"PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation"

By Ziming Zhou, Tobias Alam, Sahil Vemuri, and Yuewen Hou

Eager vs. Compiled: The Core Tension

Arbitrary Python Code (torch, dict, conditionals, …)

Interpreted Execution

Flexible. Pythonic. Research-driven.

PyTorch

Graph-like Representation (e.g., Tensorflow APIs only)

Compiler | Optimization Passes

Execution

Scalable. Structured. Production-ready.

TensorFlow

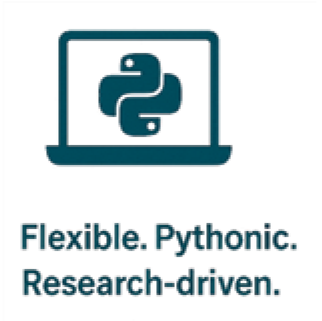
Eager "Just write Python"

Compiled "Make it Optimized"

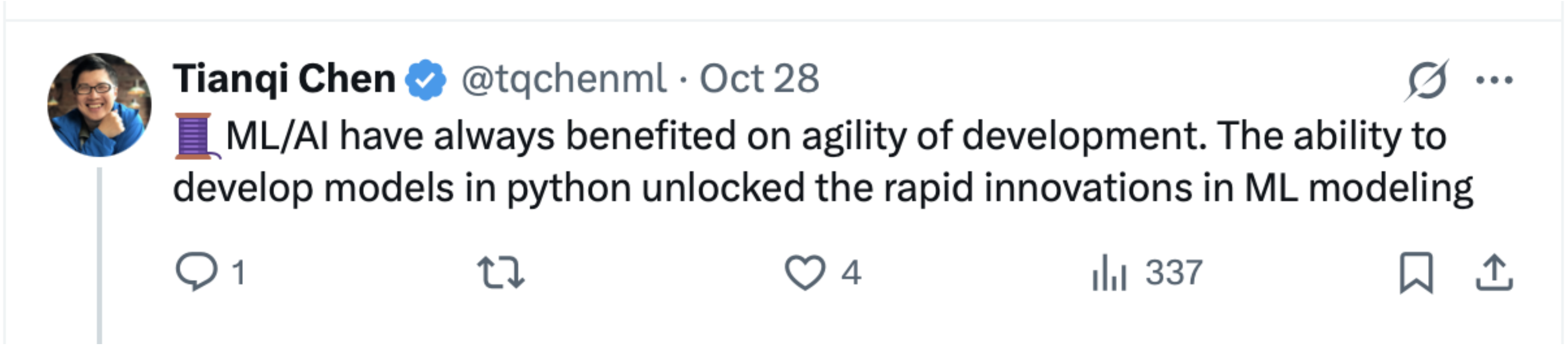
Eager Won: Worse is Better

Arbitrary Python Code (torch, dict, conditionals, ...)

Interpreted Execution



PyTorch

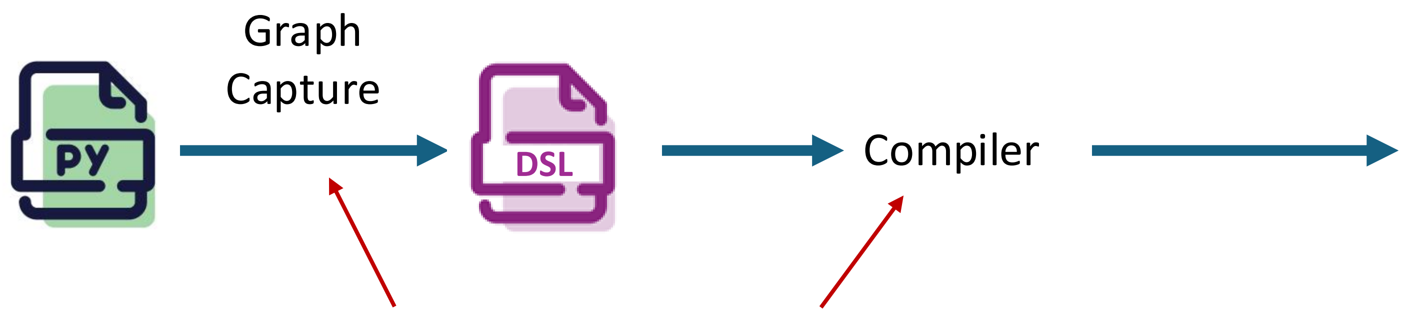

Tianqi Chen @tqchenml · Oct 28

🧵 ML/AI have always benefited on agility of development. The ability to develop models in python unlocked the rapid innovations in ML modeling

Eager "Just write Python"

Compiled "Make it Optimized"

AI Infra Drives a more "Compiled" Approach



Now people wants:

- Auto Parallelism
- Auto Kernel Performance Tuning
- Multi-device Support
- Overlap of Computation and Communication

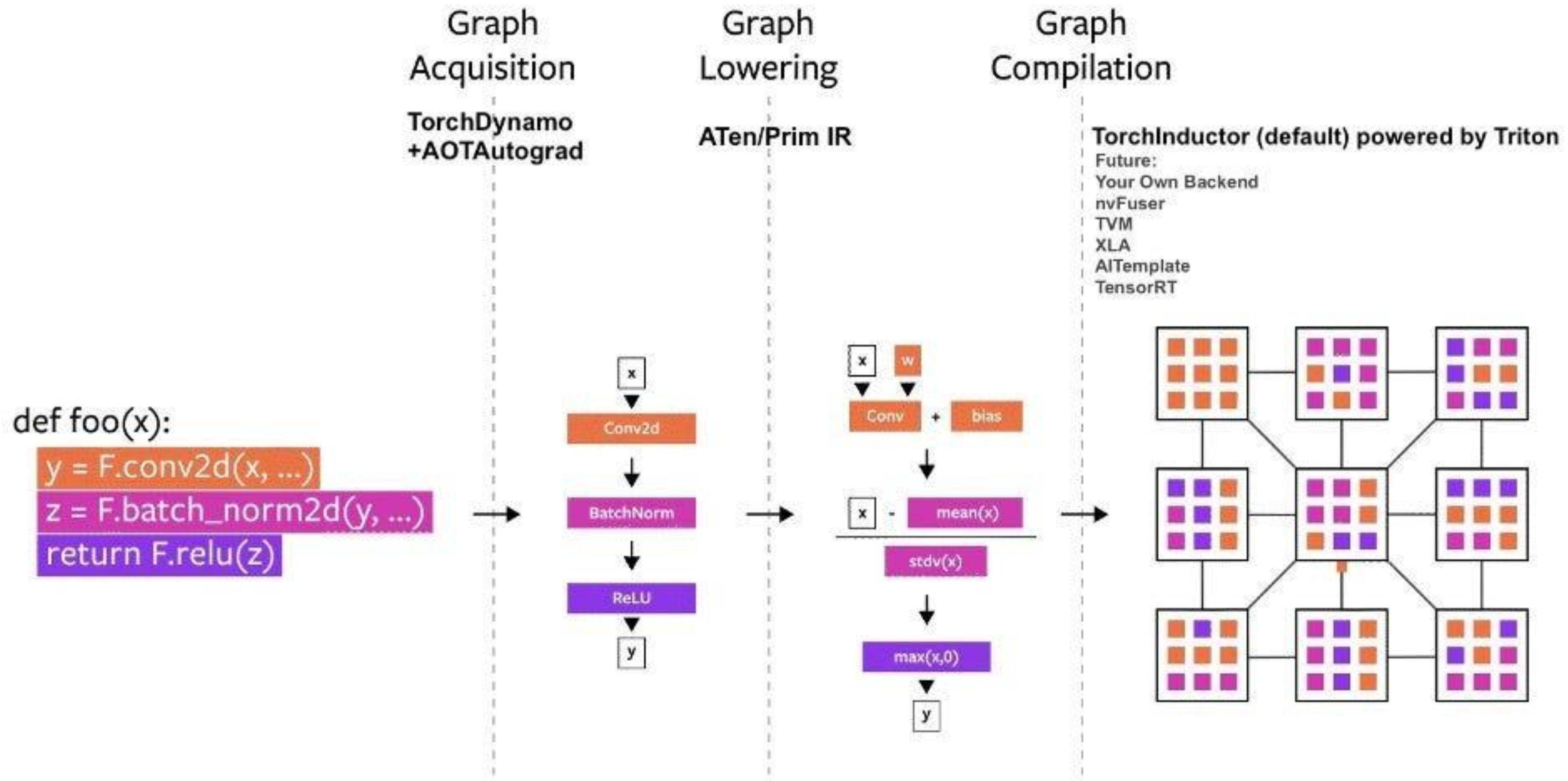
Both are very challenging tasks!

- Non-trivial effort to implement everything, when everything moves so fast
- Hackability and debuggability are hard to maintain

PyTorch Core Philosophy: Best-effort, Transparency, and Modularity

Eager
Compiled

Torch Compiler Ecosystem



TorchDynamo

TorchDynamo Bytecode Analysis -VariableTrackers

TorchDynamo symbolically evaluates Python bytecode

- Each Python object is tracked by a Variable Tracker
    - `torch.*ops` - `TorchVariable`
    - `torch.tensor` - `TensorVariable`
    - `Python builtinvariables` - `BuiltInVariable`
    - `Python lists/dicts` - `ListVariable, DictVariable`
- Operations on a TensorVariable adds a FX node in the graph

```
LOAD_GLOBALtorch []
LOAD_ATTR clamp_min [TorchVariable(<module 'torch' from '/scratch/anijain/work/pytorch/torch/__init__.py'>)]
LOAD_FASTa [TorchVariable(<built-in method clamp_min of type object at 0x7f258f1d2b80>)]
LOAD_FASTb [TorchVariable(<built-in method clamp_min of type object at 0x7f258f1d2b80>), TensorVariable()]
CALL_FUNCTION2 [TorchVariable(<built-in method clamp_min of type object at 0x7f258f1d2b80>), TensorVariable(), ConstantVariable(int)]
LOAD_CONST3 [TensorVariable()]
BINARY_MULTIPLY None [TensorVariable(), ConstantVariable(int)]
RETURN_VALUE None [TensorVariable()]
```

TorchDynamo’s Symbolic Evaluation of Python Bytecode

TorchDynamo Guards and Graph Breaks

- Guards - set of conditions observed during JIT compilation
- Used to determine whether graph can be reused from cache
- Graph breaks used on encountering unsupported Python construct
- TorchDynamo is called again on the continuation function at its invocation

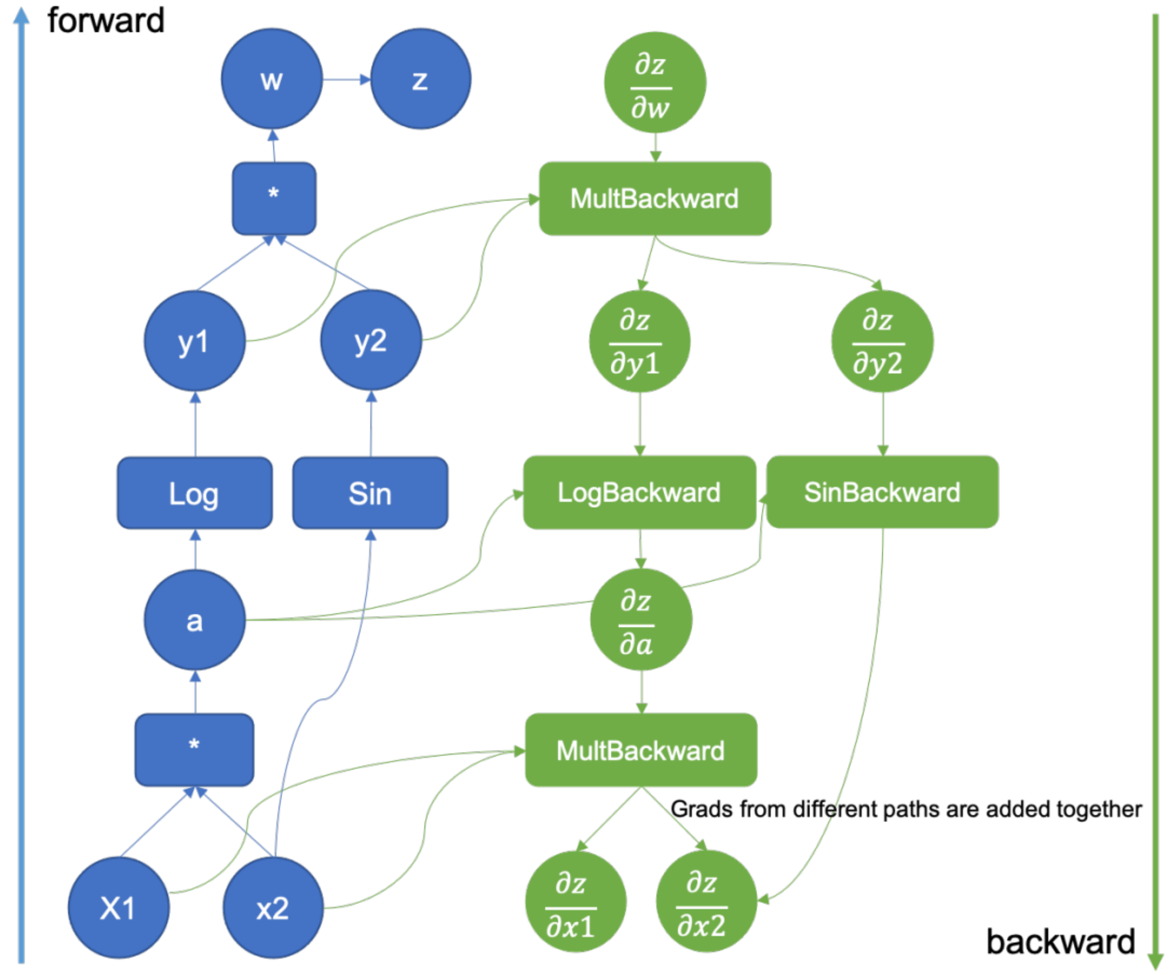
```
def f(x):                      def f(x) after Dynamo's rewrite:
   a = foo(x)       —▶         a = call_fx_graph_1(x)   # compiled prefix
   <unsupported>                 <run in Python>        # graph break
   b = bar(a)                    b = cont(a)            # continuation call

def cont(a):
   b = bar(a)
   return b
# cont(...) will itself be Dynamo-compiled on first call.
```

AOTAutograd

Autograd

Automatically determine the gradient of the model!

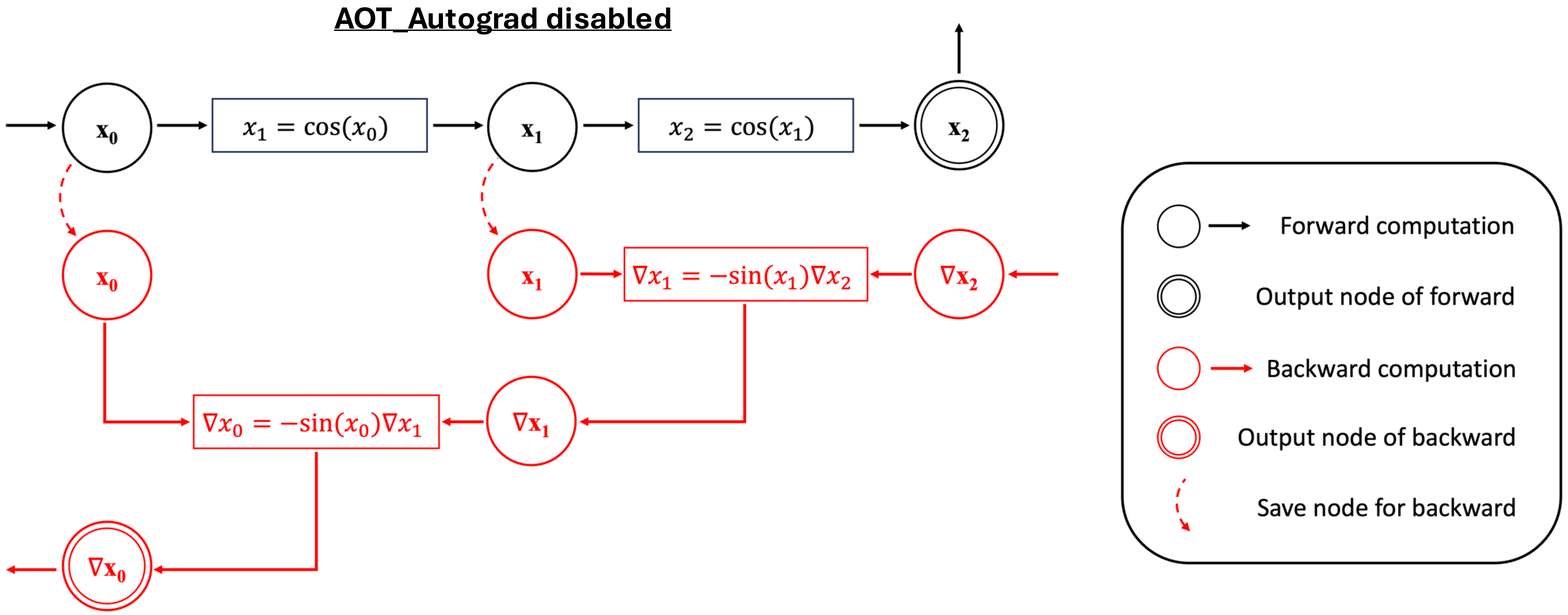
- Model at this stage represented as graph
- Autograd traverses the directed acyclic graph starting at the root node
- Leaf nodes are processed according to the chain rule



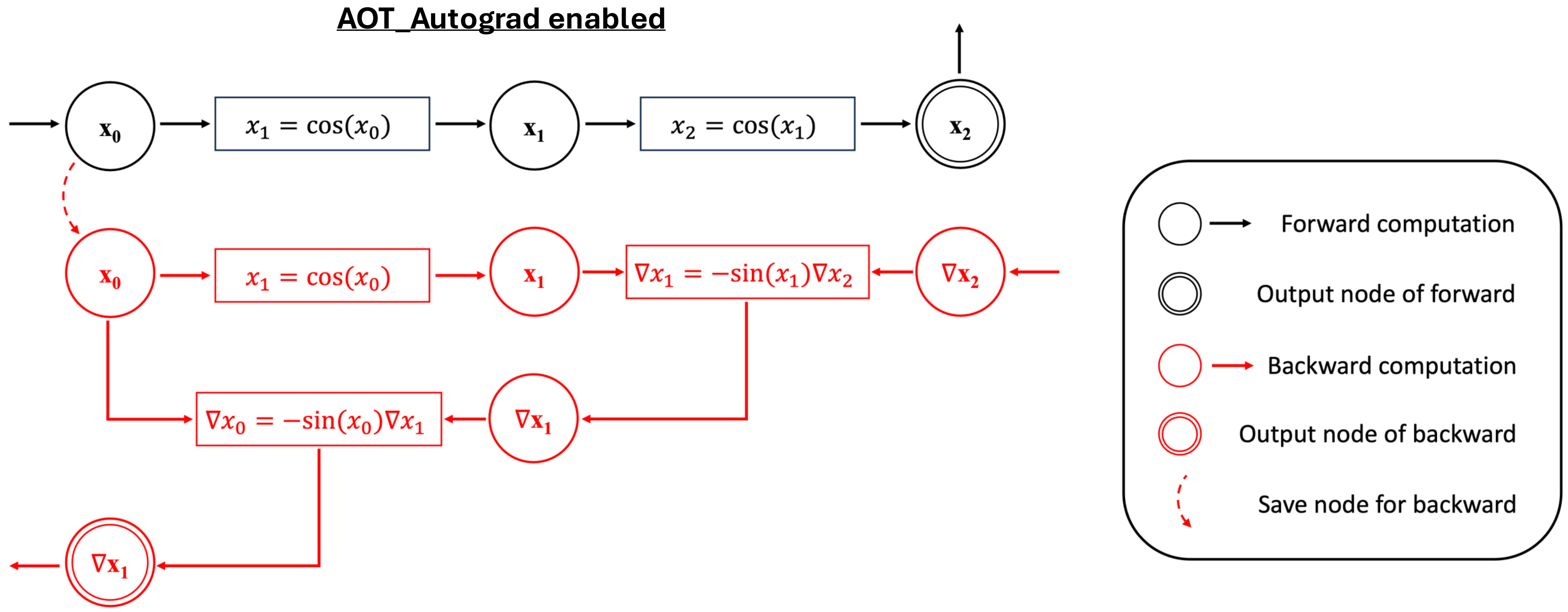
forward
w
z
*
y1
y2
Log
Sin
a
*
X1
x2
$\frac{\partial z}{\partial w}$
MultBackward
$\frac{\partial z}{\partial y1}$
$\frac{\partial z}{\partial y2}$
LogBackward
SinBackward
$\frac{\partial z}{\partial a}$
MultBackward
Grads from different paths are added together
$\frac{\partial z}{\partial x1}$
$\frac{\partial z}{\partial x2}$
backward

Source: PyTorch 2.0 — pytorch.org

Eager Autograd Misses Opportunities

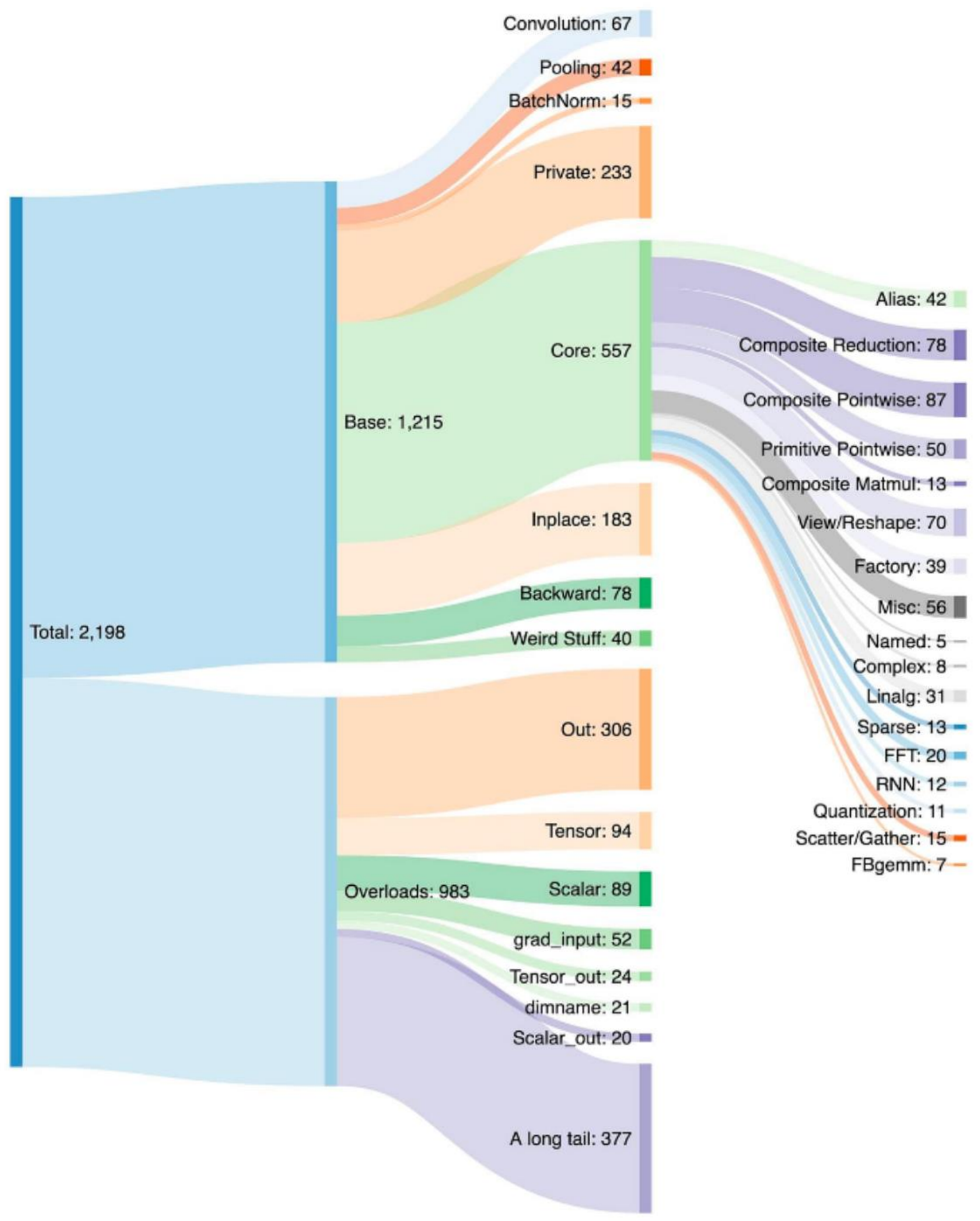


Eager Autograd Misses Opportunities



PrimTorch

- A canonicalization of all ~2000 PyTorch operators to a small set of ~250 primitive operators
- Makes developing a custom backend for PyTorch much easier



Source: PyTorch 2.0 — pytorch.org

Functionalization

If you have: `x.add_(y)`, convert into `x_new = x.add(y)`

What if you have an alias? **`x2 = x[0]; x.add_(y)`** Must

update all aliases! Functionalization knows to do this:

```
x2_new = x2.add(y[0])
x_new = x.add(y)
```

Note: must know if operators mutate or not! Captured by JIT schema

Functionalization

If you have: **`x.add_(y)`**, convert into **`x_new = x.add(y)`**

What if you have an alias? **x2 = x[0]; x.add_(y)** Must

update all aliases! Functionalization knows to do this:

```
x2_new = x2.add(y[0])

x_new = x.add(y)
```

Note: must know if operators mutate or not! Captured by JIT schema

Pure, single-assignment dataflow (SSA-like)

TorchInductor

Design Principles

- PyTorch Native
  - Similar abstractions to PyTorch eager to allow support for nearly all of PyTorch, with a thin translation layer.
- Python First
  - Implemented in Python to make it easier for python users to contribute since the majority of PyTorch users most comfortable in Python.
- Breadth First
  - Intentional early focus on a wide variety of operators, hardware, and optimization. This makes it more general purpose.
- Reuse State-of-the-Art Languages
  - Increasing popularity of OpenAI Triton DSL for GPU Kernels; typical CPU kernels written in C++/OpenMP.
  - Generates both Triton and C++ as output code to leverage and make it more understandable.

Decompositions

- Instead of lowering all PyTorch operators to TorchInductor’s IR, many are decomposed into simpler, easier-to-handle ops.
- Happens using AOTAutograd.

Example:

```
log2_scale = 1 / math.log(2)

@register_decomposition(torch.ops.aten.log2)
def log2(x):
  return torch.log(x) * log2_scale
```

Lowerings and Define-By-Run Loop-Level IR

- In the next phase, the FX graph is lowered into TorchInductor’s define-by-run IR, which uses executable Python code for flexibility and concise lowerings.
- This Python-based IR is easy to construct, analyze, and generate code from e.g., Triton or C++.

Example:

```
def inner_fn_buf0(index):
  i0, i1 = index
  tmp0 = ops.load("arg0_1", i0 * s1 + i1)
  tmp1 = ops.log(tmp0)
  tmp2 = ops.constant(1.4426950408889634, torch.float32)
  tmp3 = ops.mul(tmp1, tmp2)
  return tmp3

buf0_ir = TensorBox(StorageBox(ComputedBuffer(
  name='buf0',
  layout=FixedLayout('cuda', torch.float32,
                     size=[s0, s1], stride=[s1, 1]),
  data=Pointwise(inner_fn=inner_fn_buf0,
                 ranges=[s0, s1], ...))))
```

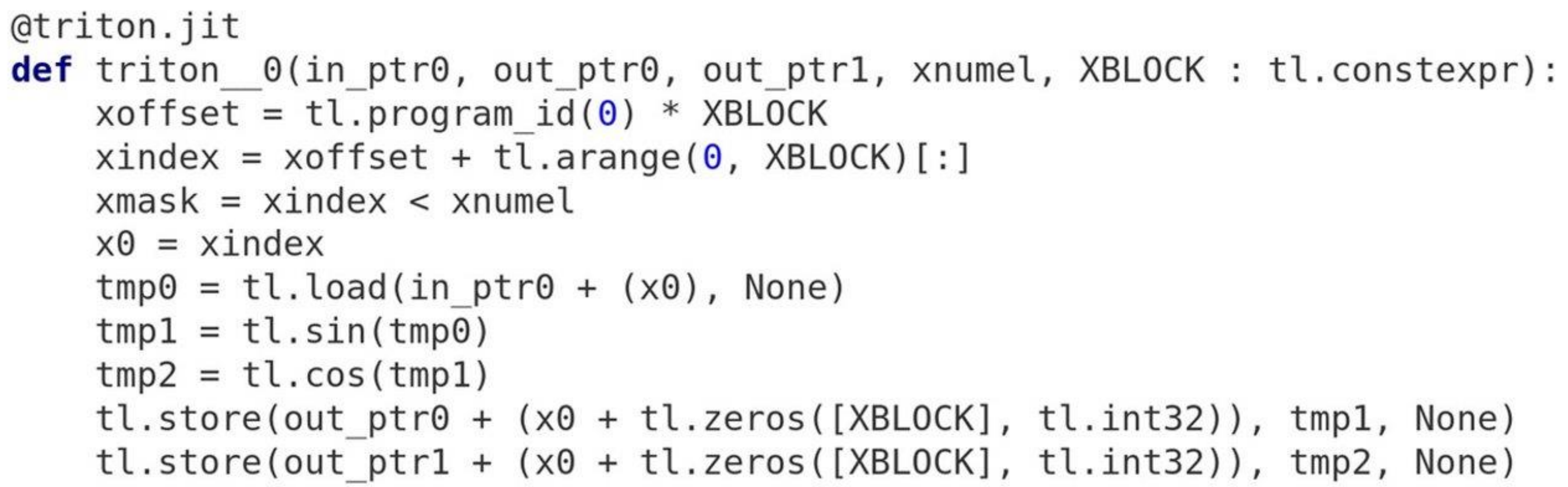
Scheduling

- Scheduling decides fusion, kernel order, and memory reuse.
- Buffers become scheduler nodes with dependency edges based on memory accesses.
- TorchInductor greedily fuses nodes using legality and efficiency scores.

Triton Codegen

- TorchInductor converts its IR into optimized Triton kernels, simplifying indexing and applying common subexpression elimination.
- Pointwise kernels operate element-wise on blocks of data with masking for non-divisible sizes.
- Reduction kernels use either persistent reductions (small tensors in registers/shared memory) or block-based reductions (accumulating over a block and calling Triton reduction).

Example:



```python
@triton.jit
def triton__0(in_ptr0, out_ptr0, out_ptr1, xnumel, XBLOCK : tl.constexpr):
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + (x0), None)
    tmp1 = tl.sin(tmp0)
    tmp2 = tl.cos(tmp1)
    tl.store(out_ptr0 + (x0 + tl.zeros([XBLOCK], tl.int32)), tmp1, None)
    tl.store(out_ptr1 + (x0 + tl.zeros([XBLOCK], tl.int32)), tmp2, None)
```

- Complex ops like matmuls and convolutions use Jinja-based templates combining handwritten and generated Triton code for precise control.

C++ Codegen

- For CPU backend, TorchInductor generates C++ code using OpenMP with vectorized and non-vectorized variants.
- The vectorized path uses PyTorch's *at::vec::Vectorized* class for SIMD execution on 16 elements at a time.
- The non-vectorized path emits standard C++ with STL functions for general compatibility.
- Both variants use #pragma omp for parallelization, with reductions mapped to OpenMP or manual loops.

Wrapper Codegen

- Generates Python or C++ wrappers to launch kernels, manage tensor sizes, and handle memory.
- Can use CUDA Graphs to minimize overhead by recording and replaying kernel launches safely.

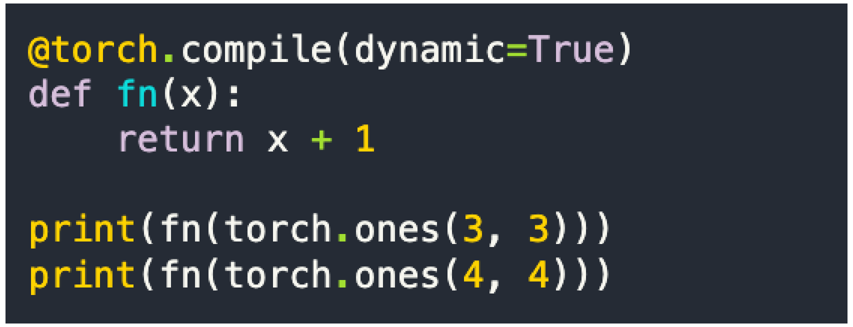
Dynamic Shapes

Why?

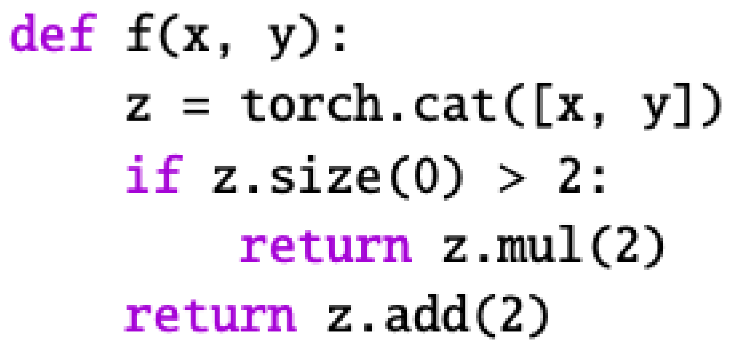
- Varying length of batch size and sequence length in LLM serving
- Data-dependent output shapes. E.g. : return all non-zero element of a tensor
- Sparse-representation.

How?

- Symbolic Shape Guards
- Supported by meta functions to propagate shape information of all PyTorch operation
- Optimized Dynamic Shapes Reasoning



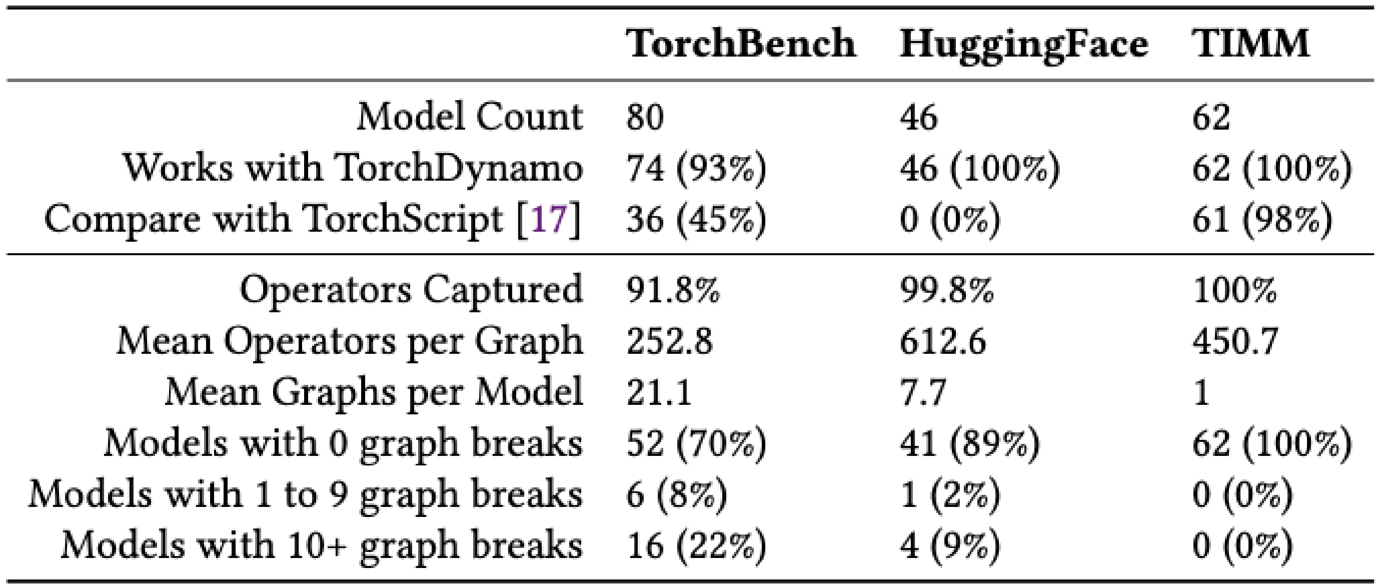
```
@torch.compile(dynamic=True)
def fn(x):
    return x + 1

print(fn(torch.ones(3, 3)))
print(fn(torch.ones(4, 4)))
```



```
def f(x, y):
    z = torch.cat([x, y])
    if z.size(0) > 2:
        return z.mul(2)
    return z.add(2)
```

Experimental Result

Ability to capture graph

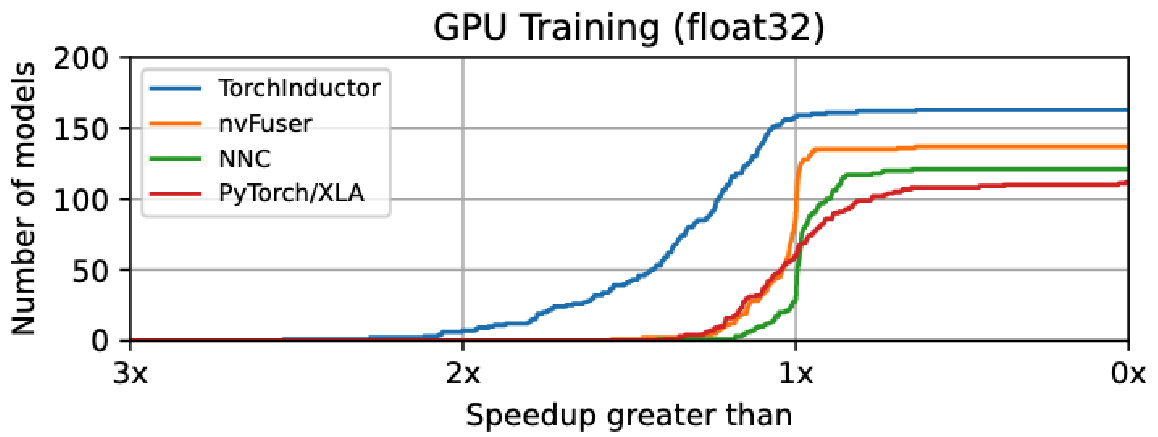


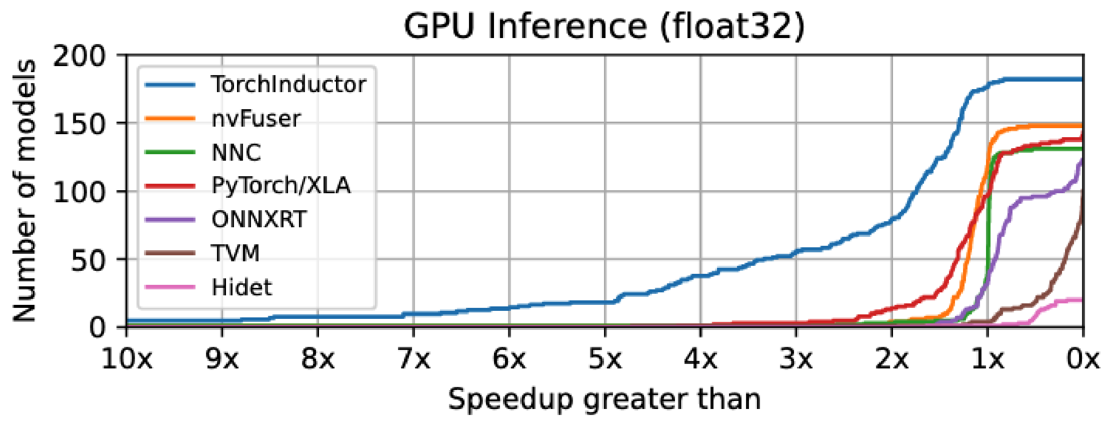
	TorchBench	HuggingFace	TIMM
Model Count	80	46	62
Works with TorchDynamo	74 (93%)	46 (100%)	62 (100%)
Compare with TorchScript [17]	36 (45%)	0 (0%)	61 (98%)
Operators Captured	91.8%	99.8%	100%
Mean Operators per Graph	252.8	612.6	450.7
Mean Graphs per Model	21.1	7.7	1
Models with 0 graph breaks	52 (70%)	41 (89%)	62 (100%)
Models with 1 to 9 graph breaks	6 (8%)	1 (2%)	0 (0%)
Models with 10+ graph breaks	16 (22%)	4 (9%)	0 (0%)

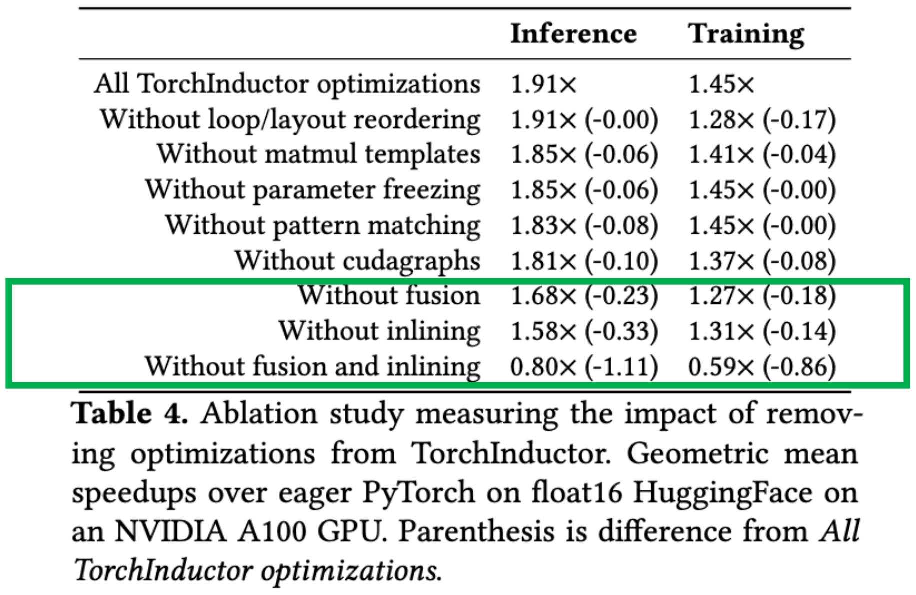
Table 1. TorchDynamo statistics from each benchmark suite, measured using float32 inference on an NVIDIA A100 GPU.

TorchBench is the most representative benchmark

Speedup







	Inference	**Training**
All TorchInductor optimizations	1.91×	1.45×
Without loop/layout reordering	1.91× (-0.00)	1.28× (-0.17)
Without matmul templates	1.85× (-0.06)	1.41× (-0.04)
Without parameter freezing	1.85× (-0.06)	1.45× (-0.00)
Without pattern matching	1.83× (-0.08)	1.45× (-0.00)
Without cudagraphs	1.81× (-0.10)	1.37× (-0.08)
Without fusion	1.68× (-0.23)	1.27× (-0.18)
Without inlining	1.58× (-0.33)	1.31× (-0.14)
Without fusion and inlining	0.80× (-1.11)	0.59× (-0.86)

Table 4. Ablation study measuring the impact of removing optimizations from TorchInductor. Geometric mean speedups over eager PyTorch on float16 HuggingFace on an NVIDIA A100 GPU. Parenthesis is difference from *All TorchInductor optimizations.*

Conclusion: compiler as a Tool Kit

- torch.compile
  - Designed initially for ordinary researchers to gain “free” performance boosts
  - Not much fine-grained control (e.g., backend, parameters)

- Some problems:
  - Debugging opaqueness;
  - Implementing custom optimization passes is still difficult;
  - A lot of optimization design looks “half-baked”;

- **Graph capture + Flexible control on what to do with the graph**
- **PyTorch 2.9: Piecewise CUDAGraph (to better support custom ops)**
- **GraphMend: Code Transformations for Fixing Graph Breaks in PyTorch 2**