点击小眼睛开启蜘蛛网特效

Pytorch拓展进阶(二)：Pytorch结合C++以及Cuda拓展

Oldpan 2018年6月12日 20条评论 31,197次阅读 34人点赞

前言

之前的文章中：Pytorch拓展进阶(一)：Pytorch结合C以及Cuda语言 (https://oldpan.me/archives/pytorch-combine-c-and-cuda)。我们简单说明了如何简单利用C语言去拓展Pytorch并且利用编写底层的 `.cu` 语言。这篇文章我们说明如何利用C++和Cuda去拓展Pytorch，同样实现我们的自定义功能。

为何使用C++

之前已经提到了什么我们要拓展，而不是直接使用Pytorch提供的python函数去构建算法函数。很简单，因为效率以及速度-还有深度的自定义。

> Pytorch虽然已经使用了NVIDIA cuDNN、Intel MKL和NNPACK这些底层来加快训练速度，但是在某些情况下，比如我们要实现一些特定算法，光靠组合Pytorch已有的操作是不够的。这是因为Pytorch虽然在特定操作上经过了很好的优化，但是对于Pytorch已经写好的这些操作，假如我们组合起来，组成我们的新的算法，Pytorch才不管你的算法的具体执行流程，一般Pytorch只会按照设计好的操作去使用GPU的通道，然后通道不能充分利用或者直接超负载，然后python解释器也不能对此进行优化，导致程序执行速度反而变慢了。

那么之前已经说到了利用c语言可以，那么C++拓展和C语言拓展的区别是什么呢?

C++是趋势。

`Pytorch-v1.0` 即将出现，在官方的介绍中，有这么一段话：

> On the backend side of things, PyTorch will see some changes, which might affect user-written C and C++ extensions. We are replacing (or refactoring) the backend ATen library to incorporate features and optimizations from Caffe2.

大致意思就是，C语言底层的库和C++底层的库会因为结合caffe2而有所改变，但是接口应该变动不会太大，上面提到了 `replacing` 和 `refacoring` 比较耐人寻味。Aten是Pytorch现在使用的C++拓展专用库，Pytorch的设计者想去重构这个库以去适应caffe2.

那么，C++拓展的功能，相比C来说，应该是Pytorch更看重的一点(当然C还是能拓展的)，所以我们今天说一说C++拓展，长远来看，是值得去学习的。

以一个例子开始

同样，我们首先设计一个普通的神经网络层：

这个层叫做LLTM，即*Long-Long-Term-Memory*。一个经典的RNN构造。继承 `nn.Module` ，然后按照我们平常的进行定义即可。

```
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.Tensor(3 * state_size, input_features + state_size))
        self.bias = torch.nn.Parameter(torch.Tensor(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 = F.sigmoid(gates[0])
        output_gate = F.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 = F.tanh(new_cell) * output_gate

        return new_h, new_cell
```

定义好了我们这样使用:

```
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))
```

上面的程序当然是可以执行的，但是我们要注意，可以执行和效率是两码事，我们之所以要用拓展，是因为我们要提高我们算法运行的速度和实现我们所有想要实现自定义功能。

用C++写这个例子

在编写C++程序之前我们需要安装pybind这个python-C++拓展库。

pybind11

安装pybind11很简单，执行下面两个命令就行：

```
pip install pytest
pip install pybind11
```

注意：可能还需要安装 `python3-dev` 和 `pyyaml` 。

用C++进行编写

好了，上面使我们的python版实现过程，现在我们改成C++版，当然我们首先编写一下简单的sigmoid功能函数：

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

#include <iostream>

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

上面的程序，引用的头文件 `<torch.h>` 很重要，这个头文件是在Pytorch安装后已经包含的，所以我们用尖括号括起来，这个头文件里面都有啥：

```
#pragma once

#include <Python.h>

#include <ATen/ATen.h>
#include <pybind11/pybind11.h>
#include <torch/csrc/THP_export.h>
#include <torch/csrc/utils/pybind.h>

namespace torch {

// NOTE: This API is currently highly experimental and may change drastically
// in the near future.

/// Returns a `Type` object for the given backend (e.g. `at::kCPU`) and
/// `ScalarType` (e.g. `at::kDouble`).
THP_CLASS at::Type& getType(at::Backend backend, at::ScalarType type);

/// Returns a `Type` object for the CPU backend and the given `ScalarType`
/// (e.g. `at::kDouble`). Equivalent to `getType(kCPU, type)`.
THP_CLASS at::Type& CPU(at::ScalarType type);

/// Returns a `Type` object for the CUDA backend and the given `ScalarType`
/// (e.g. `at::kDouble`). Equivalent to `getType(kCUDA, type)`.
THP_CLASS at::Type& CUDA(at::ScalarType type);

/// Sets the `requires_grad` property of the given `Tensor`.
THP_CLASS void set_requires_grad(at::Tensor& tensor, bool requires_grad) noexcept;

/// Returns the `requires_grad` of the given `Tensor`.
THP_CLASS bool requires_grad(const at::Tensor& tensor) noexcept;
} // namespace torch
```

显然，这个头文件中大概有三个东西：

- Aten库，这个库就是pytorch操作Tensor的C++接口
- pybind11,这个是用来将python和C++结合起来
- 一些头文件，用来整合Aten和pybind11

好了，我们开始编写整个forward函数：

```
#include <vector>

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};
}
```

然后我们编写backward函数，需要注意的是Pytorch的C++接口并不会自动实现反向求导，需要我们自己去写，当然怎么写不用细究：

```
// 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};
}
```

绑定到Python

现在我们要把刚才写的C++绑定到Python上，我们在上面的 `.cpp` 文件最下面加上：

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

然后我们的文件目录大概是这样：

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

在 `setup.py` 中写这些信息，我们使用setuptools去编译我们的C++代码， `CppExtension` 和 `BuildExtension` 可以很方便地让我们实现对C++和Cuda的编译：

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


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


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

执行 `python setup.py install` 后就可以看到：

```
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/
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=~/
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.lin
copying build/lib.linux-x86_64-3.6/lltm.cpython-36m-x86_64-linux-gnu.so -> build/bdist.linux-x8
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
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
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
```

注意：编译有可能失败，这时候需要安装 `python3-dev` 和 `pyyaml` 等拓展组件。

引用C++代码

编译好后，我们就可以：

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

然后我们就可以定义我们的自定义层 (%E8%87%AA%E5%AE%9A%E4%B9%89%E5%B1%82)了：

```
import math
import torch

# Our module!
import lltm

class LLTMFunction(torch.nn.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, old_cell]
        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, d_gates = 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.Tensor(3 * state_size, input_features + state_size))
        self.bias = torch.nn.Parameter(torch.Tensor(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)
```

官方的结果中，程序在CPU和GPU中运行，C++版的运行速度要大于直接使用pytorch编写层的速度。注意，我们只编写了C++代码但是却可以在CPU中和GPU中跑，为什么，这就要归功于Aten的设计，Aten就是pytorch的C++版，使用Aten编写出来的tensor，只要在程序中 `.cuda()` ，就可以将Tensor移到GPU当中了。

但是这样移到GPU中和直接编写cuda语言是不一样的。

编写CUDA代码

之前我们说明了如何写 `C++` 代码，现在我们来编写如何去写 `.cu` 代码然后去和 `C++` 代码结合。C++和cuda代码结合其实和C语言是类似的，需要我们使用C++来写接口函数和python相连，然后使用C++去调用cuda程序。

首先我们写C++部分的接口程序：

```
// lltm_cuda.cpp
#include <torch/torch.h>

#include <vector>

// 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

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

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)");
```

注意上面的代码是接口程序，并没有具体的实现，而下面的代码才是核心的cuda代码：

```
// lltm_cuda_kernel.cu
#include <ATen/ATen.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));
}

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);
}


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];
  }
}


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};
}


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_forward_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};
}
```

只需要关注核心的代码即可，上面的代码后缀是 `.cu` ，我们可以看到上面代码中也有C++的特性，那是因为cuda的nvcc编译器不仅支持c语言也支持C++语言的语法，我们需要注意的只是数据类型一定要写正确。返回的Tensor顺序不能错。在自己设计cuda程序的时候一定要注意内存问题。

具体的详细信心可以查看官方文档，这里只是简单介绍。

然后我们同样编写 `setup.py` ：

```
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
    })
```

和之前不同的是我们采用的拓展变为了 `CUDAExtension` 和包含文件包含了 `.cu` 。

编译完成后，根据官网的性能测试，比起之前单纯使用C++在GPU上跑速度又提升了一阶。

官方代码地址:https://github.com/pytorch/extension-cpp (https://github.com/pytorch/extension-cpp)

后记

使用C++和C都可以拓展pytorch实现自定义功能或者设计自己的算法。Pytorch拓展这些其实还是比较容易的，唯一的缺点就是，官方几乎没有这方面的文档说明，也就是接口说明，需要我们自己去研究。不过我们平时所需要的接口也就那么几种，多多编写熟悉了就就好。

参考资料

https://blog.csdn.net/fitzzhang/article/details/78988682 (https://blog.csdn.net/fitzzhang/article/details/78988682)
https://discuss.pytorch.org/t/compiling-an-extension-with-cuda-files/302/8 (https://discuss.pytorch.org/t/compiling-an-extension-with-cuda-files/302/8)
http://blog.christianperone.com/2018/03/pytorch-internal-architecture-tour/#python-intro (http://blog.christianperone.com/2018/03/pytorch-internal-architecture-tour/#python-intro)
https://devblogs.nvidia.com/even-easier-introduction-cuda/ (https://devblogs.nvidia.com/even-easier-introduction-cuda/)
http://pybind11.readthedocs.io/en/master/index.html (http://pybind11.readthedocs.io/en/ma

ster/index.html)
https://pytorch.org/tutorials/advanced/cpp_extension.html (https://pytorch.org/tutorials/advanced/cpp_extension.html)

CUDA (HTTPS://OLDPAN.ME/LABELS/CUDA)

C语言 (HTTPS://OLDPAN.ME/LABELS/C%E8%AF%AD%E8%A8%80)

PYTORCH (HTTPS://OLDPAN.ME/LABELS/PYTORCH)

机器学习 (HTTPS://OLDPAN.ME/LABELS/%E6%9C%BA%E5%99%A8%E5%AD%A6%E4%B9%A0)

深度学习 (HTTPS://OLDPAN.ME/LABELS/%E6%B7%B1%E5%BA%A6%E5%AD%A6%E4%B9%A0)

<上一篇 > 教你30分钟安装cuda环境下的torch(非Pytorch)
(https://oldpan.me/archives/torch-install)

<下一篇> 马尔科夫随机场(MRF)在图像处理中的应用-图像分割、纹理迁移
(https://oldpan.me/archives/markov-random-field-deeplearning)

猜你喜欢

1. 暂无相关文章

小 小星星
2021年5月27日 下午10:08 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-47651)

大佬你好，我想问一下，这个官方的代码，/cuda文件夹下的setup.py我一直安装不成功，load函数不能用，这个你知道是怎么回事吗，我的环境是Ubuntu18.04，cuda：11.1，python：3.7，pytorch：1.8.1，期待回，谢谢

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=47651#respond)

小 小龙
2020年12月20日 下午11:51 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-47175)

楼主,您好,我的代码老是报如下错误:
RuntimeError: CUDA error: an illegal memory access was encountered (copy_kernel_cuda at /pytorch/aten/src/ATen/native/cuda/Copy.cu:187)
有时候会出现,有时候又不会出现,不知您有遇到过没

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=47175#respond)

Pingback： PyTorch中的C++扩展实现-中国站长社区 (https://www.zgzzsq.com/338246.html)

G Galway
2020年2月13日 上午12:17 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-21608)

发现CPP扩展的效率并没有比Python端API的效率高，跑了几个简单的对比实验，结果如下
模型	内存	速度	GPU利用率
nn.LSTM	4989M	5.11it/s	保持在94%到96%，几乎没浮动
pytorch类实现	5155M	0.72it/s	在0%到100%跳动，浮动大
C++扩展实现	7121M	0.70it/s	在0%到100%跳动，浮动大
结果是，内存和运行效率上，C++扩展实现的LSTM表现都要比python API的差。如果能把for写到C++端，效率应该会高很多，但不知道怎么写（太菜鸡了……）

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=21608#respond)

O Oldpan (https://oldpan.me)
2020年2月15日 下午9:13 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-21883)

效率当然不能只看语言了，python端不论数据载入和算法代码都很完善了(底层也是C++毕竟)，官方也建议没有特殊需要不建议使用C++前端

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=21883#respond)

S Stone (http://none)
2020年1月8日 下午4:21 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-18145)

感谢博主。请问可以通过pycuda直接将前传反传的cuda内核写到自定义层吗，比如https://vimsky.com/examples/detail/python-method-pycuda.compiler.SourceModule.html 我试了一下，反传报错，不能编译通过

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=18145#respond)

O Oldpan (https://oldpan.me)
2020年1月9日 上午11:08 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-18214)

这个可能不行，接口数据结构都不一样，还是原生的CUDA写吧

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=18214#respond)

K KingsleyDec
2020年2月18日 下午6:02 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-22184)

我也遇到了这个问题，编译显示通过并加载到site-package中，但是在交互模式下不能import lltm， 但是可以通过jtl load 并且能够使用，不知道是什么原因？

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=22184#respond)

Y yhhhcf
2019年12月5日 上午3:50 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-15793)

楼主好，我跑了下原github的代码，发现python extensions在backward部分的效率要高于cpp extensions, 请问这可能是什么原因呢？具体效率如下：
Python: Forward: 229.836/252.881 us | Backward 432.014/465.167 us
CPP: Forward: 211.477/240.314 us | Backward 580.311/682.030 us

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=15793#respond)

O Oldpan (https://oldpan.me)
2019年12月6日 下午9:56 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-15981)

看一下两个运行程序的CPU占用率，是一样么

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=15981#respond)

T

Tinet

2019年11月29日 上午10:49 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-15363)

想请教一下，刚刚尝试了c++那段代码。python setup.py install之后显示成功安装。但是在python里面import lltm的时候却报ModuleNotFoundError: No module named 'lltm'. 请问是我漏掉了什么东西么。非常感谢！

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=15363#respond)

O

Oldpan (https://oldpan.me)

2019年12月2日 下午5:02 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-15520)

安装成功看看有没有生成编译好的东西，再看看路径对不对

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=15520#respond)

B

bubu

2021年4月19日 下午8:51 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-47601)

build文件夹下应该有什么文件呢？报错无法找到...\build\temp.win-amd64-3.6\Release\fused_bias_act.obj”文件

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=47601#respond)

F

FourierTransform

2019年9月16日 下午6:38 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-9488)

这个清除中间变量的文章在哪儿呢 咋找不到啊

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=9488#respond)

B

boxiayizhanke

2019年6月10日 下午4:13 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-6122)

博主你好，想问一下,这个pytorch c++拓展必须是1.0版本以上吗
我在0.4版本上出现这个错误
lltm_cuda.cpp:1:29: fatal error: torch/extension.h:

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=6122#respond)

R

Researching Dexter

2019年6月5日 下午3:34 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-6022)

就是说我只要指定了初始的index，程序就会自动按指定的那个grid和block大小去并行执行，是这个意思吗

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=6022#respond)

O Oldpan (https://oldpan.me)
2019年6月6日 下午11:14 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-6046)

嗯，大概是每个gird每个block每个thread去执行一次相当于for循环中的一次

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=6046#respond)

R Researching Dexter
2019年6月7日 上午10:52 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-6048)

非常感谢，我知道了。

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=6048#respond)

R Researching Dexter
2019年6月3日 下午9:42 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-5958)

这个cuda编程不用写循环吗，我看代码中只是指定了一个index，我不是很明白，我是个新手希望你可以解答一些

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=5958#respond)

O Oldpan (https://oldpan.me)
2019年6月5日 上午12:12 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus#comment-5994)

这个我也说不明白的..因为是并行编程,和平常的串行编程的思想要转化一下，for在并行中不用写就已经被多个thread执行了

回复 (https://oldpan.me/archives/pytorch-cuda-c-plus-plus?replytocom=5994#respond)

发表评论

邮箱地址不会被公开。 必填项已用*标注

昵称 *

邮箱 *

网站

发表评论

评论审核已启用。您的评论可能需要一段时间后才能被显示。

 ()  ()  (HTTPS://GITHUB.COM/OLDPAN)  ()

友链 (HTTPS://OLDPAN.ME/FRIENDS-URL) 资源 ()

RSS (HTTPS://OLDPAN.ME/FEED)

洛奇 (HTTPS://OLDPAN.ME/MABINOGI-MOBLIE-RELEASE)

陕ICP备17018520号-1 (HTTPS://BEIAN.MIIT.GOV.CN)

网站已运行1849天8小时53分57秒