# 教程：Python中使用C++/CUDA｜以PointNet中的ball query 为例


JeffWang
中科院自动化研究所 模式识别与智能系统博士在读
关注他





Python语言的好处就是简单明了易上手，但是速度却比C++和CUDA编程慢了不少，尤其是在深度学习火热的今天，训练和inference效率也是极为重要的。本文主要介绍了如何在Python中使用C++和CUDA，以达到加速部分算子的目的。

本文以PointNet中的ball query操作为例，讲解具体的实现步骤。

> ball query问题：我们有1000个点 $P \in \mathbb{R}^{1000,3}$ ，3代表坐标 $x, y, z$ 。给定5个中心坐标点 $C \in \mathbb{R}^{5,3}$ ,以每个中心坐标点 $C_i$ 为球心,我都给定一个固定的半径 $r$ 画一个球，每个球内都有点集 $P$ 落入其中的点，我需要找到每个球中离球心最近的 $K$ 个点。
> 不难看出，这个问题可以并行求解，5个中心点可以并行的找出离自己最近的$K$ 个点。所以可以借助CUDA和C++来加速该算法的实现。

本文实现代码如下，代码十分轻量简洁：

GitHub - JeffWang987/Python_Using_Cpp_CUD...
github.com/JeffWang987/Python_Using_...

## 总体流程（简要）

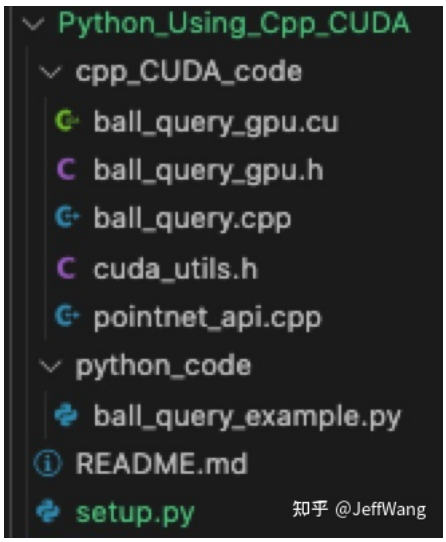
（1）为我们需要的**总体框架**编写C++代码（针对ball query中的5个中心点）

（2）在CUDA文件中编写**单个样本**实现的算法：（针对ball query中的1个中心点）

（3）用**pybind11**绑定c++和python

（4）利用**setuptools**编译c++和cuda文件，python可直接调用

整体文件夹划分如下：




## 编写C++代码：框架

c++这块主要是整体框架的编写，在ball_query.cpp中核心代码如下：已写上注释。可以看出并不并在最后调用CUDA代

码ball_query_kernel_launcher_cuda进行算法的实现。

```
#include <torch/serialize/tensor.h>
#include <vector>
#include <THC/THC.h>
#include <cuda.h>
#include <cuda_runtime_api.h>
#include "ball_query_gpu.h"

....省略.....
int ball_query_wrapper_cpp(int b, int n, int m, float radius, int nsample,
    at::Tensor new_xyz_tensor, at::Tensor xyz_tensor, at::Tensor idx_tensor) {
    // b: batch，n为输入点集P的个数，m是输出中心点的个数，radius是每个球的半径，
    // nsample是每个球内最多允许找的点数，new_xyz_tensor[b,m,3]代表中心点坐标，
    // xyz_tensor[b,n,3]代表原始点集合P的坐标，
    // idx_tensor[b,m,nsample]为输出的m个中心点聚合的nsample个点的idx

    // 检查输入是否为contiguous的torch.cuda变量
    CHECK_INPUT(new_xyz_tensor);
    CHECK_INPUT(xyz_tensor);

    // 建立指针
    const float *new_xyz = new_xyz_tensor.data<float>();
    const float *xyz = xyz_tensor.data<float>();
    int *idx = idx_tensor.data<int>();

    // 放入到CUDA中进行具体的算法实现
    ball_query_kernel_launcher_cuda(b, n, m, radius, nsample, new_xyz, xyz, idx);
    return 1;
}
```

## CUDA代码：并行算法

CUDA还是使用C++编程，只不过文件后缀是.cu。在CUDA编程时，我们可以定义普通的C++类型函数，也可以定义kernel函数，kernel函数由N个不同的CUDA线程并行执行N次。我们在调用kernel函数时，需要用<<<blocks, threads>>>来定义执行该kernel所需要使用的线程数量。（参考捏太阳：一、CUDA C++ 编程指导）

> （1）__host__ int foo(int a){}与C或者C++中的foo(int a){}相同，是由CPU调用，由CPU执行的函数，__host__可缺省。
> （2）__global__ int foo(int a){}表示一个内核函数，是一组由GPU执行的并行计算任务，以foo<<>>(a)的形式或者driver API的形式调用。目前__global__函数必须由CPU调用，并将并行计算任务发射到GPU的任务调用单元。随着GPU可编程能力的进一步提高，未来可能可以由GPU调用。
> （3）__device__ int foo(int a){}则表示一个由GPU中一个线程调用的函数。由于Tesla架构的GPU允许线程调用函数，因此实际上是将__device__ 函数以__inline形式展开后直接编译到二进制代码中实现的，并不是真正的函数。

CUDA部分的主函数(*_host_*被缺省了)如下：

```
void ball_query_kernel_launcher_cuda(int b, int n, int m, float radius, int nsample, \
    const float *new_xyz, const float *xyz, int *idx) {

    // cudaError_t变量用来记录CUDA的err信息，在最后需要check
    cudaError_t err;
    // divup定义在cuda_utils.h,DIVUP(m, t)相当于把m个点平均划分给t个block中的线程，每个bloc
    // THREADS_PER_BLOCK=256，假设我有m=1024个点，那就是我需要4个block，一共256*4个线程去处
    // dim3 是CUDA定义的变量格式，三维
    dim3 blocks(DIVUP(m, THREADS_PER_BLOCK), b);  // blockIdx.x(col), blockIdx.y(row)
    dim3 threads(THREADS_PER_BLOCK);

    // 可函数需要用<<<blocks, threads>>>去指定调用的块数和线程数，总共调用的线程数为blocks*
                                                                    nsample, new_xyz, xyz
```

```
    // 如果cuda操作错误，则打印错误信息
    err = cudaGetLastError();
    if (cudaSuccess != err) {
        fprintf(stderr, "CUDA kernel failed : %s\n", cudaGetErrorString(err));
        exit(-1);
    }
}
```

主函数调用了__global__内核函数ball_query_kernel_fast如下：首先按照thread确定这是数组中的哪一个元素，然后针对该元素进行算法编写：此处用的算法很简单，只是循环一遍 $\boldsymbol{P}$ 中所有点，然后算一下距离。

```
// CUDA使用__global__来定义kernel
__global__ void ball_query_kernel_cuda(int b, int n, int m, float radius, int nsample,
    const float *__restrict__ new_xyz, const float *__restrict__ xyz, int *__restrict_
    // threadIdx是一个三维的向量，可以用.x .y .z分别调用其三个维度。此处我们只初始化了第一个维
    // blockIdx也是三维向量。我们初始化用的DIVUP(m, THREADS_PER_BLOCK), b分别对应blockIdx.
    // blockDim代表block的长度
    int bs_idx = blockIdx.y;
    int pt_idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (bs_idx >= b || pt_idx >= m) return;

    // 针对指针数据，利用+的操作来确定数组首地址，相当于取new_xyz[bi,ni]
    new_xyz += bs_idx * m * 3 + pt_idx * 3;
    xyz += bs_idx * n * 3;
    idx += bs_idx * m * nsample + pt_idx * nsample;

    float radius2 = radius * radius;
    float new_x = new_xyz[0];
    float new_y = new_xyz[1];
    float new_z = new_xyz[2];

    int cnt = 0;
    for (int k = 0; k < n; ++k) {
        float x = xyz[k * 3 + 0];
        float y = xyz[k * 3 + 1];
        float z = xyz[k * 3 + 2];
        // 算法很简单，循环一遍所有数据算距离
        float d2 = (new_x - x) * (new_x - x) + (new_y - y) * (new_y - y) + (new_z - z)
        if (d2 < radius2){
            if (cnt == 0){
                for (int l = 0; l < nsample; ++l) {
                    idx[l] = k;
                }
            }
            idx[cnt] = k;
            ++cnt;
            if (cnt >= nsample) break;
        }
    }
}
```

## pybind把c++ api提供给python

在pointnet_api.cpp（不一定要单独再写一个cpp文件，这样是为了代码层次清晰一点，完全可以在上面的cpp文件写如下代码）中用pybind确定c++给python的api，语法很简单，第一个参数是python调用的函数名，第二个参数是c++函数地址，第三个变量是函数description。

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("ball_query_wrapper", &ball_query_wrapper_cpp, "ball_query_wrapper_cpp");
```

### setuptools将c++和cuda编译成python所需二进制文件

在setup.py文件中用setuptools将c++和CUDA编译成python包。

最后运行python setup.py develop得到编译后的so文件保存在该目录下（如果用python setup.py install则保存在系统路径中）

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

if __name__ == '__main__':

    setup(

        .........

        ext_modules=[
            CUDAExtension(
                name="cpp_CUDA_code.pointnet_cuda",
                sources=[
                    "cpp_CUDA_code/pointnet_api.cpp",
                    "cpp_CUDA_code/ball_query.cpp",
                    "cpp_CUDA_code/ball_query_gpu.cu",
                ]
            ),
        ],
    )
```

### 在python文件中调用编译好的c++、CUDA函数

最后一步，直接在python文件里面import之前编译好的函数，然后使用它！值得注意的是，此处import的模块名**pointnet_cuda**正是我们在setup.py里定义的（见上面加粗部分），除此之外，使用的函数名**ball_query_wrapper**是我们在pybind里定义的。

```
from cpp_CUDA_code import pointnet_cuda as pointnet

if __name__ == '__main__':
    batch_size = 4
    N = 100000
    npoint = 2
    radius = 3
    nsample = 5
    xyz = torch.rand([batch_size, N, 3]) * 100  # 0~100 均匀分布
    new_xyz = torch.rand([batch_size, npoint, 3]) * 100
    idx = torch.cuda.IntTensor(batch_size, npoint, nsample).zero_()
    pointnet.ball_query_wrapper(batch_size, N, npoint, radius, nsample, new_xyz.cuda()
    print(idx)
```

写的比较潦草，如果有问题可以在评论区提问，我会继续完善。

更多的细节可以参考pytorch给的tutorial：

Custom C++ and CUDA Extensions
pytorch.org/tutorials/advanced/cpp_extension.html#w...

编辑于 2021-10-09 00:22

写评论 | CrazyVertigo 关注了作者

**4 条评论** 默认 最新

**柯南**
大佬好，最近看PointNet中的ball query，取idx是按照序号的升序从前至后取的，您的这里的实现是按照从中心由近及远来取的是吗
2021-12-18 回复 赞

**柯南** ‣ **JeffWang**
谢谢大佬，可是还是有点疑问，这样取的话比如说如果idx前512个已经完全可以取够16个，那样后面512是不是就没用到过🤔
2021-12-24 回复 赞

**JeffWang** 作者 ‣ **柯南**
这里的例子是这样的
2021-12-24 回复 赞

展开其他 1 条回复 >

**文章被以下专栏收录**

**大佬们的搬运工**

**推荐阅读**

**Python数据分析及可视化实例之基本语法**
civilpy

**python主成分分析小实例**
牧羊的男孩儿

**pytho list**
黄哥

已赞同 34 ▼ 4 条评论 分享 喜欢 收藏 申请转载 ···