PyTorch 源码解读之 cpp_extension：揭秘 C++/CUDA 算子实现和调用全流程

CV开发者都爱看的 极市平台 2022-12-17 22:00:37 发表于广东 手机阅读

作者丨OpenMMLab@知乎（已授权）

来源丨https://zhuanlan.zhihu.com/p/348555597

编辑丨极市平台

极市导读

本文结合 Python 与 C++ 各自的优点，在 PyTorch 中加入 C++ / CUDA的扩展，详细解释了 C++/CUDA 算子实现和调用全流程，让大家更好地使用工具而不为工具所束缚。 >>加入极市CV技术交流群，走在计算机视觉的最前沿

“Python 用户友好却运行效率低”，“C++ 运行效率较高，但实现一个功能代码量会远大于 Python”。平常学习工作中你是否常听到类似的说法？在 Python 大行其道的今天，你是否经常也会面临代码的瓶颈，而为运行加速而烦恼呢？“我的代码刚跑 10 步，隔壁同学的已经跑完第一个 epoch 了。”--这究竟是人性的扭曲还是科学的沦丧？荀子有言“君子性非异也，善假于物也”。本期《**源码解读**》带你走进“**Pytorch 中 (神秘) 的 C++ / CUDA 扩展**"。

- 本期主题：结合 Python 与 C++ 各自的优点，在 PyTorch 中加入 C++ / CUDA的扩展，而让我们自己更好地使用工具而不为工具所束缚。
- 代码来源：MMCV, PyTorch。

  https://github.com/open-mmlab/mmcv

  https://github.com/pytorch/pytorch
- 注：C++ / CUDA 扩展一般有”预编译“ 与 ”实时编译“ （just-in-time, JIT）模式。本期主要介绍”预编译“模式。

1. 由扩展的调用方式说起

当你想为自己的代码添加扩展进行加速时，我们可以先来看看经典的例子中是怎么处理的。对检测或分割稍有了解的同学应该知道，nms 的计算是最常见的用到了 C++ / CUDA 扩展的算子。

```python
from mmcv import _ext as ext_module
from torch.autograd import Function

def nms(boxes, scores, iou_threshold, offset=0):
    inds = NMSop.apply(boxes, scores, iou_threshold, offset)
    dets = torch.cat((boxes[inds], scores[inds].reshape(-1, 1)), dim=1)
    return dets, inds

class NMSop(torch.autograd.Function):
    @staticmethod
    def forward(ctx, bboxes, scores, iou_threshold, offset):
        inds = ext_module.nms(
            bboxes, scores, iou_threshold=float(iou_threshold), offset=of
        return inds

    @staticmethod
    def symbolic(g, bboxes, scores, iou_threshold, offset):
        pass  # onnx 转换相关
```

Function (见往期内容https://zhuanlan.zhihu.com/p/321449610)。 `NMSop` 的 `forward` 函数内核调用的是 `mmcv._ext.nms` 模块，但实际上我们在 MMCV 源码（https://github.com/open-mmlab/mmcv）中是看不到 `_ext` module 的。只有在编译好的mmcv 库 (`MMCV_WITH_OPS=True python setup.py build_ext \--inplace`) 会出现 `mmcv/_ext.cpython-xxx.so` 文件，只有这时在 Python 中运行 `import mmcv._ext` 才会成功。看来 C++ 扩展是通过 setup.py 来执行编译的。

2. setup.py -- 扩展的编译文件

基于预编译的扩展由于需要编译，而 `setup.py` 文件正是基于 `setuptools` 的编译脚本。因此一个 Python package 的扩展是可以在 `setup.py` 文件中找到其蛛丝马迹的。这里我们截取

一段mmcv的 setup.py 文件（https://github.com/open-mmlab/mmcv/blob/master/setup.py），

```
setup(
    name='mmcv',
    install_requires=install_requires,
    # 需要编译的c++/cuda扩展
    ext_modules=get_extensions(),
    # cmdclass 为python setup.py --build_ext命令指定行为
    cmdclass={'build_ext':  torch.utils.cpp_extension.BuildExtension})
```

这里可以看到 setup 函数中一个主要的参数 ext_modules ，该参数需要指定为一个 Extension 列表，代表实际需要编译的扩展。目前该参数由 get_extensions 函数获得。其中 get_extensions 函数定义如下（节选）

```
def get_extensions():
    extensions = []
    ext_name = 'mmcv._ext'
    from torch.utils.cpp_extension import (CUDAExtension, CppExtension)

    if torch.cuda.is_available():
        # CUDA编译扩展
        extra_compile_args['nvcc'] = [cuda_args] if cuda_args else []
        # 编译./mmcv/ops/csrc/pytorch文件夹中的所有文件
        op_files = glob.glob('./mmcv/ops/csrc/pytorch/*')
        extension = CUDAExtension
    else:
        # C++ 编译扩展
        op_files = glob.glob('./mmcv/ops/csrc/pytorch/*.cpp')
        extension = CppExtension
     include_path = os.path.abspath('./mmcv/ops/csrc')
     ext_ops = extension(
         name=ext_name,  # 扩展模块名
         sources=op_files,  # 扩展文件
         include_dirs=[include_path],  # 头文件地址
         define_macros=define_macros,  # 预定义宏
         extra_compile_args=extra_compile_args)  # 其他编译选项
```

```
        extensions.append(ext_ops)
    return extensions
```

在上述代码中我们终于看到了 `mmcv._ext`，该名字正是新定义的扩展的名字。由此我们便知道上文中提到的 `mmcv._ext` 模块实际上是在 setup.py 文件中指定其模块名字的。另外我们发现用于生成扩展的函数会随系统环境不同而有所区别，当系统中没有 CUDA 时会调用 `CppExtension`，且只编译所有 .cpp文件，反之则调用 `CUDAExtension`。其实 `CppExtension` 与 `CUDAExtension` 都是基于 `setuptools.Extension` 的扩展，这两个函数都额外将系统目录中的 `torch/include` 加入到 C++ 编译时的 `include_dirs` 中，另外 `CUDAExtension` 会额外将 CUDA 相关的库以及头文件加到默认编译搜索路径中。由 setup.py 文件我们还了解到送给编译的其他信息，如扩展文件的源文件地址，在 MMCV 中则是存放于 `./mmcv/ops/csrc/pytorch/` 中。其他信息如 `include_dirs`，`define_macros`，`extra_compile_args` 则会在 `torch/utils/cpp_extension.py:BuildExtension` 一并形成最终的 gcc /nvcc 的命令。

```
class BuildExtension(build_ext, object):
    # 只显示核心代码
    def build_extensions(self):
        # 检查二进制接口兼容性
        self._check_abi()

        # 注册 cuda 代码 (.cu, .cuh)
        self.compiler.src_extensions += ['.cu', '.cuh']
        def unix_wrap_compile(obj, src, ext, cc_args, extra_postargs, pp_
            try:
                original_compiler = self.compiler.compiler_so
                if _is_cuda_file(src):
                    # 对 cuda 文件调用 nvcc 命令
                    nvcc = _join_cuda_home('bin', 'nvcc')
                    self.compiler.set_executable('compiler_so', nvcc)
                    if isinstance(cflags, dict):
                        cflags = cflags['nvcc']
                    cflags = COMMON_NVCC_FLAGS + ['--compiler-options',
                                                  "'-fPIC'"] + cflags + _
                elif isinstance(cflags, dict):
                    # 默认 c++ 程序的 flags
                    cflags = cflags['cxx']
```

```
                # 强制性使用 --std=c++11
                if not any(flag.startswith('-std=') for flag in cflags):
                    cflags.append('-std=c++11')
                # c++ / cuda 程序编译入口
                original_compile(obj, src, ext, cc_args, cflags, pp_opts)
            finally:
                # 将之前覆盖的默认编译器还原
                self.compiler.set_executable('compiler_so', original_comp
```

以上过程了解清楚之后我们运行 `MMCV_WITH_OPS=True python setup.py build_ext \--inplace` 指令。

```
/usr/local/cuda-10.0/bin/nvcc -DMMCV_WITH_CUDA -I/home-to-/mmcv/mmcv/ops/c

gcc -pthread -B compiler_compat -Wl,--sysroot=/ -Wsign-compare -DNDEBUG -g
...
```

在上述运行结果中我们可以看到

1. 编译器对 CUDA 文件自动调用 nvcc 而对 .cpp 文件则调用 gcc
2. 被 `CUDAExtension` 包装过后系统自动加入了 Python，PyTorch， CUDA 等库中的头文件以及库地址，系统架构信息(`-gencode`)与编译优化信息(`-O3` 等)
3. 通过 `-DTORCH_EXTENSION_NAME=_ext` 将 `TORCH_EXTENSION_NAME` 宏赋值为 `_ext` 。这看来也绝非是闲来之笔，欲知后事如何，我们看下一节分解

3. PYBIND11_MODULE -- Python 与 C++ 的桥梁

上文说到通过 setup.py 我们编译了扩展文件。可是目前仍然有个疑问，为什么编译出来的 C++ / CUDA 二进制文件可以在 Python 中直接被调用呢？再次检测编译的所有文件，发现其中有个文件 pybind.cpp （https://github.com/open-mmlab/mmcv/blob/master/mmcv/ops/csrc/pytorch/pybind.cpp）十分可疑，其打开后大致形式如下。

```
#include <torch/extension.h>
// 函数声明，具体实现在其他文件
Tensor nms(Tensor boxes, Tensor scores, float iou_threshold, int offset);
```

```
4
5  PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
6    m.def("nms", &nms, "nms (CPU/CUDA) ", py::arg("boxes"), py::arg("scores"
7          py::arg("iou_threshold"), py::arg("offset"));
8  }
```

这里 PYBIND11_MODULE 是一个宏，定义在 pybind11 库中(见pybind11/include/pybind11/pybind11.h) （https://github.com/pybind/pybind11/blob/master/include/pybind11/pybind11.h） 。而 pybind11 是一个用来在 C++ 代码中创建 Python的连接的库。找到了源头，我们进一步分析。

这里 PYBIND11_MODULE 的作用是为 C++ 代码接入 Python 解释器提供入口。以上述代码为例， TORCH_EXTENSION_NAME 正是在上文 gcc 编译过程中出现的宏，对应为extension的 name 变量。因此在这里会被解释成 _ext （注意没有双引号） 。 m 则代表 TORCH_EXTENSION_NAME 所对应的模块实例（实际上可以指定为任何名字）。 {} 中的每个 m.def 都定义了一个 _ext 的成员函数，其一般形式为 m.def("函数名", 具体 C++ 实现的函数指针, "文档", 参数列表) 。通过这种形式， nms 也就顺利地成为了 mmcv._ext 的成员函数，其具体实现为已经定义好的 nms 函数（对这个函数的分析我们会在下节讲到）。在 Python 中也就可以运行 from mmcv._ext import nms 了。如果对这里的定义仍然不清楚，我们可以把该宏用 C++ 编译器展开一下：

```
1  Tensor nms(Tensor boxes, Tensor scores, float iou_threshold, int offset);
2  static void pybind11_init__ext(pybind11::module &);
3  extern "C" __attribute__ ((visibility("default"))) PyObject *PyInit__ext(
4  {
5      // 省略部分代码
6      auto m = pybind11::module("_ext");  // m 变量的初始化是在宏内部
7      try { pybind11_init__ext(m); return m.ptr(); }
8  }
9  void pybind11_init__ext(pybind11::module &m) {
10    // 添加成员函数
11    m.def("nms", &nms, "nms (CPU/CUDA) ", py::arg("boxes"), py::arg("score
12        py::arg("iou_threshold"), py::arg("offset"));
13 }
```

其中 `PyObject *PyInit_` 定义在 `Python.h` 中，正是 C++ 中声明 Python module 的官方方法（可见官方 Python 文档）。这里 `PyInit_` 后接的 `_ext` 其实就是 `TORCH_EXTENSION_NAME` 宏解释得到。意味着新声明了一个 名为 `_ext` 的 Python module。

4. cpp/cu文件 -- 算子的具体实现

通过对 `PYBIND11_MODULE` 的分析后，我们了解了 `mmcv._ext.nms` 具体的实现是一个声明为 `Tensor nms(Tensor boxes, Tensor scores, float iou_threshold, int offset);` 的函数。该函数定义在 mmcv/ops/csrc/pytorch/nms.cpp 中（https://github.com/open-mmlab/mmcv/blob/master/mmcv/ops/csrc/pytorch/nms.cpp）

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

Tensor nms(Tensor boxes, Tensor scores, float iou_threshold, int offset)
  if (boxes.device().is_cuda()) {
    // cuda 实现
    return nms_cuda(boxes, scores, iou_threshold, offset);
  } else {
    // c++ 实现
    return nms_cpu(boxes, scores, iou_threshold, offset);
  }
}
```

可以看到这时实际的实现方式针对设备的不同分为了 `nms_cuda` 与 `nms_cpu` 两种。这里我们先来看 cpp 的实现。

4.1 CPP 算子实现

```
#include <torch/extension.h>
using namespace at; // 适当改写
Tensor nms_cpu(Tensor boxes, Tensor scores, float iou_threshold, int offs
  // 仅显示核心代码
  for (int64_t _i = 0; _i < nboxes; _i++) {
    // 遍历所有检测框，称为主检测框
    if (select[_i] == false) continue;
    for (int64_t _j = _i + 1; _j < nboxes; _j++) {
```

```
        // 对每个主检测框，遍历其他检测框，称为次检测框
        // 这里只用遍历上三角元素即可，节省计算
        if (select[_j] == false) continue;
        auto ovr = inter / (iarea + areas[j] - inter);
        // 如果次检测框与主检测框 iou 过大，则去除该次检测框
        if (ovr >= iou_threshold) select[_j] = false;
      }
    }
    return order_t.masked_select(select_t);
}
```

以上即为 `nms_cpu` 的核心代码，对该算法想要有进一步了解的同学可以参看源码 。这里出现了两个 `for` 循环，实现上这正是我们希望实现 `nms` 的 C++ / CUDA 扩展的原因。对于有一定 C++ 基础的同学来说代码应该较好理解 （注意这里 `int64_t` 可理解为 `C99` 规约的为支持不同平台的int64类型的 `typedef` 定义，可直接理解为 `int64` ），但这里同时也出现了一些新的变量类型，最典型的是 `Tensor` 数据类型。

其实这里 `Tensor` 数据类型由 `torch/extension.h` 支持，来源于 pytorch 中 C++ API 中三大 namespace（ `at` ， `torch` 与 `c10`)中的 `at` 。

> 小知识点： `at` ， `torch` 与 `c10` 这三个命名空间中 `at` 代表 ATen (A Tensor Library)，负责声明和定义Tensor运算相关的逻辑，是pytorch扩展c++接口中最常用到的命名空间， `c10` (Caffe Tensor Library)其实是 ATen 的基础，包含了PyTorch的核心抽象、Tensor和Storage数据结构的实际实现。 `torch` 命名空间下定义的 Tensor 相比于ATen 增加自动求导功能，但 c++ 扩展中一般不常见）

该类型功能十分强大，基本支持 PyTorch 中 Tensor 的所有运算方式（如 +, -, *, /, >, < 等运算符， `.view` ， `.reshape` ， `.unsqueeze` 等维度变化功能等）。Tensor 的 API 接口可见官方链接。当然除了 Tensor 类型外 `at` 命名空间也支持几乎所有和 Tensor 有关的函数 （如 `at::ones` ， `at::zeros` ， `at::where` 等）， ATen 的 API 接口可见官方链接（https://pytorch.org/cppdocs/api/classat_1_1_tensor.html#exhale-class-classat-1-1-tensor ） 。 基本上只要在程序中加入了 `#include <torch/extension.h>` 就可以在 C++ 中调用所有 PyTorch 支持的功能。

4.2 CUDA 算子实现

4.2.1 (番外篇) CUDA 编程基础

该部分内容部分参考 CUDA编程入门极简教程，感兴趣的同学可以看原文

https://blog.csdn.net/xiaohu2022/article/details/79599947。

基本概念

CUDA 是建立在 NVIDIA GPU上的一个通用并行计算平台和编程模型，CUDA编程可以利用GPUs 的并行计算引擎来更加高效地解决比较复杂的计算难题。CUDA 的语法和 C++ 大多部分情况下是一致的，其默认文件名后缀是 .cu，默认头文件名后缀是 .cuh。CUDA 编程是异构的，即CPU负责处理逻辑复杂的串行程序，而 GPU 重点处理数据密集型的并行计算程序，从而发挥最大功效。其中 CPU 所在位置称为为主机端（host），而 GPU 所在位置称为设备端（device）。

CUDA程序的设计流程

一般而言，CUDA 程序执行会依照如下流程：

1. 分配 host 内存，并进行数据初始化
2. 分配 device 内存，并从 host 将数据拷贝到 device 上
3. 调用 CUDA 的核函数在 device 上完成指定的运算
4. 将 device 上的运算结果拷贝到 host 上
5. 释放 device 和 host 上分配的内存

而对 PyTorch 的 CUDA 扩展来说， CUDA 扩展传入和传出的 Tensor 都已经在 GPU 上，因此这里的 5 个步骤只有第 3 步了，这会为我们省下比较宝贵的时间而将更多注意力放到具体的程序实现上。

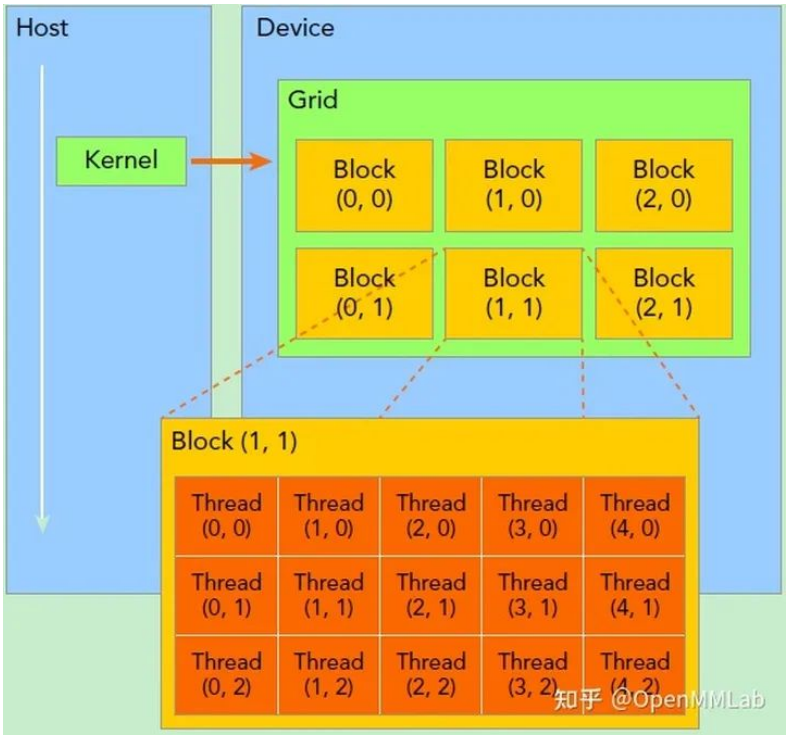
CUDA 中指定函数设备关键字

由于 CUDA 编程为异步，因此函数的定义与调用很可能不在同一个 device 上面，因此 CUDA 中骑过增加额外函数类型来规约函数的定义与调用设备。- `__global__` ： 在 device 上执行，从 host 中调用（一些特定的 GPU 也可以从 device 上调用），返回类型必须是 void，不支持可变参数参数，不能成为类成员函数。注意用 `__global__` 定义的 kernel 是异步的，这意味着 host 不会等待 kernel 执行完就执行下一步。- `__device__` ： 在 device 上执行，单仅可以从 device 中调用，不可以和 `__global__` 同时用。- `__host__` ： 在 host 上执行，仅可以从 host 上调用，一般省略不写，不可以和 `__global__` 同时用，但可和 `__device__` ，此时函数会在 device 和 host 都编译。

CUDA 中线程逻辑架构形式

一旦一个 kernel 在 device 上执行，device 上很多经量级的线程会被启动，一个 kernel 所启动的所有线程分成两级架构。所有线程归为一个网格（grid），同一个网格上的线程共享相同的全局内存空间，而网格又可以分为很多线程块（block），一个线程块里面包含很多线程。线程两层组织结构如下图所示，这是一个 gird 和 block 均为 2-dim 的线程组织。grid 和 block 都是定义为 dim3 类型的变量，dim3 可以看成是包含三个无符号整数（x，y，z）成员的结构体变量，在定义时，可定义为一维或二维，剩下维度缺少值为 1。当然这里的 grid, block 层次划分实际上只是逻辑层次，线程在 GPU 中的流处理器 (SM) 中是用“线程束”管理的，一个线程束包含 32 个线程。因此一般在设计 block 时要保证其线程个数为 32 的整数倍。

> 为了更好地理解这里的线程架构，我们可以直接将一个kernel 开辟的所有线程理解为一个小区，这个小区就被称为 grid，而该小区 (grid) 是由不同楼栋 (block)组成的，每个楼栋 (block)有其在小区内的三维坐标 (x, y, z)。在每个楼栋中的所有线程按其在该 block 的三维坐标 (x, y, z)来进行索引。



CUDA 中核函数调用

核函数 (kernel) 是在 device上线程中并行执行的函数，核函数用 `__global__` 符号声明，在调用时需要用 `<<<grid, block>>>` 来指定 kernel 要执行的线程数量。这里 `grid` 与 `block` 都需要提前定义好，在 CUDA 中，每一个线程都要执行核函数，并且每个线程会分配一个唯一的线程号 thread ID，这个 ID 值可以通过核函数的内置变量 threadIdx 来获得。下面代码即为在上图线程逻辑架构下的核函数调用方式。

```
dim3 grid(3, 2);
dim3 block(5, 3);
kernel_fun<<< grid, block >>>(prams...);
```

CUDA 中核函数编写

核函数调用过程中将需要并行执行的部分用不同的的线程进行完成。因此在实际 CUDA 的核函数中，系统定义了两个两个内置的坐标变量 `blockIdx` 与 `threadIdx` 来唯一标识一个线程，它们都是 dim3 类型变量（包括 `x`，`y`，`z` 成员），其中 `blockIdx` 指明线程所在grid中的位置，而 `threaIdx` 指明线程所在block中的位置，这里 grid 与 block 正是在核函数调用过程中定义好的，在核函数的定义中也有 dim3 类型变量 `gridDim` 与 `blockDim` 来分别指定 grid 与 block 的维度。如下核函数为矩阵相加的 CUDA 代码。程序执行过程中会按 `blockIdx` 与 `threadIdx` 的坐标信息将该核函数分配给不同的线程来完成，因此实现高效并行化计算。以下为一个较为典型的矩阵相加的核函数设计。

```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
```

至此我们完成了一般 CUDA 算子实现的基础，在下一小节中我们再来分析 nms CUDA 算子的实例。

4.2.2 CUDA 算子实例

```
// 以下程序适当改写，只显示核心代码
#include <torch/extension.h>
using namespace at;


#define DIVUP(m, n) ((m) / (n) + ((m) % (n) > 0))
```

```
int const threadsPerBlock = sizeof(unsigned long long int) * 8; // 64
__device__ inline bool devIoU(float const *const a, float const *const b
                              const int offset, const float threshold) {
  // 定义在 device 上的函数，用于返回iou
  // 定义省略
}

__global__ void nms_cuda(const int n_boxes, const float iou_threshold,
                         const int offset, const float *dev_boxes,
                         unsigned long long *dev_mask) {
  // 核函数
  const int row_start = blockIdx.y;  // block 在 grid 中的位置
  const int col_start = blockIdx.x;  // block 在 grid 中的位置
  const int tid = threadIdx.x;  // thread ID (0-63)

  if (row_start > col_start) return;

  const int row_size =
      fminf(n_boxes - row_start * threadsPerBlock, threadsPerBlock);
  const int col_size =
      fminf(n_boxes - col_start * threadsPerBlock, threadsPerBlock);

  //  Shared memory 只会被一个 block 中的所有线程共享
  __shared__ float block_boxes[threadsPerBlock * 4];
  if (tid < col_size) {
    block_boxes[tid * 4 + 0] =
        dev_boxes[(threadsPerBlock * col_start + tid) * 4 + 0];
    block_boxes[tid * 4 + 1] =
        dev_boxes[(threadsPerBlock * col_start + tid) * 4 + 1];
    block_boxes[tid * 4 + 2] =
        dev_boxes[(threadsPerBlock * col_start + tid) * 4 + 2];
    block_boxes[tid * 4 + 3] =
        dev_boxes[(threadsPerBlock * col_start + tid) * 4 + 3];
  }
  __syncthreads();

  if (tid < row_size) {
    const int cur_box_idx = threadsPerBlock * row_start + tid;
    const float *cur_box = dev_boxes + cur_box_idx * 4;
    int i = 0;
```

```
    unsigned long long int t = 0;
    int start = 0;
    if (row_start == col_start) {
      start = tid + 1;  // 每个 bbox 只需要和上三角元素计算（节省计算）
    }
    for (i = start; i < col_size; i++) {
      if (devIoU(cur_box, block_boxes + i * 4, offset, iou_threshold)) {
        t |= 1ULL << i;
      }
    }
    dev_mask[cur_box_idx * gridDim.y + col_start] = t;
  }
}

Tensor NMSCUDAKernelLauncher(Tensor boxes, Tensor scores, float iou_thre
                             int offset) {
  // cuda 程序入口
  at::cuda::CUDAGuard device_guard(boxes.device());  // 指定默认的显卡

  auto order_t = std::get<1>(scores.sort(0, /*descending=*/true));
  auto boxes_sorted = boxes.index_select(0, order_t);

  int boxes_num = boxes.size(0);
  // 这里将
  const int col_blocks = DIVUP(boxes_num, threadsPerBlock);
  // mask 是用来存储 bboxes 之间两两 iou 是否大于阈值的一个 mask
  // 本来长度应该是 (boxes_num, boxes_num)，但这里采用位存储的方式，一个 LongLo
  // 个 bool 值，因此存储空间可以缩小64倍，只用开辟 (boxes_num, boxes_num/64)长

  Tensor mask =
      at::empty({boxes_num, col_blocks}, boxes.options().dtype(at::kLong
  dim3 blocks(col_blocks, col_blocks);
  dim3 threads(threadsPerBlock);  // 每 64 个线程放到一个 block 中，遍历一个L
  cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  // 更完整的核函数调用 <<< blocks, threads, shared_memory, stream>>>
  nms_cuda<<<blocks, threads, 0, stream>>>(
      boxes_num, iou_threshold, offset, boxes_sorted.data_ptr<float>(),
      (unsigned long long*)mask.data_ptr<int64_t>());

  at::Tensor mask_cpu = mask.to(at::kCPU);
```

```
  unsigned long long* mask_host =
      (unsigned long long*)mask_cpu.data_ptr<int64_t>();

  std::vector<unsigned long long> remv(col_blocks);
  memset(&remv[0], 0, sizeof(unsigned long long) * col_blocks);

  at::Tensor keep_t =
      at::zeros({boxes_num}, boxes.options().dtype(at::kBool).device(at:
  bool* keep = keep_t.data_ptr<bool>();

  for (int i = 0; i < boxes_num; i++) {
    int nblock = i / threadsPerBlock;
    int inblock = i % threadsPerBlock;

    if (!(remv[nblock] & (1ULL << inblock))) {
      keep[i] = true;
      // set every overlap box with bit 1 in remv
      unsigned long long* p = mask_host + i * col_blocks;
      for (int j = nblock; j < col_blocks; j++) {
        remv[j] |= p[j];
      }
    }
  }

  AT_CUDA_CHECK(cudaGetLastError());
  return order_t.masked_select(keep_t.to(at::kCUDA));
}
```

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

点击阅读原文进入CV社区

获取更多技术干货

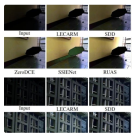
阅读原文

喜欢此内容的人还喜欢

极市平台

ICCV23｜将隐式神经表征用于低光增强，北大张健团队提出NeRCo

极市平台



ICCV 2023｜南开程明明团队提出适用于SR任务的新颖注意力机制（已开源）

极市平台