Codegen + Structured Kernels Overview

Ed has a really great overview of code-generation and why we have it in PyTorch: check out his podcast: https://pytorch-dev-podcast.simplecast.com/episodes/code-generation.

This document will go over our codegen subsystem + structured kernels in more detail, and involve you using gdb to jump through the different code-generated files that are part of a call into torch.add().

What it is

We have a code-generation pipeline that runs as part of the PyTorch build - it reads in some yaml files, and spits out a bunch of C++ files.

Why we have it

So, why do we have codegen? One big motivating factor is to reduce boilerplate. PyTorch has a lot of operators, and there's a lot of stuff that should "just work" for every operator. We don't want to make someone hand-write all of that functionality whenever a new operator is added. Instead, we code-generate it.

A (non-exhaustive) list of functionality (we need all of this for every operator, so multiply by ~2000):

- bindings to python
- The frontend C++ API
- autograd support
- registering kernels to the dispatcher
- other stuff
  - special logic for factory functions
  - torch.jit.trace functionality

Inputs

We have a yaml file, native_functions.yaml, which describes metadata about each operator that gets consumed by the codegen: https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/n

We're going to focus on the operator torch.add(a, b, out=c), which corresponds to the yaml entry add.out:

```
- func: add.out(Tensor self, Tensor other, *, Scalar alpha=1, Tensor(a!) out) -> Tensor(a!)
  device_check: NoCheck   # TensorIterator
  structured: True
  structured_inherits: TensorIteratorBase
  dispatch:
    CPU, CUDA: add_out
    SparseCPU: add_out_sparse_cpu
    SparseCUDA: add_out_sparse_cuda
```

```
    SparseCsrCPU: add_out_sparse_csr_cpu
    SparseCsrCUDA: add_out_sparse_csr_cuda
    MkldnnCPU: mkldnn_add_out
```

There's public documentation on each of the different pieces of yaml here: https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/README.md

The codegen is written in a functional style using python dataclasses to represent the different inputs/intermediates/outputs. For example, each entry in native_functions.yaml is represented in the codegen as a NativeFunction object: https://github.com/pytorch/pytorch/blob/e1bd4963e23835e307792eb3e49cb5f1b8f771c5/tools/codegen/model.p

Finally, one of the main entry points to the codegen is in tools/codegen/gen.py (there's also a separate entry point for the autograd codegen pipeline). You can see the part of the file where we generate the C++ API for example, Functions.h: https://github.com/pytorch/pytorch/blob/e1bd4963e23835e307792eb3e49cb5f1b8f771c5/tools/codegen/gen.py# It reads in a template file, aten/src/ATen/templates/Functions.h (https://github.com/pytorch/pytorch/blob/e1b and generates the file build/aten/src/ATen/Functions.h:

Exercise 0: Full Stack Trace of torch.add

For this exercise you'll need to have pytorch built with debug symbols. I usually do that with USE_CUDA=0 DEBUG=1 python setup.py develop (The USE_CUDA=0 is because we don't need it, and building with cuda takes a long time).

We're going to run a small python program using gdb to view the full stack trace. Create a python script, tmp.py, with the following:

```
import torch
a = torch.tensor([1, 1])
b = torch.tensor([1, 1])
c = torch.add(a, b)
```

Run gdb python to start up gdb. We're going to set a breakpoint in the add kernel - to do that, in the gdb prompt, type break structured_add_out::impl. Then run your script inside of gdb with run tmp.py.

Gdb should pause inside of the add kernel. Type bt to view the current stack trace.

Ignoring the first ~10 function calls through the python interpreter, you should see a stack trace that looks something like the following:

```
(gdb) bt
#0  0x00007fffdb6a0280 in at::native::structured_add_out::impl(at::Tensor const&, at::Tensor
#1  0x00007fffdcbdae41 in at::(anonymous namespace)::wrapper_add_Tensor (self=..., other=...
    at aten/src/ATen/RegisterCPU.cpp:367
#2  0x00007fffdcd281e4 in c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunct
```

```
    at ../aten/src/ATen/core/boxing/impl/WrapFunctionIntoFunctor.h:13
#3  c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10:
    functor=0x555555f7bff0, args#0=..., args#1=..., args#2=...)
    at ../aten/src/ATen/core/boxing/impl/make_boxed_from_unboxed_functor.h:423
#4  0x00007fffdc968cd1 in c10::callUnboxedKernelFunction<at::Tensor, at::Tensor const&, at::
    unboxed_kernel_func=0x7fffdcd28122 <c10::impl::wrap_kernel_functor_unboxed_<c10::impl::c
    at ../aten/src/ATen/core/boxing/KernelFunction_impl.h:57
#5  0x00007fffdc787a10 in c10::KernelFunction::call<at::Tensor, at::Tensor const&, at::Tensc
    at ../aten/src/ATen/core/boxing/KernelFunction_impl.h:67
#6  c10::Dispatcher::redispatch<at::Tensor, at::Tensor const&, at::Tensor const&, c10::Scala
    this=0x7fffed44c700 <c10::Dispatcher::realSingleton()::_singleton>, op=..., currentDispa
    at ../aten/src/ATen/core/dispatch/Dispatcher.h:536
#7  0x00007fffdc3273cf in c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor
    at ../aten/src/ATen/core/dispatch/Dispatcher.h:398
#8  at::_ops::add_Tensor::redispatch (dispatchKeySet=..., self=..., other=..., alpha=...)
    at aten/src/ATen/Operators.cpp:1620
#9  0x00007fffdf4f3669 in at::redispatch::add (dispatchKeySet=..., self=..., other=..., alph
    at aten/src/ATen/RedispatchFunctions.h:484
#10 0x00007fffdf3e78e7 in torch::autograd::VariableType::(anonymous namespace)::<lambda()>::
#11 0x00007fffdf3e7ce8 in torch::autograd::VariableType::(anonymous namespace)::add_Tensor (
    self=..., other=..., alpha=...) at ../torch/csrc/autograd/generated/VariableType_2.cpp:2
#12 0x00007fffdf4a5245 in c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFuncti
    args#1=..., args#0=..., this=0x555556db7d10)
    at ../aten/src/ATen/core/boxing/impl/WrapFunctionIntoFunctor.h:13
#13 c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10:
    dispatchKeySet=..., args#0=..., args#1=..., args#2=...)
    at ../aten/src/ATen/core/boxing/impl/make_boxed_from_unboxed_functor.h:440
#14 0x00007fffdc968cd1 in c10::callUnboxedKernelFunction<at::Tensor, at::Tensor const&, at::
    unboxed_kernel_func=0x7fffdf4a5158 <c10::impl::wrap_kernel_functor_unboxed_<c10::impl::c
    functor=0x555556db7d10, dispatchKeySet=...) at ../aten/src/ATen/core/boxing/KernelFuncti
#15 0x00007fffdc3270c7 in c10::KernelFunction::call<at::Tensor, at::Tensor const&, at::Tensc
    at ../aten/src/ATen/core/boxing/KernelFunction_impl.h:67
#16 c10::Dispatcher::call<at::Tensor, at::Tensor const&, at::Tensor const&, c10::Scalar cons
    this=0x7fffed44c700 <c10::Dispatcher::realSingleton()::_singleton>)
    at ../aten/src/ATen/core/dispatch/Dispatcher.h:527
--Type <RET> for more, q to quit, c to continue without paging--
#17 c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor const&, c10::Scalar c
    this=<optimized out>) at ../aten/src/ATen/core/dispatch/Dispatcher.h:393
#18 at::_ops::add_Tensor::call (self=..., other=..., alpha=...) at aten/src/ATen/Operators.c
#19 0x00007fffee2afbc5 in at::Tensor::add (this=0x7fffffffcec0, other=..., alpha=...)
    at aten/src/ATen/core/TensorBody.h:1924
#20 0x00007fffee38db89 in torch::autograd::<lambda(const at::Tensor&, const at::Tensor&, cor
    __closure=0x7fffffffceb8, self=..., other=..., alpha=...)
    at ../torch/csrc/autograd/generated/python_torch_functions.cpp:7633
#21 0x00007fffee38e17f in torch::autograd::THPVariable_add (self_=0x0, args=0x7ffff75b3a08,
    at ../torch/csrc/autograd/generated/python_torch_functions.cpp:7635
```

That's a lot of function calls! We're going walk through the main pieces that are relevant to codegen and where they live. For each piece, I listed the relevant numbers in the gdb stack trace.

(1) Python Bindings

#21: torch::autograd::THPVariable_add

This is the first stop that we hit after going through the python interpreter: python bindings. This is the code that interfaces directly with cpython to bind our C++ functions to python.

You can see a snippet of the function below: Its job is basically to take all of the PyObjects that it was handed from cpython, parse them into actual C++ types (like at::Tensor), and call into the C++ API. It does that below by calling into the Tensor add method: self.add(other, alpha).

```
static PyObject * THPVariable_add(PyObject* self_, PyObject* args, PyObject* kwargs)
{
  HANDLE_TH_ERRORS
  static PythonArgParser parser({
    "add(Tensor input, Scalar alpha, Tensor other, *, Tensor out=None)|deprecated",
    "add(Tensor input, Tensor other, *, Scalar alpha=1, Tensor out=None)",
  }, /*traceable=*/true);

  ParsedArgs<4> parsed_args;
  auto _r = parser.parse(nullptr, args, kwargs, parsed_args);
  ...
        auto dispatch_add = [](const at::Tensor & self, const at::Tensor & other, const at::
          pybind11::gil_scoped_release no_gil;
          return self.add(other, alpha);
        };
        return wrap(dispatch_add(_r.tensor(0), _r.tensor(1), _r.scalar(2)));
  ...
  Py_RETURN_NONE;
  END_HANDLE_TH_ERRORS
}
```

These are all codegen'd and live in `torch/csrc/autograd/generated/python_torch_functions.cpp`.

(2) C++ API

#19: at::Tensor::add #18: at::_ops::add_Tensor::call

The next stop is the C++ method API, which is one of the top-level API's for calling into the dispatcher. The dispatcher then looks at all of the arguments + any thread-local state to figure out which kernel to dispatch to. https://fb.quip.com/crnrAjLMKVbs has some more details about the dispatcher key-calculation process.

In `build/aten/src/ATen/TensorBody.h`:

```
//namespace at
inline at::Tensor Tensor::add(const at::Tensor & other, const at::Scalar & alpha) const {
    return at::_ops::add_Tensor::call(const_cast<Tensor&>(*this), other, alpha);
}
```

In `build/aten/src/ATen/Operators.cpp`:

```
//namespace at::_ops
at::Tensor add_Tensor::call(const at::Tensor & self, const at::Tensor & other, const at::Sca
    static auto op = c10::Dispatcher::singleton()
        .findSchemaOrThrow("aten::add", "Tensor")
        .typed<at::Tensor (const at::Tensor &, const at::Scalar &)>();
    return op.call(self, other, alpha);
}
```

(3) Autograd kernel

#11: torch::autograd::VariableType::(anonymous namespace)::add_Tensor

After a bunch of dispatcher-related functions, the dispatcher eventually takes us to the autograd add kernel. The autograd kernel:

- saves some metadata for autograd
- re-invokes the dispatcher by calling at::redispatch::add(ks & c10::after_autograd_keyset, self_, other_, alpha);

```
// namespace at::VariableType
at::Tensor add_Tensor(c10::DispatchKeySet ks, const at::Tensor & self, const at::Tensor & ot
  ...
}

TORCH_LIBRARY_IMPL(aten, Autograd, m) {
  m.impl("add.Tensor",TORCH_FN(VariableType::add_Tensor));
}
```

You can find the autograd add kernel, along with the code that registers it to the dispatcher, in torch/csrc/autograd/generated/VariableTypeEverything.cpp

The autograd kernel ends up calling back into the C++ API (by calling at::redispatch::add), which then calls back into the dispatcher and calculates the next kernel to dispatch to.

(4) CPU kernel

#1: at::(anonymous namespace)::wrapper__add_Tensor #0: at::native::structured_add_out::impl

After a few more function hops through the dispatcher, we eventually dispatch to the CPU add kernel, which has to actually carry out the computation. The

code for the cpu kernel (and the code that registers the kernel to the dispatcher) looks like this:

```
at::Tensor wrapper_add_Tensor(const at::Tensor & self, const at::Tensor & other, const at::S
  structured_add_out_functional op;
  op.meta(self, other, alpha);
  op.impl(self, other, alpha, op.outputs_[0]);
  return std::move(op.outputs_[0]);
}

TORCH_LIBRARY_IMPL(aten, CPU, m) {
  m.impl("add.Tensor", TORCH_FN(wrapper_add_Tensor));
}
```

This code looks a little funky; it calls into a meta() and impl() function that are defined elsewhere. This is because add is implemented as a structured kernel - a new way of implementing operators in pytorch.

That code is some scaffolding that contains a call to the hand-written "cpu add" kernel. The call to op.impl() corresponds directly to the add kernel written here, in BinaryOps.cpp (https://github.com/pytorch/pytorch/blob/e1bd4963e23835e307792eb3e49cb5f1b8f771c5/aten
(Note: there's a bit more indirection inside of the handwritten kernel before reaching main part of the add kernel, which lives here, in BinaryOpsKernel.cpp (https://github.com/pytorch/pytorch/blob/e1bd4963e23835e307792eb3e49cb5f1b8f771c5/aten/src/ATen/native/

The code-generated code above lives in the code-generated file build/aten/src/ATen/RegisterCPU.cpp.

So, the code above calls into our hand-written CPU add kernel, returns a new output tensor containing the result, and we're done!

Takeaway

The main takeaway from the exercise above is that:

- A lot of stuff happens when you call an operator
- ...most of which is code-generated! A lot of this logic is *really* similar across PyTorch's ~2000 operators, and ripe for abstracting over (through something like code generation).

Sometimes when you're working on / debugging a feature, it can be useful to know which bits of logic are codegen'd, and where that logic lives.

Structured Kernels

Another big part of the codegen is "structured kernels" - a new way of writing kernels in PyTorch, which uses some clever factoring + a bunch of codegen to reduce the amount of boilerplate required when writing kernels. torch.add is implemented as a "structured kernel", so we're going to walk through the bits of it related to structured kernels.

The process of implementing an operator as a structured kernel involves writing two functions:

- A "meta" function, which asserts that the inputs have the correct shape/dtype and figures out what size the output tensor should be.
- An "impl" function, which does the actual computation. There will be a separate impl() function for every backend (cpu, cuda, xla, etc).

The codegen is responsible for taking these two functions, and plugging them together in the right way to create all 3 variants of the operator for you:

- at::add() (functional version)
- at::add_() (inplace version)
- at::add_out() (out= version)

Helpful reading: this presentation on structured kernels, including a diagram on the class hierarchy (which will be useful in the exercise further down). https://drive.google.com/file/d/16qPvpCF4Jbh7ss2lCQMk5hmcyzJvUyQj/view?usp=sharing

See also: the structured kernels RFC contains a more detailed overview of what they are and what the codegen creates: https://github.com/pytorch/rfcs/blob/rfc-0005/RFC-0005-structured-kernel-definitions.md

Structured Kernel codegen output example: torch.add

The CPU kernel for the torch.add operator lives in https://github.com/pytorch/pytorch/blob/master/aten/src/A
and has two components:

Meta function:

```
// expands to structured_add_Tensor::meta() { ... }
TORCH_META_FUNC2(add, Tensor) (
  const Tensor& self, const Tensor& other, const Scalar& alpha
) {
  build_borrowing_binary_op(maybe_get_output(), self, other);
  native::alpha_check(dtype(), alpha);
}
```

Impl function:

```
// expands to structured_add_out::impl() { ... }
TORCH_IMPL_FUNC(add_out) (
  const Tensor& self, const Tensor& other, const Scalar& alpha, const Tensor& result
) {
  add_stub(device_type(), *this, alpha);
  TORCH_INTERNAL_ASSERT(result.scalar_type() == output().dtype());
}
```

So, the code above implements the two functions structured_add_Tensor::meta() and structured_add_out::impl(), but where are they declared? The codegen

creates declarations for them.

In NativeMetaFunctions.h:

```
// namespace at::meta
struct TORCH_API structured_add_Tensor : public TensorIteratorBase {
    void meta(const at::Tensor & self, const at::Tensor & other, const at::Scalar & alpha);
};
```

In NativeFunctions.h:

```
// namespace at::native
struct TORCH_API structured_add_out : public at::meta::structured_add_Tensor {
    void impl(const at::Tensor & self, const at::Tensor & other, const at::Scalar & alpha, c
};
```

You can see that the codegen generated declarations for the two functions, and we hand-implemented them ourselves in BinaryOps.cpp. But how does the codegen use them?

The code-generated logic that stitches them together lives in the code-generated file RegisterCPU.cpp, and looks like this:

```
// functional version
at::Tensor wrapper_add_Tensor(const at::Tensor & self, const at::Tensor & other, const at::S
  structured_add_out_functional op;
  op.meta(self, other, alpha);
  op.impl(self, other, alpha, op.outputs_[0]);
  return std::move(op.outputs_[0]);
}

// inplace version
at::Tensor & wrapper_add__Tensor(at::Tensor & self, const at::Tensor & other, const at::Scal
  structured_add_out_inplace op(self);
  op.meta(self, other, alpha);
  op.impl(self, other, alpha, op.outputs_[0]);
  return self;
}

// out= version
at::Tensor & wrapper_add_out_out(const at::Tensor & self, const at::Tensor & other, const at
  structured_add_out_out op(out);
  op.meta(self, other, alpha);
  op.impl(self, other, alpha, op.outputs_[0]);
  return out;
}

// registering the 3 kernels above to the dispatcher, under the CPU Dispatch Key.
TORCH_LIBRARY_IMPL(aten, CPU, m) {
```

```
  ...
  m.impl("add.Tensor", TORCH_FN(wrapper_add_Tensor));
  m.impl("add.out", TORCH_FN(wrapper_add_out_out));
  m.impl("add_.Tensor", TORCH_FN(wrapper_add__Tensor));
}
```

This is the "final" output - the 3 operators that we needed. The codegen created 3 new kernels, each of which call into our meta() and impl() functions. The only difference between the 3 is that they use different classes, each of which has a different implementation of set_output(). You can also find the definition of all 3 of these classes in RegisterCPU.cpp, but below is the example for structured_add_out_functional:

```
struct structured_add_out_functional final : public at::native::structured_add_out {

    void set_output(int64_t output_idx, IntArrayRef sizes, IntArrayRef strides,
                    TensorOptions options, DimnameList names) override {

        if (strides.empty()) {
            outputs_[output_idx] = at::native::empty_cpu(sizes, optTypeMetaToScalarType(opti
        } else {
            // TODO: assert options.memory_format_opt() is nullopt (debug only?)
            outputs_[output_idx] = at::native::empty_strided_cpu(sizes, strides, optTypeMeta
        }

        if (!names.empty()) {
          namedinference::propagate_names(outputs_[output_idx], names);
        }
        // super must happen after, so that downstream can use maybe_get_output
        // to retrieve the output
        at::native::structured_add_out::set_output(output_idx, sizes, strides, options, name
    }

    const Tensor& maybe_get_output(int64_t output_idx) override {
        return outputs_[output_idx];
    }
    std::array<Tensor, 1> outputs_;
};
```

You can see that it has its own definition of set_output() - in this case, it's implementing the functional at::add kernel, so it needs to allocate a new tensor as the output (it does that using at::native::empty_cpu()).

That class corresponds to one of the leaves of the class hierarchy - a picture of the full class hierarchy can be found in the linked presentation (https://drive.google.com/file/d/16qPvpCF4Jbh7ss2lCQMk5hmcyzJvUyQj/view?usp=sharing)