# 自定义CUDA算子用于PyTorch

## TL;DR

自定义CUDA算子并使用PyTorch调用，通常有三种方式

- JIT（Just-In-Time）。利用PyTorch的JIT机制，在运行时动态编译并使用自定义的CUDA算子
- Setuptools。利用python的Setuptools，预先编译C++代码，并作为python包安装到环境中
- 手动编译与调用。当做普通的C++项目，使用CMake等方式进行编译并生成动态链接库，再使用python调用动态链接库

---

## Content

## 定义Python调用C++的接口

假设我们自定义了一个矩阵相乘的CUDA算子，现在想使用python进行调用，并传入PyTorch的tensor。那么我们先需要在C++中将从python传入的tensor解析成C++可以使用的形式，而PyTorch已经为我们构建好了C++的接口。

C++ *gemm_wrapper*

```cpp
#include <torch/extension.h>  // 一键导入所有与C++ PyTorch相关的头文件

void gemm_torch_wrapper(torch::Tensor &c, const torch::Tensor &a,
const torch::Tensor &b) {
        int n = c.size(0);
        int m = c.size(1);
        int k = a.size(1);
        gemm_gpu_mulblock((float *)c.data_ptr(), (const float
*)a.data_ptr(), (const float *)b.data_ptr(), n, m, k)
}
```

另外，为了实现python和C++两种语言的数据交换，通常使用pybind11库。

C++ *pybind11_interface*

```
PYBIND11_MODULE(name, m) {
        m.def("gemm", &gemm_torch_warpper, "launch gemm_gpu_mulblock");
}
```

## 编译CUDA算子与接口

在定义好接口后，我们就可以将CUDA算子进行编译和调用了。通常来说有三种方式，但三种方式的本质其实是一致的，简单来说都是将CUDA算子编译成动态链接库，然后使用python来调用这些动态链接库。

## 即时编译（JIT, Just-In-Time）

PyTorch提供了一种动态编译的机制，可以在运行时实时编译C++代码，并进行调用。

*JIT.py*

```python
import torch
import os
from torch.utils.cpp_extension import load
module_path = os.path.dirname(__file__)
gemm_jit_module = load(
    name="gemm_jit",
    extra_include_paths=[os.path.join(module_path, "include")],  # explicitly specify the include path
    sources=[
        os.path.join(module_path, "pytorch_wrapper/gemm_op.cpp"),
        os.path.join(module_path, "kernels/gemm_kernel.cu")
    ],
    verbose=True
)

a = torch.randn(640, 640, device="cuda")
b = torch.randn(640, 640, device="cuda")
c = torch.zeros(640, 640, device="cuda")
# JIT (Just-in-time Compilation)
gemm_jit_module.gemm(c, a, b)
```

不过需要注意的是，考虑到pybind11的命名机制，即接口`PYBIND11_MODULE(name, m)`中的`name`，我们将这里的`name`换成PyTorch的一个宏定义`TORCH_EXTENSION_NAME`，这样在使用PyTorch的JIT机制时，才能正确调用。

## Setuptools

与封装其他python包类似，我们也可以用Setuptools来构建一个包。由于使用了PyTorch的Tensor，因此在编写 `setup.py` 时也需要调用PyTorch相关的函数。

*setup.py*

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

setup(
    name="gemm-st",  # python package name, used by pip
    include_dirs=['include'],
    ext_modules=[
        CUDAExtension(
            "gemmst",   # actual module name when calling
            ["kernels/gemm_kernel.cu", "pytorch_wrapper/gemm_op.cpp"]
        )
    ],
    cmdclass={
        'build_ext': BuildExtension
    },
    version="0.1"
)
```

python调用时则与平时通过pip安装的包无异，不过需要注意两点

- 先导入torch，因为依赖于pytorch
- C++端pybind11接口函数的命名，即 `name` ，需要与setup.py中的Extension名称一致，如上面 `CUDAExtension` 中的 `gemmst` ，因为Setuptools会按Extension中的名称来生成动态链接库文件，而pybind11的接口依赖于这个文件，因此名称很重要。当然，如果你手动把生成的动态链接库名称改为pybind11接口函数中的 `name` ，效果也一样。

*python_package.py*

```python
import torch
import gemmst  # gemmst is the actual module name
# define tensor a, b, c in CUDA
gemmst.gemm(c, a, b)
```

## 手动编译与调用（进阶但不必要）

我们都知道，Python是解释型的脚本语言，C++是编译型语言，但Python的底层其实就是C。因此从Python中调用C++代码，本质上就是将C++代码先编译成动态链接库，然后再利用Python加载动态链接库，从而实现调用C++编译好的内容。万变不离其宗，这三种方法本质上都是如此

- JIT是在运行时实时编译，将动态链接库文件保存到临时目录中
- Setuptools是预先编译，但会额外生成一些python包信息（如egg信息），更方便调用
- 手动编译也是余弦编译，不过只会生成动态链接库，调用起来会稍微麻烦

这样来说，我们完全可以先手动将C++代码编译成动态链接库，比如使用CMake进行构建。

*CUDA+PyTorch+Pybind11+CMake*

```
cmake_minimum_required(VERSION 3.20)
project(gemm LANGUAGES CXX CUDA)  # activate support for CUDA

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CUDA_STANDARD 17)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}")

# activate support for PyTorch
# For Conda environment, we need to manually specify the Python include directory and PyTorch's cmake prefix path
find_package(Python COMPONENTS Interpreter Development)  # see https://cmake.org/cmake/help/latest/module/FindPython3.html for more details
message("Python_EXECUTABLE ${Python_EXECUTABLE}")
message("Python_INCLUDE_DIRS ${Python_INCLUDE_DIRS}")
execute_process(COMMAND ${Python_EXECUTABLE} -c "import torch; print(torch.utils.cmake_prefix_path)" OUTPUT_VARIABLE PYTORCH_CMAKE_PREFIX_PATH OUTPUT_STRIP_TRAILING_WHITESPACE)
list(APPEND CMAKE_PREFIX_PATH "${PYTORCH_CMAKE_PREFIX_PATH}/Torch")
message("PYTORCH_CMAKE_PREFIX_PATH ${PYTORCH_CMAKE_PREFIX_PATH}")
message("CMAKE_PREFIX_PATH ${CMAKE_PREFIX_PATH}")
find_package(Torch REQUIRED)
find_library(TORCH_PYTHON_LIBRARY torch_python PATHS "${TORCH_INSTALL_PREFIX}/lib")
include_directories("${Python_INCLUDE_DIRS}")

# Add pybind11 support
```

```
execute_process(COMMAND ${Python_EXECUTABLE} -c "import pybind11; print(pybind11.get_cmake_dir())" OUTPUT_VARIABLE pybind11_DIR OUTPUT_STRIP_TRAILING_WHITESPACE)
find_package(pybind11 REQUIRED)

# add custom header files
include_directories("include")

# add source files
file(GLOB SOURCES
    "${CMAKE_CURRENT_SOURCE_DIR}/kernels/gemm_kernel.cu"
    "${CMAKE_CURRENT_SOURCE_DIR}/pytorch_wrapper/gemm_op.cpp"
)

# generate a shared library rather than an executable application
add_library(gemm SHARED ${SOURCES})

# dynamic link library
target_link_libraries(gemm PUBLIC ${CUDA_LIBRARIES} ${TORCH_LIBRARIES} ${TORCH_PYTHON_LIBRARY} pybind11::module)

# set the module name and extension to things like setuptools do.
set_target_properties(gemm PROPERTIES PREFIX "${PYTHON_MODULE_PREFIX}" SUFFIX "${PYTHON_MODULE_EXTENSION}")
```

这里需要注意的也是动态链接库的命名，最后一行将动态链接库的命名设置为类似setuptools默认的规则。

在调用时，由于使用了pybind11接口，而不是直接定义函数指针，因此不能使用`ctypes.cdll`，而是直接将编译好的动态链接库文件放到python文件同级目录下，然后根据动态链接库的名称进行导入即可。

*manually_import*

```
# for dll file named like "gemm.cpython-38-x86_64-linux-gnu.so"
import gemm
gemm.gemm(c, a, b)
```

此外，你也可以用PyTorch自带的`TORCH_LIBRARY`来定义C++接口并使用`torch.ops`来导入和调用。

*pytorch_binding_cpp_interface*

```
// define python binding
// PYBIND11_MODULE(gemm, m) {
//     m.def("gemm", &gemm_torch_warpper, "launch
gemm_gpu_mulblock");
// }

TORCH_LIBRARY(gemmpt, m) {
    m.def("gemm", &gemm_torch_warpper);
}
```

*pytorch_binding_py_call*

```
import torch
torch.ops.load_library("gemm.cpython-38-x86_64-linux-gnu.so")
torch.ops.gemmpt.gemm(c, a, b)
```

此时，就不用管动态链接库的命名了，只需要通过`TORCH_LIBRARY`中的`name`来调用即可。

---

## Supplementary Readings

1. FindPython — CMake 3.29.3 Documentation
2. 知 详解PyTorch编译并调用自定义CUDA算子的三种方式 - 知乎