PyTorch 2: Faster machine learning through dynamic Python bytecode transformation

Jason Ansel

Principal Research Scientist, Meta

Meta

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

Lazy Tensors (Torch XLA)

- Graph capture through deferred execution
- High overheads
- Performance cliffs

PyTorch Models Are Not Static Graphs

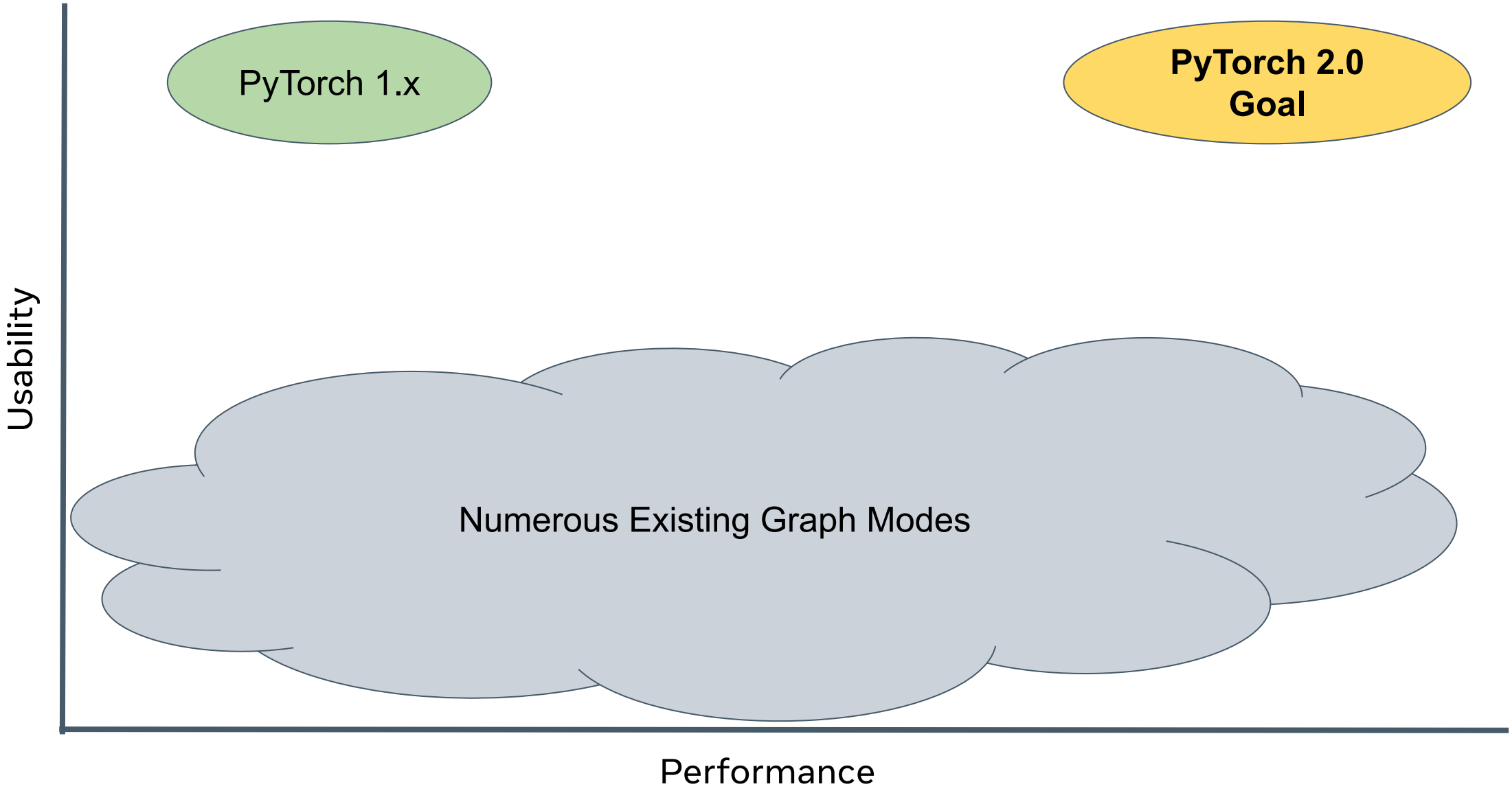
Due to history of being an eager model framework, PyTorch users have written models in ways where whole program graphs are impossible

In our benchmark suite 20% of models, do one (or more) of:

- 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

All of these violate the assumptions of most graph mode backends.

PyTorch Usability/Performance Tradeoff



INTRODUCING

```
cmodel = torch.compile(model)
```

OVERHEAD OPTIMIZED MODE

```
cmodel = torch.compile(
                model,
                mode="reduce-overhead"
            )
```

AUTOTUNING MODE

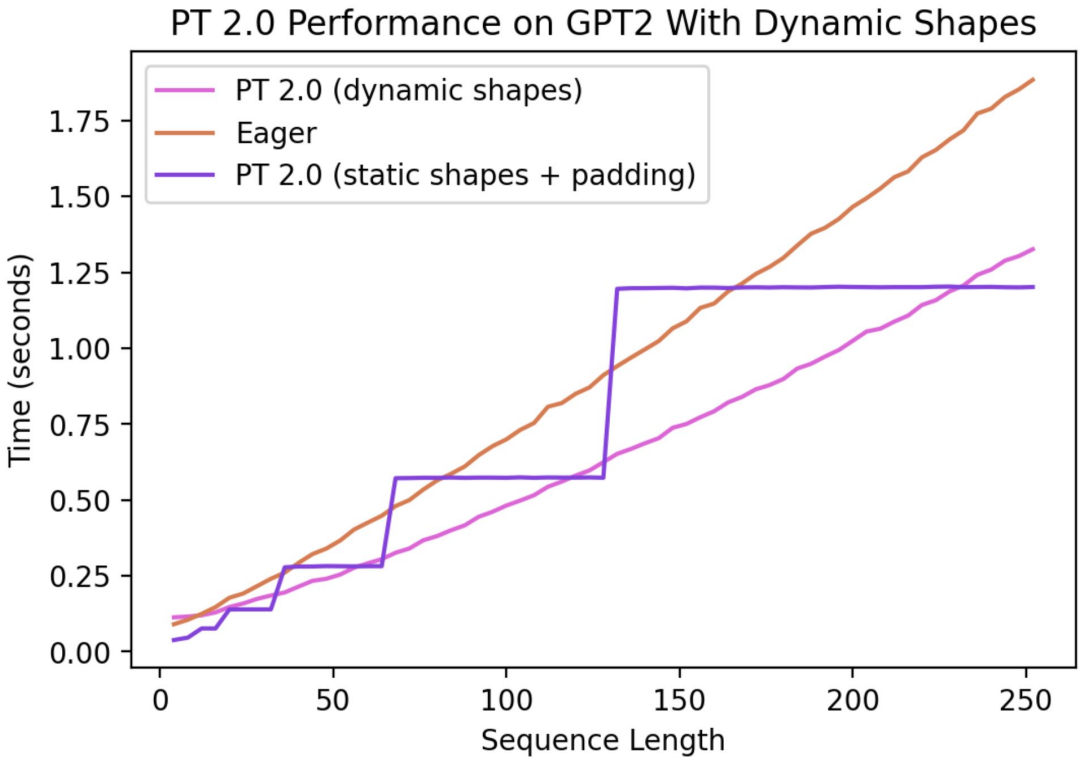
```
cmodel = torch.compile(
            model,
            mode="max-autotune"
        )
```

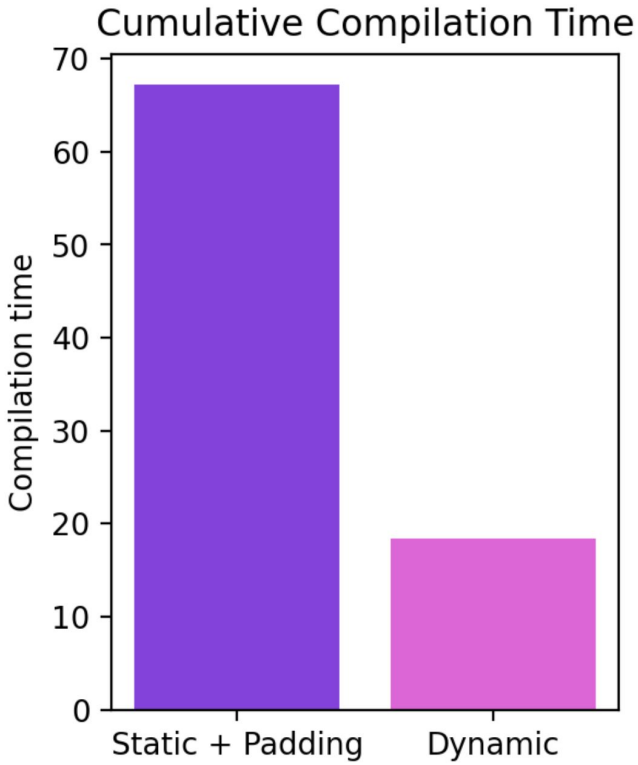
DIFFERENT BACKENDS

```
cmodel = torch.compile(
                model,
                mode="max-autotune",
                backend="inductor"
            )
```

DYNAMIC SHAPES

```
cmodel = torch.compile(model, dynamic=True)
```





FULLGRAPH MODE

```
cmodel = torch.compile(
            model,
            fullgraph=True
        )
```

A SMOOTHER TRANSITION

Eager	torch.compile default (Partial Graphs)	torch.compile with fullgraph=True
Full Python Flexibility	Full Python Flexibility	Restricted Python
User doesn’t change code	User doesn’t change code	User has to significantly modify code
Full Framework overhead	Negligible Framework overhead	No Framework overhead
No code fusion	Code fusion on parts of the graph	Global code fusion and static analysis
Cannot do static analysis	Static analysis, but only in parts	Advanced Distributed algorithms
	No pipeline parallel and automated distributed placement	Mobile
	No Mobile	

IN THE NEAR FUTURE

```
torch.export(model)
```

2.0 is fully backward-compatible by definition!

TorchDynamo:
Out-of-the-box graph capture for PyTorch

torch.compile() with a user-defined backend

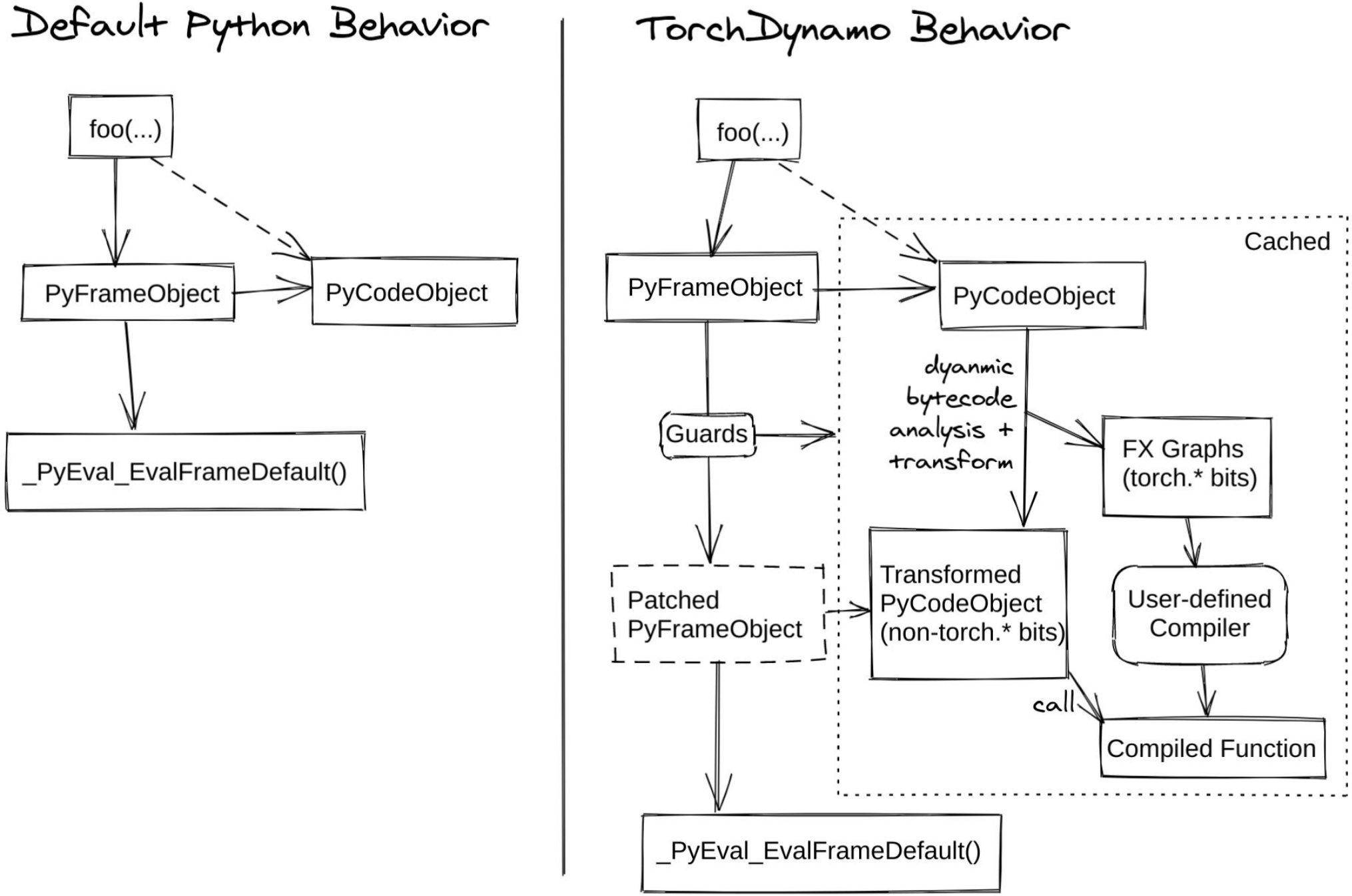
```python
from typing import List
import torch

def my_compiler(gm: torch.fx.GraphModule,
                example_inputs: List[torch.Tensor]):
    print("my_compiler() called with FX graph:")
    gm.graph.print_tabular()
    return gm  # return a python callable

@torch.compile(backend=my_compiler)
def toy_example(a, b):
    x = a / (torch.abs(a) + 1)
    if b.sum() < 0:
        b = b * -1
    return x * b

for _ in range(100):
    toy_example(torch.randn(10), torch.randn(10))
```

Output:

```
my_compiler() called with FX graph:
opcode         name     target                args
-------------  -------  ----------------  ----------------
placeholder    a        a                 ()
placeholder    b        b                 ()
call_function  abs_1    torch.abs         (a,)
call_function  add      operator.add      (abs_1, 1)
call_function  truediv  operator.truediv  (a, add)
call_method    sum_1    sum               (b,)
call_function  lt       operator.lt       (sum_1, 0)
output         output   output            ((truediv, lt),)
```

```
my_compiler() called with FX graph:
opcode         name     target                args
-------------  ------   ----------------  -----------
placeholder    b        b                 ()
placeholder    x        x                 ()
call_function  mul      operator.mul      (b, -1)
call_function  mul_1    operator.mul      (x, mul)
output         output   output            ((mul_1,),)
```

```
my_compiler() called with FX graph:
opcode         name     target                args
-------------  ------   ----------------  ---------
placeholder    b        b                 ()
placeholder    x        x                 ()
call_function  mul      operator.mul      (x, b)
output         output   output            ((mul,),)
```

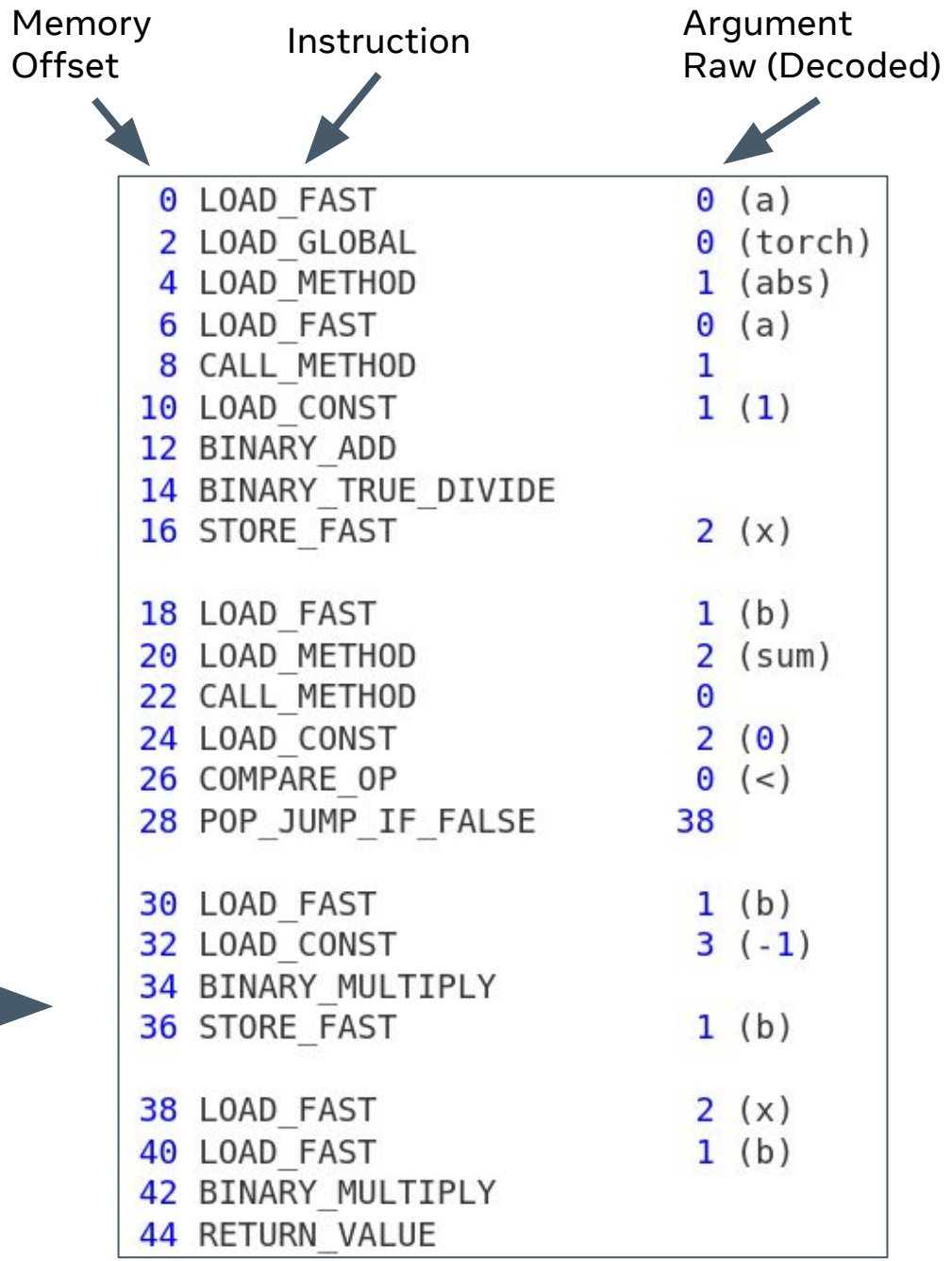

Default Python Behavior
foo(...)
PyFrameObject
PyCodeObject
_PyEval_EvalFrameDefault()
TorchDynamo Behavior
foo(...)
PyFrameObject
PyCodeObject
Cached
dyanmic bytecode analysis + transform
Guards
FX Graphs (torch.* bits)
Patched PyFrameObject
Transformed PyCodeObject (non-torch.* bits)
User-defined Compiler
call
Compiled Function
_PyEval_EvalFrameDefault()

Back to Our Toy Example

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

When toy_example() is called, TorchDynamo takes control:

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



Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

Stack

```
TensorVariable(proxy=<a>,
               source=<local> "a",
               guards={<TENSOR_MATCH "a">})
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a        a
placeholder    b        b
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

Stack

```
TensorVariable(proxy=<a>,
               source=<local> "a",
               guards={<TENSOR_MATCH "a">})
```

```
TorchVariable(
    value=torch,
    source=<global> "torch"
    guards={<FUNCTION_MATCH "torch">})
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a        a
placeholder    b        b
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

Stack

```
TensorVariable(proxy=<a>,
               source=<local> "a",
               guards={<TENSOR_MATCH "a">})
```

```
TorchVariable(
    value=torch.abs,
    source=<global> "torch"."abs"
    guards={<FUNCTION_MATCH "torch">})

NULL
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a        a
placeholder    b        b
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

Stack

```
TensorVariable(proxy=<a>,
               source=<local> "a",
               guards={<TENSOR_MATCH "a">})

TorchVariable(
    value=torch.abs,
    source=<global> "torch.abs"
    guards={<FUNCTION_MATCH "torch">})

NULL
```

```
TensorVariable(proxy=<a>,
               source=<local> "a",
               guards={<TENSOR_MATCH "a">})
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a        a
placeholder    b        b
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

Stack

```
TensorVariable(proxy=<a>,
               source=<local> "a",
               guards={<TENSOR_MATCH "a">})
```

```
TensorVariable(
    proxy=<abs_1>,
    guards={<FUNCTION_MATCH "torch">})

NULL


TensorVariable(proxy=<a>,
               source=<local> "a",
               guards={<TENSOR_MATCH "a">})
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a        a
placeholder    b        b
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

Stack

```
TensorVariable(proxy=<a>,
               source=<local> "a",
               guards={<TENSOR_MATCH "a">})

TensorVariable(
    proxy=<abs_1>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

```
ConstantVariable(value=1)
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a        a
placeholder    b        b
call_function  abs_1    torch.abs        (a,)
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

Stack

```
TensorVariable(proxy=<a>,
               source=<local> "a",
               guards={<TENSOR_MATCH "a">})
```

```
TensorVariable(
    proxy=<abs_1>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})


ConstantVariable(value=1)
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a        a
placeholder    b        b
call_function  abs_1    torch.abs        (a,)
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

Stack

```
TensorVariable(proxy=<a>,
    proxy=<truediv>,  source=<local> "a",
    guards={<TENSOR_MATCH "a">,  <TENSOR_MATCH "a">})
            <FUNCTION_MATCH "torch">})
TensorVariable(
    proxy=<add>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a        a
placeholder    b        b
call_function  abs_1    torch.abs        (a,)
call_function  add      operator.add     (abs 1, 1)
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

```
"x": TensorVariable(
    proxy=<truediv>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

Stack

```
TensorVariable(
    proxy=<truediv>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a         a
placeholder    b         b
call_function  abs_1     torch.abs         (a,)
call_function  add       operator.add      (abs_1, 1)
call_function  truediv   operator.truediv  (a, add)
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})


"x": TensorVariable(
    proxy=<truediv>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

Stack

```
TensorVariable(
    proxy=<b>,
    guards={<TENSOR_MATCH "b">})
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a         a
placeholder    b         b
call_function  abs_1     torch.abs         (a,)
call_function  add       operator.add      (abs_1, 1)
call_function  truediv   operator.truediv  (a, add)
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})


"x": TensorVariable(
    proxy=<truediv>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

Stack

```
GetAttrVariable(
    obj=TensorVariable(
            proxy=<b>,
            guards={<TENSOR_MATCH "b">}),
    name="sum",
    guards={<TENSOR_MATCH "b">})

NULL
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a         a
placeholder    b         b
call_function  abs_1     torch.abs         (a,)
call_function  add       operator.add      (abs_1, 1)
call_function  truediv   operator.truediv  (a, add)
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})


"x": TensorVariable(
    proxy=<truediv>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

Stack

```
TensorVariable(
    proxy=<sum>,
    guards={<TENSOR_MATCH "b">})
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a         a
placeholder    b         b
call_function  abs_1     torch.abs         (a,)
call_function  add       operator.add      (abs_1, 1)
call_function  truediv   operator.truediv  (a, add)
call_method    sum_1     sum               (b,)
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})


"x": TensorVariable(
    proxy=<truediv>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

Stack

```
TensorVariable(
    proxy=<sum>,
    guards={<TENSOR_MATCH "b">})
```

```
ConstantVariable(0)
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a         a
placeholder    b         b
call_function  abs_1     torch.abs         (a,)
call_function  add       operator.add      (abs_1, 1)
call_function  truediv   operator.truediv  (a, add)
call_method    sum_1     sum               (b,)
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})

"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})


"x": TensorVariable(
    proxy=<truediv>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

Stack

```
TensorVariable(
    proxy=<lt>,
    guards={<TENSOR_MATCH "b">})
```

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a         a
placeholder    b         b
call_function  abs_1     torch.abs         (a,)
call_function  add       operator.add      (abs_1, 1)
call_function  truediv   operator.truediv  (a, add)
call_method    sum 1     sum               (b,)
call_function  lt        operator.lt       (sum_1, 0)
```

Symbolic Locals

```
"a": TensorVariable(
          proxy=<a>,
          source=<local> "a",
          guards={<TENSOR_MATCH "a">})
```

```
"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})
```

LIVE (unchanged)

```
"x": TensorVariable(
    proxy=<truediv>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

LIVE (from graph)

Stack

```
TensorVariable(
   proxy=<lt>,
   guards={<TENSOR_MATCH "b">})
```

LIVE (from graph)

```
 0 LOAD_FAST                0 (a)
 2 LOAD_GLOBAL              0 (torch)
 4 LOAD_METHOD              1 (abs)
 6 LOAD_FAST                0 (a)
 8 CALL_METHOD              1
10 LOAD_CONST               1 (1)
12 BINARY_ADD
14 BINARY_TRUE_DIVIDE
16 STORE_FAST               2 (x)

18 LOAD_FAST                1 (b)
20 LOAD_METHOD              2 (sum)
22 CALL_METHOD              0
24 LOAD_CONST               2 (0)
26 COMPARE_OP               0 (<)
28 POP_JUMP_IF_FALSE       38

30 LOAD_FAST                1 (b)
32 LOAD_CONST               3 (-1)
34 BINARY_MULTIPLY
36 STORE_FAST               1 (b)

38 LOAD_FAST                2 (x)
40 LOAD_FAST                1 (b)
42 BINARY_MULTIPLY
44 RETURN_VALUE
```

FX Graph

```
placeholder    a         a
placeholder    b         b
call_function  abs_1     torch.abs         (a,)
call_function  add       operator.add      (abs_1, 1)
call_function  truediv   operator.truediv  (a, add)
call_method    sum_1     sum               (b,)
call_function  lt        operator.lt       (sum_1, 0)
```

Symbolic Locals

```
"b": TensorVariable(
          proxy=<b>,
          source=<local> "b",
          guards={<TENSOR_MATCH "b">})

"x": TensorVariable(
    proxy=<truediv>,
    guards={<TENSOR_MATCH "a">,
            <FUNCTION_MATCH "torch">})
```

FX Graph

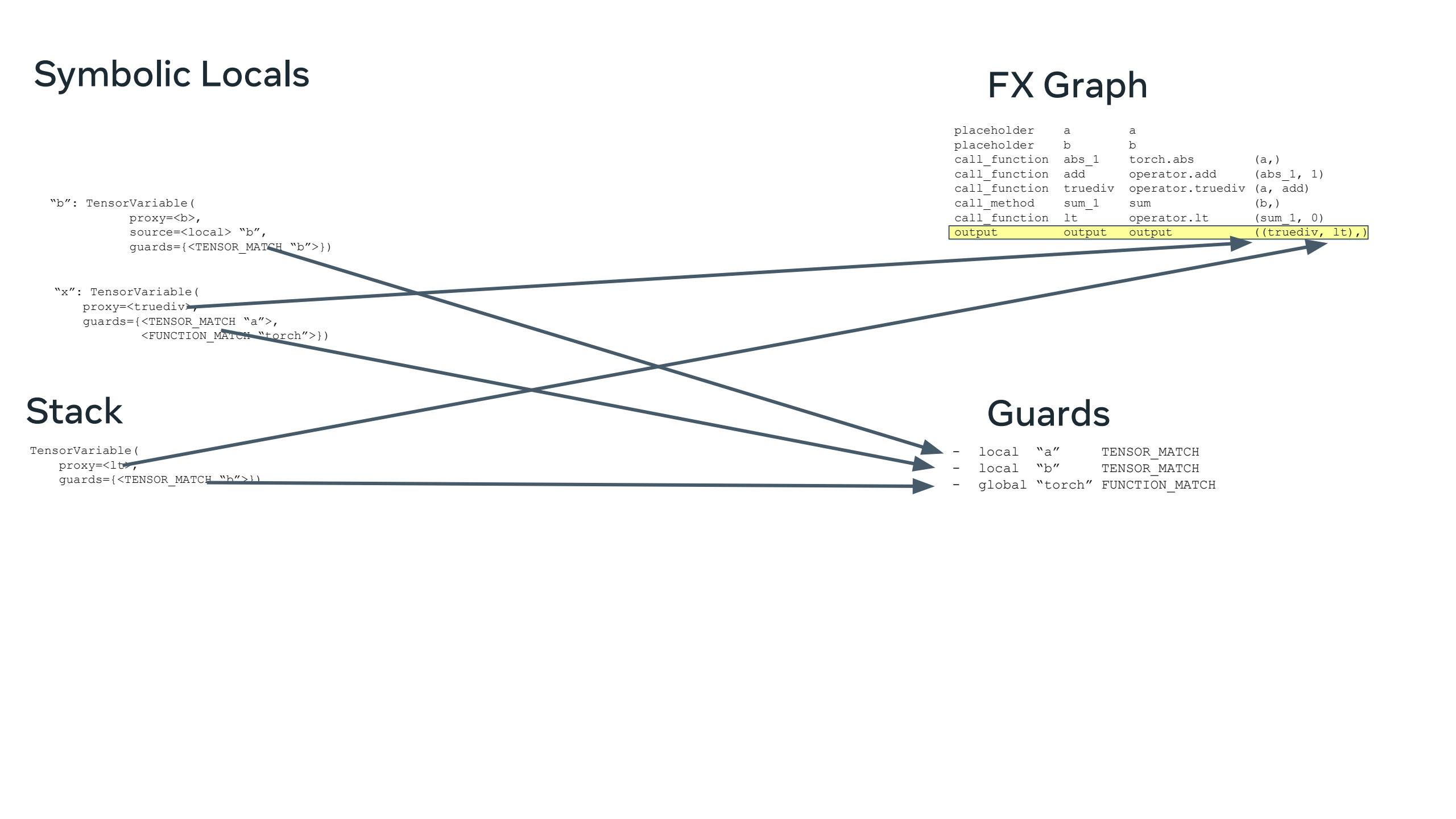
```
placeholder    a        a
placeholder    b        b
call_function  abs_1    torch.abs         (a,)
call_function  add      operator.add      (abs_1, 1)
call_function  truediv  operator.truediv  (a, add)
call_method    sum_1    sum               (b,)
call_function  lt       operator.lt       (sum_1, 0)
output         output   output            ((truediv, lt),)
```

Stack

```
TensorVariable(
    proxy=<lt>,
    guards={<TENSOR_MATCH "b">})
```

Guards

```
-  local  "a"     TENSOR_MATCH
-  local  "b"     TENSOR_MATCH
-  global "torch" FUNCTION_MATCH
```



Guards

```
-  local  "a"      TENSOR_MATCH
-  local  "b"      TENSOR_MATCH
-  global "torch"  FUNCTION_MATCH
```

FX Graph

```
placeholder    a        a
placeholder    b        b
call_function  abs_1    torch.abs         (a,)
call_function  add      operator.add      (abs_1, 1)
call_function  truediv  operator.truediv  (a, add)
call_method    sum_1    sum               (b,)
call_function  lt       operator.lt       (sum_1, 0)
output         output   output            ((truediv, lt),)
```

Result of my_compiler()

Output Bytecode

```
 0 LOAD_GLOBAL              3 (__compiled_fn_0)
 2 LOAD_FAST                0 (a)
 4 LOAD_FAST                1 (b)
 6 CALL_FUNCTION            2
 8 UNPACK_SEQUENCE          2
10 STORE_FAST               2 (x)
12 POP_JUMP_IF_FALSE       24
14 LOAD_GLOBAL              4 (__resume_at_30_1)
16 LOAD_FAST                1 (b)
18 LOAD_FAST                2 (x)
20 CALL_FUNCTION            2
22 RETURN_VALUE
24 LOAD_GLOBAL              5 (__resume_at_38_2)
26 LOAD_FAST                1 (b)
28 LOAD_FAST                2 (x)
30 CALL_FUNCTION            2
32 RETURN_VALUE
```

Graph Inputs

Restore stack/local state from graph outputs

The bytecode we stopped at (and couldn't handle)

Generated resume_at_<offset> functions.
Create new frames, so the process starts again recursively

```
def __resume_at_30(b, x):
    JUMP_ABSOLUTE <offset 30>
    ... original bytecode of toy_example ...
```

```
def __resume_at_38(b, x):
    JUMP_ABSOLUTE <offset 38>
    ... original bytecode of toy_example ...
```

Supporting More Complex Things

- **Function calls:** Inlined + guards
- **Comprehensions**: Inlined
- **List/tuple/dict/slice/NamedTuple/etc:** Handled symbolically
- **Loops:** Unrolled + guards
- **Control flow:** Specialized + guards
- **Lambdas/inline function definitions/generators:** Deferred or inlined
- **Tensor properties (dtype/device/shapes(optional)/etc):** Specialized + constant folding + guards
- **Some side-effects/list-mutation:** Defer + apply after Graph
- **Closures:** Handled symbolically, materialized when needed
- **Break graph on:**
  - Data-dependent control flow
    - Most control flow gets unrolled away
  - External Python C-extensions (numpy, etc)
  - Conversions to Python types (.tolist(), .item())
  - Other uncommon things

Let’s talk about guards

If any of the guards fail, the graph will be recaptured and recompiled.

TENSOR_MATCH checks:

- Python class of the tensor (tensor subclassing, etc)
- dtype
- device
- requires_grad
- dispatch_key (with thread-local includes/exclude)
- ndim
- sizes* (optional)
- strides* (optional)

FUNCTION_MATCH checks:

- id(obj) hasn’t changed

Guards

```
- local  “a”     TENSOR_MATCH
- local  “b”     TENSOR_MATCH
- global “torch” FUNCTION_MATCH
```

TorchDynamo has 15 types of guards (types, lists, attributes, dicts, consts, nn.Modules, mutation, etc)

*For sizes/strides you can disable this specialization by setting:

```
torchdynamo.config.dynamic_shapes = True
```

Training Support with AotAutograd

Horace He (@cHHillee)

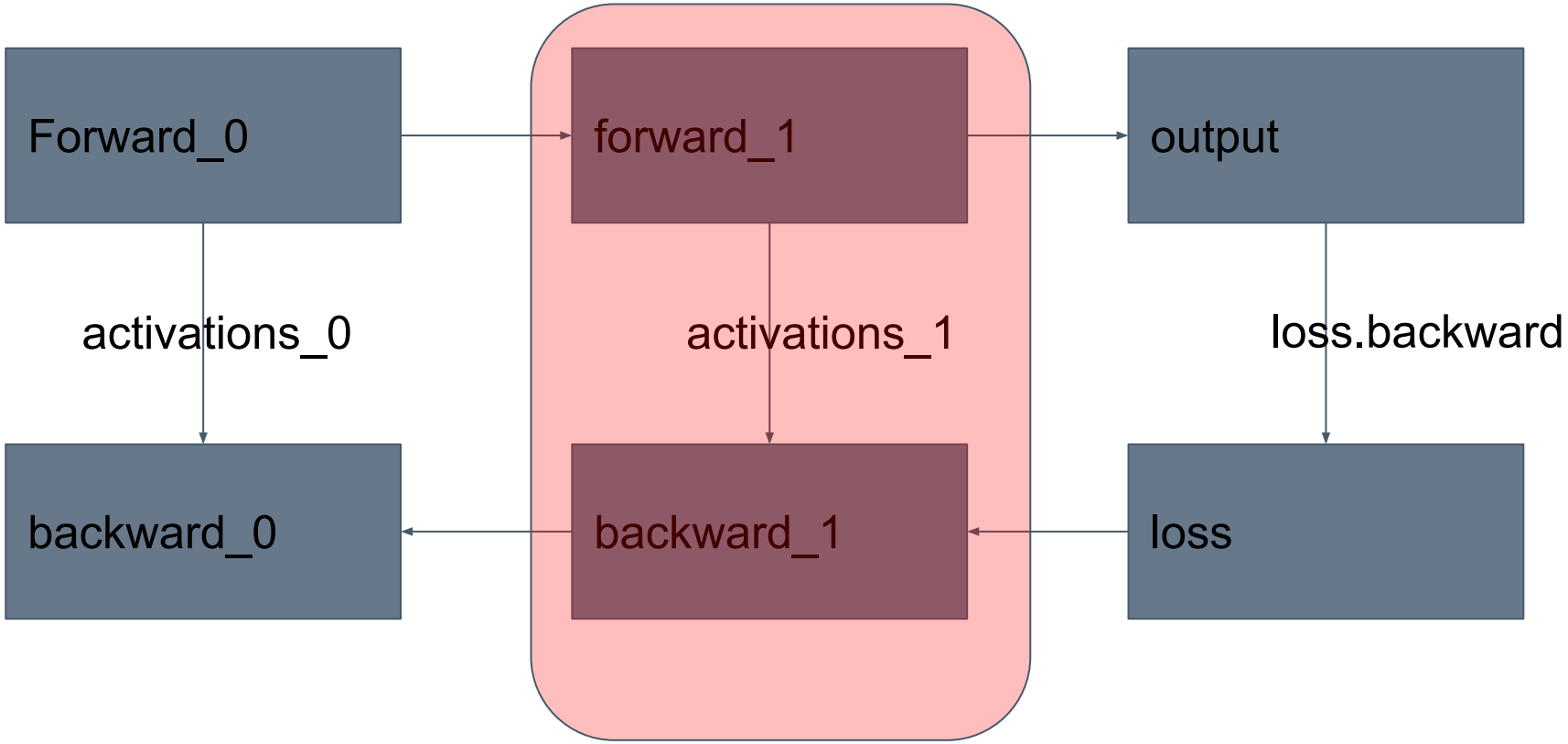
Handling Training/Backwards

- TorchDynamo captures the forwards, but we still need backwards
- Backwards in PyTorch is done through dynamic autograd tape
- We need to capture the dynamic autograd behavior at compile time
- Key challenge: partial graphs not full graphs

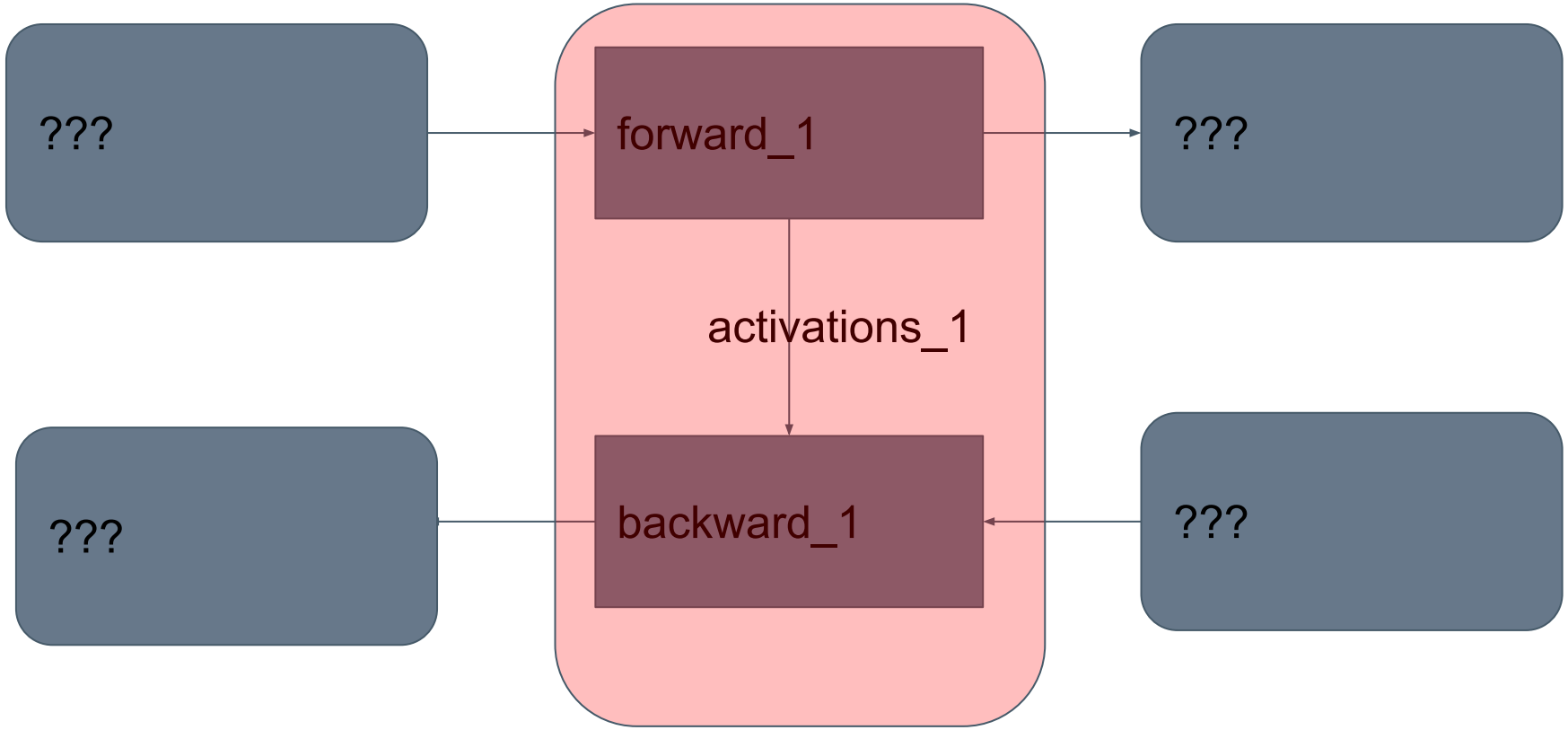
AOT Autograd

- Traces the behavior of the PyTorch autograd tape
- Allows partitioning the forwards and backwards
- Works on partial graph fragments

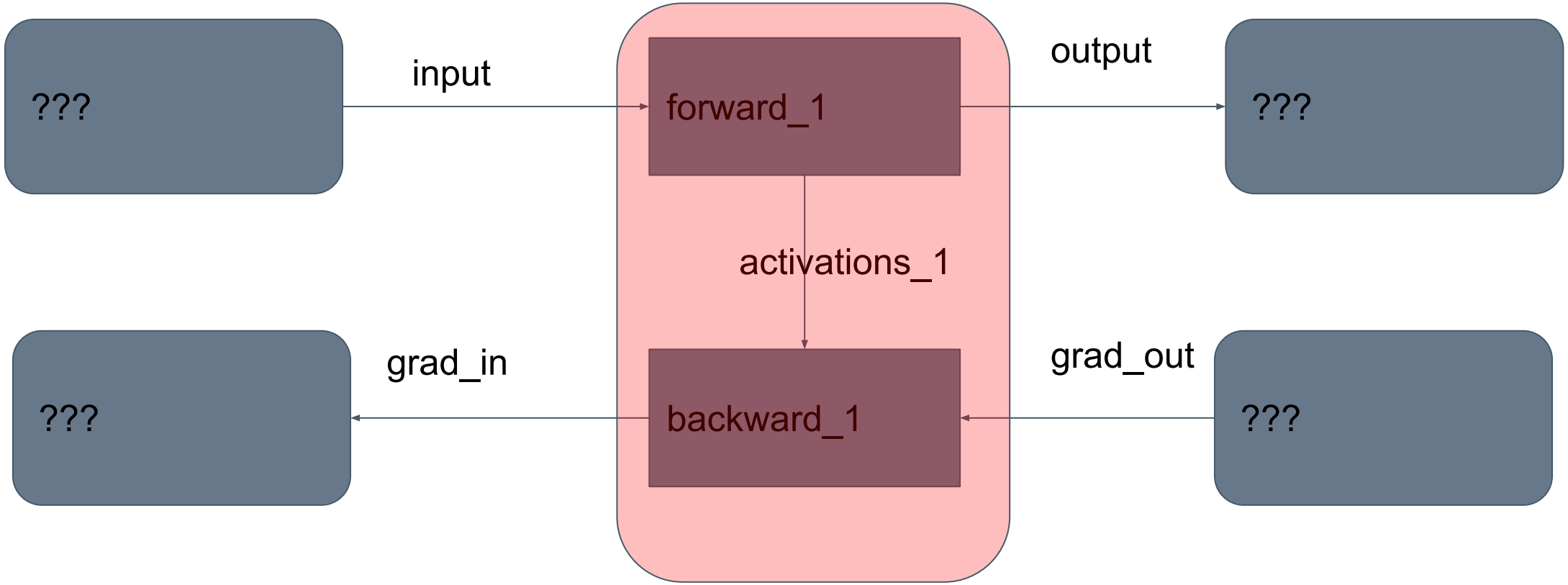
How does autograd work?



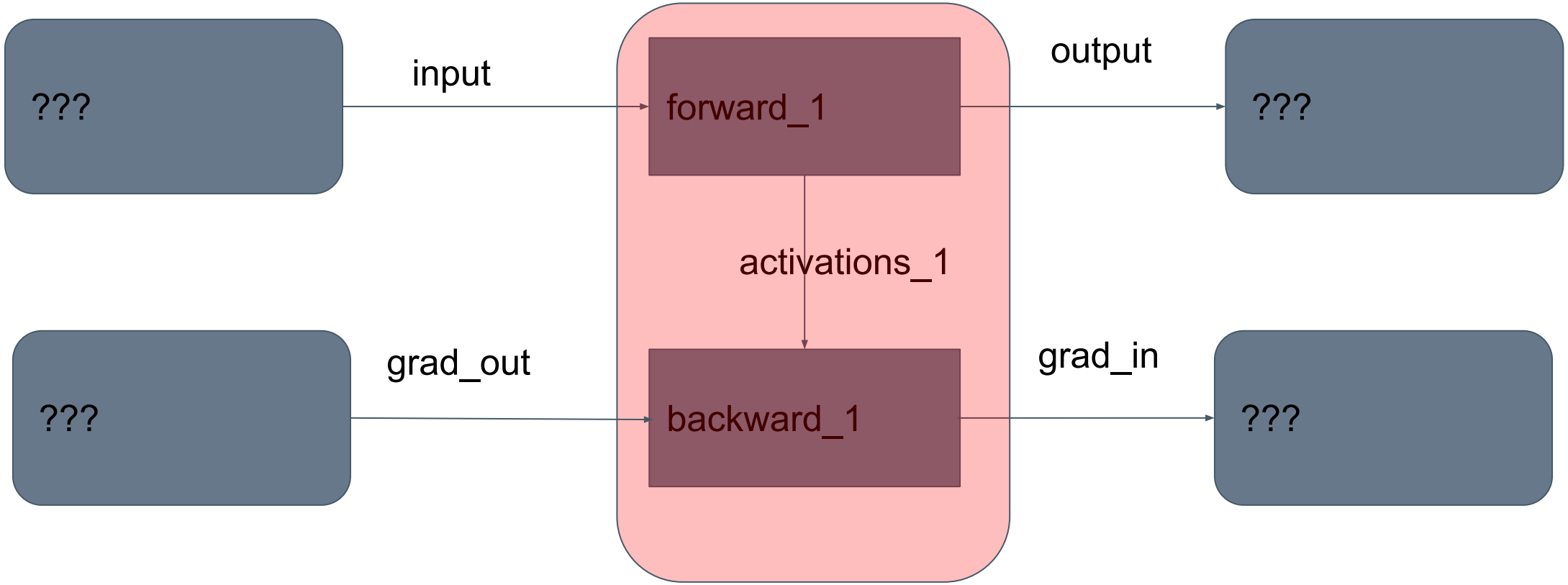
How does autograd work?



How does autograd work?



How does autograd work?



Perform a little switcheroo

Example Function

```
def f(x):
    y = x.sin()
    z = y.sin()
    return z
```

```
def graph(x, g_z):
    y = x.sin()
    z = y.sin()
    r0 = y.cos()
    r1 = g_z * r0
    r2 = x.cos()
    g_x = r1 * r2
    return z, g_x
```

Example Function

Forward 0

Forward 1

Backward 1

Backward 0

grad_in

```
def graph(x, g_z):
    y = x.sin()
    z = y.sin()
    r0 = y.cos()
    r1 = g_z * r0
    r2 = x.cos()
    g_x = r1 * r2
    return z, g_x
```

grad_out

Graph Partitioning

```
def graph(x, g_z):
    y = x.sin()
    z = y.sin()
    r0 = y.cos()
    r1 = g_z * r0
    r2 = x.cos()
    g_x = r1 * r2
    return z, g_x
```

No single way to do this - providing combined graph and then partitioning gives compilers control

```
def graph_forward(x):
    y = x.sin()
    z = y.sin()
    r0 = y.cos()
    r2 = x.cos()
    return z, r0, r2

                          Activations
def graph_backward(r0, r2, g_z):
    r1 = g_z * r0
    g_x = r1 * r2
    return g_x
```

TorchInductor:
A PyTorch Native Compiler

TORCHINDUCTOR PRINCIPLES

PyTorch Native

Similar abstractions to PyTorch eager to allow support for nearly all of PyTorch, with a thin translation layer.

Python First

A pure python compiler makes TorchInductor easy to understand and hackable by users. Generates Triton and C++.

Breadth First

Early focus on supporting a wide variety of operators, hardware, and optimization. A general purpose compiler, that can scale.

TORCHINDUCTOR TECHNOLOGIES

Define-By-Run Loop-Level IR

Direct use of Python functions in IR definitions allows for rapidly defining lowering with little boilerplate.

Dynamic Shapes & Strides

Uses SymPy to reason about shapes, indexing, and managing guards. Symbolic shapes from the ground up.

Reuse State-Of-The-Art Languages

Generates output code in languages popular for writing handwritten kernels:

- Triton for GPUs
- C++/OpenMP for CPUs

What is Triton?

A new programming language for highly performant GPU kernels

- Higher level than CUDA
- Lower level than preexisting DSLs
- Allows non-experts to write fast custom kernels

Users define tensors (i.e., blocks of data) in SRAM, and modify them using torch-like operators

PYTHONIC INTERFACE

Like in Numba, kernels are defined in Python using the triton.jit decorator

LOW-LEVEL MEMORY CONTROL

Users can construct tensors of pointers and dereference them element-wise

Optimizing Compiler

Blocked program representation allows the Triton compiler to generate extremely efficient code

https://triton-lang.org
https://github.com/openai/triton
by Philippe Tillet @ OpenAI

Triton: an intermediate language and compiler for tiled neural network computations

Philippe Tillet, H. T. Kung, David Cox

In Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)

https://doi.org/10.1145/3315508.3329973

DEFINE-BY-RUN (DBR) LOOP-LEVEL IR

x.permute(1, 0) + x[2, :] becomes:

```
def inner_fn(index: List[sympy.Expr]):
    i1, i0 = index
    tmp0 = ops.load("x", i1 + i0*size1)
    tmp1 = ops.load("x", 2*size1 + i0)
    tmp2 = ops.add(tmp0, tmp1)
    return tmp2
```

```
torchinductor.ir.Pointwise(
        device=torch.device(...),
        dtype=torch.float32,
        inner_fn=inner_fn,
        ranges=[size0, size1],
)
```

Override `ops` to do analysis and backend codegen.

TORCHINDUCTOR COMPILER STACK

AotAutograd

Decomposes into smaller operator set

Capture forwards + backwards

Some inductor specific decomps included in this step

Inductor Graph Lowerings

Remove views, broadcasting, and simplify indexing

Rematerialize vs reuse decisions

Layout tuning and optimization

Loop order

Inductor Scheduling

Horizontal / vertical fusion decisions

Reduction fusions

Tiling

Memory planning and buffer reuse

In-place memory buffers

Autotuning / kernel selection

Wrapper Codegen

Outer code that calls kernels and allocates memory

(Replaces interpreter)

Backend Codegen

Triton

C++

AotAutograd	**Graph Lowerings**	**Scheduling**	**Wrapper Codegen**
Operator Decomposition Capture forwards + backwards Functionalization	Remove views, broadcasting, and simplify indexing Rematerialize vs reuse decisions Layouts and loop order	Horizontal / vertical fusion decision Tiling / Kernel selection Memory planning and buffer reuse	Invokes kernels and allocates memory
			Backend Codegen Triton or C++

TorchInductor Example

Input Code	ATen FX Graph	Define-by-run IR	Scheduling/Fusion	Output Triton	Output Wrapper

```
import torch
```

Run with:
TORCH_COMPILE_DEBUG=1 python inductor_demo.py

```
@torch.compile(dynamic=True)
def toy_example(x):
    y = x.sin()
    z = y.cos()
    return y, z


toy_example(torch.randn([8192, 1024], device="cuda"))
```

TorchInductor Example

Input Code	**ATen FX Graph**	Define-by-run IR	Scheduling/Fusion	Output Triton	Output Wrapper

```
def forward(self, arg0_1: f32[s0, s1]):
    # File: inductor_demo.py:6, code: y = x.sin()
    sin: f32[s0, s1] = torch.ops.aten.sin.default(arg0_1)

    # File: inductor_demo.py:7, code: z = y.cos()
    cos: f32[s0, s1] = torch.ops.aten.cos.default(sin)
    return (sin, cos)
```

TorchInductor Example

Input Code	ATen FX Graph	**Define-by-run IR**	Scheduling/Fusion	Output Triton	Output Wrapper

```
def inner_fn_buf0(index):
  i0, i1 = index
  tmp0 = ops.load(arg0_1, i1 + i0 * s1)
  tmp1 = ops.sin(tmp0)
  return tmp1

def inner_fn_buf1(index):
  i0, i1 = index
  tmp0 = ops.load(buf0, i1 + i0 * s1)
  tmp1 = ops.cos(tmp0)
  return tmp1
```

```
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], ...))))

buf1_ir = TensorBox(StorageBox(ComputedBuffer(
  name='buf1',
  layout=FixedLayout('cuda', torch.float32,
                     size=[s0, s1], stride=[s1, 1]),
  data=Pointwise(inner_fn=inner_fn_buf1,
                 ranges=[s0, s1], ...))))
```

TorchInductor Example

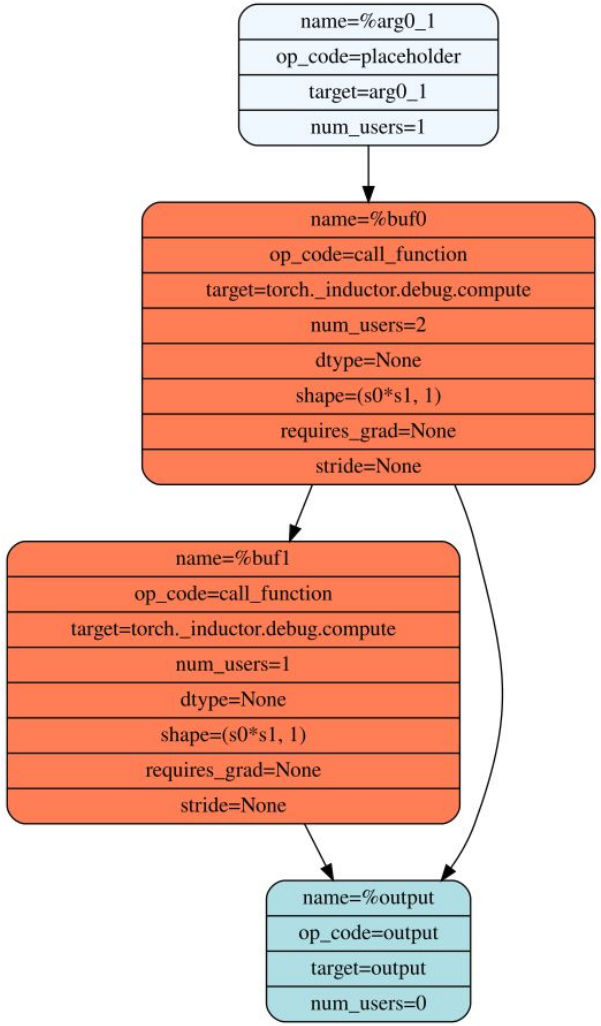
Input Code	ATen FX Graph	Define-by-run IR	**Scheduling/Fusion**	Output Triton	Output Wrapper

torch/_inductor/scheduler.py

Scheduler.can_fuse(buf0, buf1)
True

Scheduler.score_fusion(buf0, buf1)
(True, True, 33554432, -1)

- True/True is category of fusion (pointwise+pointwise)
- 33554432 is estimated memory bandwidth saved by fusion: 8192*1024*4
- -1 is distance in input graph



TorchInductor Example

Input Code	ATen FX Graph	Define-by-run IR	Scheduling/Fusion	**Output Triton**	Output Wrapper

```
@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)
```

TorchInductor Example

Input Code	ATen FX Graph	Define-by-run IR	Scheduling/Fusion	Output Triton	**Output Wrapper**

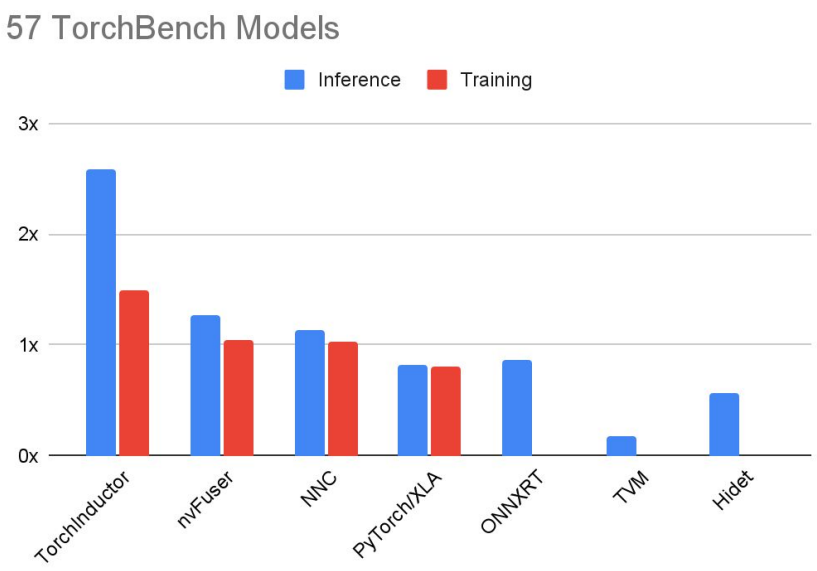
```
def call(args):
    arg0_1, = args
    args.clear()
    arg0_1_size = arg0_1.size()
    s0 = arg0_1_size[0]
    s1 = arg0_1_size[1]
    buf0 = empty_strided((s0, s1), (s1, 1), device='cuda', dtype=torch.float32)
    buf1 = empty_strided((s0, s1), (s1, 1), device='cuda', dtype=torch.float32)
    triton__0_xnumel = s0*s1
    triton__0.run(arg0_1, buf0, buf1, triton__0_xnumel, grid=grid(triton__0_xnumel))
    return (buf0, buf1, )
```

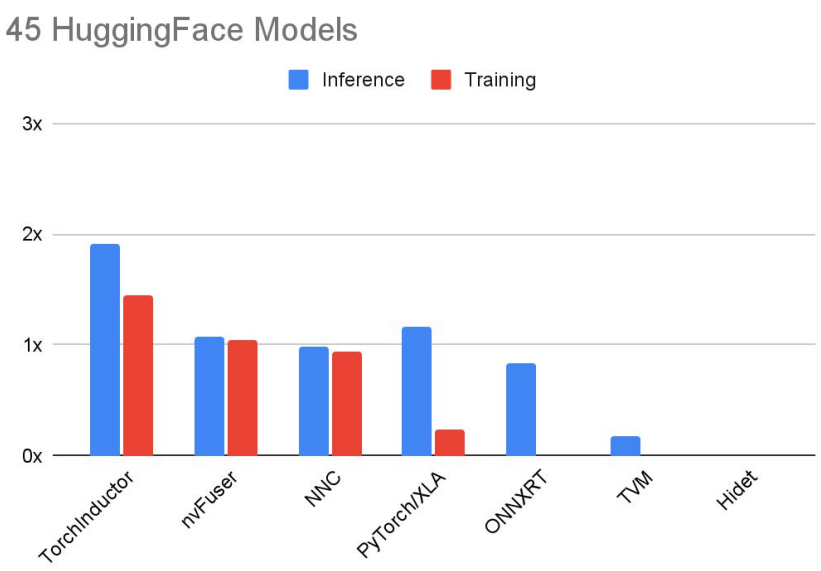
TorchInductor Example: C++ Output

Change device='cuda' to device='cpu'

```
extern "C" void kernel(const float* __restrict__ in_ptr0,
                       float* __restrict__ out_ptr0,
                       float* __restrict__ out_ptr1,
                       const long ks0,
                       const long ks1)
{
    #pragma omp parallel num_threads(8)
    {
        {
            #pragma omp for
            for(long i0=0; i0<((ks0*ks1) / 16); i0+=1)
            {
                auto tmp0 = at::vec::Vectorized<float>::loadu(in_ptr0 + 16*i0);
                auto tmp1 = tmp0.sin();
                auto tmp2 = tmp1.cos();
                tmp1.store(out_ptr0 + 16*i0);
                tmp2.store(out_ptr1 + 16*i0);
            }
            #pragma omp for simd simdlen(8)
            for(long i0=16*(((ks0*ks1) / 16)); i0<ks0*ks1; i0+=1)
            {
                auto tmp0 = in_ptr0[i0];
                auto tmp1 = std::sin(tmp0);
                auto tmp2 = std::cos(tmp1);
                out_ptr0[i0] = tmp1;
                out_ptr1[i0] = tmp2;
            }
        }
    }
}
```

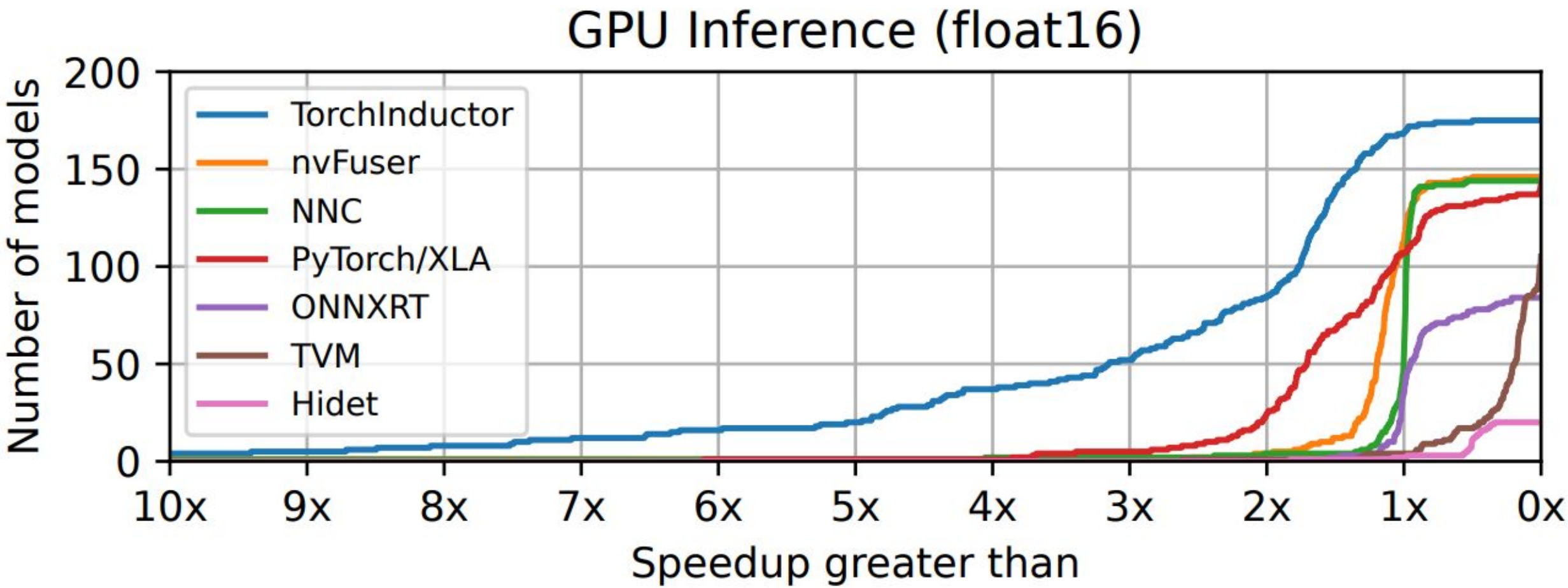
NVIDIA A100 PERFORMANCE





Geomean speedup over PyTorch eager using float16
Higher is better

NVIDIA A100 PERFORMANCE



Cumulative distribution function of speedups over PyTorch eager.

NVIDIA A100 PERFORMANCE

	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)

Geomean speedup over PyTorch eager on 45 models from HuggingFace using fp16

Sylvain Gugger
the primary maintainer of HuggingFace transformers:

"With just one line of code to add, PyTorch 2.0 gives a speedup between 1.5x and 2.x in training Transformers models.

This is the most exciting thing since mixed precision training was introduced!"

Luca Antiga the CTO of grid.ai and one of the primary maintainers of PyTorch Lightning

> “PyTorch 2.0 embodies the future of deep learning frameworks.
>
> The possibility to capture a PyTorch program with effectively no user intervention and get massive on-device speedups and program manipulation out of the box unlocks a whole new dimension for AI developers. ”

Ross Wightman the primary maintainer of TIMM

"It just works out of the box with majority of TIMM models for inference and train workloads with no code changes."

PyTorch 2.0:

https://pytorch.org/get-started/pytorch-2.0/

Live PyTorch 2.0 Q&A Series:

https://www.youtube.com/@PyTorch

Code:

https://github.com/pytorch/pytorch/tree/master/torch/_dynamo

https://github.com/pytorch/pytorch/tree/master/torch/_functorch/aot_autograd.py

https://github.com/pytorch/pytorch/tree/master/torch/_inductor