Howto: Writing PyTorch & Caffe2 Operators

So you want to write a new operator or a new kernel for an existing operator. How do you do that and what API should you use? So glad you asked.

native_functions.yaml vs custom operators

All operators that are part of the public API of PyTorch are defined in `native_functions.yaml`. Just add an entry there and write the corresponding C++ kernel function. It's very easy and there is a good introduction at https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/README.md.

So when should you not use `native_functions.yaml`?

There's four main use cases

- You're writing a new operator that isn't supposed to be part of the public PyTorch API.
- You're writing a new operator but don't want to change the core pytorch code base, say you're developing a shared library with operators.
- You're writing a C++ extension for PyTorch or you're using inline c++ in your .py model files.
- You're writing a backend library like XLA or ORT that adds new kernels to all operators defined in `native_functions.yaml`.

For these use cases, the custom operator API is the better solution.

What is the price for using the custom operator API instead of `native_functions.yaml`?

If you're just using the custom operator API to add new kernels for existing operators (e.g. the XLA/ORT example above), then you're fine and don't pay any price. If, however, you define a new operator purely using the custom op API, i.e. your operator never shows up in `native_functions.yaml`, then you need to be aware of a few caveats.

- It will not get a C++ API generated. There will not be `Tensor::your_op()` methods or `at::your_op()` functions to call your operator.
- The API for calling the operator from Python looks a little bit different. It needs to be called through `torch.ops.your_op()` instead of `torch._C`.
- Setting up autograd for custom operators is harder. You don't get it automatically but need to use `torch::autograd::Function` to implement autograd (example). Note also that `torch::autograd::Function` does not work together with dispatch yet, so if you have different kernels for different backends (say CPU and CUDA), you need to manually write if/else statements for that.

Writing custom operators

So, you've read all above but still want to use the custom operator API? Great. Here's how you do it.

There's two ways you can write kernels for a PyTorch operator. You can write them as functions or as lambdas.

As functions

This is probably the most simple way to write an operator. Just write a kernel function and register it with the PyTorch operator library.

```
namespace { Tensor my_kernel_cpu(const Tensor& a, const Tensor& b) {...} }

static auto registry = torch::RegisterOperators()
   .op("my_namespace::my_op",  torch::RegisterOperators::options()
       .kernel<decltype(my_kernel_cpu), &my_kernel_cpu>(CPU()));
```

It is recommended to put your kernel into an anonymous namespace because that allows for better linker optimizations and smaller binary size. The dispatch key argument (i.e. `CPU()`) takes care that this kernel is only called for tensors from the CPU backend, more on that below.

As lambdas

Very short and simple kernels can be written as lambdas directly in the registration call:

```
static auto registry = torch::RegisterOperators()
    .op("my_namespace::my_op", torch::RegisterOperators::options()
        .kernel(CPU(), [] (const Tensor& a) -> Tensor{...}));
```

These lambdas must be stateless, i.e. not have a closure. The registration will fail if the lambda has a closure.

Catch-all kernels

You can register catch-all kernels that are called for every backend. This disables dispatch for this operator and just always calls into the kernel you provide. You cannot combine catch-all kernels and regular device-bound kernels for the same operator.

```
namespace { Tensor my_kernel_fallback(Tensor a, Tensor b) {...} }

static auto registry = torch::RegisterOperators()
   .op("my_namespace::my_op", torch::RegisterOperators::options()
       .catchAllKernel<decltype(my_kernel_fallback), &my_kernel_fallback>());
```

The other ways of specifying kernels mentioned above (as functions, functors or lambdas) also work with `catchAllKernel()`.

Syntactic Sugar

You can use the following syntactic sugar to define a catch-all kernel function more easily:

```
namespace { Tensor my_kernel_cpu(const Tensor& a, const Tensor& b) {...}

static auto registry = torch::RegisterOperators()
 .op("my_namespace::my_op", &my_kernel_cpu);
```

or for lambdas:

```
static auto registry = torch::RegisterOperators()
 .op("my_namespace::my_op", [] (Tensor a, Tensor b) {...});
```

Chaining

Multiple operator registrations can be chained into the same registry by calling `.op()` multiple times:

```
static auto registry = torch::RegisterOperators()
    .op("my_namespace::my_op_1", torch::RegisterOperators::options()
        .kernel<MyKernel1>(CPU()))
    .op("my_namespace::my_op_2", torch::RegisterOperators::options()
        .kernel<MyKernel2>(CPU()));
```

Multiple Backends

You can register different kernels for the same operator for different backends.

```
namespace {
Tensor my_kernel_cpu(const Tensor& a, const Tensor& b) {...}
Tensor my_kernel_cuda(const Tensor& a, const Tensor& b) {...}
}

static auto registry = torch::RegisterOperators()
   .op("my_namespace::my_op",  torch::RegisterOperators::options()
       .kernel<decltype(my_kernel_cpu), &my_kernel_cpu>(CPU()))
   .op("my_namespace::my_op",  torch::RegisterOperators::options()
       .kernel<decltype(my_kernel_cuda), &my_kernel_cuda>(CUDA()));
```

Note that here, the CPU and CUDA kernel were registered directly next to each other, but that's not necessary. You could even put them into different shared libraries if you want and as long as both are loaded into your process, things will work as you expect.

The operator schema

Explicitly defining the schema

All examples above automatically inferred the operator schema from the kernel function/lambda. Sometimes, however, you want to specify the schema manually. To specify annotations for example, or default values for arguments (default values will not be inferred from the c++ kernel function), or simply for documentation purposes or to make sure the schema matches your expectations.

```
namespace { Tensor my_kernel_cpu(const Tensor& a, const Tensor& b) {...} }

static auto registry = torch::RegisterOperators()
   .op("my_namespace::my_op(Tensor a, Tensor b) -> Tensor",
       torch::RegisterOperators::options()
         .kernel<decltype(my_kernel_cpu), &my_kernel_cpu>(CPU()));
```

Or with annotations:

```
namespace {
    Tensor my_kernel_cpu(const Tensor& a, int64_t b, at::optional<int64_t> c) {...}
}

static auto registry = torch::RegisterOperators()
   .op("my_namespace::my_op(Tensor(a) x, int y = 3, int? z = None) -> Tensor(a|b)",
       torch::RegisterOperators::options()
         .kernel<decltype(my_kernel_cpu), &my_kernel_cpu>(CPU()));
```

If the schema is explicitly specified but doesn't match the kernel signature, you will get an error when registering it.

Multiple outputs

The kernel function can either return `void` or a single element like `Tensor` in the examples above, or it can return multiple values using `std::tuple` as shown in the following example:

```
namespace {
  std::tuple<Tensor, int64_t, Tensor>
     my_kernel_cpu(const Tensor& a, const Tensor& b, int64_t c) {...}
}

static auto registry = torch::RegisterOperators()
   .op("my_namespace::my_op", torch::RegisterOperators::options()
       .kernel<decltype(my_kernel_cpu), &my_kernel_cpu>(CPU()));
```

Supported Input and output types

The kernel function can take any of the following types as inputs or outputs:

- `at::Tensor`
- `double` (note: `float` is not supported)
- `int64_t` (note: other integer types like `int`, `uint64_t`, `int32_t`, `...` are not supported)
- `bool`
- `c10::string_view`
- `at::Scalar` (this is a type that can hold either an integer or a floating point value)
- `at::optional<T>` with T being any type from the list above

The kernel function can take and return list inputs by using `torch::List<T>`. `T` must be one of the supported types from above excluding `at::Scalar`.

The kernel function can take and return dicts by using `torch::Dict<Key, Value>`. `Key` must be `int64_t`, `c10::string_view`, `double` or `bool`, and `Value` must be from the list of supported types above excluding `at::Scalar`.

When taken as input, any of these types can be taken by value (i.e. `Tensor`) or by const-reference (i.e. `const Tensor&`). We recommend taking all arguments by value, even Tensors. They will be moved in, so there is no performance overhead.

If you need another type, it might work but not be officially supported (yet). Please reach out to Sebastian Messmer and we'll see what we can do.

Overloads

When multiple kernels are registered for the same operator, they must have the same schema or registration will fail. *Note: This also includes schema properties like annotations or default arguments. If one kernel specifies a schema with annotations or a default argument, all kernels for this operator must do this. Schemas automatically inferred from kernel functions will not have annotations or default arguments. This means to use annotations or default arguments, all kernels for this operator must explicitly specify the schema.*

If you want to reuse the same operator name for a different schema, you can use overloads. Overloads must be named and the name is appended to the operator name after a dot:

```
namespace {
  Tensor my_kernel_cpu_1(const Tensor& a) {...}
  Tensor my_kernel_cpu_2(const Tensor& a, const Tensor& b) {...}
}

static auto registry = torch::RegisterOperators()
   .op("my_namespace::my_op.overload1(Tensor a) -> Tensor",
       torch::RegisterOperators::options()
         .kernel<decltype(my_kernel_cpu_1), &my_kernel_cpu>(CPU()))
   .op("my_namespace::my_op.overload2(Tensor a, Tensor b) -> Tensor",
       torch::RegisterOperators::options()
         .kernel<decltype(my_kernel_cpu_2), &my_kernel_cpu>(CPU()));
```

Kernels registered for the same overload must have exactly matching schemas, but kernels registered for different overloads are allowed to have different schemas. This also works when different overloads come from different shared libraries.

Schema-only operators

You can register an operator without a kernel:

```
static auto registry = torch::RegisterOperators()
   .op("my_namespace::my_op(Tensor a, Tensor b) -> Tensor");
```

In this case, you must explicitly specify the full schema and you must not specify a dispatch key. This is useful to define the interface of an operator when you don't know a kernel yet. As mentioned above in the "Overloads" section, you will get an error if any kernel registered for this operator has a mismatching signature.

Calling custom operators

From PyTorch/JIT

All registered operators are automatically available to PyTorch and JIT under `torch.ops.XXX`. If your operator was `my_namespace::my_op`, you can call it from python or JIT using `torch.ops.my_namespace.my_op(a, b)`.

From caffe2

Custom operators are not available to the caffe2 frontend by default, but there's a simple macro you can add if you want to make it available. To expose a CPU kernel:

```
// Expose "my_namespace::my_op" custom operator to caffe2.
// In caffe2, the operator will be called "MyCaffe2OperatorName".
C10_EXPORT_C10_OP_TO_CAFFE2_CPU(
    MyCaffe2OperatorName, "my_namespace::my_op")
```

And to expose a CUDA kernel:

```
C10_EXPORT_C10_OP_TO_CAFFE2_CUDA(
    MyCaffe2OperatorName, "my_namespace::my_op")
```

Note that this doesn't autogenerate a caffe2 operator schema for you (yet). If there's need, we might consider adding that in future, but for now you have to write the caffe2 `OPERATOR_SCHEMA` macro manually if you need it.

Also, there's some requirements on the operator schema for it to be callable from caffe2. Some of these restrictions are just because the functionality isn't implemented. If you have a use case that is blocked by them, please reach out to Sebastian Messmer.

- There must be either one or more arguments of type `Tensor`, or one argument of type `Tensor[]`. You cannot have both `Tensor` and `Tensor[]`.
- Except for `Tensor` or `Tensor[]`, only arguments of type `int`, `double` and `bool` are supported. These can be in any position in the argument list and will be read from the caffe2 operator arguments, based on the argument name in the operator schema.
- We do not support lists (`int[]`, `double[]` or `bool[]`) or optionals (`int?`, `double?`, `bool?`) yet.
- The operator must return a single `Tensor` or multiple tensors as in `(Tensor, Tensor, Tensor)`. It cannot return a list `Tensor[]`, optional `Tensor?` or any primitive types.