Table of Contents

CUSTOM C++ AND CUDA EXTENSIONS

Author: Peter Goldsborough

PyTorch provides a plethora of operations related to neural networks, arbitrary tensor algebra, data wrangling and other purposes. However, you may still find yourself in need of a more customized operation. For example, you might want to use a novel activation function you found in a paper, or implement an operation you developed as part of your research.

The easiest way of integrating such a custom operation in PyTorch is to write it in Python by extending `Function` and `Module` as outlined here. This gives you the full power of automatic differentiation (spares you from writing derivative functions) as well as the usual expressiveness of Python. However, there may be times when your operation is better implemented in C++. For example, your code may need to be *really* fast because it is called very frequently in your model or is very expensive even for few calls. Another plausible reason is that it depends on or interacts with other C or C++ libraries. To address such cases, PyTorch provides a very easy way of writing custom *C++ extensions*.

C++ extensions are a mechanism we have developed to allow users (you) to create PyTorch operators defined *out-of-source*, i.e. separate from the PyTorch backend. This approach is *different* from the way native PyTorch operations are implemented. C++ extensions are intended to spare you much of the boilerplate associated with integrating an operation with PyTorch's backend while providing you with a high degree of flexibility for your PyTorch-based projects. Nevertheless, once you have defined your operation as a C++ extension, turning it into a native PyTorch function is largely a matter of code organization, which you can tackle after the fact if you decide to contribute your operation upstream.

Motivation and Example

The rest of this note will walk through a practical example of writing and using a C++ (and CUDA) extension. If you are being chased or someone will fire you if you don't get that op done by the end of the day, you can skip this section and head straight to the implementation details in the next section.

Let's say you've come up with a new kind of recurrent unit that you found to have superior properties compared to the state of the art. This recurrent unit is similar to an LSTM, but differs in that it lacks a *forget gate* and uses an *Exponential Linear Unit* (ELU) as its internal activation function. Because this unit never forgets, we'll call it *LLTM*, or *Long-Long-Term-Memory* unit.

The two ways in which LLTMs differ from vanilla LSTMs are significant enough that we can't configure PyTorch's `LSTMCell` for our purposes, so we'll have to create a custom cell. The first and easiest approach for this – and likely in all cases a good first step – is to implement our desired functionality in plain PyTorch with Python. For this, we need to subclass `torch.nn.Module` and implement the forward pass of the LLTM. This would look something like this:

```
class LLTM(torch.nn.Module):
    def __init__(self, input_features, state_size):
        super(LLTM, self).__init__()
        self.input_features = input_features
        self.state_size = state_size
        # 3 * state_size for input gate, output gate and candidate cell gate.
        # input_features + state_size because we will multiply with [input, h].
        self.weights = torch.nn.Parameter(
            torch.empty(3 * state_size, input_features + state_size))
        self.bias = torch.nn.Parameter(torch.empty(3 * state_size))
        self.reset_parameters()

    def reset_parameters(self):
        stdv = 1.0 / math.sqrt(self.state_size)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, +stdv)

    def forward(self, input, state):
        old_h, old_cell = state
        X = torch.cat([old_h, input], dim=1)

        # Compute the input, output and candidate cell gates with one MM.
        gate_weights = F.linear(X, self.weights, self.bias)
        # Split the combined gate weight matrix into its components.
        gates = gate_weights.chunk(3, dim=1)

        input_gate = torch.sigmoid(gates[0])
        output_gate = torch.sigmoid(gates[1])
        # Here we use an ELU instead of the usual tanh.
        candidate_cell = F.elu(gates[2])

        # Compute the new cell state.
        new_cell = old_cell + candidate_cell * input_gate
        # Compute the new hidden state and output.
        new_h = torch.tanh(new_cell) * output_gate

        return new_h, new_cell
```

which we could then use as expected:

```
import torch

X = torch.randn(batch_size, input_features)
h = torch.randn(batch_size, state_size)
C = torch.randn(batch_size, state_size)

rnn = LLTM(input_features, state_size)

new_h, new_C = rnn(X, (h, C))
```

Naturally, if at all possible and plausible, you should use this approach to extend PyTorch. Since PyTorch has highly optimized implementations of its operations for CPU *and* GPU, powered by libraries such as NVIDIA cuDNN, Intel MKL or NNPACK, PyTorch code like above will often be fast enough. However, we can also see why, under certain circumstances, there is room for further performance improvements. The most obvious reason is that PyTorch has no knowledge of the *algorithm* you are implementing. It knows only of the individual operations you use to compose your algorithm. As such, PyTorch must execute your operations individually, one after the other. Since each individual call to the implementation (or *kernel*) of an operation, which may involve the launch of a CUDA kernel, has a certain amount of overhead, this overhead may become significant across many function calls. Furthermore, the Python interpreter that is running our code can itself slow down our program.

A definite method of speeding things up is therefore to rewrite parts in C++ (or CUDA) and *fuse* particular groups of operations. Fusing means combining the implementations of many functions into a single function, which profits from fewer kernel launches as well as other optimizations we can perform with increased visibility of the global flow of data.

Let's see how we can use C++ extensions to implement a *fused* version of the LLTM. We'll begin by writing it in plain C++, using the ATen library that powers much of PyTorch's backend, and see how easily it lets us translate our Python code. We'll then speed things up even more by moving parts of the model to CUDA kernel to benefit from the massive parallelism GPUs provide.

Writing a C++ Extension

C++ extensions come in two flavors: They can be built "ahead of time" with `setuptools`, or "just in time" via `torch.utils.cpp_extension.load()`. We'll begin with the first approach and discuss the latter later.

Building with `setuptools`

For the "ahead of time" flavor, we build our C++ extension by writing a `setup.py` script that uses setuptools to compile our C++ code. For the LLTM, it looks as simple as this:

```
from setuptools import setup, Extension
from torch.utils import cpp_extension

setup(name='lltm_cpp',
      ext_modules=[cpp_extension.CppExtension('lltm_cpp', ['lltm.cpp'])],
      cmdclass={'build_ext': cpp_extension.BuildExtension})
```

In this code, `CppExtension` is a convenience wrapper around `setuptools.Extension` that passes the correct include paths and sets the language of the extension to C++. The equivalent vanilla `setuptools` code would simply be:

```
Extension(
   name='lltm_cpp',
   sources=['lltm.cpp'],
   include_dirs=cpp_extension.include_paths(),
   language='c++')
```

`BuildExtension` performs a number of required configuration steps and checks and also manages mixed compilation in the case of mixed C++/CUDA extensions. And that's all we really need to know about building C++ extensions for now! Let's now take a look at the implementation of our C++ extension, which goes into `lltm.cpp`.

Writing the C++ Op

Let's start implementing the LLTM in C++! One function we'll need for the backward pass is the derivative of the sigmoid. This is a small enough piece of code to discuss the overall environment that is available to us when writing C++ extensions:

```
#include <torch/extension.h>

#include <iostream>

torch::Tensor d_sigmoid(torch::Tensor z) {
  auto s = torch::sigmoid(z);
  return (1 - s) * s;
}
```

`<torch/extension.h>` is the one-stop header to include all the necessary PyTorch bits to write C++ extensions. It includes:

- The ATen library, which is our primary API for tensor computation,
- pybind11, which is how we create Python bindings for our C++ code,
- Headers that manage the details of interaction between ATen and pybind11.

The implementation of `d_sigmoid()` shows how to use the ATen API. PyTorch's tensor and variable interface is generated automatically from the ATen library, so we can more or less translate our Python implementation 1:1 into C++. Our primary datatype for all computations will be `torch::Tensor`. Its full API can be inspected here. Notice also that we can include `<iostream>` or *any other C or C++ header* – we have the full power of C++11 at our disposal.

Note that CUDA-11.5 nvcc will hit internal compiler error while parsing torch/extension.h on Windows. To workaround the issue, move python binding logic to pure C++ file. Example use:

```
#include <ATen/ATen.h>
at::Tensor SigmoidAlphaBlendForwardCuda(....)
```

Instead of:

```
#include <torch/extension.h>
torch::Tensor SigmoidAlphaBlendForwardCuda(...)
```

Currently open issue for nvcc bug here. Complete workaround code example here.

Forward Pass

Next we can port our entire forward pass to C++:

```
#include <vector>

std::vector<at::Tensor> lltm_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  auto X = torch::cat({old_h, input}, /*dim=*/1);

  auto gate_weights = torch::addmm(bias, X, weights.transpose(0, 1));
  auto gates = gate_weights.chunk(3, /*dim=*/1);

  auto input_gate = torch::sigmoid(gates[0]);
  auto output_gate = torch::sigmoid(gates[1]);
  auto candidate_cell = torch::elu(gates[2], /*alpha=*/1.0);

  auto new_cell = old_cell + candidate_cell * input_gate;
  auto new_h = torch::tanh(new_cell) * output_gate;

  return {new_h,
          new_cell,
          input_gate,
          output_gate,
          candidate_cell,
          X,
          gate_weights};
}
```

Backward Pass

The C++ extension API currently does not provide a way of automatically generating a backwards function for us. As such, we have to also implement the backward pass of our LLTM, which computes the derivative of the loss with respect to each input of the forward pass. Ultimately, we will plop both the forward and backward function into a `torch.autograd.Function` to create a nice Python binding. The backward function is slightly more involved, so we'll not dig deeper into the code (if you are interested, Alex Graves' thesis is a good read for more information on this):

```
// tanh'(z) = 1 - tanh^2(z)
torch::Tensor d_tanh(torch::Tensor z) {
  return 1 - z.tanh().pow(2);
}

// elu'(z) = relu'(z) + { alpha * exp(z) if (alpha * (exp(z) - 1)) < 0, else 0}
torch::Tensor d_elu(torch::Tensor z, torch::Scalar alpha = 1.0) {
  auto e = z.exp();
  auto mask = (alpha * (e - 1)) < 0;
  return (z > 0).type_as(z) + mask.type_as(z) * (alpha * e);
}

std::vector<torch::Tensor> lltm_backward(
    torch::Tensor grad_h,
    torch::Tensor grad_cell,
    torch::Tensor new_cell,
    torch::Tensor input_gate,
    torch::Tensor output_gate,
    torch::Tensor candidate_cell,
    torch::Tensor X,
    torch::Tensor gate_weights,
    torch::Tensor weights) {
  auto d_output_gate = torch::tanh(new_cell) * grad_h;
  auto d_tanh_new_cell = output_gate * grad_h;
  auto d_new_cell = d_tanh(new_cell) * d_tanh_new_cell + grad_cell;

  auto d_old_cell = d_new_cell;
  auto d_candidate_cell = input_gate * d_new_cell;
  auto d_input_gate = candidate_cell * d_new_cell;

  auto gates = gate_weights.chunk(3, /*dim=*/1);
  d_input_gate *= d_sigmoid(gates[0]);
  d_output_gate *= d_sigmoid(gates[1]);
  d_candidate_cell *= d_elu(gates[2]);

  auto d_gates =
      torch::cat({d_input_gate, d_output_gate, d_candidate_cell}, /*dim=*/1);

  auto d_weights = d_gates.t().mm(X);
  auto d_bias = d_gates.sum(/*dim=*/0, /*keepdim=*/true);

  auto d_X = d_gates.mm(weights);
  const auto state_size = grad_h.size(1);
  auto d_old_h = d_X.slice(/*dim=*/1, 0, state_size);
  auto d_input = d_X.slice(/*dim=*/1, state_size);

  return {d_old_h, d_input, d_weights, d_bias, d_old_cell};
}
```

Binding to Python

Once you have your operation written in C++ and ATen, you can use pybind11 to bind your C++ functions or classes into Python in a very simple manner. Questions or issues you have about this part of PyTorch C++ extensions will largely be addressed by pybind11 documentation.

For our extensions, the necessary binding code spans only four lines:

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("forward", &lltm_forward, "LLTM forward");
  m.def("backward", &lltm_backward, "LLTM backward");
}
```

One bit to note here is the macro `TORCH_EXTENSION_NAME`. The torch extension build will define it as the name you give your extension in the `setup.py` script. In this case, the value of `TORCH_EXTENSION_NAME` would be "lltm_cpp". This is to avoid having to maintain the name of the extension in two places (the build script and your C++ code), as a mismatch between the two can lead to nasty and hard to track issues.

Using Your Extension

We are now set to import our extension in PyTorch. At this point, your directory structure could look something like this:

```
pytorch/
  lltm-extension/
    lltm.cpp
    setup.py
```

Now, run `python setup.py install` to build and install your extension. This should look something like this:

```
running install
running bdist_egg
running egg_info
creating lltm_cpp.egg-info
writing lltm_cpp.egg-info/PKG-INFO
writing dependency_links to lltm_cpp.egg-info/dependency_links.txt
writing top-level names to lltm_cpp.egg-info/top_level.txt
writing manifest file 'lltm_cpp.egg-info/SOURCES.txt'
reading manifest file 'lltm_cpp.egg-info/SOURCES.txt'
writing manifest file 'lltm_cpp.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
running build_ext
building 'lltm_cpp' extension
creating build
creating build/temp.linux-x86_64-3.7
gcc -pthread -B ~/local/miniconda/compiler_compat -Wl,--sysroot=/ -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
prototypes -fPIC -I~/local/miniconda/lib/python3.7/site-packages/torch/include -I~/local/miniconda/lib/python3.7/site-
packages/torch/include/torch/csrc/api/include -I~/local/miniconda/lib/python3.7/site-packages/torch/include/TH -
I~/local/miniconda/lib/python3.7/site-packages/torch/include/THC -I~/local/miniconda/include/python3.7m -c lltm.cpp -o
build/temp.linux-x86_64-3.7/lltm.o -DTORCH_API_INCLUDE_EXTENSION_H -DTORCH_EXTENSION_NAME=lltm_cpp -
D_GLIBCXX_USE_CXX11_ABI=1 -std=c++11
cc1plus: warning: command line option ‘-Wstrict-prototypes’ is valid for C/ObjC but not for C++
creating build/lib.linux-x86_64-3.7
g++ -pthread -shared -B ~/local/miniconda/compiler_compat -L~/local/miniconda/lib -Wl,-rpath=~/local/miniconda/lib -Wl,--
no-as-needed -Wl,--sysroot=/ build/temp.linux-x86_64-3.7/lltm.o -o build/lib.linux-x86_64-3.7/lltm_cpp.cpython-37m-x86_64-
linux-gnu.so
creating build/bdist.linux-x86_64
creating build/bdist.linux-x86_64/egg
copying build/lib.linux-x86_64-3.7/lltm_cpp.cpython-37m-x86_64-linux-gnu.so -> build/bdist.linux-x86_64/egg
creating stub loader for lltm_cpp.cpython-37m-x86_64-linux-gnu.so
byte-compiling build/bdist.linux-x86_64/egg/lltm_cpp.py to lltm_cpp.cpython-37.pyc
creating build/bdist.linux-x86_64/egg/EGG-INFO
copying lltm_cpp.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-INFO
copying lltm_cpp.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying lltm_cpp.egg-info/dependency_links.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying lltm_cpp.egg-info/top_level.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
writing build/bdist.linux-x86_64/egg/EGG-INFO/native_libs.txt
zip_safe flag not set; analyzing archive contents...
__pycache__.lltm_cpp.cpython-37: module references __file__
creating 'dist/lltm_cpp-0.0.0-py3.7-linux-x86_64.egg' and adding 'build/bdist.linux-x86_64/egg' to it
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
Processing lltm_cpp-0.0.0-py3.7-linux-x86_64.egg
removing '~/local/miniconda/lib/python3.7/site-packages/lltm_cpp-0.0.0-py3.7-linux-x86_64.egg' (and everything under it)
creating ~/local/miniconda/lib/python3.7/site-packages/lltm_cpp-0.0.0-py3.7-linux-x86_64.egg
Extracting lltm_cpp-0.0.0-py3.7-linux-x86_64.egg to ~/local/miniconda/lib/python3.7/site-packages
lltm-cpp 0.0.0 is already the active version in easy-install.pth

Installed ~/local/miniconda/lib/python3.7/site-packages/lltm_cpp-0.0.0-py3.7-linux-x86_64.egg
Processing dependencies for lltm-cpp==0.0.0
Finished processing dependencies for lltm-cpp==0.0.0
```

A small note on compilers: Due to ABI versioning issues, the compiler you use to build your C++ extension must be *ABI-compatible* with the compiler PyTorch was built with. In practice, this means that you must use GCC version 4.9 and above on Linux. For Ubuntu 16.04 and other more-recent Linux distributions, this should be the default compiler already. On MacOS, you must use clang (which does not have any ABI versioning issues). In the worst case, you can build PyTorch from source with your compiler and then build the extension with that same compiler.

Once your extension is built, you can simply import it in Python, using the name you specified in your `setup.py` script. Just be sure to `import torch` first, as this will resolve some symbols that the dynamic linker must see:

```
In [1]: import torch
In [2]: import lltm_cpp
In [3]: lltm_cpp.forward
Out[3]: <function lltm.PyCapsule.forward>
```

If we call `help()` on the function or module, we can see that its signature matches our C++ code:

```
In[4] help(lltm_cpp.forward)
forward(...) method of builtins.PyCapsule instance
    forward(arg0: torch::Tensor, arg1: torch::Tensor, arg2: torch::Tensor, arg3: torch::Tensor, arg4: torch::Tensor) ->
List[torch::Tensor]

    LLTM forward
```

Since we are now able to call our C++ functions from Python, we can wrap them with `torch.autograd.Function` and `torch.nn.Module` to make them first class citizens of PyTorch:

```
import math
import torch

# Our module!
import lltm_cpp

class LLTMFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weights, bias, old_h, old_cell):
        outputs = lltm_cpp.forward(input, weights, bias, old_h, old_cell)
        new_h, new_cell = outputs[:2]
        variables = outputs[1:] + [weights]
        ctx.save_for_backward(*variables)

        return new_h, new_cell

    @staticmethod
    def backward(ctx, grad_h, grad_cell):
        outputs = lltm_cpp.backward(
            grad_h.contiguous(), grad_cell.contiguous(), *ctx.saved_tensors)
        d_old_h, d_input, d_weights, d_bias, d_old_cell = outputs
        return d_input, d_weights, d_bias, d_old_h, d_old_cell


class LLTM(torch.nn.Module):
    def __init__(self, input_features, state_size):
        super(LLTM, self).__init__()
        self.input_features = input_features
        self.state_size = state_size
        self.weights = torch.nn.Parameter(
            torch.empty(3 * state_size, input_features + state_size))
        self.bias = torch.nn.Parameter(torch.empty(3 * state_size))
        self.reset_parameters()

    def reset_parameters(self):
        stdv = 1.0 / math.sqrt(self.state_size)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, +stdv)

    def forward(self, input, state):
        return LLTMFunction.apply(input, self.weights, self.bias, *state)
```

Performance Comparison

Now that we are able to use and call our C++ code from PyTorch, we can run a small benchmark to see how much performance we gained from rewriting our op in C++. We'll run the LLTM forwards and backwards a few times and measure the duration:

```
import time

import torch

batch_size = 16
input_features = 32
state_size = 128

X = torch.randn(batch_size, input_features)
h = torch.randn(batch_size, state_size)
C = torch.randn(batch_size, state_size)

rnn = LLTM(input_features, state_size)

forward = 0
backward = 0
for _ in range(100000):
    start = time.time()
    new_h, new_C = rnn(X, (h, C))
    forward += time.time() - start

    start = time.time()
    (new_h.sum() + new_C.sum()).backward()
    backward += time.time() - start

print('Forward: {:.3f} s | Backward {:.3f} s'.format(forward, backward))
```

If we run this code with the original LLTM we wrote in pure Python at the start of this post, we get the following numbers (on my machine):

```
Forward: 506.480 us | Backward 444.694 us
```

and with our new C++ version:

```
Forward: 349.335 us | Backward 443.523 us
```

We can already see a significant speedup for the forward function (more than 30%). For the backward function, a speedup is visible, albeit not a major one. The backward pass I wrote above was not particularly optimized and could definitely be improved. Also, PyTorch's automatic differentiation engine can automatically parallelize computation graphs, may use a more efficient flow of operations overall, and is also implemented in C++, so it's expected to be fast. Nevertheless, this is a good start.

Performance on GPU Devices

A wonderful fact about PyTorch's *ATen* backend is that it abstracts the computing device you are running on. This means the same code we wrote for CPU can *also* run on GPU, and individual operations will correspondingly dispatch to GPU-optimized implementations. For certain operations like matrix multiply (like `mm` or `addmm`), this is a big win. Let's take a look at how much performance we gain from running our C++ code with CUDA tensors. No changes to our implementation are required, we simply need to put our tensors in GPU memory from Python, with either adding `device=cuda_device` argument at creation time or using `.to(cuda_device)` after creation:

```
import torch

assert torch.cuda.is_available()
cuda_device = torch.device("cuda")  # device object representing GPU

batch_size = 16
input_features = 32
state_size = 128

# Note the device=cuda_device arguments here
X = torch.randn(batch_size, input_features, device=cuda_device)
h = torch.randn(batch_size, state_size, device=cuda_device)
C = torch.randn(batch_size, state_size, device=cuda_device)

rnn = LLTM(input_features, state_size).to(cuda_device)

forward = 0
backward = 0
for _ in range(100000):
    start = time.time()
    new_h, new_C = rnn(X, (h, C))
    torch.cuda.synchronize()
    forward += time.time() - start

    start = time.time()
    (new_h.sum() + new_C.sum()).backward()
    torch.cuda.synchronize()
    backward += time.time() - start

print('Forward: {:.3f} us | Backward {:.3f} us'.format(forward * 1e6/1e5, backward * 1e6/1e5))
```

Once more comparing our plain PyTorch code with our C++ version, now both running on CUDA devices, we again see performance gains. For Python/PyTorch:

```
Forward: 187.719 us | Backward 410.815 us
```

And C++/ATen:

```
Forward: 149.802 us | Backward 393.458 us
```

That's a great overall speedup compared to non-CUDA code. However, we can pull even more performance out of our C++ code by writing custom CUDA kernels, which we'll dive into soon. Before that, let's discuss another way of building your C++ extensions.

JIT Compiling Extensions

Previously, I mentioned there were two ways of building C++ extensions: using `setuptools` or just in time (JIT). Having covered the former, let's elaborate on the latter. The JIT compilation mechanism provides you with a way of compiling and loading your extensions on the fly by calling a simple function in PyTorch's API called `torch.utils.cpp_extension.load()`. For the LLTM, this would look as simple as this:

```
from torch.utils.cpp_extension import load

lltm_cpp = load(name="lltm_cpp", sources=["lltm.cpp"])
```

Here, we provide the function with the same information as for `setuptools`. In the background, this will do the following:

1. Create a temporary directory `/tmp/torch_extensions/lltm`,
2. Emit a Ninja build file into that temporary directory,

3. Compile your source files into a shared library,

4. Import this shared library as a Python module.

In fact, if you pass `verbose=True` to `cpp_extension.load()`, you will be informed about the process:

```
Using /tmp/torch_extensions as PyTorch extensions root...
Emitting ninja build file /tmp/torch_extensions/lltm_cpp/build.ninja...
Building extension module lltm_cpp...
Loading extension module lltm_cpp...
```

The resulting Python module will be exactly the same as produced by setuptools, but removes the requirement of having to maintain a separate `setup.py` build file. If your setup is more complicated and you do need the full power of `setuptools`, you *can* write your own `setup.py` – but in many cases this JIT technique will do just fine. The first time you run through this line, it will take some time, as the extension is compiling in the background. Since we use the Ninja build system to build your sources, re-compilation is incremental and thus re-loading the extension when you run your Python module a second time is fast and has low overhead if you didn't change the extension's source files.

Writing a Mixed C++/CUDA extension

To really take our implementation to the next level, we can hand-write parts of our forward and backward passes with custom CUDA kernels. For the LLTM, this has the prospect of being particularly effective, as there are a large number of pointwise operations in sequence, that can all be fused and parallelized in a single CUDA kernel. Let's see how we could write such a CUDA kernel and integrate it with PyTorch using this extension mechanism.

The general strategy for writing a CUDA extension is to first write a C++ file which defines the functions that will be called from Python, and binds those functions to Python with pybind11. Furthermore, this file will also *declare* functions that are defined in CUDA (`.cu`) files. The C++ functions will then do some checks and ultimately forward its calls to the CUDA functions. In the CUDA files, we write our actual CUDA kernels. The `cpp_extension` package will then take care of compiling the C++ sources with a C++ compiler like `gcc` and the CUDA sources with NVIDIA's `nvcc` compiler. This ensures that each compiler takes care of files it knows best to compile. Ultimately, they will be linked into one shared library that is available to us from Python code.

We'll start with the C++ file, which we'll call `lltm_cuda.cpp`, for example:

```
#include <torch/extension.h>

#include <vector>

// CUDA forward declarations

std::vector<torch::Tensor> lltm_cuda_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell);

std::vector<torch::Tensor> lltm_cuda_backward(
    torch::Tensor grad_h,
    torch::Tensor grad_cell,
    torch::Tensor new_cell,
    torch::Tensor input_gate,
    torch::Tensor output_gate,
    torch::Tensor candidate_cell,
    torch::Tensor X,
    torch::Tensor gate_weights,
    torch::Tensor weights);

// C++ interface

#define CHECK_CUDA(x) TORCH_CHECK(x.device().is_cuda(), #x " must be a CUDA tensor")
#define CHECK_CONTIGUOUS(x) TORCH_CHECK(x.is_contiguous(), #x " must be contiguous")
#define CHECK_INPUT(x) CHECK_CUDA(x); CHECK_CONTIGUOUS(x)

std::vector<torch::Tensor> lltm_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  CHECK_INPUT(input);
  CHECK_INPUT(weights);
  CHECK_INPUT(bias);
  CHECK_INPUT(old_h);
  CHECK_INPUT(old_cell);

  return lltm_cuda_forward(input, weights, bias, old_h, old_cell);
}

std::vector<torch::Tensor> lltm_backward(
    torch::Tensor grad_h,
    torch::Tensor grad_cell,
    torch::Tensor new_cell,
    torch::Tensor input_gate,
    torch::Tensor output_gate,
    torch::Tensor candidate_cell,
    torch::Tensor X,
    torch::Tensor gate_weights,
    torch::Tensor weights) {
  CHECK_INPUT(grad_h);
  CHECK_INPUT(grad_cell);
  CHECK_INPUT(input_gate);
  CHECK_INPUT(output_gate);
  CHECK_INPUT(candidate_cell);
  CHECK_INPUT(X);
  CHECK_INPUT(gate_weights);
  CHECK_INPUT(weights);

  return lltm_cuda_backward(
      grad_h,
      grad_cell,
      new_cell,
      input_gate,
      output_gate,
      candidate_cell,
      X,
      gate_weights,
      weights);
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("forward", &lltm_forward, "LLTM forward (CUDA)");
  m.def("backward", &lltm_backward, "LLTM backward (CUDA)");
}
```

As you can see, it is largely boilerplate, checks and forwarding to functions that we'll define in the CUDA file. We'll name this file `lltm_cuda_kernel.cu` (note the `.cu` extension!). NVCC can reasonably compile C++11, thus we still have ATen and the C++ standard library available to us (but not `torch.h`). Note that `setuptools` cannot handle files with the same name but different extensions, so if you use the `setup.py` method instead of the JIT method, you must give your CUDA file a different name than your C++ file (for the JIT method, `lltm.cpp` and `lltm.cu` would work fine). Let's take a small peek at what this file will look like:

```
#include <torch/extension.h>

#include <cuda.h>
#include <cuda_runtime.h>

#include <vector>

template <typename scalar_t>
__device__ __forceinline__ scalar_t sigmoid(scalar_t z) {
  return 1.0 / (1.0 + exp(-z));
}
```

Here we see the headers I just described, as well as the fact that we are using CUDA-specific declarations like `__device__` and `__forceinline__` and functions like `exp`. Let's continue with a few more helper functions that we'll need:

```
template <typename scalar_t>
__device__ __forceinline__ scalar_t d_sigmoid(scalar_t z) {
  const auto s = sigmoid(z);
  return (1.0 - s) * s;
}

template <typename scalar_t>
__device__ __forceinline__ scalar_t d_tanh(scalar_t z) {
  const auto t = tanh(z);
  return 1 - (t * t);
}

template <typename scalar_t>
__device__ __forceinline__ scalar_t elu(scalar_t z, scalar_t alpha = 1.0) {
  return fmax(0.0, z) + fmin(0.0, alpha * (exp(z) - 1.0));
}

template <typename scalar_t>
__device__ __forceinline__ scalar_t d_elu(scalar_t z, scalar_t alpha = 1.0) {
  const auto e = exp(z);
  const auto d_relu = z < 0.0 ? 0.0 : 1.0;
  return d_relu + (((alpha * (e - 1.0)) < 0.0) ? (alpha * e) : 0.0);
}
```

To now actually implement a function, we'll again need two things: one function that performs operations we don't wish to explicitly write by hand and calls into CUDA kernels, and then the actual CUDA kernel for the parts we want to speed up. For the forward pass, the first function should look like this:

```
std::vector<torch::Tensor> lltm_cuda_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  auto X = torch::cat({old_h, input}, /*dim=*/1);
  auto gates = torch::addmm(bias, X, weights.transpose(0, 1));

  const auto batch_size = old_cell.size(0);
  const auto state_size = old_cell.size(1);

  auto new_h = torch::zeros_like(old_cell);
  auto new_cell = torch::zeros_like(old_cell);
  auto input_gate = torch::zeros_like(old_cell);
  auto output_gate = torch::zeros_like(old_cell);
  auto candidate_cell = torch::zeros_like(old_cell);

  const int threads = 1024;
  const dim3 blocks((state_size + threads - 1) / threads, batch_size);

  AT_DISPATCH_FLOATING_TYPES(gates.type(), "lltm_forward_cuda", ([&] {
    lltm_cuda_forward_kernel<scalar_t><<<blocks, threads>>>(
        gates.data<scalar_t>(),
        old_cell.data<scalar_t>(),
        new_h.data<scalar_t>(),
        new_cell.data<scalar_t>(),
        input_gate.data<scalar_t>(),
        output_gate.data<scalar_t>(),
        candidate_cell.data<scalar_t>(),
        state_size);
  }));

  return {new_h, new_cell, input_gate, output_gate, candidate_cell, X, gates};
}
```

The main point of interest here is the `AT_DISPATCH_FLOATING_TYPES` macro and the kernel launch (indicated by the `<<<...>>>`). While ATen abstracts away the device and datatype of the tensors we deal with, a tensor will, at runtime, still be backed by memory of a concrete type on a concrete device. As such, we need a way of determining at runtime what type a tensor is and then selectively call functions with the corresponding correct type signature. Done manually, this would (conceptually) look something like this:

```
switch (tensor.type().scalarType()) {
  case torch::ScalarType::Double:
    return function<double>(tensor.data<double>());
  case torch::ScalarType::Float:
    return function<float>(tensor.data<float>());
  ...
}
```

The purpose of `AT_DISPATCH_FLOATING_TYPES` is to take care of this dispatch for us. It takes a type (`gates.type()` in our case), a name (for error messages) and a lambda function. Inside this lambda function, the type alias `scalar_t` is available and is defined as the type that the tensor actually is at runtime in that context. As such, if we have a template function (which our CUDA kernel will be), we can instantiate it with this `scalar_t` alias, and the correct function will be called. In this case, we also want to retrieve the data pointers of the tensors as pointers of that `scalar_t` type. If you wanted to dispatch over all types and not just floating point types (`Float` and `Double`), you can use `AT_DISPATCH_ALL_TYPES`.

Note that we perform some operations with plain ATen. These operations will still run on the GPU, but using ATen's default implementations. This makes sense because ATen will use highly optimized routines for things like matrix multiplies (e.g. `addmm`) or convolutions which would be much harder to implement and improve ourselves.

As for the kernel launch itself, we are here specifying that each CUDA block will have 1024 threads, and that the entire GPU grid is split into as many blocks of `1 x 1024` threads as are required to fill our matrices with one thread per component. For example, if our state size was 2048 and our batch size 4, we'd launch a total of `4 x 2 = 8` blocks with each 1024 threads. If you've never heard of CUDA "blocks" or "grids" before, an introductory read about CUDA may help.

The actual CUDA kernel is fairly simple (if you've ever programmed GPUs before):

```
template <typename scalar_t>
__global__ void lltm_cuda_forward_kernel(
    const scalar_t* __restrict__ gates,
    const scalar_t* __restrict__ old_cell,
    scalar_t* __restrict__ new_h,
    scalar_t* __restrict__ new_cell,
    scalar_t* __restrict__ input_gate,
    scalar_t* __restrict__ output_gate,
    scalar_t* __restrict__ candidate_cell,
    size_t state_size) {
  const int column = blockIdx.x * blockDim.x + threadIdx.x;
  const int index = blockIdx.y * state_size + column;
  const int gates_row = blockIdx.y * (state_size * 3);
  if (column < state_size) {
    input_gate[index] = sigmoid(gates[gates_row + column]);
    output_gate[index] = sigmoid(gates[gates_row + state_size + column]);
    candidate_cell[index] = elu(gates[gates_row + 2 * state_size + column]);
    new_cell[index] =
        old_cell[index] + candidate_cell[index] * input_gate[index];
    new_h[index] = tanh(new_cell[index]) * output_gate[index];
  }
}
```

What's primarily interesting here is that we are able to compute all of these pointwise operations entirely in parallel for each individual component in our gate matrices. If you imagine having to do this with a giant `for` loop over a million elements in serial, you can see why this would be much faster.

Using accessors

You can see in the CUDA kernel that we work directly on pointers with the right type. Indeed, working directly with high level type agnostic tensors inside cuda kernels would be very inefficient.

However, this comes at a cost of ease of use and readability, especially for highly dimensional data. In our example, we know for example that the contiguous `gates` tensor has 3 dimensions:

1. batch, size of `batch_size` and stride of `3*state_size`
2. row, size of `3` and stride of `state_size`
3. index, size of `state_size` and stride of `1`

How can we access the element `gates[n][row][column]` inside the kernel then? It turns out that you need the strides to access your element with some simple arithmetic.

```
gates.data<scalar_t>()[n*3*state_size + row*state_size + column]
```

In addition to being verbose, this expression needs stride to be explicitly known, and thus passed to the kernel function within its arguments. You can see that in the case of kernel functions accepting multiple tensors with different sizes you will end up with a very long list of arguments.

Fortunately for us, ATen provides accessors that are created with a single dynamic check that a Tensor is the type and number of dimensions. Accessors then expose an API for accessing the Tensor elements efficiently without having to convert to a single pointer:

```
torch::Tensor foo = torch::rand({12, 12});

// assert foo is 2-dimensional and holds floats.
auto foo_a = foo.accessor<float,2>();
float trace = 0;

for(int i = 0; i < foo_a.size(0); i++) {
  // use the accessor foo_a to get tensor data.
  trace += foo_a[i][i];
}
```

Accessor objects have a relatively high level interface, with `.size()` and `.stride()` methods and multi-dimensional indexing. The `.accessor<>` interface is designed to access data efficiently on cpu tensor. The equivalent for cuda tensors are `packed_accessor64<>` and `packed_accessor32<>` , which produce Packed Accessors with either 64-bit or 32-bit integer indexing.

The fundamental difference with Accessor is that a Packed Accessor copies size and stride data inside of its structure instead of pointing to it. It allows us to pass it to a CUDA kernel function and use its interface inside it.

We can design a function that takes Packed Accessors instead of pointers.

```
__global__ void lltm_cuda_forward_kernel(
    const torch::PackedTensorAccessor32<scalar_t,3,torch::RestrictPtrTraits> gates,
    const torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> old_cell,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> new_h,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> new_cell,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> input_gate,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> output_gate,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> candidate_cell)
```

Let's decompose the template used here. the first two arguments `scalar_t` and `2` are the same as regular Accessor. The argument `torch::RestrictPtrTraits` indicates that the `__restrict__` keyword must be used. Note also that we've used the `PackedAccessor32` variant which store the sizes and strides in an `int32_t`. This is important as using the 64-bit variant (`PackedAccessor64`) can make the kernel slower.

The function declaration becomes

```
template <typename scalar_t>
__global__ void lltm_cuda_forward_kernel(
    const torch::PackedTensorAccessor32<scalar_t,3,torch::RestrictPtrTraits> gates,
    const torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> old_cell,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> new_h,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> new_cell,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> input_gate,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> output_gate,
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> candidate_cell) {
  //batch index
  const int n = blockIdx.y;
  // column index
  const int c = blockIdx.x * blockDim.x + threadIdx.x;
  if (c < gates.size(2)){
    input_gate[n][c] = sigmoid(gates[n][0][c]);
    output_gate[n][c] = sigmoid(gates[n][1][c]);
    candidate_cell[n][c] = elu(gates[n][2][c]);
    new_cell[n][c] =
        old_cell[n][c] + candidate_cell[n][c] * input_gate[n][c];
    new_h[n][c] = tanh(new_cell[n][c]) * output_gate[n][c];
  }
}
```

The implementation is much more readable! This function is then called by creating Packed Accessors with the `.packed_accessor32<>` method within the host function.

```
std::vector<torch::Tensor> lltm_cuda_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  auto X = torch::cat({old_h, input}, /*dim=*/1);
  auto gate_weights = torch::addmm(bias, X, weights.transpose(0, 1));

  const auto batch_size = old_cell.size(0);
  const auto state_size = old_cell.size(1);

  auto gates = gate_weights.reshape({batch_size, 3, state_size});
  auto new_h = torch::zeros_like(old_cell);
  auto new_cell = torch::zeros_like(old_cell);
  auto input_gate = torch::zeros_like(old_cell);
  auto output_gate = torch::zeros_like(old_cell);
  auto candidate_cell = torch::zeros_like(old_cell);

  const int threads = 1024;
  const dim3 blocks((state_size + threads - 1) / threads, batch_size);

  AT_DISPATCH_FLOATING_TYPES(gates.type(), "lltm_forward_cuda", ([&] {
    lltm_cuda_forward_kernel<scalar_t><<<blocks, threads>>>(
        gates.packed_accessor32<scalar_t,3,torch::RestrictPtrTraits>(),
        old_cell.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        new_h.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        new_cell.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        input_gate.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        output_gate.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        candidate_cell.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>());
  }));

  return {new_h, new_cell, input_gate, output_gate, candidate_cell, X, gates};
}
```

The backwards pass follows much the same pattern and I won't elaborate further on it:

```
template <typename scalar_t>
__global__ void lltm_cuda_backward_kernel(
    torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> d_old_cell,
    torch::PackedTensorAccessor32<scalar_t,3,torch::RestrictPtrTraits> d_gates,
    const torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> grad_h,
    const torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> grad_cell,
    const torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> new_cell,
    const torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> input_gate,
    const torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> output_gate,
    const torch::PackedTensorAccessor32<scalar_t,2,torch::RestrictPtrTraits> candidate_cell,
    const torch::PackedTensorAccessor32<scalar_t,3,torch::RestrictPtrTraits> gate_weights) {
  //batch index
  const int n = blockIdx.y;
  // column index
  const int c = blockIdx.x * blockDim.x + threadIdx.x;
  if (c < d_gates.size(2)){
    const auto d_output_gate = tanh(new_cell[n][c]) * grad_h[n][c];
    const auto d_tanh_new_cell = output_gate[n][c] * grad_h[n][c];
    const auto d_new_cell =
        d_tanh(new_cell[n][c]) * d_tanh_new_cell + grad_cell[n][c];


    d_old_cell[n][c] = d_new_cell;
    const auto d_candidate_cell = input_gate[n][c] * d_new_cell;
    const auto d_input_gate = candidate_cell[n][c] * d_new_cell;

    d_gates[n][0][c] =
        d_input_gate * d_sigmoid(gate_weights[n][0][c]);
    d_gates[n][1][c] =
        d_output_gate * d_sigmoid(gate_weights[n][1][c]);
    d_gates[n][2][c] =
        d_candidate_cell * d_elu(gate_weights[n][2][c]);
  }
}

std::vector<torch::Tensor> lltm_cuda_backward(
    torch::Tensor grad_h,
    torch::Tensor grad_cell,
    torch::Tensor new_cell,
    torch::Tensor input_gate,
    torch::Tensor output_gate,
    torch::Tensor candidate_cell,
    torch::Tensor X,
    torch::Tensor gates,
    torch::Tensor weights) {
  auto d_old_cell = torch::zeros_like(new_cell);
  auto d_gates = torch::zeros_like(gates);

  const auto batch_size = new_cell.size(0);
  const auto state_size = new_cell.size(1);

  const int threads = 1024;
  const dim3 blocks((state_size + threads - 1) / threads, batch_size);

  AT_DISPATCH_FLOATING_TYPES(X.type(), "lltm_backward_cuda", ([&] {
    lltm_cuda_backward_kernel<scalar_t><<<blocks, threads>>>(
        d_old_cell.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        d_gates.packed_accessor32<scalar_t,3,torch::RestrictPtrTraits>(),
        grad_h.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        grad_cell.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        new_cell.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        input_gate.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        output_gate.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        candidate_cell.packed_accessor32<scalar_t,2,torch::RestrictPtrTraits>(),
        gates.packed_accessor32<scalar_t,3,torch::RestrictPtrTraits>());
  }));

  auto d_gate_weights = d_gates.reshape({batch_size, 3*state_size});
  auto d_weights = d_gate_weights.t().mm(X);
  auto d_bias = d_gate_weights.sum(/*dim=*/0, /*keepdim=*/true);

  auto d_X = d_gate_weights.mm(weights);
  auto d_old_h = d_X.slice(/*dim=*/1, 0, state_size);
  auto d_input = d_X.slice(/*dim=*/1, state_size);

  return {d_old_h, d_input, d_weights, d_bias, d_old_cell, d_gates};
}
```

Integrating a C++/CUDA Operation with PyTorch

Integration of our CUDA-enabled op with PyTorch is again very straightforward. If you want to write a `setup.py` script, it could look like this:

```
from setuptools import setup
from torch.utils.cpp_extension import BuildExtension, CUDAExtension

setup(
    name='lltm',
    ext_modules=[
        CUDAExtension('lltm_cuda', [
            'lltm_cuda.cpp',
            'lltm_cuda_kernel.cu',
        ])
    ],
    cmdclass={
        'build_ext': BuildExtension
    })
```

Instead of `CppExtension()`, we now use `CUDAExtension()`. We can just specify the `.cu` file along with the `.cpp` files – the library takes care of all the hassle this entails for you. The JIT mechanism is even simpler:

```
from torch.utils.cpp_extension import load

lltm = load(name='lltm', sources=['lltm_cuda.cpp', 'lltm_cuda_kernel.cu'])
```

Performance Comparison

Our hope was that parallelizing and fusing the pointwise operations of our code with CUDA would improve the performance of our LLTM. Let's see if that holds true. We can run the code I listed earlier to run a benchmark. Our fastest version earlier was the CUDA-based C++ code:

```
Forward: 149.802 us | Backward 393.458 us
```

And now with our custom CUDA kernel:

```
Forward: 129.431 us | Backward 304.641 us
```

More performance increases!

Conclusion

You should now be equipped with a good overview of PyTorch's C++ extension mechanism as well as a motivation for using them. You can find the code examples displayed in this note here. If you have questions, please use the forums. Also be sure to check our FAQ in case you run into any issues.

Previous Next

Contributing Brand Guidelines

Terms | Privacy