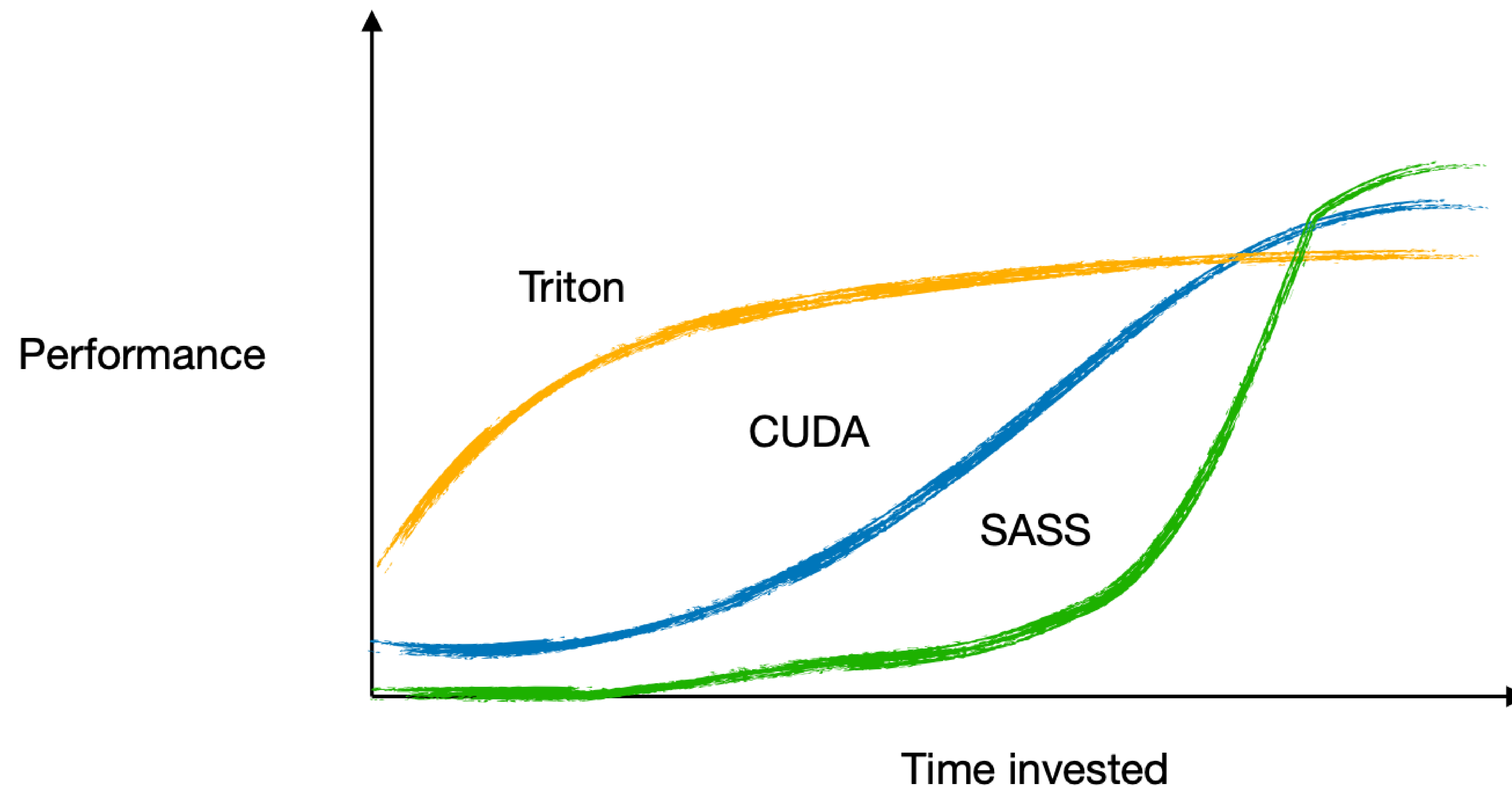



```
import triton.language as tl
Import triton

@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



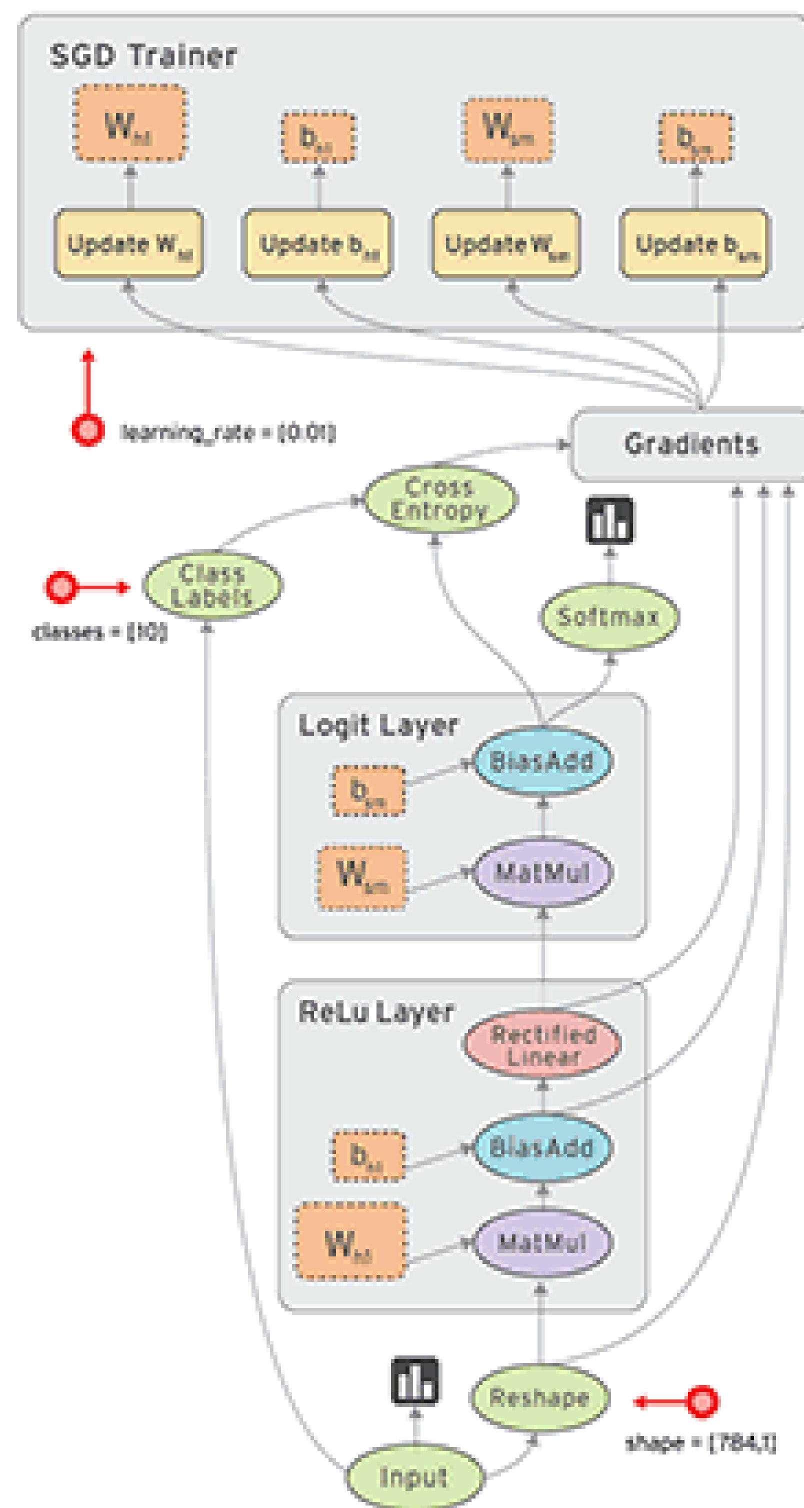
SASS = streaming assembly

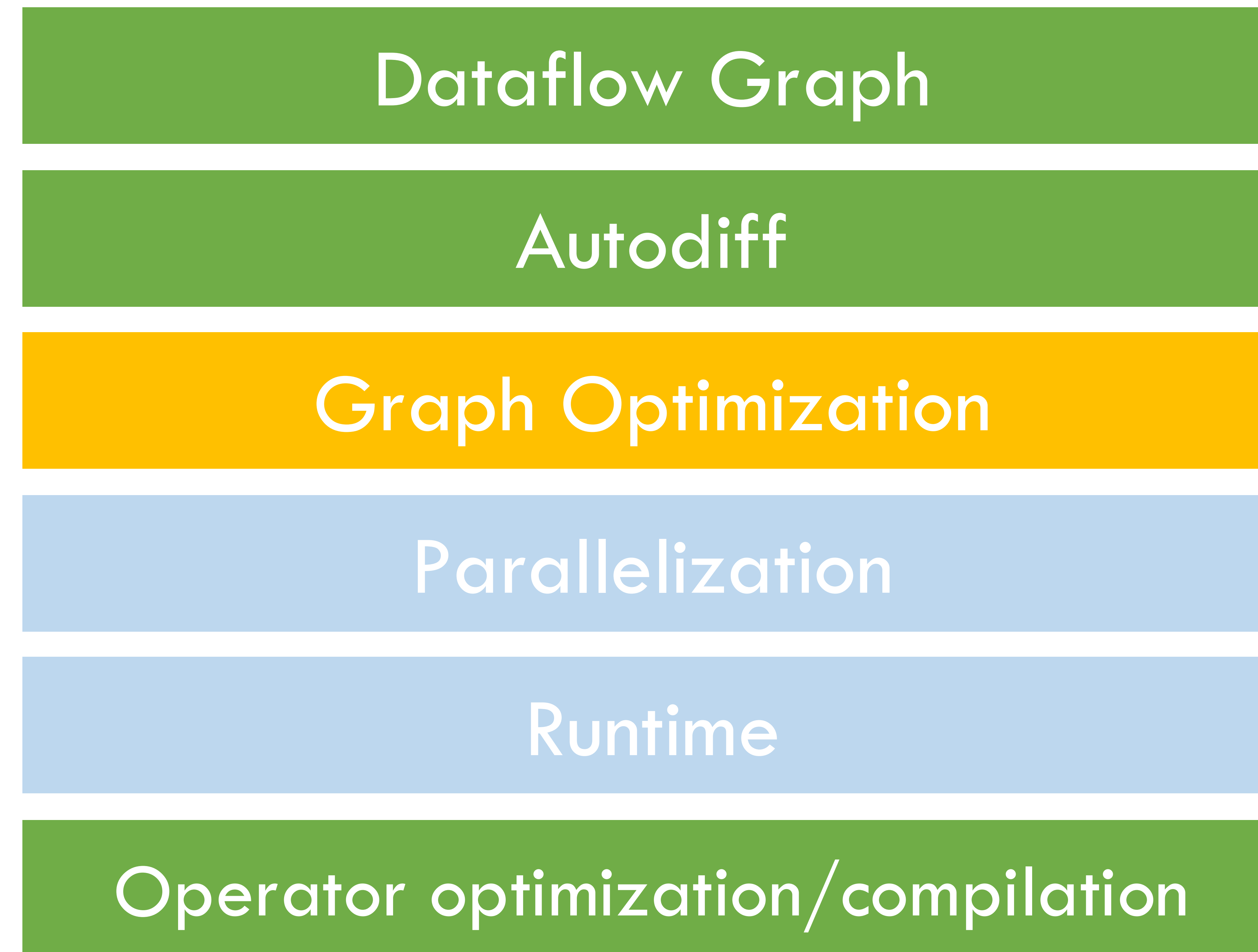
Summary: Operator Optimization

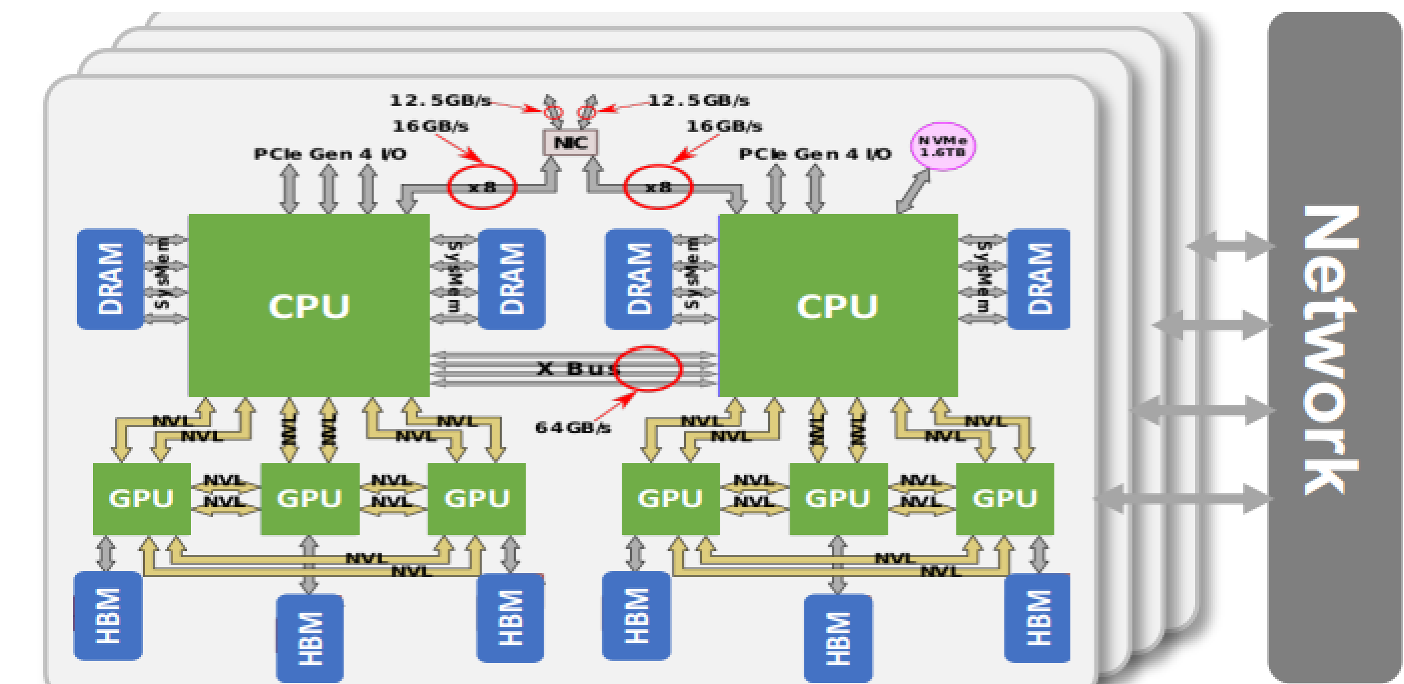
Goal: to make individual operator run fast on diverse devices

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



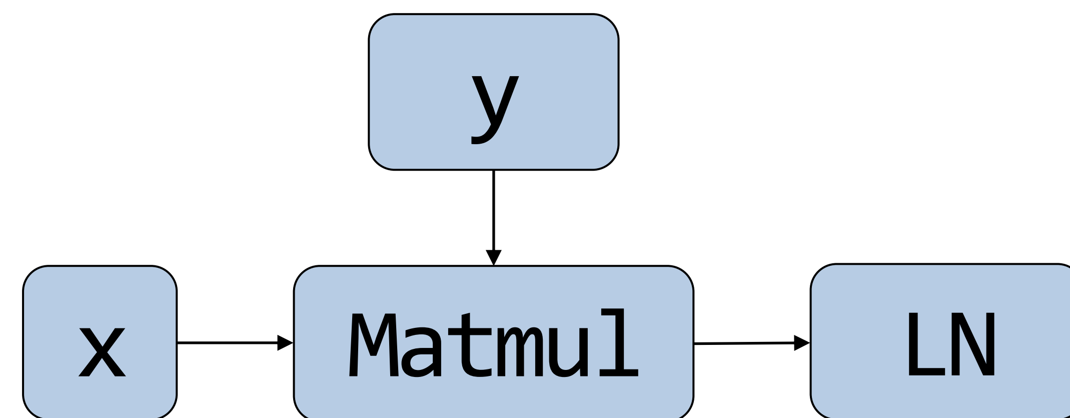


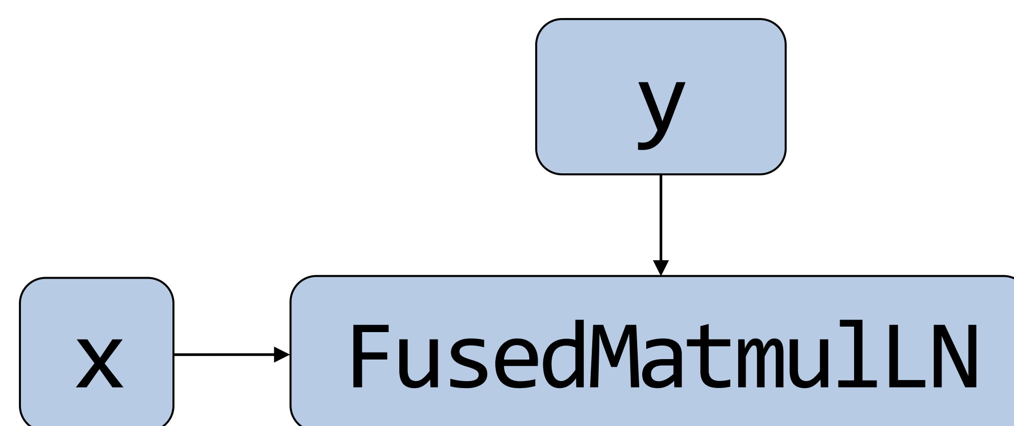


Recall Our Goal

- 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

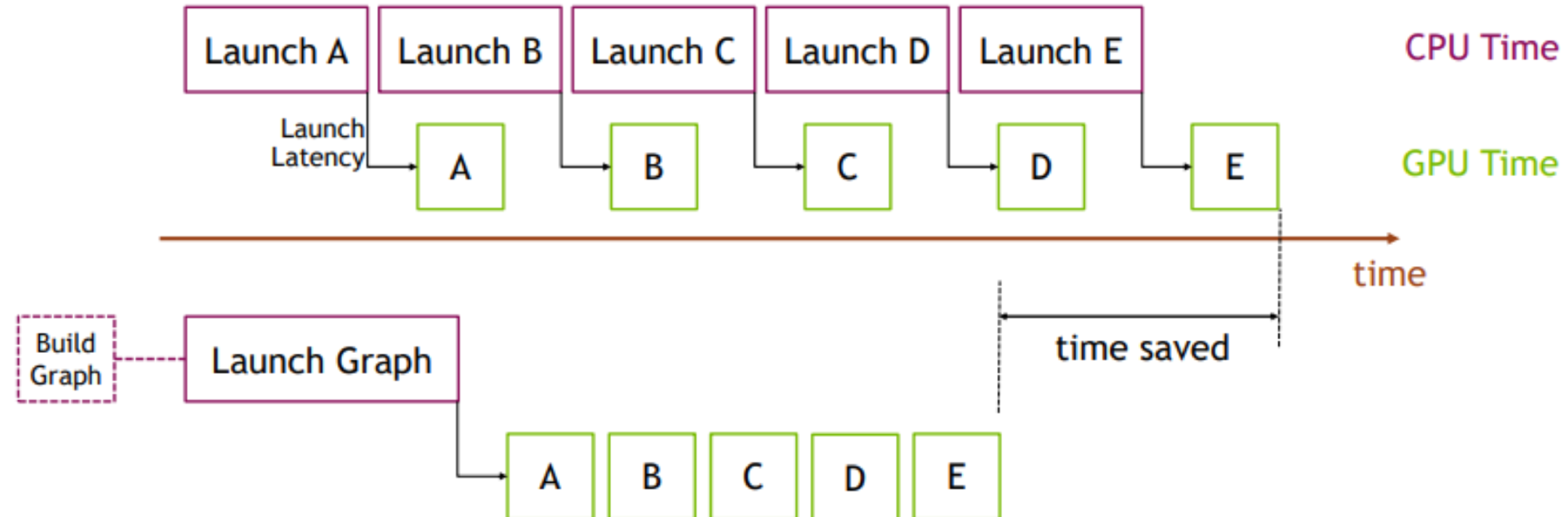




- Why operator fusion improves performance?
  - Reduce I/O
  - Reduce kernel launching
- Cons:
  - Requiring many fused ops: FusedABCOp
  - At some point, codebase becomes unmanageable

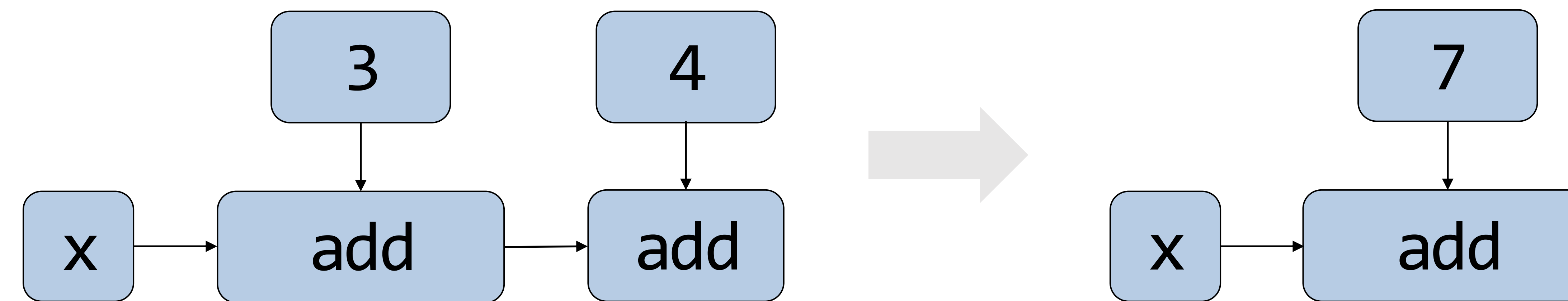
One trade-off in Practice results in “CUDA Graph”

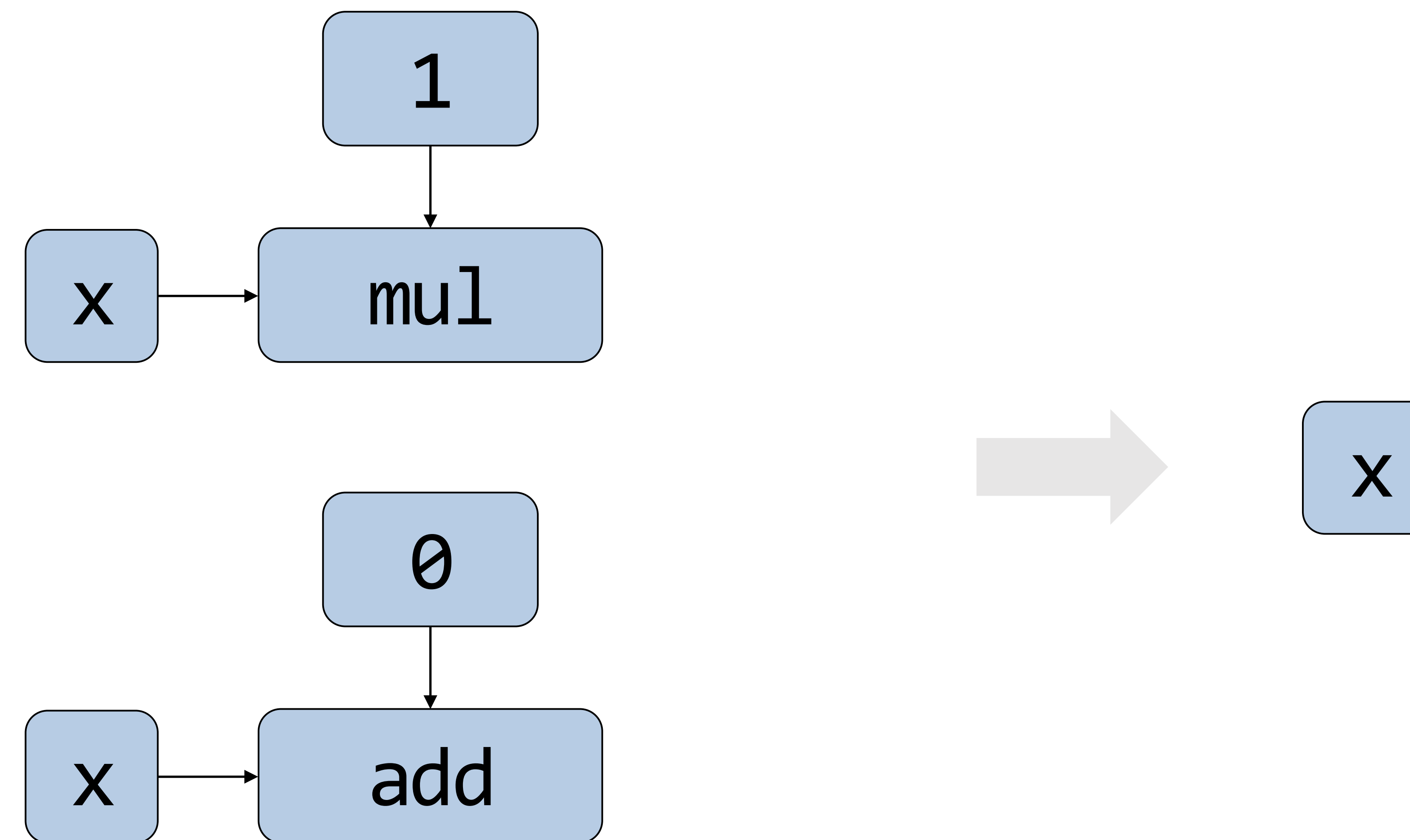
- Users are allowed to program using primitives with high-level APIs
- Graph is captured at CUDA level



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

Graph Optimization Templates: Constant Folding





Common Subexpression Elimination (CSE)

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

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

CSE hit

Dead Code Elimination (DCE)

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

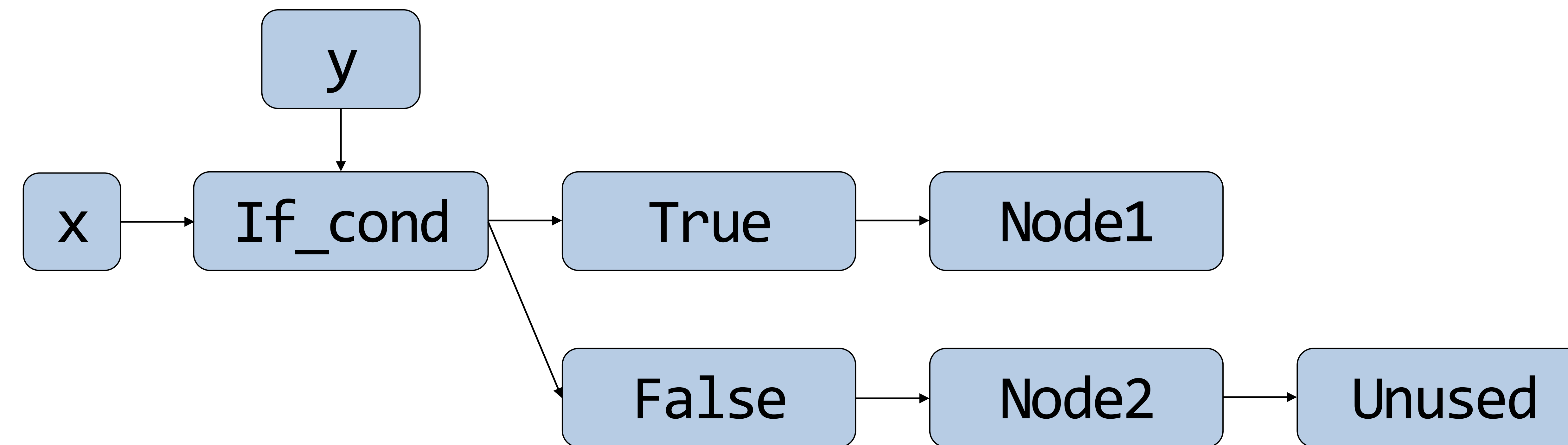
$$\begin{gathered} \ldots \\ c^3 = a^1 + b^2 \\ d^1 = a^1 \\ e^2 = b^2 \\ \require{cancel}\cancel{f^3 = d^1 + e^2} \\ f^3 = c^3 \\ d^4 = x^4 \\ \ldots. \end{gathered}$$

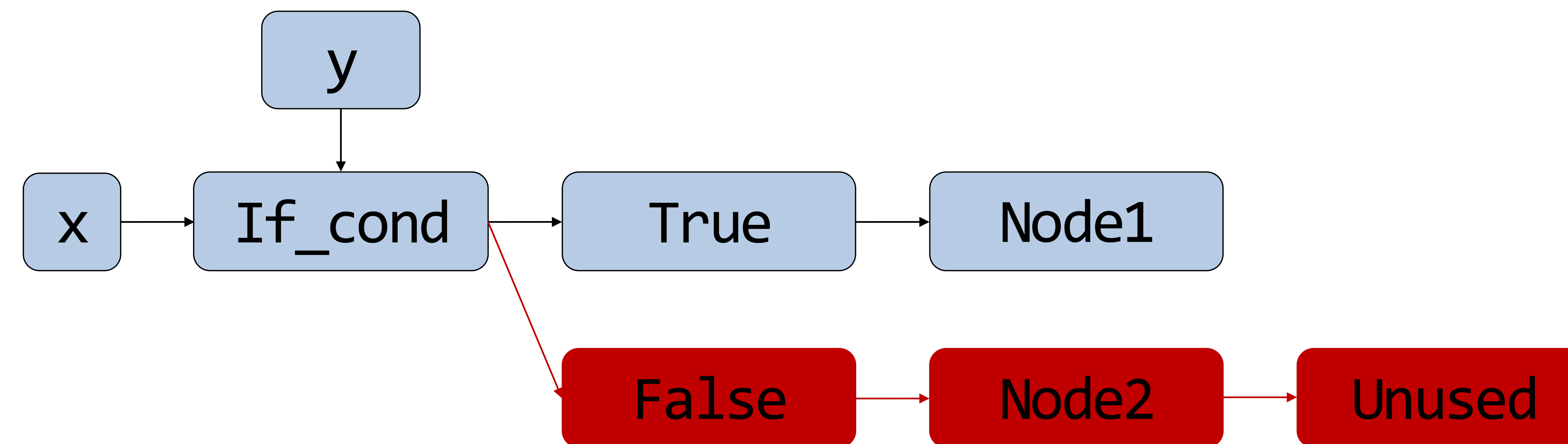
CSE hit

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

DCE hit

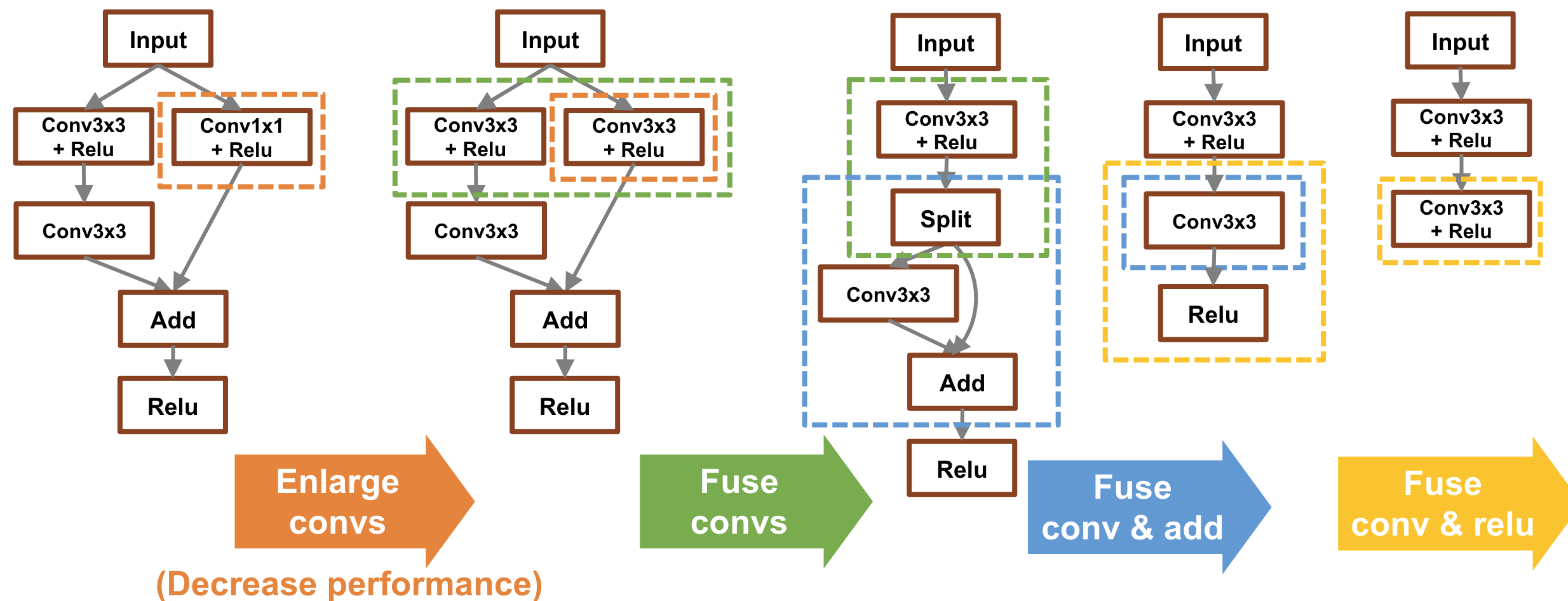
More templates for CSE and DCE





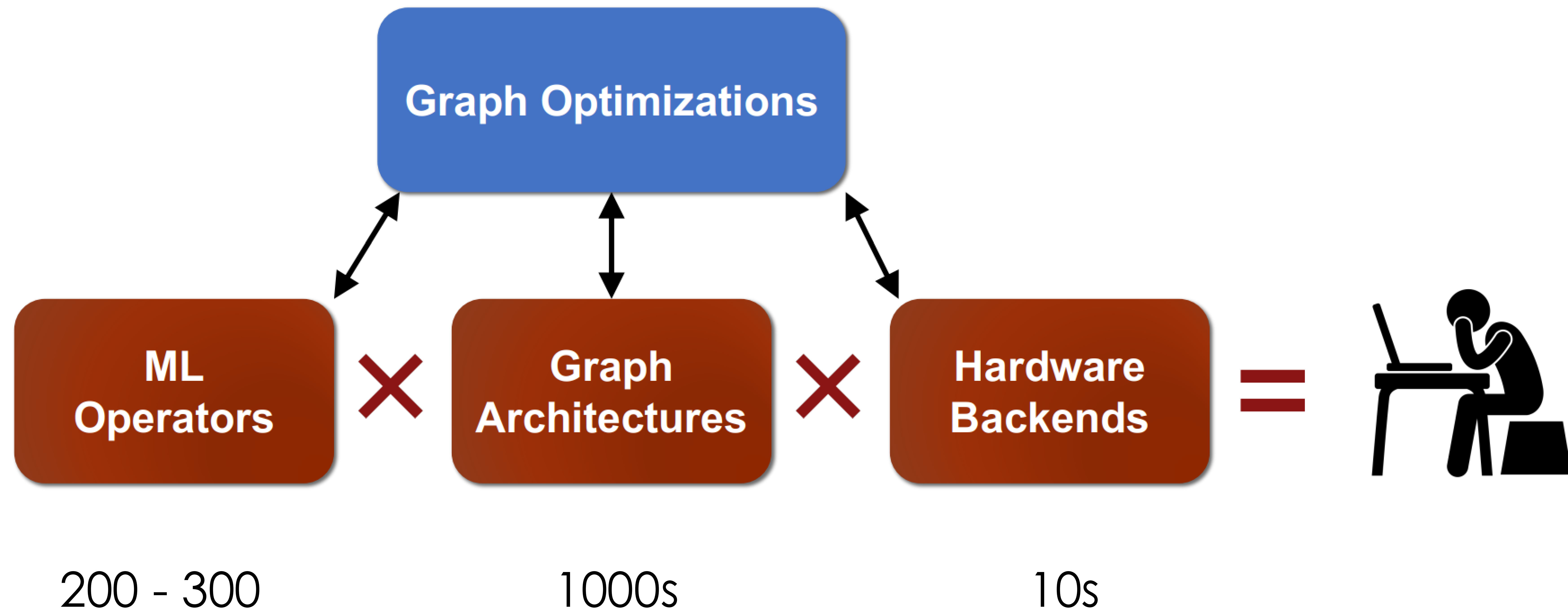
How to ensure performance gain?

- Greedily apply graph optimizations



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

Problems of Template-based Graph Optimizations



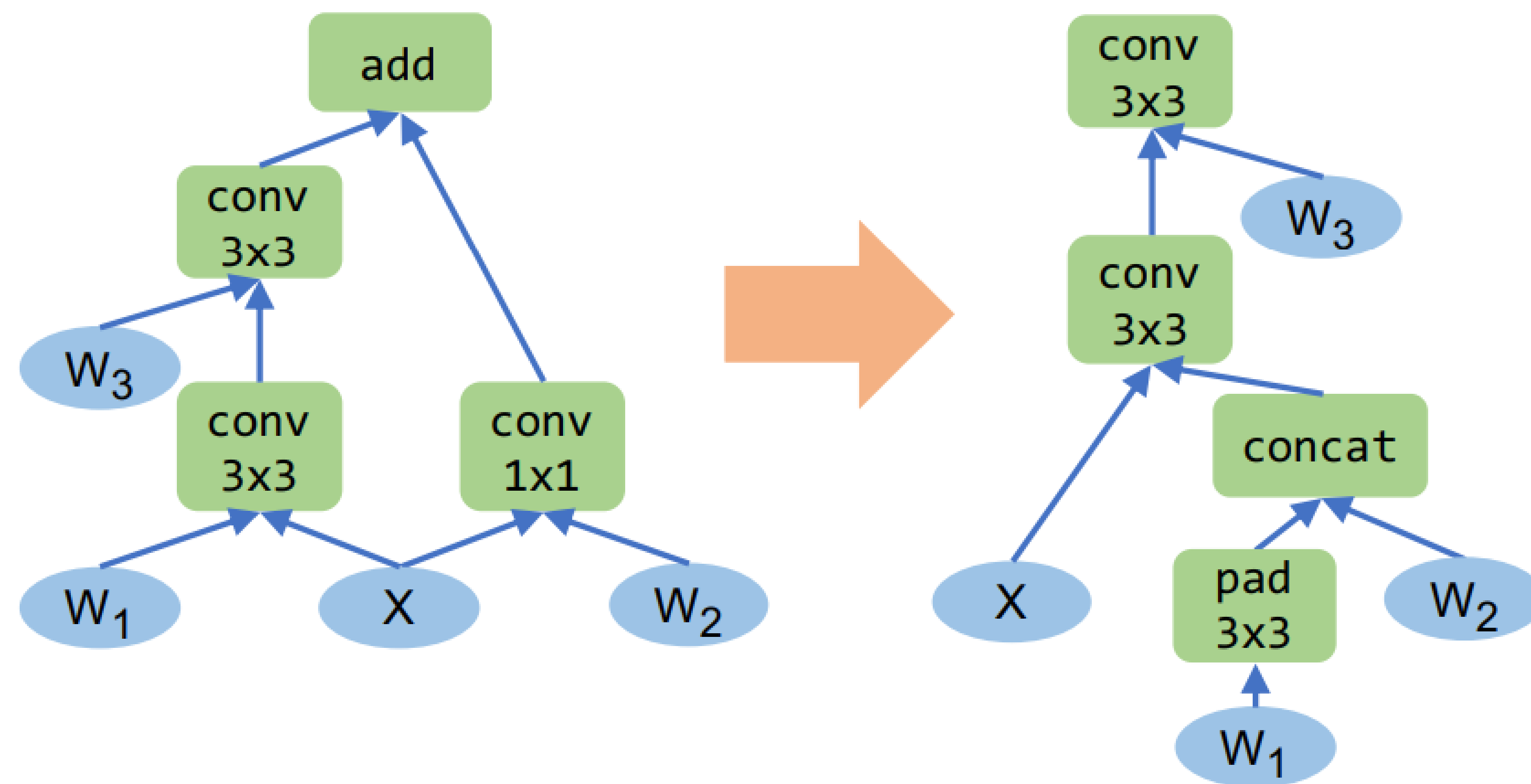
Problem: Infeasible to manually design graph optimizations for all cases

Problems of Template-based Graph Optimizations

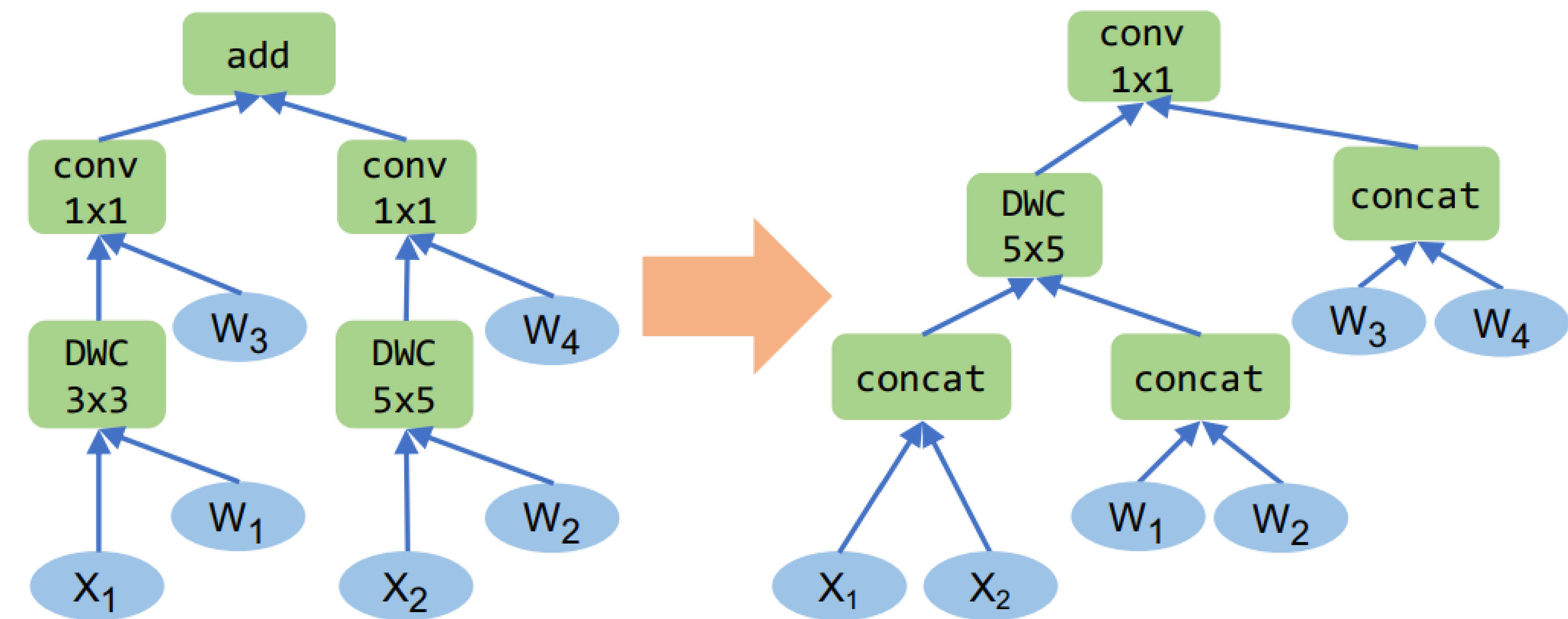
Robustness
Experts' heuristics do not apply to all DNNs/hardware

Scalability
New operators and graph structures require more rules

Performance
Miss subtle optimizations for specific DNNs/hardware



Only apply to **specific hardware**

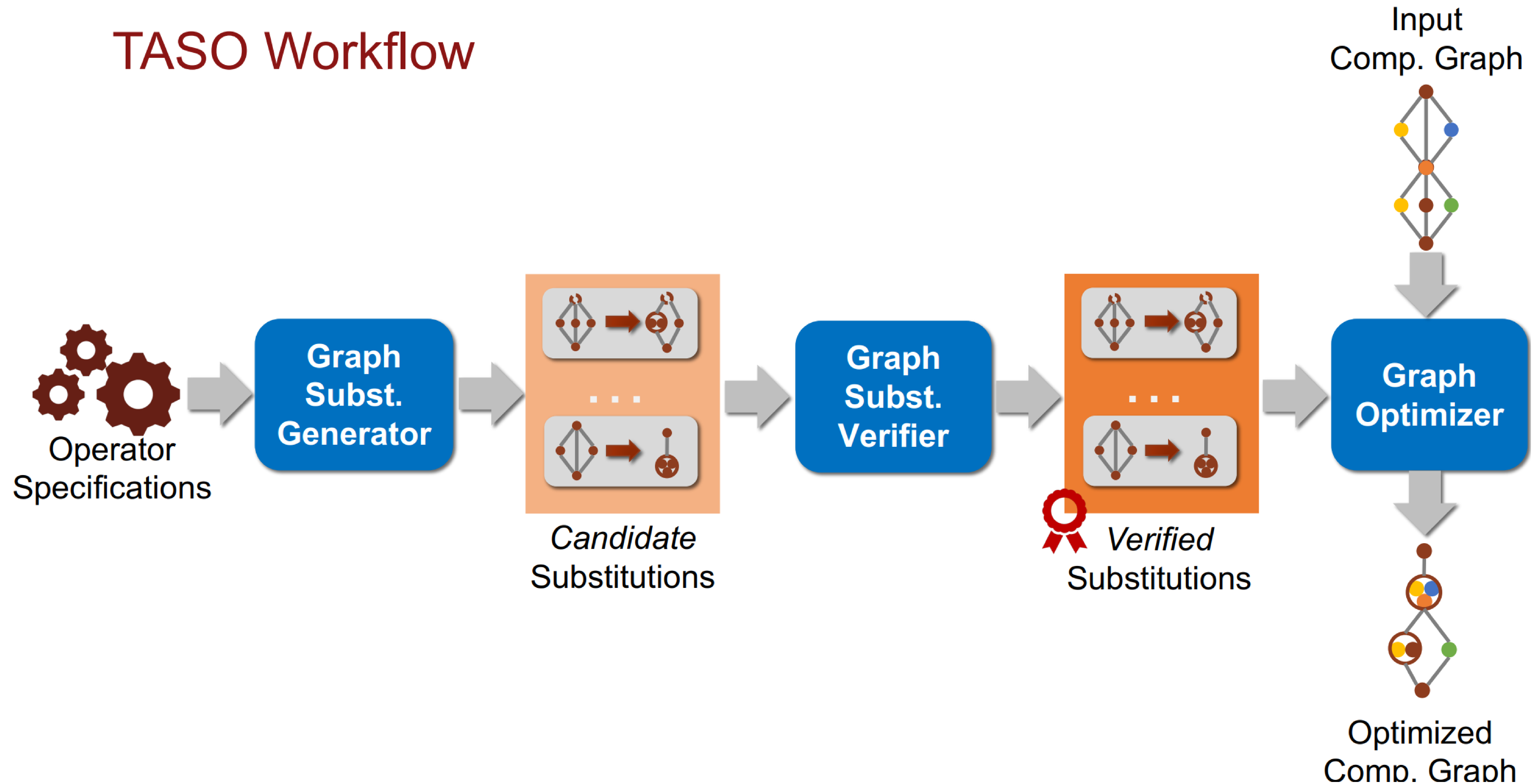


Only apply to **specialized graph structures**

Automate Graph Transformation

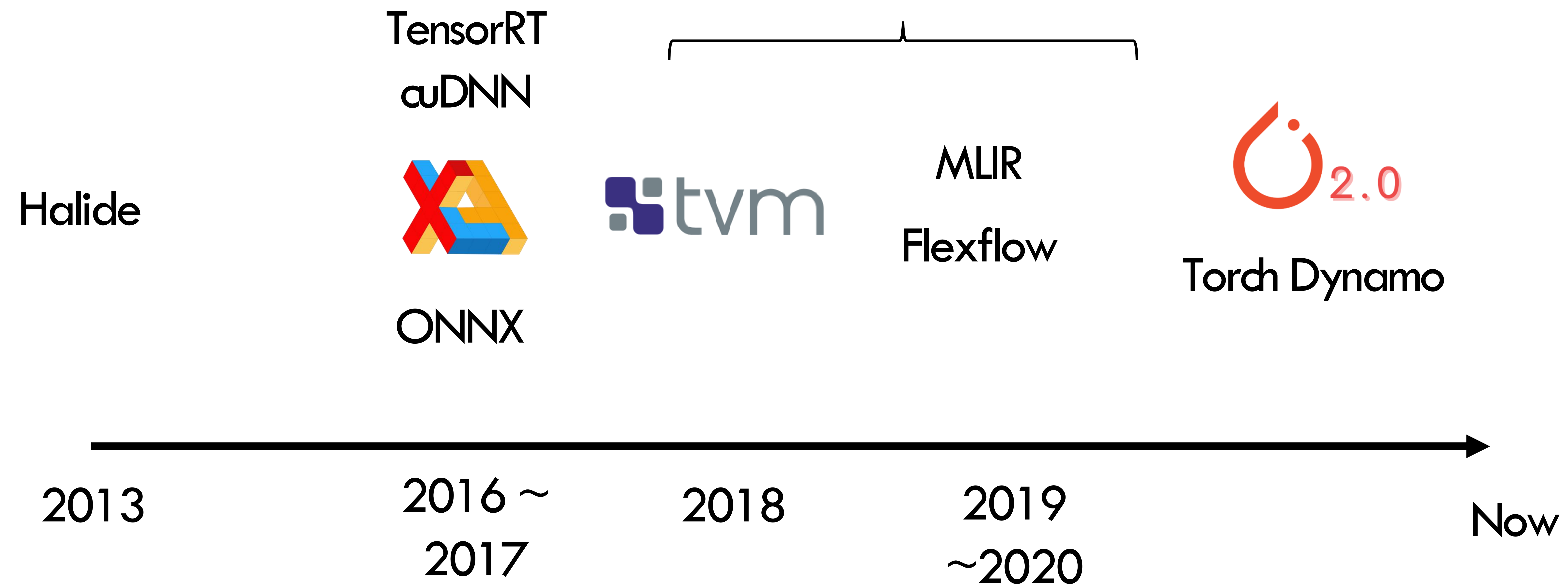
Key idea: replace manually-designed graph optimizations with automated generation and verification of graph substitutions for tensor algebra

Enumerate and Verify ALL possible graph



ML Compiler Retrospective

Q: why the community shifts away from compiler



Big Picture: Where We Are