Custom Class Support in PT2 (OSS)

Background

PyTorch uses a framework called TorchBind to bind custom c++ classes, which may carry mutable state, to Python. This past half, we have built infrastructure support for these TorchBind classes within the PT2 stack (torch.compile and torch.export).

How to support existing torchBind classes in PT2 stack?

Concretely, there are a few steps:

1. Implement an `__obj_flatten__` method to the C++ custom class implementation to allow us to inspect its states and guard the changes. The method should return a tuple of tuple of attribute_name, value (tuple[tuple[str, value] * n].
2. Register a python fake class using `@torch._library.register_fake_class`
    a. Implement “fake methods” of each of the class’s c++ methods, which should have the same schema as the C++ implementation.
    b. Additionally, implement an `__obj_unflatten__` classmethod in the Python fake class to tell us how to create a fake class from the flattened states returned by `__obj_flatten__`.

Here is a breakdown of the diff. Following the guide in Extending TorchScript with Custom C++ Classes, we can create a thread-safe tensor queue (copied from fbgemm) and build it.

C/C++

```
// Thread-safe Tensor Queue
struct TensorQueue : torch::CustomClassHolder {
 ...
 private:
  std::deque<at::Tensor> queue_;
  std::mutex mutex_;
  at::Tensor init_tensor_;
};
// The torch binding code
TORCH_LIBRARY(MyCustomClass, m) {
  m.class_<TensorQueue>("TensorQueue")
      .def(torch::init<at::Tensor>())
      .def("push", &TensorQueue::push)
      .def("pop", &TensorQueue::pop)
      .def("top", &TensorQueue::top)
      .def("size", &TensorQueue::size)
      .def("clone_queue", &TensorQueue::clone_queue)
      .def("get_raw_queue", &TensorQueue::get_raw_queue)
      .def_pickle(
          // __getstate__
          [](const c10::intrusive_ptr<TensorQueue>& self)
              -> c10::Dict<std::string, at::Tensor> {
            return self->serialize();
```

```
          },
          // __setstate__
          [](c10::Dict<std::string, at::Tensor> data)
              -> c10::intrusive_ptr<TensorQueue> {
            return c10::make_intrusive<TensorQueue>(std::move(data));
          });
}
```

Step 1: Add an `__obj_flatten__` method to the C++ custom class implementation:

C/C++

```
// Thread-safe Tensor Queue
struct TensorQueue : torch::CustomClassHolder {
  ...
  std::tuple<std::tuple<std::string, std::vector<at::Tensor>>, std::tuple<std::string, at::Tensor>> __obj_flatten__() {
    return std::tuple(std::tuple("queue", this->get_raw_queue()), std::tuple("init_tensor_",
this->init_tensor_.clone()));
  }
  ...
}

TORCH_LIBRARY(MyCustomClass, m) {
  m.class_<TensorQueue>("TensorQueue")
      .def(torch::init<at::Tensor>())
      .def("__obj_flatten__", &TensorQueue::__obj_flatten__)
      ...
}
```

Step 2a: Register a fake class in Python that implements each method.

Python

```
# namespace::class_name
@torch._library.register_fake_class("MyCustomClass::TensorQueue")
class FakeTensorQueue:
    def __init__(
        self,
        queue: List[torch.Tensor],
        init_tensor_: torch.Tensor
    ) -> None:
        self.queue = queue
        self.init_tensor_ = init_tensor_

    def push(self, tensor: torch.Tensor) -> None:
        self.queue.append(tensor)

    def pop(self) -> torch.Tensor:
        if len(self.queue) > 0:
            return self.queue.pop(0)
        return self.init_tensor_

    def size(self) -> int:
           return len(self.queue)
```

Step 2b: Implement an `__obj_unflatten__` classmethod in Python.

Python

```
# namespace::class_name
@torch._library.register_fake_class("MyCustomClass::TensorQueue")
class FakeTensorQueue:
  ...
    @classmethod
    def __obj_unflatten__(cls, flattened_tq):
        return cls(**dict(flattened_tq))

  ...
```

That's it! Now we can create a module that uses this object and run it with torch.compile or torch.export:

Python

```
import torch

torch.ops.load_library("//caffe2/test:test_torchbind_cpp_impl")
tq = torch.classes.MyCustomClass.TensorQueue(torch.empty(0).fill_(-1))

class Mod(torch.nn.Module):
    def forward(self, tq, x):
        tq.push(x.sin())
        tq.push(x.cos())
        poped_t = tq.pop()
        assert torch.allclose(poped_t, x.sin())
        return tq, poped_t

tq, poped_t = torch.compile(Mod(), backend="eager", fullgraph=True)(tq, torch.randn(2, 3))
assert tq.size() == 1

exported_program = torch.export.export(Mod(), (tq, torch.randn(2, 3),), strict=False)
exported_program.module()(tq, torch.randn(2, 3))
```

We can also implement custom ops that take custom classes as inputs. For example, we could register a custom op `for_each_add_(tq, tensor)`

C/C++

```
struct TensorQueue : torch::CustomClassHolder {
 ...
 void for_each_add_(at::Tensor inc) {
    for (auto& t : queue_) {
      t.add_(inc);
    }
  }
 ...
}


TORCH_LIBRARY_FRAGMENT(MyCustomClass, m) {
  m.class_<TensorQueue>("TensorQueue")
    .def("for_each_add_", &TensorQueue::for_each_add_);

  m.def(
```

```
      "for_each_add_(__torch__.torch.classes.MyCustomClass.TensorQueue foo, Tensor inc) -> ()");

}

void for_each_add_(c10::intrusive_ptr<TensorQueue> tq, at::Tensor inc) {
  tq->for_each_add_(inc);
}

TORCH_LIBRARY_IMPL(MyCustomClass, CPU, m) {
  m.impl("for_each_add_", for_each_add_);
}
```

Since the fake class is implemented in python, we require the fake implementation of custom op **must also be registered in python**:

Python

```
@torch.library.register_fake("MyCustomClass::for_each_add_")
def fake_for_each_add_(tq, inc):
    tq.for_each_add_(inc)
```

After re-compilation, we can export the custom op with:

Python

```
class ForEachAdd(torch.nn.Module):
  def forward(self, tq: torch.ScriptObject, a: torch.Tensor) -> torch.ScriptObject:
    torch.ops.MyCustomClass.for_each_add_(tq, a)
    return tq
mod = ForEachAdd()
tq = empty_tensor_queue()
qlen = 10
for i in range(qlen):
  tq.push(torch.zeros(1))

ep = torch.export.export(mod, (tq, torch.ones(1)), strict=False)

assertExpectedInline(ep.graph_module.code.strip(), """\
def forward(self, token, tq, a):
    with_effects = torch._higher_order_ops.effects.with_effects(token, torch.ops.MyCustomClass.for_each_add_.default,
tq, a);  token = a = None
    getitem = with_effects[0];  with_effects = None
    return (getitem, tq)""")

output_tq = ep.module()(tq, torch.ones(1))
assertEqual(output_tq.size(), qlen)
assertEqual(output_tq.top(), torch.ones(1))
```

Why do we need to make a Fake Class?

Tracing with real custom object has several major downsides:

1. Operators on real objects can be **time consuming** e.g. the custom object might be reading from the network or loading data from the disk.

2. We don't want to mutate the real custom object or create **side-effects** to the environment while tracing.
3. It cannot support **dynamic shape**.

However, it may be difficult for users to write a fake class: the original class uses some third-party library (e.g. koski) that determines the output shape of the methods, or is complicated and written by others. Besides, users may not care about the limitations listed above. In this case, please **reach out to us!**

The Coverage Matrix:

	torch.compile:				export	
	eager	aot_eager	inductor	AOTInductor	non-strict	strict
Custom Op with torchbind input	Y	Y	Y	Y	Y	N
obj.call_method	Y	Y	Y	Y	Y	
Dynamic shaped tensor in script object	Y	Y	Y	N	Y	

Known Limitations:

Aliasing and mutation:
Currently, aot_autograd cannot functionalize the script object's methods due to lack of accurate schema that shows the aliasing/mutation relationship.

This causes our current support to have the limitation that: once a tensor is passed into the script object method or returned from a script object method, **it's not safe to do in-place mutation** to the tensor in the python program anymore. In the following example,

```python
tq.push(a)
a.sin_()
b = tq.pop()
return a + b

# aot_autograd captured graph
```

```
call_torchbind( tq, 'push', clone)
sin = torch.ops.aten.sin.default(a);  clone = None
poped = call_torchbind(tq, 'pop');  getitem = tq = None
return sin + poped
```

`sin_` is replaced by `sin` as a result of functionalization but the mutation to the content of tq is not showing up in the graph causing the captured graph to produce a different result than original program.

“is” operator
Besides mutation, we should not rely on comparing a tensor with a tensor contained inside a script object with `is`. For example,

```
Python
# will return a + 1
tq.push(a)
if tq.pop() is a:
       return a + 1
return a - 1

# aot_autograd captured graph:
call_torchbind(tq, 'push', a)
return a - 1
```

The captured graph of aot_autograd returns `a - 1` while eager mode returns `a + 1`. This is because `“is”` compares real tensors in eager mode but compares FunctionalTensor wrappers during aot_autograd tracing. The wrapper comparison fails to capture the information that a and top.pop() are actually referring to the same tensor.

Constant burning-in graph:
For methods that return non-tensor e.g. int, float, string, tuple. They will be treated as constants and burned in the graph (consistent with how export deals with constant inputs). Concretely, we’ll see the following:

```
Python
def f(tq, x):
  return x + tq.size()

# captured graph for tq = [torch.randn(3,3), torch.randn(3, 3)]
```

```
def graph(tq, x):
  sz = call_torchbind(tq, "size", (,), {})
  return x + 2 # instead of x + sz
```

How do things work under the hood?

1. Fakify custom object (dynamo and aot_autograd)

Before tracing, when receiving a torch bind object, we'll:

1. Call `__obj_flatten__` on the torchbind object to get a `flattend_obj.`
2. If we're in dynamo, install guards on `flattened_obj.`Specifically, we guard the `flattened_obj` directly as if it's the input.
3. Fakify the tensors in `flattened_obj`.
4. Retrieve the fake class registered with `register_fake_class`.
5. Call `fake_class.__obj_unflatten__` to create the fake object.
6. If we're in dynamo, we additionally create TorchScriptObjectVariable that's backed by the FakeScriptObject to simulate the behavior of torchbind object .

During tracing, method calls will use the fake methods to create example values for downstream to use. Custom ops that take torch bind object inputs will call the python fake implementation and call the methods of fake script object.

2. Method Calls are turned into Operators

Each method call of a custom object potentially mutates its states. Therefore, we need to record all the methods of the custom object into the graph. The way we do it is to turn method calls into a higher order operator `call_torchbind`. The signature of the operator looks like:

Python

```
call_torchbind(custom_obj: torch.ScriptObject, method_name: str, args: Tuple[Args], kwargs, Dict[str, Args])
```

For the above example on `Mod`, dynamo will record a graph that looks like the following:

Python

```
def graph(tq, a, b):
  call_torchbind(tq, "push", (a,), {})
```

```
  torch.ops.MyCustomClass.queue_push(tq, (b,), {})
  torch.ops.MyCustomClass.queue_push(tq, (b.sin(),), {})
  popped1 = torch.ops.MyCustomClass.queue_pop(tq, (), {})
  popped2 = call_torchbind(tq, "pop", (), {})
  return (popped1, popped2)
```

In **dynamo**, this is done by constructing a TorchScriptObjectVariable and turns all the call_methods call into a higher order op call.

In **aot_autograd**, this is done by intercepting the `ScriptMethod.__call__.`

3. Functionalize call_torchbind and torchbind operators:

As custom objects carry state, it is possible for operations on these custom objects to mutate the underlying states. In order to prevent downstream optimization passes from accidentally reordering or deleting these method calls (e.g. when the method doesn't have output), we made use of **effect tokens** to thread data dependency explicitly in the graph. A token is passed as an input to the graph, and between each operator call which uses a custom class through the `with_effects` higher order operator, and outputted as a result of the graph. The schema of the `with_effects` operator is:

```
Python
with_effects(token, operator, *args, **kwargs)
```

For the above example on `Mod`, the functionalized AOTAutograph graph will look like the following:

```
Python
def graph(token, tq, a, b):
  token1 = with_effects(token, call_torchbind, tq, "push", a)
  token2 = with_effects(token1, torch.ops.MyCustomClass.queue_push.default, tq, b)
  token3 = with_effects(token2, torch.ops.MyCustomClass.queue_push.default, tq, b.sin())
  token4, popped1 = with_effects(token3, torch.ops.MyCustomClass.queue_pop.default, tq)
  token5, popped2 = with_effects(token4, call_torchbind, tq, "pop")
  return token5, tq, popped1, popped2
```

Remove mutations and aliases of inputs in method calls (TODO). Custom classes' methods have schemas but these schemas are auto-generated and don't have mutation or aliasing information attached to it. We can leverage the auto_functionalize infra built for custom ops to auto functionalize the method calls, where we assume all the inputs can be mutated and rely on the backend's (e.g. inductor's) re-replacing pass to remove the data copies.

4. Inductor

Once we get to the Inductor IR, `with_effects` calls are converted to an `EffectfulKernel`. During scheduling, we will create a `StarDep` between each EffectfulKernel so that they don't get reordered. The buffers look something like this:

Python

```
buf2: ExternKernelSchedulerNode(
  EffectfulKernel(
    inputs=[TorchBindObject("arg1_1"), InputBuffer("arg2_1")],
    constant_args=("push",),
    op_overload=call_torchbind,
  )
)
buf2.writes = [StarDep(name='buf2', mode=None)]
buf2.unmet_dependencies = []
buf2.met_dependencies = [StarDep(name='arg1_1', mode=None), StarDep(name='arg2_1', mode=None)]
buf2.users = [NodeUser(node=ExternKernelSchedulerNode(name='buf3'), can_inplace=False, is_weak=False)]


buf3: ExternKernelSchedulerNode(
  EffectfulKernel(
    inputs=[TorchBindObject("arg1_1"), InputBuffer("arg3_1")],
    op_overload=MyCustomClass.queue_push.default,
  )
)
buf3.writes = [StarDep(name='buf3', mode=None)]
buf3.unmet_dependencies = [StarDep(name='buf2', mode=None)]
buf3.met_dependencies = [StarDep(name='arg1_1', mode=None), StarDep(name='arg3_1', mode=None)]
buf3.users = [NodeUser(node=ExternKernelSchedulerNode(name='buf5'), can_inplace=False, is_weak=False)]
```

The inductor generated code looks something like:

Python

```
import torch
from torch._inductor.async_compile import AsyncCompile

empty_strided_cpu = torch._C._dynamo.guards._empty_strided_cpu
async_compile = AsyncCompile()

cpp_fused_sin_0 = async_compile.cpp_pybinding(['const float*', 'float*'], '''
#include "/tmp/torchinductor_angelayi/ky/cky2bufythacofebk7ujv36e4pxyqcqbpsy5r4vojoprjiwcwfxf.h"
extern "C"  void kernel(const float* in_ptr0,
                        float* out_ptr0)
{
    {
        #pragma omp simd simdlen(8)
        for(long x0=static_cast<long>(0L); x0<static_cast<long>(6L); x0+=static_cast<long>(1L))
        {
            auto tmp0 = in_ptr0[static_cast<long>(x0)];
```

```
            auto tmp1 = std::sin(tmp0);
            out_ptr0[static_cast<long>(x0)] = tmp1;
        }
    }
}
''')

def call(args):
    arg1_1, arg2_1, arg3_1 = args
    args.clear()
    assert_size_stride(arg2_1, (2, 3), (3, 1))
    assert_size_stride(arg3_1, (2, 3), (3, 1))
    # Source Nodes: [call_torchbind], Original ATen: []
    torch.ops.higher_order.call_torchbind(arg1_1, 'push', arg2_1)
    del arg2_1
    # Source Nodes: [queue_push], Original ATen: []
    torch.ops.MyCustomClass.queue_push.default(arg1_1, arg3_1)
    buf4 = empty_strided_cpu((2, 3), (3, 1), torch.float32)
    cpp_fused_sin_0(arg3_1, buf4)
    del arg3_1
    # Source Nodes: [queue_push_1, sin], Original ATen: [aten.sin]
    torch.ops.MyCustomClass.queue_push.default(arg1_1, buf4)
    del buf4
    # Source Nodes: [popped1], Original ATen: []
    buf6 = torch.ops.MyCustomClass.queue_pop.default(arg1_1)
    buf7 = buf6
    # Source Nodes: [popped2], Original ATen: []
    buf8 = torch.ops.higher_order.call_torchbind(arg1_1, 'pop')
    del buf6
    buf9 = buf8
    return (buf7, buf9, )
```

Next Steps:

1. Strict-mode export
2. Make is comparison faithful to eager semantic in non-strict export
3. Scale to more custom classes used in production
4. Make export’ dynamic shape API works
5. Support calling constructor in execution
6. Checking the schema of user provided methods against C++ schema
7. Auto_functionalize the method calls