Improving ExecuTorch Custom Ops

Richard Zou Mengwei Liu
Feb 7, 2024

Context

A user has a model that uses an e.g. torchvision custom operator (torchvision.ops.nms) [1, 2]. Custom ops don't work with executorch out of the box:

1. it's unclear who is responsible (torchvision, the user that is trying to ship the model on device, or the executorch team) for making the kernel work with executorch
2. it's unclear how to package the custom op (on the Open Source side) for use on device
3. executorch has strict contracts for how a custom op is implemented that make code reuse between executorch and server difficult.

In general, PyTorch domain libraries (like torchvision) have special custom ops while fbcode has implementations for roughly 4000+ custom ops. We'd like to improve the workflow of getting custom ops to work with executorch by proposing solutions to the above questions.

On ownership and support

Who is responsible for adding executorch support for torchvision.ops.nms? Options include:

- torchvision (i.e. the custom op library maintainers)
- executorch
- a user who is trying to get a model that uses a torchvision custom op to work with executorch

In general, this is a question of priority and resources. Not all custom ops in a library will be used with executorch, and it depends on how many users want to use torchvision with executorch (compare with TorchScript support: this was a large ask from torchvision users, so the team prioritized it.).

This should be decided on a case-by-case basis: the ExecuTorch team should be prepared to field support requests, but if a custom op library has enough executorch user demand then they should own it.

For torchvision, a reasonable compromise for the medium-term is that we put the executorch kernel for nms (and other custom ops) into the torchvision repo, but the executorch team maintains the interaction.

On (Open Source) Packaging

How do we package torchvision ops for use with executorch?

- torchvision has libtorchvision.so that includes CPU kernels for the custom op. This will not interact with executorch (which depends on libtorch.so, which is big and not portable).
- We distribute a separate library (how? source files, a string, or an .a static archive) for the executorch kernels for the torchvision ops.
- Open question if these are available if a user `pip install torchvision` or if they should be downloaded from a separate source.

Option 1:

ExecuTorch kernels live in torchvision repo. This means torchvision now depends on ExecuTorch as well.

Potential issue:

A user wants to build an application that runs torchvision ops on executorch, they first pull in ExecuTorch as a dependency:

app -> libexecutorch.a/so

Then they pull in torchvision because they need the op to run their model

app -> libtorchvision.a/so -> libexecutorch.a/so

The user needs to make sure the 2 libexecutorch are compatible.

On custom op authoring

We will not require users to write custom ops in a “unified style” that works for executorch and server.

ExecuTorch has different (and stricter) custom op requirements than regular PyTorch:

- ExecuTorch has a different calling convention: the implementation of an op needs to accept a RuntimeContext object and executorch::Tensor (not at::Tensor!).
- The implementation of custom ops may not call at:: ops or most Tensor methods.
- ExecuTorch requires operators to have an out= variant.
- ExecuTorch has a different mechanism for throwing exceptions and allocating memory
- Memory allocation

Say that a custom op that matches these requirements is “executorch-compliant”.

Most of the above stems from the design of ExecuTorch: it doesn’t include all of PyTorch (for build size reasons) and has a different C++ Tensor representation. It seems out of the question to relax executorch’s constraints.

We should not enforce all future PyTorch custom ops to be written like ExecuTorch custom ops, or promote this as a default path. It is already difficult enough (see this manual) to write a C++ custom op for PyTorch; the above three constraints are unnatural for users.

How should users write executorch kernels?

Two options:

1. We can duplicate e.g. the torchvision nms kernel by writing one for server and one for executorch
2. We can try to reuse as much code as possible (see the next section)

xExecutorch should provide an “aten compatibility” wrapper for code-reuse

If the user is motivated to use a custom op on both executorch and server AND wants to reuse code, then they should go through the following workflow.

- We should provide an ATen Compatibility wrapper for an executorch kernel that makes it runnable in regular PyTorch.
- This wrapper should be distributed (as “libexecutorch.so” or header-only) that third-party libraries can depend on. (or upstream this to PyTorch (very unlikely)))
- We’ll document this in PyTorch as the recommendation for “how to write a custom op that also works on executorch”.

It would look something like the following (perhaps with more sugar on top).

```
C/C++
#include <et_server_compat.h>

TORCH_LIBRARY(torchvision, m) {
  m.impl("nms_out", DispatchKey::CPU,
executorch::kernel_wrapper(nms_executorch_kernel));
}

// kernel written in the executorch style
Tensor nms_executorch_kernel(executorch::RuntimeContext* ctx, Tensor img, ...)
{
  ...
```

```
}
```

What this would mean for torchvision custom ops is:

- torchvision takes a dependency on executorch
- We refactor each custom op to look something like the above.
- NB: the nms server kernel and the nms executorch kernel may only share a subroutine. That's OK, we can use the wrapper on that.

If this is too much work, then just duplicate the kernel.

We should bring the executorch custom op registration API closer to PyTorch's

Today, executorch's custom op registration requires a user to define a yaml, then go through a codegen step. Compare this to regular PyTorch custom ops, which only requires users to define a TORCH_LIBRARY static initializer in their cpp file. We can improve the executorch custom op UX by bringing it closer to PyTorch's.

```C/C++
// in nms_executorch.cpp
EXECUTORCH_LIBRARY(torchvision, m) {
  m.impl("nms_out", &kernel)
}

// in nms.cpp
TORCH_LIBRARY(torchvision, m) {
  m.impl("nms_out", DispatchKey::CPU,
executorch::wrapper(nms_executorch_kernel));
}
```

Note:

- The prototype implementation of the proposed API inevitably uses C++17 features and thus needs some refactoring to be able to compile on C++11, for what core ExecuTorch is on.
- Note that in the short term this API will only be used on custom ops, where DevX/UX is a concern. For core ATen ops kernel registration it still goes through the old flow.
- Selective build is something we are ignoring right now, with the reasoning that similar functionality can be replaced by optionally linking the kernel library in the build system.

It’s doubtful how much more value to select custom ops, so we decided not to bifurcate the API.

On quantization

We want an easy to use custom ops API that works on both PT2 and ExecuTorch, in order to allow ExecuTorch users to hack different quantization schemes.

For example, if a user wants to quantize the weights into int8 and use their own mm_float32_int8 (instead of aten::mm which doesn’t take mixed dtypes) that takes a float tensor and an int8 tensor, they should be able to do it without too much effort. The API should be easy to understand, not much boilerplate code needs to be written, and the iteration speed (compile & build) should be fast. The same piece of code should be used in PyTorch eager mode as well as ExecuTorch.

A lot of the requirements here align with the API proposal. Related doc
LLM quantization: meeting diverse user needs