TensorRT基础-概述

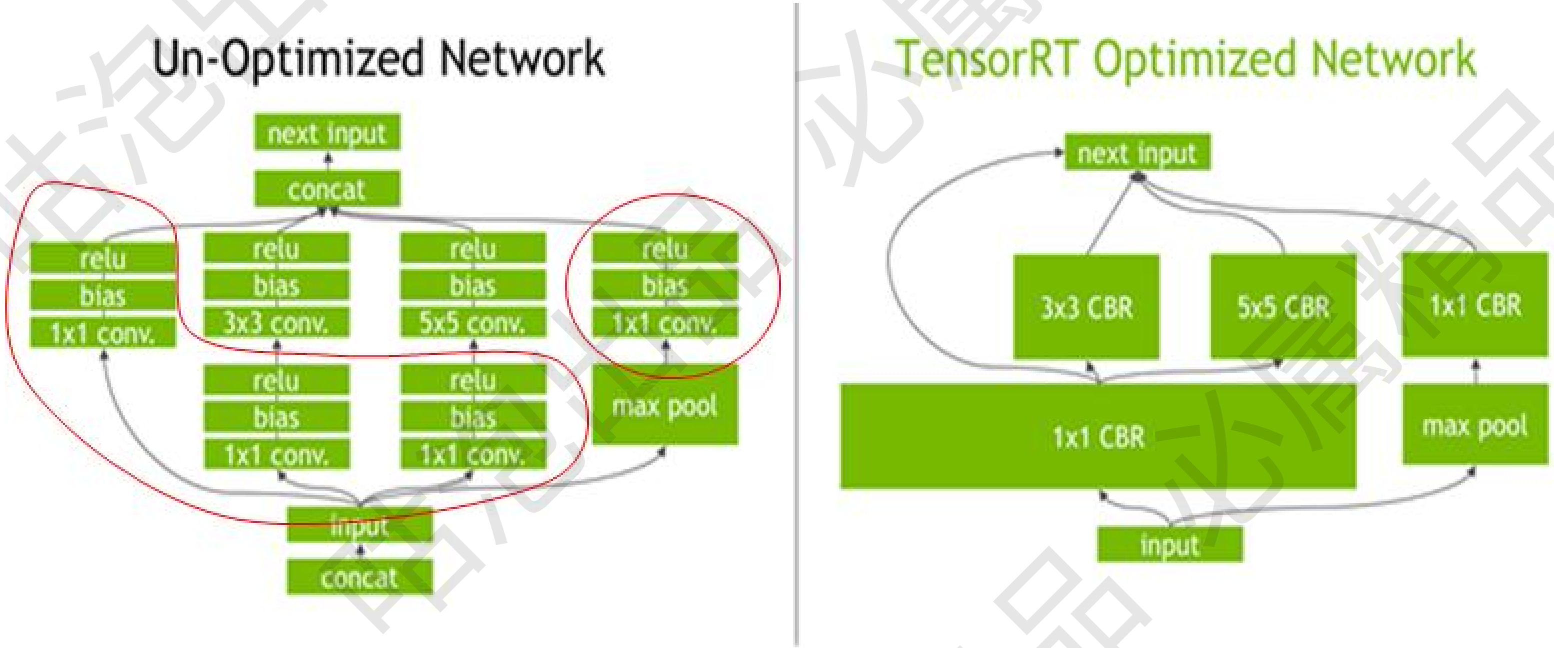
TensorRT基础

1. TensorRT的核心在于对模型算子的优化（合并算子、利用GPU特性选择特定核函数等多种策略），通过tensorRT，能够在Nvidia系列GPU上获得最好的性能
2. 因此tensorRT的模型，需要在目标GPU上实际运行的方式选择最优算法和配置
3. 也因此tensorRT生成的模型只能在特定条件下运行（编译的trt版本、cuda版本、编译时的GPU型号）
4. 主要知识点，是**模型结构定义方式、编译过程配置、推理过程实现、插件实现、onnx理解**

不跑跑，我哪知道怎么才最快

https://www.cnblogs.com/qccz123456/p/11767858.html
https://www.bilibili.com/video/BV1Xw411f7FW/

TensorRT基础



Un-Optimized Network
next input
concat
relu
bias
1x1 conv.
relu
bias
3x3 conv.
relu
bias
5x5 conv.
relu
bias
1x1 conv.
relu
bias
1x1 conv.
relu
bias
1x1 conv.
max pool
input
concat
TensorRT Optimized Network
next input
3x3 CBR
5x5 CBR
1x1 CBR
1x1 CBR
max pool
input

C++接口

TensorRT提供的C++接口

```
// Create input tensor of shape { 1, 1, 28, 28 }
ITensor* data = network->addInput(
    mParams.inputTensorNames[0].c_str(), DataType::kFLOAT, Dims3{1, mParams.inputH, mParams.inputW});
ASSERT(data);

// Create scale layer with default power/shift and specified scale parameter.
const float scaleParam = 0.0125f;
const Weights power{DataType::kFLOAT, nullptr, 0};
const Weights shift{DataType::kFLOAT, nullptr, 0};
const Weights scale{DataType::kFLOAT, &scaleParam, 1};
IScaleLayer* scale_1 = network->addScale(*data, ScaleMode::kUNIFORM, shift, scale, power);
ASSERT(scale_1);

// Add convolution layer with 20 outputs and a 5x5 filter.
IConvolutionLayer* conv1 = network->addConvolutionNd(
    *scale_1->getOutput(0), 20, Dims{2, {5, 5}}, mWeightMap["conv1filter"], mWeightMap["conv1bias"]);
ASSERT(conv1);
conv1->setStride(DimsHW{1, 1});

// Add max pooling layer with stride of 2x2 and kernel size of 2x2.
IPoolingLayer* pool1 = network->addPoolingNd(*conv1->getOutput(0), PoolingType::kMAX, Dims{2, {2, 2}});
ASSERT(pool1);
pool1->setStride(DimsHW{2, 2});

// Add second convolution layer with 50 outputs and a 5x5 filter.
IConvolutionLayer* conv2 = network->addConvolutionNd(
    *pool1->getOutput(0), 50, Dims{2, {5, 5}}, mWeightMap["conv2filter"], mWeightMap["conv2bias"]);
ASSERT(conv2);
conv2->setStride(DimsHW{1, 1});

// Add second max pooling layer with stride of 2x2 and kernel size of 2x3>
IPoolingLayer* pool2 = network->addPoolingNd(*conv2->getOutput(0), PoolingType::kMAX, Dims{2, {2, 2}});
ASSERT(pool2);
pool2->setStride(DimsHW{2, 2});

// Add fully connected layer with 500 outputs.
IFullyConnectedLayer* ip1
    = network->addFullyConnected(*pool2->getOutput(0), 500, mWeightMap["ip1filter"], mWeightMap["ip1bias"]);
ASSERT(ip1);

// Add activation layer using the ReLU algorithm.
IActivationLayer* relu1 = network->addActivation(*ip1->getOutput(0), ActivationType::kRELU);
ASSERT(relu1);
```

TensorRT-8.0.1.6/samples/sampleMNISTAPI/sampleMNISTAPI.cpp

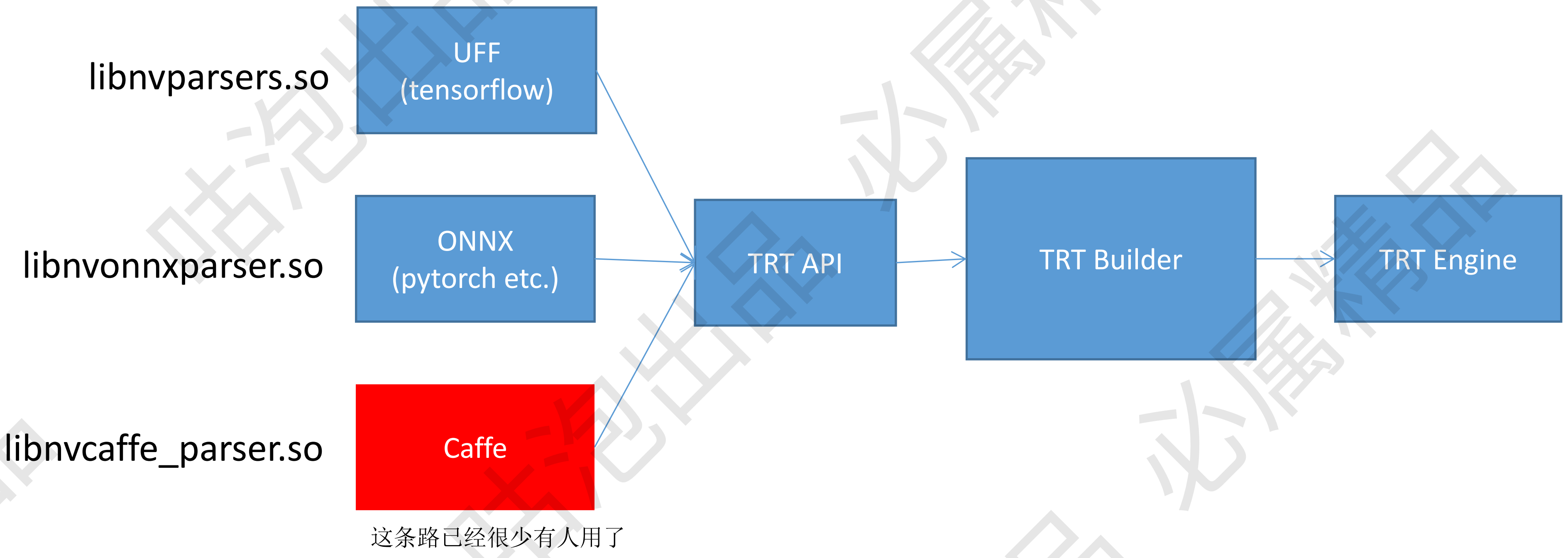
Python接口

TensorRT提供的Python接口

```
def populate_network(network, weights):
    # Configure the network layers based on the weights provided.
    input_tensor = network.add_input(name=ModelData.INPUT_NAME, dtype=ModelData.DTYPE, shape=ModelData.INPUT_SHAPE)

    conv1_w = weights['conv1.weight'].numpy()
    conv1_b = weights['conv1.bias'].numpy()
    conv1 = network.add_convolution(input=input_tensor, num_output_maps=20, kernel_shape=(5, 5), kernel=conv1_w, bias=conv1_b)
    conv1.stride = (1, 1)

    pool1 = network.add_pooling(input=conv1.get_output(0), type=trt.PoolingType.MAX, window_size=(2, 2))
    pool1.stride = (2, 2)

    conv2_w = weights['conv2.weight'].numpy()
    conv2_b = weights['conv2.bias'].numpy()
    conv2 = network.add_convolution(pool1.get_output(0), 50, (5, 5), conv2_w, conv2_b)
    conv2.stride = (1, 1)

    pool2 = network.add_pooling(conv2.get_output(0), trt.PoolingType.MAX, (2, 2))
    pool2.stride = (2, 2)

    fc1_w = weights['fc1.weight'].numpy()
    fc1_b = weights['fc1.bias'].numpy()
    fc1 = network.add_fully_connected(input=pool2.get_output(0), num_outputs=500, kernel=fc1_w, bias=fc1_b)

    relu1 = network.add_activation(input=fc1.get_output(0), type=trt.ActivationType.RELU)

    fc2_w = weights['fc2.weight'].numpy()
    fc2_b = weights['fc2.bias'].numpy()
    fc2 = network.add_fully_connected(relu1.get_output(0), ModelData.OUTPUT_SIZE, fc2_w, fc2_b)

    fc2.get_output(0).name = ModelData.OUTPUT_NAME
    network.mark_output(tensor=fc2.get_output(0))
```

TensorRT-8.0.1.6/samples/python/engine_refit_mnist/sample.py

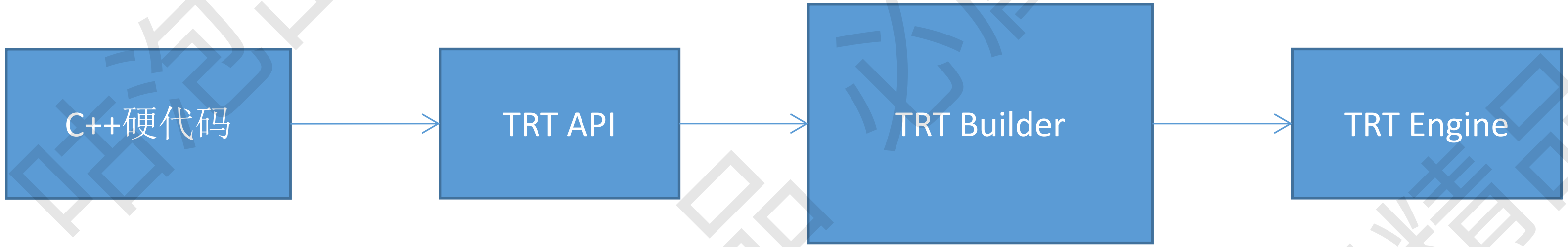
工作流程


libnvparsers.so
UFF
(tensorflow)
libnvonnxparser.so
ONNX
(pytorch etc.)
libnvcaffe_parser.so
Caffe
这条路已经很少有人用了
TRT API
TRT Builder
TRT Engine

常见方案

基于tensorRT的发布，又有人在之上做了工作
https://github.com/wang-xinyu/tensorrtx

为每个模型写硬代码，并已写好了大量的常见模型代码

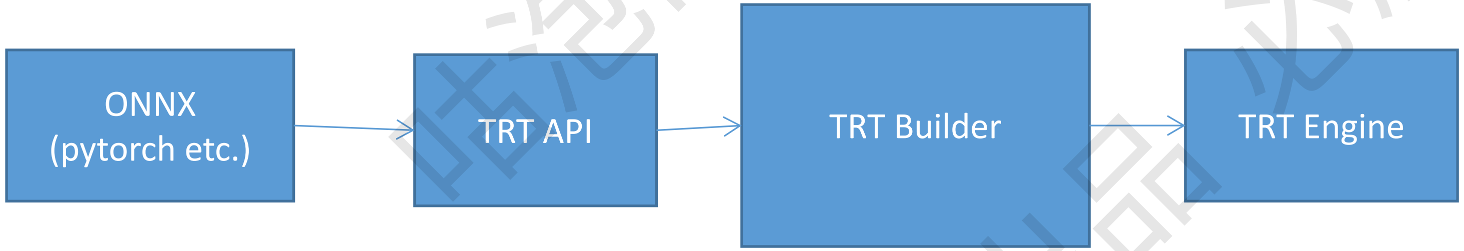



```
/* ------ yolov5 backbone------ */
auto focus0 = focus(network, weightMap, *data, 3, get_width(64, gw), 3, "model.0");
auto conv1 = convBlock(network, weightMap, *focus0->getOutput(0), get_width(128, gw), 3, 2,
auto bottleneck_CSP2 = C3(network, weightMap, *conv1->getOutput(0), get_width(128, gw), get
auto conv3 = convBlock(network, weightMap, *bottleneck_CSP2->getOutput(0), get_width(256, g
auto bottleneck_csp4 = C3(network, weightMap, *conv3->getOutput(0), get_width(256, gw), get
auto conv5 = convBlock(network, weightMap, *bottleneck_csp4->getOutput(0), get_width(512, g
auto bottleneck_csp6 = C3(network, weightMap, *conv5->getOutput(0), get_width(512, gw), get
auto conv7 = convBlock(network, weightMap, *bottleneck_csp6->getOutput(0), get_width(1024,
auto spp8 = SPP(network, weightMap, *conv7->getOutput(0), get_width(1024, gw), get_width(10
```

方案思考

本课程主要学习以onnx路线的模型编译、推理和部署，原因主要有：

若使用onnx，则导出或者修改好的onnx模型，可以**轻易的移植到其他引擎上**、例如ncnn、rknn，这一点硬代码无法做到。并且用于排查错误，修改调整时也非常方便



获取代码

对应于系列名称：tensorrt-basic

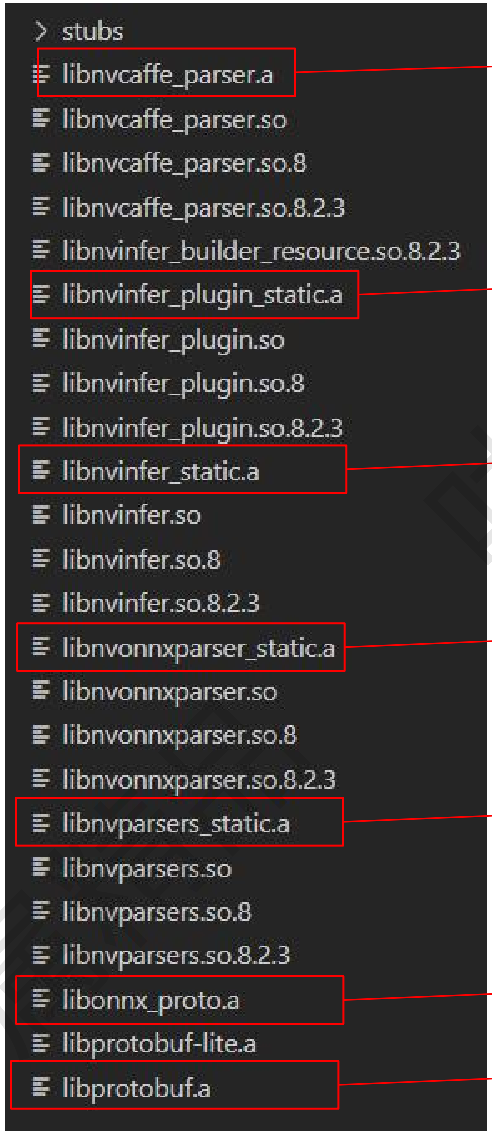
获取代码：trtpy get-series tensorrt-basic

查询系列清单：trtpy series-detail tensorrt-basic

案例清单

```
C:\Users\Administrator\cuda-driver-api>trtpy series-detail tensorrt-basic
Use cache C:\Users\Administrator/.cache/trtpy\code_template\tensorrt-basic.series.json
List templ:
chapter: 1.1, caption: hello-tensorrt, description: 开始第一个tensorRT的旅程吧，编译一个模型
chapter: 1.2, caption: hello-inference, description: 编译好的模型进行推理
chapter: 1.3, caption: cnn-and-dynamic-shape, description: CNN结构的动态shape如何控制，要点在哪里
chapter: 1.4, caption: onnx-parser, description: 使用onnx解析器读取onnx文件构建模型结构和填充权重
chapter: 1.5, caption: onnx-parser-source-code, description: 使用onnx解析器的源代码，了解解析器细节原理
chapter: 1.6, caption: onnx-editor, description: 对onnx文件进行编辑、创建、读取
chapter: 1.7, caption: hello-plugin, description: 开始第一个插件，基于onnx的插件
chapter: 1.8, caption: integrate-easyplugin, description: 对插件做封装，插件开发更简单，更容易
chapter: 1.9, caption: int8, description: tensorRT的int8标定量化
```

TensorRT的库文件一览



caffe的模型解析器，输入caffemodel，解析到tensorRT的模型结构INetworkDefinition

nvidia提供的插件，编译自这里的代码https://github.com/NVIDIA/TensorRT/tree/main/plugin

tensorRT的核心库

ONNX模型解析器，输入onnx模型，解析到tensorRT的模型结构INetworkDefinition

uff模型解析器，输入uff模型，解析到tensorRT的模型结构INetworkDefinition

onnx的proto编译后的效果，是由protoc产生的onnx.cc编译而成

protobuf的静态库

谢谢!