Codegen Task: Update an op to bypass the Dispatcher

This is a more hands-on task meant to walk you through making a change to an operator that impacts our codegen and dispatcher registration subsystems. The goal of this task is to:

- Give an introduction to our codegen subsystem
  - pre-reading: https://github.com/pytorch/pytorch/wiki/Codegen-and-Structured-Kernels
- Learn how the codegen interacts with the dispatcher
- Learn more about Structured Kernels
- See how the [codegen] / [dispatcher registration] / [call path of an operator] all change when we make a tweak to bypass the dispatcher.

You're going to modify at::add and see what happens!

Exercise: Make torch.add Bypass the Dispatcher

In this lab, you're going to update the cpu kernel of at::add to bypass the dispatcher.

Why would we want to update at::add to bypass the dispatcher? We don't - this will remove a lot of core functionality (and break any relevant tests - you won't be able to use torch.add with autograd, measure it in the profiler, or run it on cuda, amongst other things). But skipping over all that logic will also make torch.add faster!

This is meant to serve as a repeatable lab to get exposure to the codegen/dispatcher, since there aren't that many bootcamp-friendly outstanding github issues for either subsystem.

The lab will involve:

- Learning more about structured kernels
- Staring at some code-generated output
- Measuring performance cost associated with all of the flexibility that the dispatcher gives us

Bypassing the dispatcher

The dispatcher gives us a lot of flexibility - we can implement separate add kernels for different backends, provide autograd support, and hook in custom behavior like batching and tracing. But that extra flexible also comes with a performance cost!

There are some basic operators that we've are simple enough and perf-critical enough that they don't need to be part of the dispatcher. The codegen provides a hook to bypass the dispatcher through the manual_cpp_bindings keyword in native_functions.yaml. You can find some example operators that use it by grepping for that keyword in native_functions.yaml: one example is is_complex().

Here's  a diagram describing the call path in C++ before/after skipping the dispatcher.

The Change

Note 1: Before starting, I recommend you create two separate conda environments. e.g. codegen_task_before and codegen_task_after. Run a REL_WITH_DEB_INFO=1 build before your changes in the first conda environment, and then switch to the second environment when making your changes. That will make it very easy to benchmark and compare your change to the baseline.

Note 2: Feel free to ask the task owner for help if you're stuck. There's also a separate doc with tips + some gotchas that you might encounter here: https://fb.quip.com/L7PvAJOIb5iL.

The main change that you're going to make is to update the entries for add/add_/add_out in native_functions.yaml to mark them as manual_cpp_bindings.

Delete the structured/structured_inherits/structured_delegate/dispatch lines from each entry, and add in a new line: manual_cpp_bindings: True. You will also need to make the same change in derivatives.yaml.

By removing all of the structured metadata and adding manual_cpp_bindings, a few things will happen:

- The codegen will stop generating all of the structured kernel scaffolding for add:
  - Function and Method API declaration/definitions in Functions.h and TensorBody.h
  - Calls to the dispatcher (these live in Operators.cpp)
  - Structured Kernel meta() and impl() declarations, which live in NativeFunctions.h and NativeMetaFunctions.h
  - All of the class scaffolding + actual codegen'd kernels for the 3 add variants that you saw in RegisterCPU.cpp
- The codegen will now expect the existence of the following functions, which you will need to implement.
  - at:: The functions API (add and add_out)
  - at::Tensor:: The methods API (add and add_)
  - at::native:: The native API. This namespace contains all of our internal kernel definitions. (add, add_ and add_out)

How to make the changes

What should the functions look like? You can use the existing structured kernel scaffolding, but the codegen will no longer generate it for you since we're using manual_cpp_bindings. Instead, you can manually write the structured kernel scaffolding (probably by copy-pasting the output of the original codegen).

Where to make changes

There are 4 files that make up the main parts of the C++ function and method API's:

aten/src/ATen/templates/Functions.h aten/src/ATen/templates/Functions.cpp aten/src/ATen/templates/TensorBody.h aten/src/ATen/templates/TensorMethods.cpp

These files are all in the templates folder because they are ingested by the codegen. The codegen is responsible for filling the files with the function and method declarations/definitions for all of the ~2000 ops in native_functions.yaml.

The original file for the native API of add lives in aten/src/ATen/native/BinaryOps.cpp, so you can write your native kernels there.

Finally, don't bother trying to submit a PR with this change and run CI against your branch - lots of stuff will fail, because we're effectively ripping out a bunch of core functionality for torch.add: most importantly, autograd + the ability to run it on cuda. But we've done the bare minimum to make the compiler happy so we can run torch.add() and benchmark it!

Benchmarking

You can re-run the gdb the same way that we did before and see that a lot less is getting called. Instead of invoking the dispatcher to send us to the add kernel, calling at::add() takes us straight to the kernel implementation.

Test A: the profiler

Another feature that's provided directly inside of the dispatcher is the profiler, which profiles the set of aten ops that ran in your model as well as how much time was spent in them.

Run the following snippet before/after your change, and compare:

> *import torch with torch.profiler.profile() as prof: ... a = torch.ones(2) ... b = torch.ones(2) ... c = a + b ... str(prof.events())*

You should see that aten::add() no longer shows up in the profiler! The profiler is no longer aware of aten::add(), since the function bypasses the dispatcher. It also completely excludes the time spent running at::add() in the total time, since it's not aware of the call.

Test B (Optional): instruction count measurements

To see how much faster torch.add() is without the extra dispatcher overhead, we have some helpful tools for benchmarking code.

If you're interested, and for a great overview to benchmarking in PyTorch, see the guide here: https://pytorch.org/tutorials/recipes/recipes/timer_quick_start.html#instruction-counts-timer-collect-callgrind. Take a look at that guide, and create an instruction count benchmark for torch.add() before/after your change. Note that running the benchmarks requires valgrind.

You'll see a very large speedup - that's mostly because we're measuring with very small tensors, so much of the time taken is from framework overhead - i.e. the (lack of) use of the dispatcher.

Send the task owner your wall-clock times and/or instruction count measurements (you can just include the total counts, not the function call tree breakdown) - you're done!