TensorRT OnnX 插件

- 该文档受众是对tensorRT有所了解的人，如果不了解可能会比较蒙圈

1. python代码部分，插件op的处理，并顺利导出

```
import torch
import torch.nn.functional as F
import torch.nn as nn
import json

class HSwishImplementation(torch.autograd.Function):

    # 主要是这里，对于autograd.Function这种自定义实现的op，只需要添加静态方法symbolic即可，除了g以外的参数应与forward函数的除ctx以外完全一样
    # 这里演示了input->作为tensor输入，bias->作为参数输入，两者将会在tensorRT里面具有不同的处理方式
    # 对于附加属性（attributes），以 "名称_类型简写" 方式定义，类型简写，请参考：torch/onnx/symbolic_helper.py中_parse_arg函数的实现【from torch.onnx.symbolic_helper import _parse_arg】
    # 属性的定义会在对应节点生成attributes，并传给tensorRT的onnx解析器做处理
    @staticmethod
    def symbolic(g, input, bias):
        return g.op("HSwish", input, bias, info_s="string attribute", kernel_size_i=3, eps_f=3e-2)

    @staticmethod
    def forward(ctx, i, bias):
        ctx.save_for_backward(i)
        return i * F.relu6(i + 3) / 6

     # 这里省略了backward

class MemoryEfficientHSwish(nn.Module):
    def __init__(self):
        super(MemoryEfficientHSwish, self).__init__()

        # 这里我们假设有bias作为权重参数
        self.bias = nn.Parameter(torch.zeros((5, 3, 3, 1)))
        self.bias.data.fill_(3.15)

    def forward(self, x):
        # 我们假设丢一个bias进去
        return HSwishImplementation.apply(x, self.bias)

class FooModel(torch.nn.Module):
    def __init__(self):
        super(FooModel, self).__init__()
        self.hswish = MemoryEfficientHSwish()

    def forward(self, input1, input2):
        return input2 + self.hswish(input1)
```

```
dummy_input1 = torch.zeros((1, 3, 3, 3))
dummy_input2 = torch.zeros((1, 1, 3, 3))
model = FooModel()

# 这里演示了2个输入的情况，实际上你可以自己定义几个输入
torch.onnx.export(model, (dummy_input1, dummy_input2), 'test.onnx',
verbose=True)
```

2. onnx部分

- 下载：https://github.com/onnx/onnx-tensorrt/tree/master/third_party

  1. 根据这个链接找到当前onnx-tensorrt所使用和引用的onnx版本（因为onnx本身自己会更新，导致不一样）
  2. 本身onnx是一个独立的库，解析器依赖这里的onnx库，即libonnx_proto.a
  3. 编译onnx前请安装protobuf，具体步骤去搜索有的，一般是下载代码编译，要求Protobuf v3.8.x：

  ```
  # 这是protobuf的下载编译参考
  sudo apt-get install autoconf automake libtool curl
  git clone https://github.com/google/protobuf.git
  cd protobuf
  git submodule update --init --recursive
  ./autogen.sh
  ./configure
  make all -j16
  sudo make install
  sudo ldconfig
  ```

  4. 编译onnx

  ```
  cd onnx && mkdir build && cd build
  cmake .. -DONNX_NAMESPACE=onnx2trt_onnx
  make onnx_proto -j16
  cp ../onnx/onnx_pb.h onnx/onnx_pb.h
  cp ../onnx/onnxifi.h onnx/onnxifi.h
  ```

  这时候，我们得到build/libonnx_proto.a，以及build/onnx底下的各类头文件

3. onnx-tensorRT部分

- 代码下载：https://github.com/onnx/onnx-tensorrt
- 使用方式：

  1. 编译为so/dll
     - 该方法直接下载项目后，cmake编译即可，生成的so或者dll引入到项目中替代tensorRT发布时的libnvonnxparser.so或者dll
     - 该方法比较简单，但是对于需要定制修改调试时比较不方便
  2. 融入代码到工程
     - 该方法是将该代码融入到自己的工程中使用，此时已不需要libnvonnxparser.so或者dll
     - 该方法难度比较大，但是调试维护等比较方便
     - 该方法是这里推荐的做法

- 按照上面是使用方法2操作：

1. 由于这个项目还涉及到python相关api，会依赖python库，如果没有这个需求，可以只保留必要的代码部分。或者你也可以把全部代码加入到自己项目中，以下是我抽取的部分cpp和hpp

```
builtin_op_importers.cpp
builtin_op_importers.hpp
ImporterContext.hpp
ModelImporter.cpp
ModelImporter.hpp
NvOnnxParser.cpp
NvOnnxParser.h
onnx2trt.hpp
onnx2trt_utils.cpp
onnx2trt_utils.hpp
OnnxAttrs.cpp
OnnxAttrs.hpp
onnx_utils.hpp
ShapedWeights.cpp
ShapedWeights.hpp
ShapeTensor.cpp
ShapeTensor.hpp
Status.hpp
TensorOrWeights.hpp
toposort.hpp
trt_utils.hpp
utils.hpp
```

- 配置头文件目录：-I/datav/newbb/build/tensorRT7.0Base/onnx/build，代码包含头文件是#include <onnx/onnx_pb.h>
- 配置库文件目录：-L/datav/newbb/build/tensorRT7.0Base/onnx/build，引用到了libonnx_proto.a
- 编译你的项目
    1. 一定要添加的编译选项：-DONNX_ML 和 -DONNX_NAMESPACE=onnx2trt_onnx
    2. 添加protobuf和onnx的库引用：-pthread -lprotobuf -lonnx_proto，如果cmake可以直接find_package查找protobuf，这里一定要注意，如果makefile，一定要-pthread后-lprotobuf，使用多线程库
- 此时应该是可以编译完成自己的工程，还没有涉及到修改

5. 写自定义op实现

1. 他的入口是从NvOnnxParser.cpp的createNvOnnxParser_INTERNAL开始
2. 然后调用IParser->parseFromFile
3. 然后转调到ModelImporter::parseFromFile 开始做解析
4. 然后调用到ModelImporter::parse
5. 然后是ModelImporter::importModel
6. 然后是ModelImporter.cpp ::importInputs
    - ModelImporter.cpp ::importInput
    - 这里是控制输入的，如果想要对onnx的输入尺寸做修改，请修改importInput函数里面的trt_dims即可，放到其他地方是无效的
7. 然后是ModelImporter.cpp ::parseGraph

- 这里他会调用getBuiltinOpImporterMap函数，而getBuiltinOpImporterMap函数实际上是builtin_op_importers.hpp定义的，builtin_op_importers.cpp实现的，也就是所有自定义OP汇集的地方，通过getBuiltinOpImporterMap传递给解析器注册到tensorRT联系起来
- 所有的插件，都将以如下形式出现，只要按照名字和版本注册了，那么当你加载onnx的时候，都会被认识

```
DEFINE_BUILTIN_OP_IMPORTER(HSwish)
{
    layer = importPluginFromRegistry(ctx, name, "1", node.name(), f);
    RETURN_ALL_OUTPUTS(layer);
}
```

- 而这里的importPluginFromRegistry函数，则是在调用getPluginRegistry，通过registry->getPluginCreator(name, version, namespace)获取到插件的creator，然后执行creator->createPlugin，实现获得插件实例nvinfer1::IPluginV2*
- 对于自己实现一个插件，需要实现一个插件类（例如叫HSwish），继承自nvinfer1::IPluginV2Ext，实现他的虚函数接口，例如序列化反序列化enqueue（执行）等等
  - 插件分为编译时执行流程 和 推理时执行流程
  - 编译时引擎会调用插件的配置函数supportsFormat，查询插件的支持能力，例如插件是否支持fp32、fp16、int8，从而针对性选择效率最好的插件格式后，通过configureWithFormat函数通知插件固定支持方式，因此调试插件时，先把插件的每个执行流程打印出来
  - 推理时，主要信息是通过反序列化得到的，他不会在调用各种配置查询函数以及输出数量等等
- 然后实现一个creator类（例如叫HSwishCreator），继承自nvinfer1::IPluginCreator
- 通过REGISTER_TENSORRT_PLUGIN(HSwishCreator)宏，实现一个插件注册过程
- 这里就是一个插件的例子：TensorRT-7.0.0.11/samples/sampleUffPluginV2Ext/sampleUffPluginV2Ext.cpp
- **注意对于onnx，采用IPluginV2Ext实现（注意不是IPluginV2，IPluginV2不支持explicit batch），通过creator和REGISTER_TENSORRT_PLUGIN进行注册是比较推荐的，请不要使用PluginFactory和IPluginExt或者IPlugin等等，他会被抛弃，并且在明确batch_size（explicit batch）问题上会报错**

8. 一个插件的实现案例

```
DEFINE_BUILTIN_OP_IMPORTER(HSwish)
{
    //由于宏定义的原因，默认是可以使用inputs，表示输入的参数
    // 对于python案例：
    // @staticmethod
    // def symbolic(g, input, bias):
    //      return g.op("HSwish", input, bias, info_s="string attribute",
kernel_size_i=3, eps_f=3e-2)

    //这里的input是tensor，而bias是weight
    //   因此inputs.at(0).is_tensor() = true
    //   因此inputs.at(1).is_weights() = true
    // inputs是一个vector<TensorOrWeights>类型的，可以通过函数转换为ITensor和
ShapedWeights

    std::vector<nvinfer1::ITensor*> inputTensors;
    std::vector<onnx2trt::ShapedWeights> weights;
```

```
    for(int i = 0; i < inputs.size(); ++i){
        auto& item = inputs.at(i);
        if(item.is_tensor()){
            nvinfer1::ITensor* input = &convertToTensor(item, ctx);
            inputTensors.push_back(input);
        }else{
            weights.push_back(item.weights());
        }
    }

    // 获取onnx当前节点的attribute，对应的就是python上提到的info_s，
kernel_size_i，eps_f等函数
    OnnxAttrs attrs(node, ctx);
    auto info = attrs.get<std::string>("info", "");
    auto kernel_size = attrs.get<int>("kernel_size", 0);
    auto eps = attrs.get<float>("eps", 0);

    // 这个是tensorRT支持的插件属性传递的东西，实际上很费劲，因此换成了自己封装的
pluginInit方式传递
    std::vector<nvinfer1::PluginField> f;
    std::string name = "HSwish";
    // Create plugin from registry
    // 这里的ONNXPlugin::TRTPlugin是我自己封装的一个类型，继承自IPluginV2Ext，你可
以忽略或者自己封装
    ONNXPlugin::TRTPlugin* plugin =
(ONNXPlugin::TRTPlugin*)importPluginFromRegistry(ctx, name, "1",
node.name(), f);

    if(plugin == nullptr){
        printf("%s plugin was not found in the plugin registry!",
name.c_str());
        ASSERT(false, ErrorCode::kUNSUPPORTED_NODE);
    }

    // 这里的TRTInfer::Tensor也是自己封装的，把ShapedWeights的值获取出来并传递过去
    std::vector<std::shared_ptr<TRTInfer::Tensor>> weightTensors;
    for(int i = 0; i < weights.size(); ++i){
        auto& weight = weights[i];
        std::vector<int> dims(weight.shape.d, weight.shape.d +
weight.shape.nbDims);
        std::shared_ptr<TRTInfer::Tensor> dweight(new
TRTInfer::Tensor(dims));

        if(weight.type != ::ONNX_NAMESPACE::TensorProto::FLOAT){
            LOG(LFATAL) << "unsupport weight type: " << weight.type;
        }

        memcpy(dweight->cpu(), weight.values, dweight->bytes());
        weightTensors.push_back(dweight);
    }

    // 这里是直接调用自己封装的plugin初始化函数
    plugin->pluginInit(info, weightTensors);

    // 然后把插件添加到network中，他的返回值是：IPluginV2Layer*交给
RETURN_ALL_OUTPUTS实现op的注册整个流程
    auto out = ctx->network()->addPluginV2(inputTensors.data(),
inputTensors.size(), *plugin);
```

```
    RETURN_ALL_OUTPUTS(out);
}
```

结尾

有了权重和info等信息，就可以愉快的写插件了
如果觉得麻烦，可以参考tensorRTIntegrate项目中的插件部分代码，目前使用的是pluginFactory。关于onnx写插件的，未来很快就会更新上去

2020年4月1日 17:27:51