详解PyTorch编译并调用自定义CUDA算子的三种方式

极市平台 2022-12-24 22:00:50 发表于广东 手机阅读

以下文章来源于算法码上来，作者godweiyang

算法码上来

字节算法工程师，本硕专业第一，这里有算法、自然语言处理、模型加速等众多分享，...

作者丨godweiyang@知乎（已授权）

来源丨https://zhuanlan.zhihu.com/p/358778742

编辑丨极市平台

极市导读

本文为一篇实操教程，作者用最为精简最容易理解的文字描述为大家讲解了用PyTorch编译并调用自定义CUDA算子的三种方式：JIT、Setuptools、CMake。>>加入极市CV技术交流群，走在计算机视觉的最前沿

本篇教程我们主要讲解如何 **「编译并调用」** 之前我们写好的CUDA算子，完整的代码还是放在了github仓库，欢迎大家star并fork：

https://github.com/godweiyang/torch-cuda-example

我保证，这是你网上简单 **「最为精简、最容易看懂」** 的一套代码了，因为我自己也是刚入门，复杂的我也看得累。

运行环境

- NVIDIA Driver: 418.116.00
- CUDA: 11.0
- Python: 3.7.3
- PyTorch: 1.7.0+cu110
- CMake: 3.16.3
- Ninja: 1.10.0
- GCC: 8.3.0

这是我自己的运行环境，显卡是V100，其他环境不保证可以运行，但是大概率没问题，可能要做轻微修改。

代码结构

```
├── include
│   └── add2.h # cuda算子的头文件
├── kernel
│   ├── add2_kernel.cu # cuda算子的具体实现
│   └── add2.cpp # cuda算子的cpp torch封装
├── CMakeLists.txt
├── LICENSE
├── README.md
├── setup.py
├── time.py # 比较cuda算子和torch实现的时间差异
└── train.py # 使用cuda算子来训练模型
```

代码结构还是很清晰的。 `include` 文件夹用来放cuda算子的头文件（ `.h` 文件），里面是cuda算子的定义。 `kernel` 文件夹放cuda算子的具体实现（ `.cu` 文件）和cpp torch的接口封装（ `.cpp` 文件）。

最后是python端调用，我实现了两个功能。一是比较运行时间，上一篇教程详细讲过了；二是训练一个PyTorch模型，这个下一篇教程再来详细讲述。

编译cpp和cuda文件

JIT

JIT就是just-in-time，也就是即时编译，或者说动态编译，就是说在python代码运行的时候再去编译cpp和cuda文件。

JIT编译的方法上一篇教程已经演示过了，只需要在python端添加 `load` 代码即可：

```
import torch
from torch.utils.cpp_extension import load
cuda_module = load(name="add2",
                   extra_include_paths=["include"],
                   sources=["kernel/add2.cpp", "kernel/add2_kernel.cu"],
                   verbose=True)
cuda_module.torch_launch_add2(c, a, b, n)
```

需要注意的就是两个参数， `extra_include_paths` 表示包含的头文件目录， `sources` 表示需要编译的代码，一般就是 `.cpp` 和 `.cu` 文件。

cpp端用的是pybind11进行封装：

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("torch_launch_add2",
          &torch_launch_add2,
          "add2 kernel warpper");
}
```

JIT编译看起来非常的简单，运行过程中也基本没有碰到坑，非常顺利。

运行成功的话可以看到Ninja调用了三条命令来编译：

```
[1/2] nvcc -c add2_kernel.cu -o add2_kernel.cuda.o
[2/3] c++ -c add2.cpp -o add2.o
[3/3] c++ add2.o add2_kernel.cuda.o -shared -o add2.so
```

由于输出太长，我省略了多数的参数信息，并精简了指令。可以看出先是调用 `nvcc` 编译了 `.cu` ，生成了 `add2_kernel.cuda.o` ；然后调用 `c++` 编译 `add2.cpp` ，生成了 `add2.o` ；最

后调用 c++ 生成动态链接库 add2.so 。

Setuptools

第二种编译的方式是通过Setuptools，也就是编写 setup.py ，具体代码如下：

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

setup(
    name="add2",
    include_dirs=["include"],
    ext_modules=[
        CUDAExtension(
            "add2",
            ["kernel/add2.cpp", "kernel/add2_kernel.cu"],
        )
    ],
    cmdclass={
        "build_ext": BuildExtension
    }
)
```

编写方法也非常的常规，调用的是 CUDAExtension 。需要在 include_dirs 里加上头文件目录，不然会找不到头文件。

cpp端用的是pybind11进行封装：

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("torch_launch_add2",
          &torch_launch_add2,
          "add2 kernel warpper");
}
```

接着执行：

```
python3 setup.py install
```

这样就能生成动态链接库，同时将 `add2` 添加为python的模块了，可以直接 `import add2` 来调用。

如果执行正常的话，也是可以看到两条编译命令的：

```
[1/2] nvcc -c add2_kernel.cu -o add2_kernel.o
[2/2] c++ -c add2.cpp -o add2.o
```

然后会执行第三条：

```
x86_64-linux-gnu-g++ -shared add2.o add2_kernel.o -o add2.cpython-37m-x86_
```

最后同样生成了一个动态链接库，不过python端我们不需要加载这个动态链接库，因为setuptools已经帮我们把cuda算子调用的接口注册到python模块里了，直接import即可：

```
import torch
import add2
add2.torch_launch_add2(c, a, b, n)
```

需要注意的是，这里我踩了一个坑，**「`.cpp` 和 `.cu` 文件名不要相同，也最好不要取容易与python自带库重复的名字」**。此外要先 `import torch` ，然后再 `import add2` ，不然也会报错。

CMake

最后就是cmake编译的方式了，要编写一个 `CMakeLists.txt` 文件，代码如下：

```
cmake_minimum_required(VERSION 3.1 FATAL_ERROR)
# 修改为你自己的nvcc路径，或者删掉这行，如果能运行的话。
set(CMAKE_CUDA_COMPILER "/usr/local/cuda/bin/nvcc")
project(add2 LANGUAGES CXX CUDA)
```

```
5
6  find_package(Torch REQUIRED)
7  find_package(CUDA REQUIRED)
8  find_library(TORCH_PYTHON_LIBRARY torch_python PATHS "${TORCH_INSTALL_PRE
9
10 # 修改为你自己的python路径，或者删掉这行，如果能运行的话。
11 include_directories(/usr/include/python3.7)
12 include_directories(include)
13
14 set(SRCS kernel/add2.cpp kernel/add2_kernel.cu)
15 add_library(add2 SHARED ${SRCS})
16
17 target_link_libraries(add2 "${TORCH_LIBRARIES}" "${TORCH_PYTHON_LIBRARY}"
```

这里踩了好几个大坑。首先是找不到nvcc的路径，于是第3行先设置了一下，当然如果你删了也能跑那就更好。然后是找不到python的几个头文件，于是加上了第11行，同样如果你删了也能跑那就更好。最后是一个巨坑，没有链接 TORCH_PYTHON_LIBRARY ，导致动态链接库生成成功了，但是调用执行一直报错，所以加上了第8行和第17行。

cpp端用的是 TORCH_LIBRARY 进行封装：

```
1 TORCH_LIBRARY(add2, m) {
2     m.def("torch_launch_add2", torch_launch_add2);
3 }
```

这里不再使用pybind11，因为我的pybind11没有使用conda安装，会出现一些编译问题，详见：

https://github.com/pybind/pybind11/issues/1379#issuecomment-489815562

编写完后执行下面编译命令：

```
1 mkdir build
2 cd build
3 cmake -DCMAKE_PREFIX_PATH="$(python3 -c 'import torch.utils; print(torch.u
4 make
```

最后会在 `build` 目录下生成一个 `libadd2.so` ，通过如下方式在python端调用：

```
import torch
torch.ops.load_library("build/libadd2.so")
torch.ops.add2.torch_launch_add2(c, a, b, n)
```

如果编译成功的话，可以看到如下输出信息：

```
Building CXX object CMakeFiles/add2.dir/kernel/add2.cpp.o
[ 66%] Building CUDA object CMakeFiles/add2.dir/kernel/add2_kernel.cu.o
[100%] Linking CXX shared library libadd2.so
[100%] Built target add2
```

执行python

这里我实现了两个功能，代码都很简单，一个是测试时间，一个是训练模型。都可以通过参数 `--compiler` 来指定编译方式，可供选择的就是上面提到的三种：jit、setup和cmake。

比较运行时间

```
python3 time.py --compiler jit
python3 time.py --compiler setup
python3 time.py --compiler cmake
```

训练模型

```
python3 train.py --compiler jit
python3 train.py --compiler setup
python3 train.py --compiler cmake
```

总结

至此三种编译cuda算子并python调用的方式基本都囊括了，下一篇教程将讲讲PyTorch如何将自定义cuda算子加入到计算图中，并实现前向和反向传播，最终训练模型。

极市干货

技术干货：数据可视化必须注意的30个小技巧总结 | 如何高效实现矩阵乘？万文长字带你从CUDA初学者的角度入门

实操教程：Nvidia Jetson TX2使用TensorRT部署yolov5s模型 | 基于YOLOV5的数据集标注&训练，Windows/Linux/Jetson Nano多平台部署全流程