CUDA loops case study: code generation vs templates

dev-discuss.pytorch.org/t/cuda-loops-case-study-code-generation-vs-templates/302

August 25, 2021

Problem statement

It's not easy defining a pointwise operation in CUDA for TensorIterator. Because we are ahead-of-time compiling a set of kernels that must work universally for all combinations of dtypes/scalar combinations (in this Quip, I ignore the actual *iteration* aspects of TensorIterator), there is quite a bit of work you have to do:

- When defining any particular kernel, you must statically know *both* what the stored in memory type (scalar_t) and the intermediate computation types (accscalar_t) are, because they can be divergent (e.g., read out half from memory, but do computation in float.)
- You need separate kernels for tensor-tensor computation and tensor-scalar computation, as the scalar typically lives in CPU memory and must be transmitted (via CUDA parameters) to the CUDA kernel. Furthermore, the scalar should be transmitted in accscalar_t, as scalars may still have more precision than the tensor itself (even if they do not cause the type of the tensor to promote; this occurs when you do something like `half_tensor * 0.25`; the result is a half tensor, even though the constant is represented in float precision).
- Your kernels may have extra, non-Tensor arguments (e.g., alpha in add) which need to be transmitted to the kernel via CUDA parameters in accscalar_t precision.
- You cannot use GPU lambdas because NVCC will fail to deduplicate duplicate nested lambdas (as of 2020, see Use explicit templates in `gpu_kernel_with_scalars` by malfet · Pull Request #40992 · pytorch/pytorch · GitHub< 1), increasing our binary size.

Naive implementation (aka what to generate in codegen)

Here's what a naive (i.e., with flagrant code duplication) implementation of addition might look, ignoring the "no GPU lambdas" constraint, assuming `gpu_kernel` (which handles the actual iteration) as a primitive:

```
template <typename scalar_t>
void add_kernel(TensorIteratorBase& iter, const Scalar& alpha_scalar) {
  using accscalar_t = at::acc_type<scalar_t, true>;
  auto alpha = alpha_scalar.to<accscalar_t>();

  if (iter.is_cpu_scalar(1)) {
    auto a = iter.scalar_value<accscalar_t>(1);
    iter.remove_operand(1);
    gpu_kernel(iter, [=](scalar_t b) -> scalar_t { return a + b * alpha; });

  } else if (iter.is_cpu_scalar(2)) {
    auto b = iter.scalar_value<accscalar_t>(2);
    iter.remove_operand(2);
    gpu_kernel(iter, [=](scalar_t a) -> scalar_t { return a + b * alpha; });

  } else {
    gpu_kernel(iter, [=](scalar_t a, scalar_t b) -> scalar_t { return a + b * alpha;
});
  }
}
```

There are a number of things about this implementation worth calling out:

- We generate three kernels in total: Scalar-Tensor, Tensor-Scalar, Tensor-Tensor. This is a 3x binary size increase over just generating a Tensor-Tensor kernel.
    - Each lambda captures by value a different set of values (everyone captures alpha, the first lambda captures a, the second captures b); the capture by value translates into CUDA arguments.
    - Thought: As these kernels are typically memory bound, it may be possible to coalesce these kernels into a single kernel along the lines of `as + has_at ? at : 0 + (bs + has_bt ? bt : 0) * alpha` if the ternaries translate into conditional loads.
- The computation in the lambdas is always done in `accscalar_t`, as alpha is higher precision and forces promotion of all the other arguments. In other kernels, this promotion does not necessarily always happen, and so for correct behavior in half precision it is necessary to explicitly cast arguments to the intermediate computation type; e.g., `accscalar_t(a) + accscalar_t(b) * accscalar_t(alpha)`
- It is important for the lambda to explicitly take `scalar_t` and return a `scalar_t`; this is how we tell `gpu_kernel` what the expected input/output types of the tensors are. If there is a mismatch, this triggers a dynamic_casting codepath which, as currently implemented, triggers a 2x slowdown.
- At the same time, it is important that scalar a and b are represented as `accscalar_t`; otherwise you could lose precision if the scalar is at higher precision.
- Each lambda can be individually converted into a functor using the standard transformation (make every closed over value field on the functor), resulting in three structs.

In a code generator implementation of TensorIterator, we can simply choose to generate the code as seen here and call it a day.

Template implementation

The logic seen above must be replicated for every binary operator, so there is a good deal of incentive to reduce the duplication. There are two axes of duplication which are desirable to remove:

- Logic for testing if either argument is a CPU scalar, and appropriately dispatching
- The lambda itself, which is replicated three times with different values that it closes over

The current implementation shipped by PyTorch handles this by deriving the Tensor-Scalar/Scalar-Tensor lambdas from the Tensor-Tensor variant. Here is what is effectively done (I've taken this code from db2b273d1 prior to the functor-ization of the lambdas for clarity, and done some modest editing for clarity):

```
template <typename func_t>
void gpu_kernel_with_scalars(TensorIterator& iter, const func_t& f) {
  ASSERT_HOST_DEVICE_LAMBDA(func_t);

  using traits = function_traits<func_t>;
  using arg1_t = typename traits::template arg<0>::type;
  using arg2_t = typename traits::template arg<1>::type;

  if (iter.is_cpu_scalar(1)) {
    auto a = iter.scalar_value<arg1_t>(1);
    iter.remove_operand(1);
    gpu_kernel(iter, [=]GPU_LAMBDA(arg2_t b) {
      return f(a, b);
    });
  } else if (iter.is_cpu_scalar(2)) {
    auto b = iter.scalar_value<arg2_t>(2);
    iter.remove_operand(2);
    gpu_kernel(iter, [=]GPU_LAMBDA(arg1_t a) {
      return f(a, b);
    });
  } else {
    gpu_kernel(iter, f);
  }
}

// Invoked as:

gpu_kernel_with_scalars(iter, [alpha]GPU_LAMBDA(scalar_t a, scalar_t b) -> scalar_t {
    return a + alpha * b;
});
```

Unfortunately, this implementation fails to handle CPU scalars correctly in half precision. This is because the lambda which the derived Tensor-Scalar lambdas use takes in `scalar_t` arguments, but we actually want the CPU scalar to be applied directly to the computation in `accscalar_t` precision. Even if we correct the `scalar_value` invocations in the body of the code to accept `accscalar_t`, an undesirable downcast will still occur when we invoke the `f` functor.

One logical fix for this issue is to make the lambda accept `accscalar_t`:

```
gpu_kernel_with_scalars(iter, [alpha]GPU_LAMBDA(accscalar_t a, accscalar_t b) ->
scalar_t {
    return a + alpha * b;
});
```

However, this silently tanks the performance of the kernel by more than 2x (Add acc_gpu_kernel_with_scalars and port add to use it by ezyang · Pull Request #63884 · pytorch/pytorch · GitH); this is because the static type of the lambda no longer matches the type of data in memory in the tensors, and that shunts us to the dynamic_casting codepath. A final fix that *does* work is to explicitly instruct `gpu_kernel_with_scalars` what the intended input types are, and then rewrap the lambda to expose the correct static types:

```
template <typename scalar_t, typename func_t>
void gpu_kernel_with_scalars(TensorIterator& iter, const func_t& f) {
  ASSERT_HOST_DEVICE_LAMBDA(func_t);

  using traits = function_traits<func_t>;
  using arg1_t = typename traits::template arg<0>::type;
  using arg2_t = typename traits::template arg<1>::type;

  if (iter.is_cpu_scalar(1)) {
    auto a = iter.scalar_value<arg1_t>(1);
    iter.remove_operand(1);
    gpu_kernel(iter, [=]GPU_LAMBDA(scalar_t b) -> scalar_t {
      return f(a, b);
    });
  } else if (iter.is_cpu_scalar(2)) {
    auto b = iter.scalar_value<arg2_t>(2);
    iter.remove_operand(2);
    gpu_kernel(iter, [=]GPU_LAMBDA(scalar_t a) -> scalar_t {
      return f(a, b);
    });
  } else {
    gpu_kernel(iter, [=]GPU_LAMBDA(scalar_t a, scalar_t b) -> scalar_t {
      return f(a, b);
    });
  }
}

// Invoked as:

gpu_kernel_with_scalars<scalar_t>(iter, [alpha]GPU_LAMBDA(accscalar_t a, accscalar_t
b) -> scalar_t {
    return a + alpha * b;
});
```

Each inner lambda translates into a separate functor which wraps the initially provided lambda/functor.

Analysis

I originally went into writing this case study with the idea that it was not possible to implement the logic here nicely with templates, and was pleasantly surprised when I came up with a solution while explaining the difficulties with templating. I think in the short term, the modification of gpu_kernel_with_scalars to add an (optional) template argument for true memory type is probably the most expedient course of action. However, I have some other thoughts:

- The handling of dtypes when writing lambdas for CUDA kernels is extremely subtle. gpu_kernel does not provide any compile-time or runtime feedback about whether or not you have provided the correct types, so the only way to find out if you've fallen into the dynamic casting trap is by noticing that your CUDA kernel performance has tanked (granted, we make everyone working on CUDA kernels benchmark their kernels, so hopefully something like this would be noticed).
- Functorization of lambda (not shown here) is a serious impediment to the readability of our code here, so it is worth carefully analyzing *why* functorization helps reduce CUDA binary size; it really shouldn't!
- We are generating waaaay too much code for CUDA TensorIterator, and Tensor-Scalar shenanigans only make it (3x) worse. We should figure out if there are ways of compacting our AOT kernels without sacrificing performance, by taking advantage of the fact that most pointwise kernels are memory bound.