Custom C++ and CUDA Extensions

pytorch.apachecn.org/docs/1.0/cpp_extension.html

自定义 C++ 与 CUDA 拓展

> 译者：P3n9W31

Author: Peter Goldsborough

PyTorch 提供了大量与神经网络，任意张量代数(arbitrary tensor algebra），数据处理(data wrangling）和其他目的相关的操作。然而，你可能发现你还是会需要一些更加自定义的操作。例如，你有时可能希望使用一个你在某篇论文中找到的一个新型的激活函数，或者是实现一个为了你的研究所开发的新操作。

在 PyTorch 中集成这种自定义操作的最简单方法是通过 Python 语言对 `Function` 和 `Module` 进行扩写，正如在 这里所描述的那样。这种方式能让你充分的发挥自动微分(automatic differentiation）(让你不用去编写一些衍生的函数）与 Python 语言的常规情况下的表现力(usual expressiveness）的能力。但是有时候，可能在 C++ 语言中能够更好地实现你的一些操作。例如，你的代码可能因为被非常频繁的使用而需要 *十分* 快速，或者是即使调用的次数很少也会带来不小的性能负担。另一个原因是你的代码可能是建立在 C 或 C++ 语言之上的，或者你的代码需要与 C 或 C++ 语言进行交互与对接。为了解决上述的这些情况，PyTorch 提供了一种简单的用于编写自定义 *C++ 扩展* 的方法。

C++ 拓展是我们开发的一种能够让用户(你）自行创建一些 *所含资源之外* 的操作的机制，例如，与 PyTorch 的后端分离开来。这种方法与 PyTorch 原生操作的实现方式是 *不同的* 。C++ 扩展旨在为你提供与 PyTorch 后端集成操作相关的大部分样板(boilerplate），同时为基于 PyTorch 的项目提供高度灵活性。然而，一旦你将你的操作定义为了 C++ 拓展，将其转换为原生 PyTorch 函数就主要是代码组织的问题了，如果你决定在上游提供操作，则可以解决这个问题。

动机与例子

本篇文章的剩余部分将介绍一个编写和使用 C++(以及 CUDA）拓展的实例。如果你现在正在被一直催着或是在今天之前没有把该操作完成你就会被解雇的话，你可以跳过这一章节，直接去下一节的实施细节部分查看。

假设你已经找到了一种新型的循环(recurrent）的单元，它与现有技术相比具有优越的性能。该循环单元类似于 LSTM，但不同之处在于它缺少了 *遗忘门* 并使用 *指数线性单元* (ELU）作为其内部激活功能。因为这个单元永远都不会忘记，所以我们叫它 *LLTM*，或是 *长长期记忆* (Long-Long-Term-Memory）单元。

在 LLTMs 中的这两个与普通的 LSTMs 的不同点是十分重要的，以至于我们不能通过配置 PyTorch 中的 `LSTMCell` 来达到我们的目标。所以我们将只能创建一个自定义模块。第一个也是最简单的方法 - 可能在所有情况下都是良好的第一步——是使用 Python 在纯

PyTorch 中实现我们所需的功能。为此，我们需要继承 `torch.nn.Module` 并实现 LLTM 的正向传递。 这看起来就像这样：

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


        gate_weights = F.linear(X, self.weights, self.bias)

        gates = gate_weights.chunk(3, dim=1)

        input_gate = F.sigmoid(gates[0])
        output_gate = F.sigmoid(gates[1])

        candidate_cell = F.elu(gates[2])


        new_cell = old_cell + candidate_cell * input_gate

        new_h = F.tanh(new_cell) * output_gate

        return new_h, new_cell
```

我们可以按预期使用它：

```
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))
```

当然，如果可能的话，你应该使用这种方法来扩展 PyTorch。由于 PyTorch 对 CPU *与* GPU 的操作实施了高度优化，由 NVIDIA cuDNN，Intel MKL 或是 NNPACK 等库提供了支持，像上面那样的 PyTorch 代码一般情况下都是足够快速的。但是，我们也可以看到为什么在某些情况下还有进一步改进性能的空间。最明显的原因是PyTorch不了解你正在实施的 *算法* 。它只知道你用于编写算法的各个独立操作。因此，PyTorch 必须逐个执行你的操作。由于对操作的实现(或 *核*)的每次单独的调用都可能(可能涉及启动 CUDA 内核）具有一定量的开销，因此这种开销可能在许多函数的调用中变得显着。此外，运行我们的代码的 Python 解释器本身就可以减慢我们的程序。

因此，一个明显可以加快速度的方法是用 C++(或 CUDA）完成部分代码的重写部分并*融合*特定的操作组。融合意味着将许多函数的实现组合到单个函数中，这些函数会从更少的内核启动中受益，此外，这些函数还会从我们通过提高全局数据流的可见性来执行的其他优化中获益。

让我们来看看我们可以怎样使用 C++ 拓展来实现一个*融合*版本的 LLTM。我们首先使用纯 C++ 完成代码编写，使用驱动了大部分 PyTorch 后端的 ATen 库，并看看它能让我们多简单就完成 Python 代码的转换。然后我们将通过将一部分的模型移动到 CUDA 内核以从 GPU 提供的大规模并行性中受益，来进一步加快速度。

编写一个 C++ 拓展

C++ 扩展有两种形式：它们可以使用 `setuptools` 来进行“提前”构建，或者通过 `torch.utils.cpp_extension.load()` 来实现“实时”构建。我们将从第一种方法开始，稍后再讨论后者。

使用 `setuptools` 进行构建

对于"提前"这种形式，我们通过编写一个 `setup.py` 脚本来构建我们的 C++ 扩展，该脚本使用 setuptools 来编译我们的 C++ 代码。 对于 LLTM 而言，它看起来就像下面这样简单：

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

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

在这段代码中， `CppExtension` 是 `setuptools.Extension` 的一个便利的包装器(wrapper），它传递正确的包含路径并将扩展语言设置为 C++。 等效的普通 `setuptools` 代码像下面这样简单：

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

`BuildExtension` 执行许多必需的配置步骤和检查，并在混合 C++/CUDA 扩展的情况下管理混合编译。 这就是我们现在真正需要了解的关于构建 C++ 扩展的所有内容！现在让我们来看看我们的 C++ 扩展的实现，它扩展到了 `lltm.cpp` 中。

编写 C++ 操作

让我们开始用 C++ 实现 LLTM！我们向后传递所需的一个函数是 sigmoid 的导数。这是一段足够小的代码，用于讨论编写 C++ 扩展时可用的整体环境：

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

`torch / torch.h` 是一站式(one-stop）头文件，包含编写 C++ 扩展所需的所有 PyTorch 位。 这包括：

- ATen 库，我们主要的张量计算接口
- pybind11，我们为 C++ 代码创建 Python 绑定的方法
- 管理 ATen 和 pybind11 之间交互细节的头文件。

`d_sigmoid()` 的实现显示了如何使用 ATen API。PyTorch 的张量和变量接口是从 ATen 库自动生成的，因此我们可以或多或少地将我们的 Python 语言实现1:1转换为 C++ 语言实现。 我们所有计算的主要数据类型都是 `at::Tensor` 。可以在此处查看其完整的 API。另请注意，我们可以引用 `<iostream>` 或任何其他 C 或 C++ 头文件——我们可以使用 C++ 11 的全部功能。

前向传播

接下来，我们可以将整个前向传播部分移植为 C++ 代码：

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

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

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

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

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

反向传播

C++ 的扩展 API 目前不为我们提供自动生成反向函数的方法。因此，我们还必须实施 LLTM 的反向传播部分，LLTM 计算相对于正向传播的每个输入的损失的导数。最终，我们将向前和向后函数放入 `torch.autograd.Function` 以创建一个漂亮的 Python 绑定。向后功能稍微复杂一些，所以我们不会深入研究代码(如果你感兴趣，Alex Graves的论文是一个能让你了解跟多信息的好文章）：

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

// elu'(z) = relu'(z) + { alpha * exp(z) if (alpha * (exp(z) - 1)) < 0, else 0}
at::Tensor d_elu(at::Tensor z, at::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<at::Tensor> lltm_backward(
    at::Tensor grad_h,
    at::Tensor grad_cell,
    at::Tensor new_cell,
    at::Tensor input_gate,
    at::Tensor output_gate,
    at::Tensor candidate_cell,
    at::Tensor X,
    at::Tensor gate_weights,
    at::Tensor weights) {
  auto d_output_gate = at::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 =
      at::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};
}
```

与Python绑定

一旦你使用 C++ 和 ATen 编写了操作，就可以使用 pybind11 以非常简单的方式将 C++ 函数或类绑定到 Python 上。关于 PyTorch 的 C++ 扩展的这一部分的问题或疑问将主要通过pybind11文档来解决。

对于我们的扩展，必要的绑定代码只是仅仅四行：

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

有一点需要注意的是宏 `TORCH_EXTENSION_NAME` 。torch 的扩展部分将会把它定义为你在 `setup.py` 脚本中为扩展名命名的名称。在这种情况下， `TORCH_EXTENSION_NAME` 的值将为“lltm”。这是为了避免必须在两个地方(构建脚本和 C++ 代码中）维护扩展名，因为两者之间的不匹配可能会导致令人讨厌且难以跟踪的问题。

使用你的拓展

我们现在设置为 PyTorch 导入我们的扩展。 此时，你的目录结构可能如下所示：

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

现在，运行 `python setup.py install` 来构建和安装你的扩展。 运行结果应该是这样的：

```
running install
running bdist_egg
running egg_info
writing lltm.egg-info/PKG-INFO
writing dependency_links to lltm.egg-info/dependency_links.txt
writing top-level names to lltm.egg-info/top_level.txt
reading manifest file 'lltm.egg-info/SOURCES.txt'
writing manifest file 'lltm.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
running build_ext
building 'lltm' extension
gcc -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -fPIC -
I~/local/miniconda/lib/python3.6/site-packages/torch/lib/include -
I~/local/miniconda/lib/python3.6/site-packages/torch/lib/include/TH -
I~/local/miniconda/lib/python3.6/site-packages/torch/lib/include/THC -
I~/local/miniconda/include/python3.6m -c lltm.cpp -o build/temp.linux-x86_64-3.6/lltm.o -
DTORCH_EXTENSION_NAME=lltm -std=c++11
cc1plus: warning: command line option '-Wstrict-prototypes' is valid for C/ObjC but not for
C++
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-
3.6/lltm.o -o build/lib.linux-x86_64-3.6/lltm.cpython-36m-x86_64-linux-gnu.so
creating build/bdist.linux-x86_64/egg
copying build/lib.linux-x86_64-3.6/lltm_cuda.cpython-36m-x86_64-linux-gnu.so ->
build/bdist.linux-x86_64/egg
copying build/lib.linux-x86_64-3.6/lltm.cpython-36m-x86_64-linux-gnu.so -> build/bdist.linux-
x86_64/egg
creating stub loader for lltm.cpython-36m-x86_64-linux-gnu.so
byte-compiling build/bdist.linux-x86_64/egg/lltm.py to lltm.cpython-36.pyc
creating build/bdist.linux-x86_64/egg/EGG-INFO
copying lltm.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-INFO
copying lltm.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying lltm.egg-info/dependency_links.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying lltm.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.cpython-36: module references __file__
creating 'dist/lltm-0.0.0-py3.6-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-0.0.0-py3.6-linux-x86_64.egg
removing '~/local/miniconda/lib/python3.6/site-packages/lltm-0.0.0-py3.6-linux-x86_64.egg'
(and everything under it)
creating ~/local/miniconda/lib/python3.6/site-packages/lltm-0.0.0-py3.6-linux-x86_64.egg
Extracting lltm-0.0.0-py3.6-linux-x86_64.egg to ~/local/miniconda/lib/python3.6/site-packages
lltm 0.0.0 is already the active version in easy-install.pth

Installed ~/local/miniconda/lib/python3.6/site-packages/lltm-0.0.0-py3.6-linux-x86_64.egg
Processing dependencies for lltm==0.0.0
Finished processing dependencies for lltm==0.0.0
```

关于编译器的一个小注意事项：由于 ABI 版本问题，用于构建 C++ 扩展的编译器必须与 ABI 兼容，并且这里的编译器是必须是与构建 PyTorch 时采用的编译器一样的。实际上，这意味着你必须在 Linux 上使用 GCC 4.9 及更高版本。对于 Ubuntu 16.04 和其他更

新的 Linux 发行版来说，这应该是默认的编译器。在MacOS上，你必须使用clang(没有任何与ABI版本相关的问题）。在最坏的情况下，你可以使用编译器从源代码构建 PyTorch，然后使用相同的编译器构建扩展。

构建扩展后，你只需使用在 `setup.py` 脚本中指定的名称在Python中导入它。请务必首先运行 `import torch` ，因为这将解析动态链接器必须看到的一些符号：

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

如果我们在函数或模块上调用 `help()` ，我们可以看到它的签名(signature）与我们的 C++ 代码匹配：

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

    LLTM forward
```

既然我们现在能够从 Python 中调用我们的 C++ 函数，我们可以使用 `torch.autograd.Function` 和 `torch.nn.Module` 来包装(warp）它们，使它们成为 PyTorch 中的一等公民(first class citizens，关键的一部分）：

```
import math
import torch


import lltm

class LLTMFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weights, bias, old_h, old_cell):
        outputs = lltm.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.backward(
            grad_h.contiguous(), grad_cell.contiguous(), *ctx.saved_variables)
        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)
```

性能比较

现在我们可以使用并调用来自 PyTorch 的 C++ 代码，我们可以运行一个小的基准测试来看看我们在 C++ 中重写的操作的性能。我们将运行 LLTM 中的前向转播与反向传播几次并测量运行的时间：

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

如果运行我们在本文开头用纯 Python 编写原始 LLTM 的代码，我们将得到以下数字(在我的机器上）：

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

然后是运行全新的 C++ 版本的代码：

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

我们已经可以看到前向传播函数的显着加速(超过30％）。对于反向传播函数而言，我们也是可以看到加速效果的，尽管加速的效果不是很明显。我在上面写的反向传播并没有经过特别优化，它绝对还可以进行改进。此外，PyTorch 的自动差分引擎可以自动并行化计算图，可以使用更高效的整体操作流，并且这也是用 C++ 实现，因此预计运行速度会很快。尽管如此，这是一个良好的开端。

在GPU设备上的性能

关于 PyTorch 的 *ATen* 后端的一个很好的事实是它抽象了你正在运行代码的计算设备。这意味着我们为CPU编写的代码也可以在GPU上运行，并且各个操作将相应地分派到以 GPU 优化过后的实现中去。对于某些操作，如矩阵乘法(如 `mm` 或 `admm` ），这是一个很大的胜利。让我们看一下使用 CUDA 张量运行 C++ 代码可以获得多少的性能提升。我们不需要对代码作出任何改变，我们只需要将我们的张量放在 Python 中的 GPU 内存中，在创建时添加 `device = cuda_device` 参数或在创建后使用 `.to(cuda_device)` 即可：

```
import torch

assert torch.cuda.is_available()
cuda_device = torch.device("cuda")

batch_size = 16
input_features = 32
state_size = 128


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))
```

再一次将我们的普通 PyTorch 代码与我们的 C++ 版本进行比较，现在两者都运行在 CUDA 设备上，我们科技再次看到性能得到了提升。 对于 Python/PyTorch 来说：

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

然后是 C++ / ATen：

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

与非 CUDA 代码相比，这是一个很好的整体加速。但是，通过编写自定义 CUDA 内核，我们可以从 C++ 代码中得到更多的性能提升，我们将很快在下面介绍这些内核。在此之前，让我们讨论构建 C++ 扩展的另一种方法。

JIT 编译扩展

在之前，我提到有两种构建 C++ 扩展的方法：使用 `setuptools` 或者是实时(JIT）。 在对前者进行了说明之后，让我们再详细说明一下后者。JIT 编译机制通过在 PyTorch 的 API 中调用一个名为 `torch.utils.cpp_extension.load()` 的简单函数，为你提供了一种编译和加载扩展的方法。对于 LLTM，这看起来就像下面这样简单：

```
from torch.utils.cpp_extension import load

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

在这里，我们为函数提供与 `setuptools` 相同的信息。 在后台，将执行以下操作：

1. 创建临时目录 `/tmp/torch_extensions/lltm`
2. 将一个 Ninja 构建文件发送到该临时目录，
3. 将源文件编译为共享库
4. 将此共享库导入为 Python 模块

实际上，如果你将 `verbose = True` 参数传递给 `cpp_extension.load()` ，该过程在进行的过程中将会告知你：

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

生成的 Python 模块与 setuptools 生成的完全相同，但不需要维护单独的 `setup.py` 构建文件。如果你的设置更复杂并且你确实需要 `setuptools` 的全部功能，那么你可以编写自己的 `setup.py` ——但在很多情况下，这种JIT的方式就已经完全够用了。第一次运行此行代码时，将耗费一些时间，因为扩展正在后台进行编译。由于我们使用 Ninja 构建系统来构建源代码，因此重新编译的工作量是不断增加的，而当你第二次运行 Python 模块进行重新加载扩展时速度就会快得多了，而且如果你没有对扩展的源文件进行更改，需要的开销也将会很低。

编写一个 C++/CUDA 混合的拓展

为了真正将我们的实现的性能提升到一个新的水平，我们可以自定义 CUDA 内核并全手工的完成前向和反向传播中部分代码的编写。对于 LLTM 来说，这具有特别有效的前景，因为序列中存在大量逐点运算，所有这些运算都可以在单个 CUDA 内核中融合和并行化。让我们看看如何使用这种扩展机制编写这样的 CUDA 内核并将其与 PyTorch 整合到一起。

编写 CUDA 扩展的一般策略是首先编写一个 C++ 文件，该文件定义了将从 Python 中调用的函数，并使用 pybind11 将这些函数绑定到 Python 上。此外，该文件还将 *声明* 在 CUDA(`.cu` ）文件中将定义的函数。然后，C++ 函数将进行一些检查，并最终将其调用转发给 CUDA 函数。在 CUDA 文件中，我们编写了实际的 CUDA 内核。接着， `cpp_extension` 包将负责使用 C++ 编译器(如 `gcc` ）和使用 NVIDIA 的 `nvcc` 编译器的CUDA源编译 C++ 源代码。以此来确保每个编译器处理它最好编译的文件。最终，它们将链接到一个可从 Python 代码中进行访问的共享库。

我们将从 C++ 文件开始，我们将其称为 `lltm_cuda.cpp` ，例如：

```
// CUDA forward declarations

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

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

// C++ interface




std::vector<at::Tensor> lltm_forward(
    at::Tensor input,
    at::Tensor weights,
    at::Tensor bias,
    at::Tensor old_h,
    at::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<at::Tensor> lltm_backward(
    at::Tensor grad_h,
    at::Tensor grad_cell,
    at::Tensor new_cell,
    at::Tensor input_gate,
    at::Tensor output_gate,
    at::Tensor candidate_cell,
    at::Tensor X,
    at::Tensor gate_weights,
    at::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)");
}
```

正如你所看到的，它主要是一个样板(boilerplate），检查和转发到我们将在 CUDA 文件中定义的函数。我们将这个文件命名为 `lltm_cuda_kernel.cu` (注意 `.cu` 扩展名！）。NVCC 可以合理地编译 C++ 11，因此我们仍然可以使用 ATen 和 C++ 标准库(但 `torch.h` 不行）。 请注意， `setuptools` 无法处理具有相同名称但扩展名不同的文件，因此如果使用 `setup.py` 方法而不是 JIT 方法，则必须为 CUDA 文件指定与 C++ 文件不同的名称(对于JIT） 方法， `lltm.cpp` 和 `lltm.cu` 会正常工作）。 我们来看看这个文件的样子：

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

在这里，我们可以看到我刚刚描述的头文件，以及我们使用 CUDA 特定的声明，如 `__device__` 和 `__forceinline__` 以及像 `exp` 这样的函数。让我们继续使用我们将需要用到的一些辅助函数：

```
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);
}
```

现在实际上实现了一个函数，我们还需要两件事：一个函数执行我们不希望手动显式写入的操作并调用 CUDA 内核，然后实际的 CUDA 内核用于我们想要加速的部分。对于前向转播来说，第一个函数应如下所示：

```
std::vector<at::Tensor> lltm_cuda_forward(
    at::Tensor input,
    at::Tensor weights,
    at::Tensor bias,
    at::Tensor old_h,
    at::Tensor old_cell) {
  auto X = at::cat({old_h, input}, /*dim=*/1);
  auto gates = at::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 = at::zeros_like(old_cell);
  auto new_cell = at::zeros_like(old_cell);
  auto input_gate = at::zeros_like(old_cell);
  auto output_gate = at::zeros_like(old_cell);
  auto candidate_cell = at::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};
}
```

这里主要关注的是 `AT_DISPATCH_FLOATING_TYPES` 宏和内核启动(由 `<<<...>>>` 进行表示）。虽然 ATen 会对我们所处理的张量的设备和数据类型进行抽象化，但是在运行时，张量仍将由具体设备上的具体类型的内存支持。因此，我们需要一种在运行时确定张量是什么类型的方法，然后选择性地调用相应的具有正确类型签名(signature）函数。手动完成这些部分，这将(概念上）看起来像这样：

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

`AT_DISPATCH_FLOATING_TYPES` 的目的是为我们处理此次调度。它需要一个类型(在我们的例子中是 `gates.type()` ），一个名称(用于错误消息）和一个 lambda 函数。在这个 lambda 函数中，类型别名 `scalar_t` 是可用的，并且被定义为张量在该上下文中实际处于

运行时的类型。因此，如果我们有一个模板函数(我们的 CUDA 内核将作为模板函数），我们可以用这个 `scalar_t` 别名实例化它，然后正确的函数就会被调用。 在这种情况下，我们还希望检索张量的数据指针作为 `scalar_t` 类型的指针。 如果你想调度所有类型而不仅仅是浮点类型(`Float` 和 `Double` ），你可以使用 `AT_DISPATCH_ALL_TYPES` 。

请注意，我们使用普通 ATen 执行的一些操作。这些操作仍将在 GPU 上运行，但使用的是 ATen 的默认实现。这是有道理的，因为 ATen 将使用高度优化的例程来处理诸如矩阵乘法(例如 `addmm` ）或者是一些我们自己十分难以实现以及完成性能提升的操作，如卷积操作。

至于内核启动本身，我们在这里指定每个 CUDA 块将具有1024个线程，并且整个 GPU 网格被分成尽可能多的 `1 x 1024` 线程块，并以一组一个线程的方式填充我们的矩阵。例如，如果我们的状态(state）大小为2048且批量大小为4，那么我们将以每个块1024个线程完成启动，总共 `4 x 2 = 8` 个块。如果你之前从未听说过 CUDA “块”或“网格”，那么关于 CUDA 的介绍性阅读可能会有所帮助。

实际的 CUDA 内核非常简单(如果你以前进行过 GPU 编程） ：

```
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];
  }
}
```

这里最感兴趣的是，我们能够完全并行地为门矩阵中的每个单独组件计算所有的这些逐点运算。如果你能想象必须用一个巨大的 `for` 循环来连续超过一百万个元素的情况，你也可以理解为什么改进之后速度会更快了。

反向传播遵循相同的模式，在这里将不再详细说明：

```
template <typename scalar_t>
__global__ void lltm_cuda_backward_kernel(
    scalar_t* __restrict__ d_old_cell,
    scalar_t* __restrict__ d_gates,
```

```
    const scalar_t* __restrict__ grad_h,
    const scalar_t* __restrict__ grad_cell,
    const scalar_t* __restrict__ new_cell,
    const scalar_t* __restrict__ input_gate,
    const scalar_t* __restrict__ output_gate,
    const scalar_t* __restrict__ candidate_cell,
    const scalar_t* __restrict__ gate_weights,
    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) {
    const auto d_output_gate = tanh(new_cell[index]) * grad_h[index];
    const auto d_tanh_new_cell = output_gate[index] * grad_h[index];
    const auto d_new_cell =
        d_tanh(new_cell[index]) * d_tanh_new_cell + grad_cell[index];

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

    const auto input_gate_index = gates_row + column;
    const auto output_gate_index = gates_row + state_size + column;
    const auto candidate_cell_index = gates_row + 2 * state_size + column;

    d_gates[input_gate_index] =
        d_input_gate * d_sigmoid(gate_weights[input_gate_index]);
    d_gates[output_gate_index] =
        d_output_gate * d_sigmoid(gate_weights[output_gate_index]);
    d_gates[candidate_cell_index] =
        d_candidate_cell * d_elu(gate_weights[candidate_cell_index]);
  }
}

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

  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.data<scalar_t>(),
        d_gates.data<scalar_t>(),
        grad_h.contiguous().data<scalar_t>(),
        grad_cell.contiguous().data<scalar_t>(),
        new_cell.contiguous().data<scalar_t>(),
        input_gate.contiguous().data<scalar_t>(),
        output_gate.contiguous().data<scalar_t>(),
        candidate_cell.contiguous().data<scalar_t>(),
        gate_weights.contiguous().data<scalar_t>(),
        state_size);
  }));

  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);
  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};
}
```

将 C++/CUDA 操作与 PyTorch 集成

我们支持 CUDA 的操作与 PyTorch 的集成同样十分简单。如果你想写一个 `setup.py` 脚本，它可能看起来像这样：

```
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
    })
```

我们现在使用 `CUDAExtension()` 而不是 `CppExtension()` 。我们可以只指定 `.cu` 文件和 `.cpp` 文件——库可以解决所有麻烦。JIT 机制则更简单：

```
from torch.utils.cpp_extension import load

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

性能比较

我们希望并行化与融合我们代码与 CUDA 的逐点操作将改善我们的 LLTM 的性能。让我们看看是否成立。我们可以运行在前面列出的代码来进行基准测试。我们之前的最快版

本是基于 CUDA 的 C++ 代码：

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

现在使用我们的自定义 CUDA 内核：

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

性能得到了更多的提升！

结论

你现在应该对 PyTorch 的 C++ 扩展机制以及使用它们的动机有一个很好的大致上的了解了。你可以在此处中找到本文中显示的代码示例。如果你有任何疑问，请使用论坛。如果你遇到任何问题，请务必查看我们的FAQ。