TorchDynamo: Deep Dive

Animesh Jain

Software Engineer, Meta

Agenda

- Motivation
- How does it work?
- How does it support training?
- Backends
- Results

TorchDynamo: Motivation

The Great ML Framework Debate

Eager Mode

- Preferred by users
- Easier to use programming model
- Easy to debug
- PyTorch is a primarily an eager mode framework

Graph Mode

- Preferred by backends and framework builders
- Easier to optimize with a compiler
- Easier to do automated transformations

PyTorch's Many Attempts at Graph Modes

torch.jit.trace

- Record + replay
- Unsound
- Can give incorrect results because it ignores Python part of program

torch.jit.script

- AOT parses Python into graph format
- Only works on ~45% of real world models
- High effort to "TorchScript" models

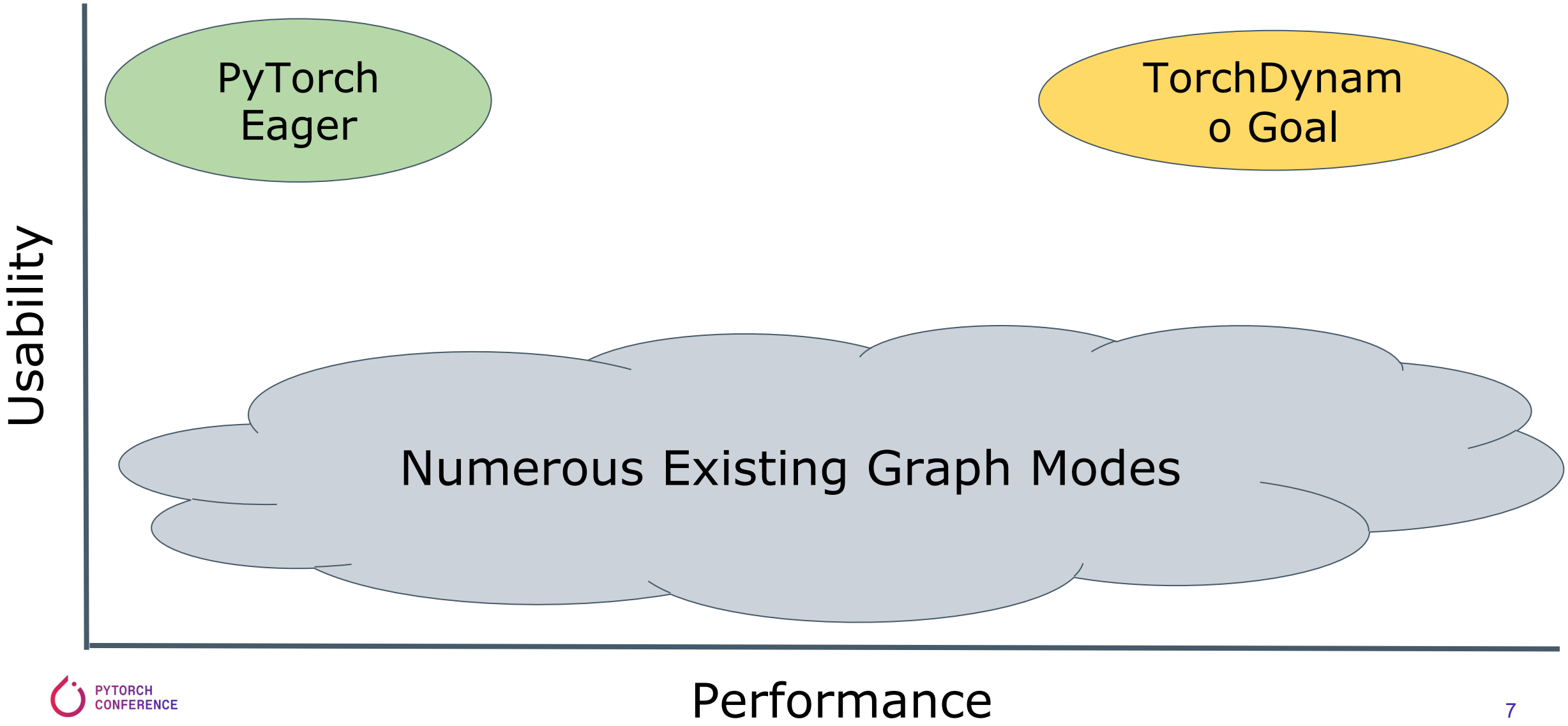
PyTorch Models Are Not Static Graphs

PyTorch users write models where program graphs are impossible

- Convert tensors native Python types (x.item(), x.tolist(), int(x), etc)
- Use other frameworks (numpy/xarray/etc) for part of their model
- Data dependent Python control flow or other dynamism
- Exceptions, closures, generators, classes, etc

These violate the assumptions of most graph mode backends

PyTorch Usability/Performance Tradeoff



7k+

Crawled GitHub Models

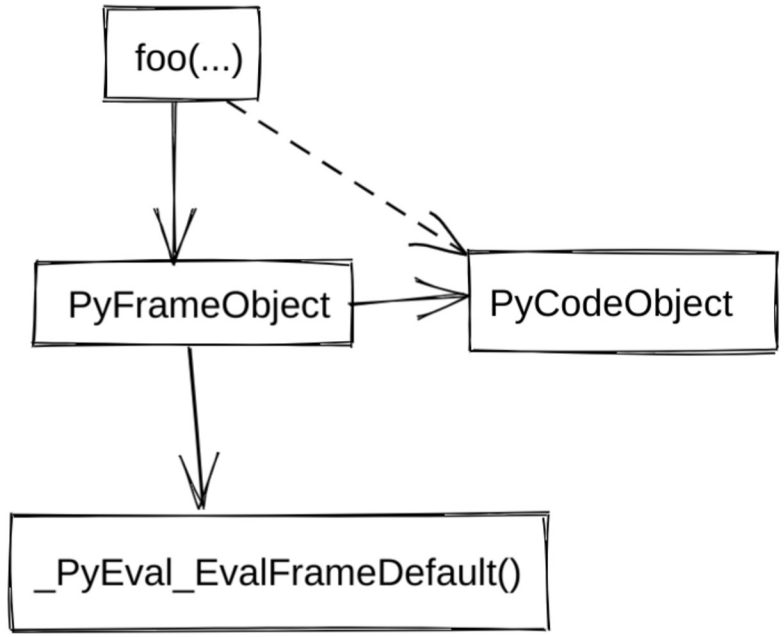
TorchDynamo

Python-level JIT compiler to make unmodified PyTorch programs run faster

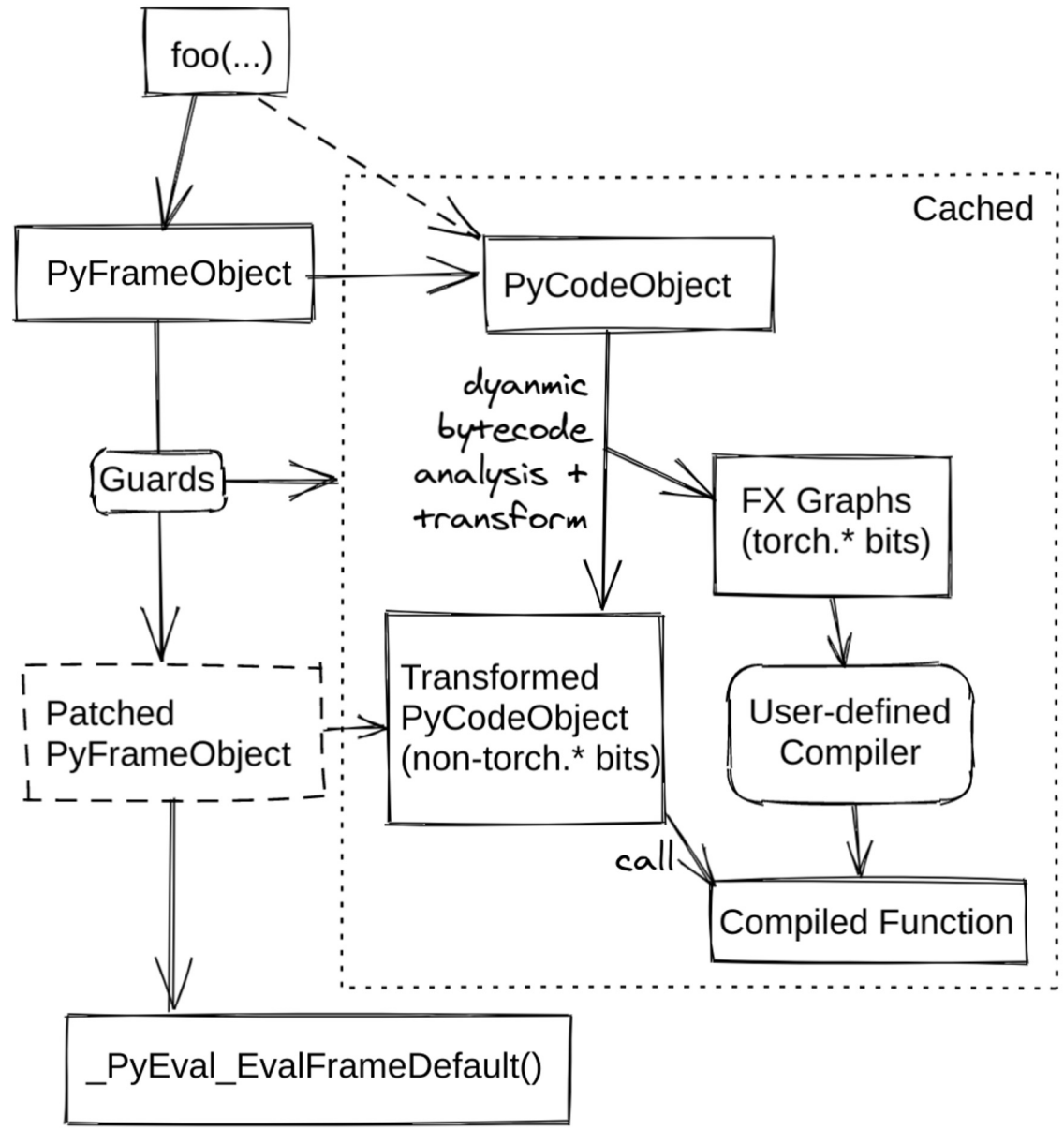
- Sprinkles graphs in eager mode
- Fallback to Python for hard-to-accelerate (non-graph) code

TorchDynamo: How does it work?

Default Python Behavior



TorchDynamo Behavior



PYTORC
CONFER

Toy Example

```python
def fn(x):
    a = torch.sin(x)
    b = torch.cos(x)
    return a * b

x = torch.randn(10)
opt_fn = torch._dynamo.optimize("eager")(fn)
opt_fn(x)
```

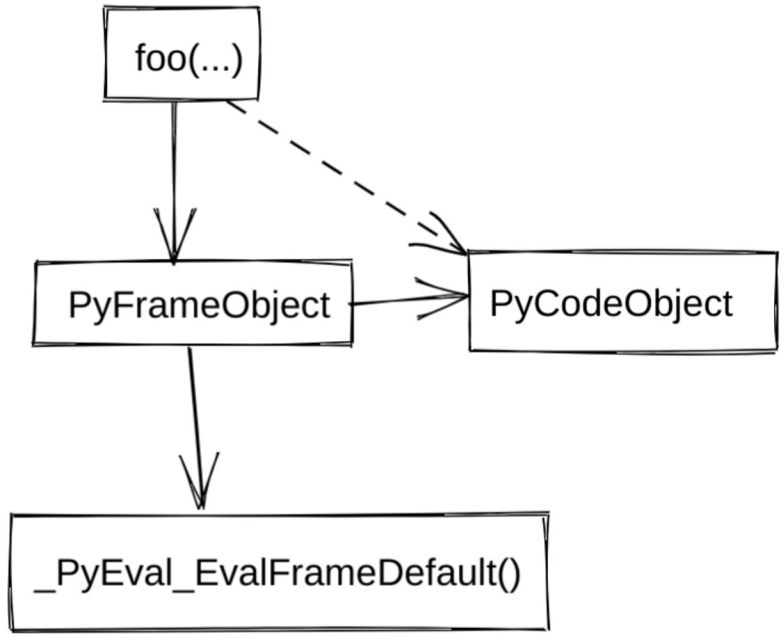
Memory Offset | Instruction | Argument Raw (Decoded)

```
0  LOAD_GLOBAL        0 (torch)
2  LOAD_METHOD        1 (sin)
4  LOAD_FAST          0 (x)
6  CALL_METHOD        1
8  STORE_FAST         2 (a)

10 LOAD_GLOBAL        0 (torch)
12 LOAD_METHOD        2 (cos)
14 LOAD_FAST          0 (x)
16 CALL_METHOD        1
18 STORE_FAST         3 (b)

20 LOAD_FAST          2 (a)
22 LOAD_FAST          3 (b)
24 BINARY_MULTIPLY
26 RETURN_VALUE
```

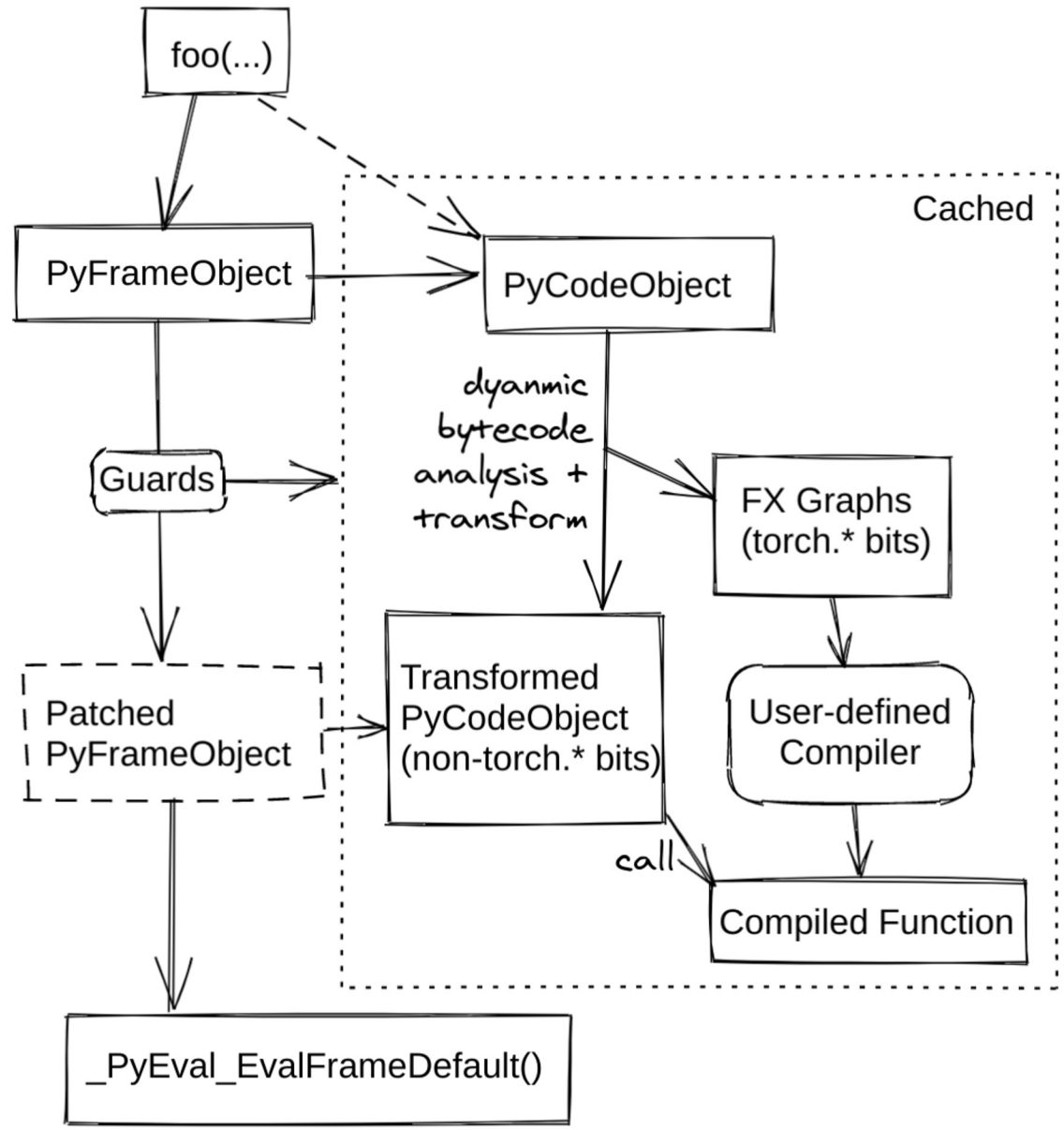
When opt_fn is called, TorchDynamo takes control:

- custom_eval_frame(PyFrameObject* frame)
  - frame->f_locals
    - {“x”: tensor([...]), “y”: tensor([...])}
  - frame->f_globals
    - {“torch”: ..., ...}
  - frame->f_code
    - Bytecode
    - ...
  - ...

Default Python Behavior



TorchDynamo Behavior



PYTORC
CONFER

TorchDynamo: Bytecode Analysis and Graph Extraction

```
 0 LOAD_GLOBAL              0 (torch)
 2 LOAD_METHOD              1 (sin)
 4 LOAD_FAST                0 (x)
 6 CALL_METHOD              1
 8 STORE_FAST               2 (a)

10 LOAD_GLOBAL              0 (torch)
12 LOAD_METHOD              2 (cos)
14 LOAD_FAST                0 (x)
16 CALL_METHOD              1
18 STORE_FAST               3 (b)

20 LOAD_FAST                2 (a)
22 LOAD_FAST                3 (b)
24 BINARY_MULTIPLY
26 RETURN_VALUE
```

Original Python Bytecode

```
def forward(self, x : torch.Tensor):
    sin = torch.sin(x)
    cos = torch.cos(x);  x = None
    mul = sin * cos;  sin = cos = None
    return (mul,)
```

Extracted Fx Graph

```
0 LOAD_GLOBAL              3 (__compiled_fn_0)
2 LOAD_FAST                0 (x)
4 CALL_FUNCTION            1
6 UNPACK_SEQUENCE          1
8 RETURN_VALUE
```

Optimized Python Bytecode

TorchDynamo: Integration with Backend Compiler

```
 0 LOAD_GLOBAL              0 (torch)
 2 LOAD_METHOD              1 (sin)
 4 LOAD_FAST                0 (x)
 6 CALL_METHOD              1
 8 STORE_FAST               2 (a)

10 LOAD_GLOBAL              0 (torch)
12 LOAD_METHOD              2 (cos)
14 LOAD_FAST                0 (x)
16 CALL_METHOD              1
18 STORE_FAST               3 (b)

20 LOAD_FAST                2 (a)
22 LOAD_FAST                3 (b)
24 BINARY_MULTIPLY
26 RETURN_VALUE
```

Original Python Bytecode

```
def forward(self, x : torch.Tensor):
    sin = torch.sin(x)
    cos = torch.cos(x);  x = None
    mul = sin * cos;  sin = cos = None
    return (mul,)
```

Extracted Fx Graph

```
def my_compiler(gm, example_inputs):
    return optimized_fn(gm, example_inputs)
```

```
0 LOAD_GLOBAL              3 (__compiled_fn_0)
2 LOAD_FAST                0 (x)
4 CALL_FUNCTION            1
6 UNPACK_SEQUENCE          1
8 RETURN_VALUE
```

Optimized Python Bytecode

Backend compilers accept a Fx graph module and return optimized callable

TorchDynamo: Control flow

```python
def fn(a, b):
    x = a / (torch.abs(a) + 1)
    if b.sum() < 0:
        b = b * -1
    return x * b
```

```python
def forward(self, a : torch.Tensor, b : torch.Tensor):
    abs_1 = torch.abs(a)
    add = abs_1 + 1;  abs_1 = None
    truediv = a / add;  a = add = None
    sum_1 = b.sum();  b = None
    lt = sum_1 < 0;  sum_1 = None
    return (truediv, lt)
```

```python
def fn(a, b):
    x, cond = compiled_fn(a, b)
    if x:
        return resume_true_br(x, b)
    else:
        return resume_false_br(x, b)
```

Optimized Python (Pseudo) Continuation

```python
def resume_true_br(x, b):
    b = b * -1
    return x * b
```

```python
def resume_false_br(x, b):
    return x * b
```

TorchDynamo will be invoked again when Python Interpreter runs these branches

- Uses continuation to handle control flow and graph breaks
- Compiles whenever a new branch is explored

TorchDynamo: How does it accelerate training?

TorchDynamo works at Python Bytecode

But PyTorch Autograd engine sits at C++ level

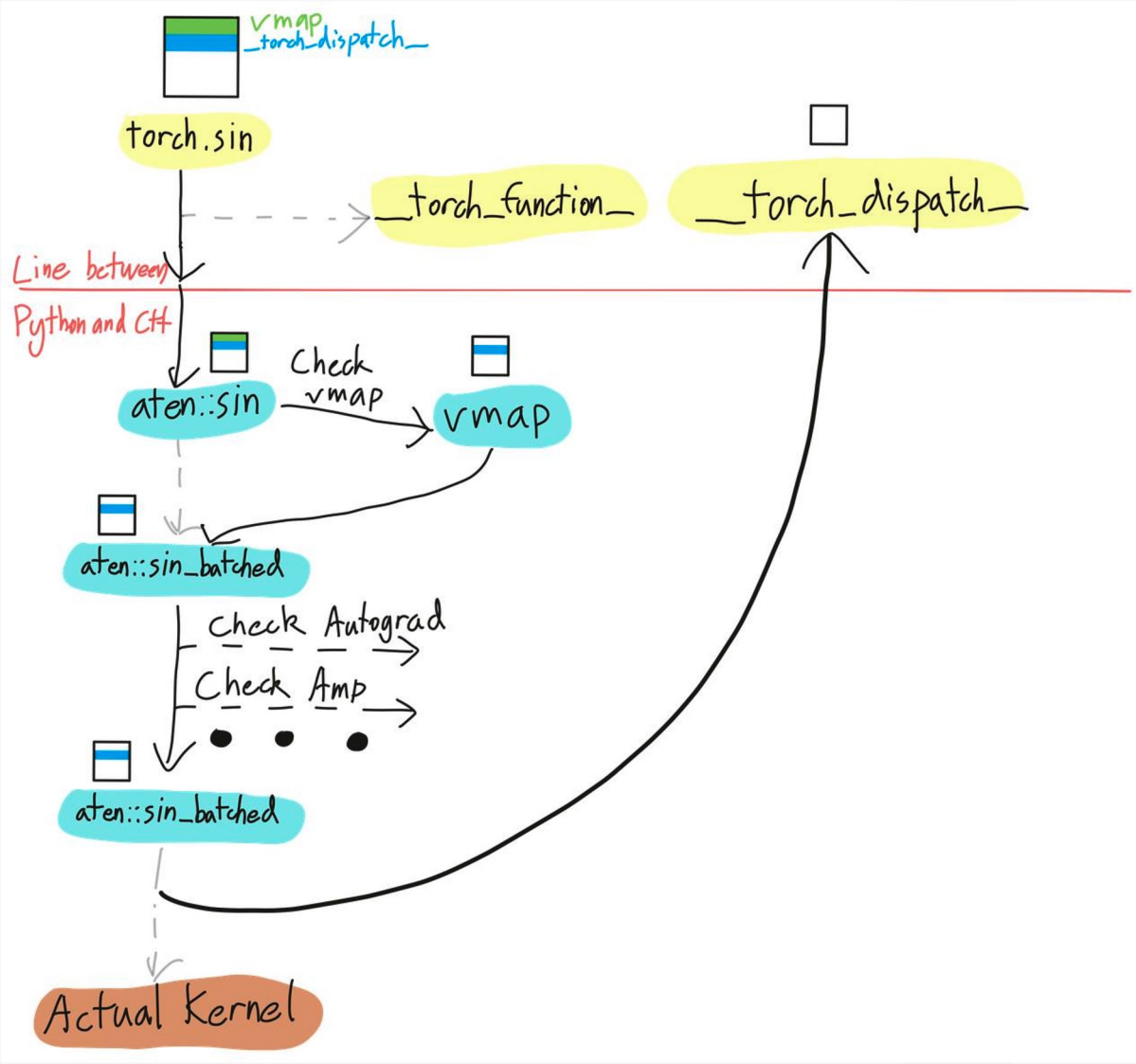
TorchDynamo works at Python Bytecode

But PyTorch Autograd engine sits at C++ level

Use AOTAutograd - Extract fwd and bwd graphs

AOTAutograd Backend For Graph Capture

- Based on __torch__dispatch__
- Traces forward and backward graphs AOT
- Fx-aten IR
- Powerful abstraction



Toy Example: AOTAutograd

```python
def fn(x):
    a = torch.sin(x)
    b = torch.cos(x)
    return a * b

x = torch.randn(10)
opt_fn = torch._dynamo.optimize("aot_eager")(fn)
opt_fn(x)
```

```
====== Forward graph ======
class GraphModule(torch.nn.Module):
    def forward(self, primals_1: f32[10]):
        sin: f32[10] = torch.ops.aten.sin.default(primals_1)

        cos: f32[10] = torch.ops.aten.cos.default(primals_1)

        mul: f32[10] = torch.ops.aten.mul.Tensor(sin, cos)
        return [mul, cos, sin, primals_1]
```

- Graphs generated AOT
- Graphs can be now optimized with backend compilers

```
====== Backward graph ======
class GraphModule(torch.nn.Module):
    def forward(self, cos: f32[10], sin: f32[10], primals_1: f32[10], tangents_1: f32[10]):
        mul_1: f32[10] = torch.ops.aten.mul.Tensor(tangents_1, sin);  sin = None
        mul_2: f32[10] = torch.ops.aten.mul.Tensor(tangents_1, cos);  tangents_1 = cos = None

        sin_1: f32[10] = torch.ops.aten.sin.default(primals_1)
        neg: f32[10] = torch.ops.aten.neg.default(sin_1);  sin_1 = None
        mul_3: f32[10] = torch.ops.aten.mul.Tensor(mul_1, neg);  mul_1 = neg = None

        cos_1: f32[10] = torch.ops.aten.cos.default(primals_1);  primals_1 = None
        mul_4: f32[10] = torch.ops.aten.mul.Tensor(mul_2, cos_1);  mul_2 = cos_1 = None
        add: f32[10] = torch.ops.aten.add.Tensor(mul_3, mul_4);  mul_3 = mul_4 = None
        return [add]
```

TorchDynamo: Backends

Backends

- list_backends() show currently supported backends
- Commonly used ones
  - `inductor` - AOTAutograd + inductor codegen
  - `eager` - pytorch native - useful for debugging
  - `aot_eager` - AOTAutograd graph capture
  - `nvfuser` - uses Nvidia nvfuser compiler for GPUs
- Writing/composing new backends is easy

```
def my_compiler(gm: torch.fx.GraphModule,
                example_inputs: List[torch.Tensor]):
    scripted = torch.jit.trace(gm, example_inputs)
    return torch.jit.optimize_for_inference(scripted)
```

TorchDynamo: Results

TorchDynamo Today

7k+	20+	1+	30+%
Crawled GitHub Models	**Inference Backends**	**Training backends**	**Geomean OOTB speedup (TB, TIMM, HF) (FP32, AMP) A100**

Conclusion

- PyTorch programs can not be fully converted to graphs
- TorchDynamo solves this problem by
  - Analyzing Python bytecode for every callable
  - Extracting Fx graphs and transforming bytecode
- AOTAutograd captures the forward but more importantly backward graphs
- Easy integration with backend compilers

Thank you