

TensorRT 教程

基于 8.6.1 版本

去年基于 8.2.3 版本的教程视频: BV15Y4y1W73E
今年更新的内容会用 ## 标出

NVIDIA DevTech. Wei Li
2023年6月12日



目录

TensorRT 简介

- TensorRT 基本特性和用法
- Workflow: 使用 TensorRT API 搭建
- Workflow: 使用 ONNX-Parser
- Workflow: 使用框架内 TensorRT 接口

插件书写

- 使用 Plugin 的简单例子
- 关键 API
- 结合使用 Parser 和 Plugin
- Plugin 高级话题
- 使用 Plugin 的例子

TensorRT 高级用法

- 多 Optimization Profile
- 多 Stream
- 多 Context
- CUDA Graph
- Timing Cache
- Algorithm Selector
- Refit
- Tactic Source
- 硬件兼容+版本兼容 ##
- 更多工具

常见优化策略

- 概述
- 性能分析工具
- 性能优化实例

开发辅助工具

- trtexec
- Netron
- polygraphy
- onnx-graphsurgeon
- Nsight Systems



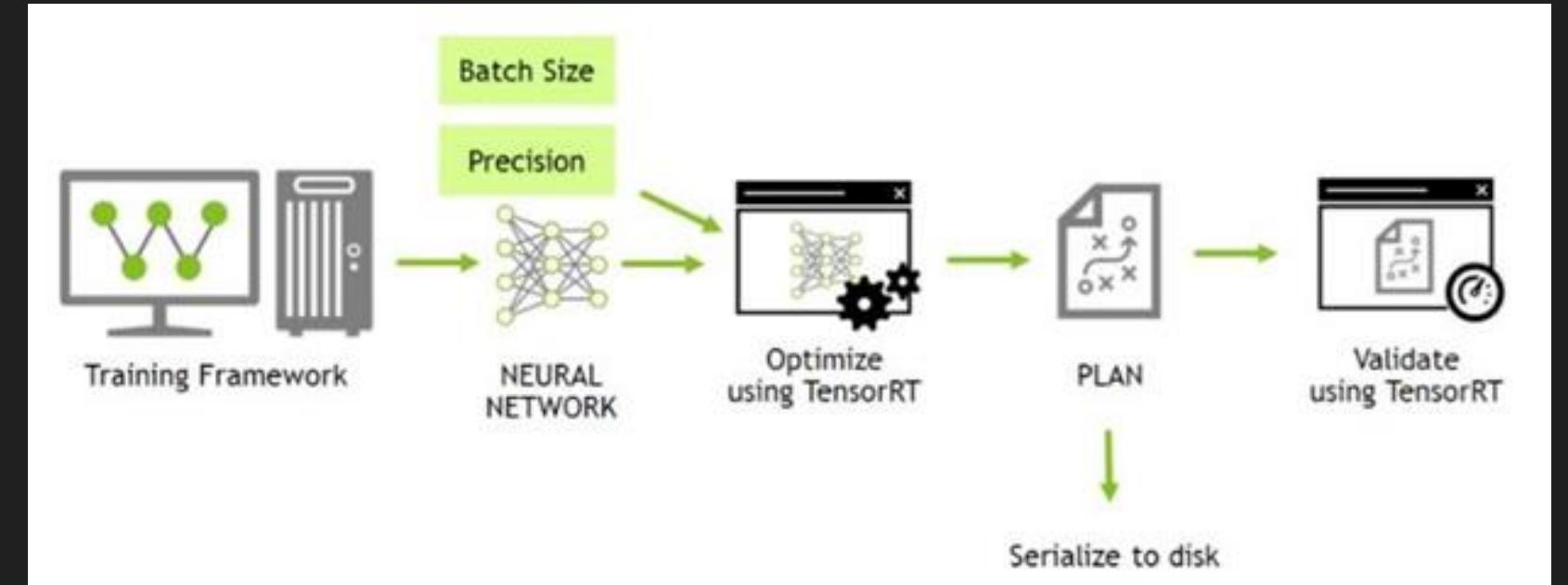
• TensorRT 简介

➤ TensorRT (最新版本 8.6.1)

- 用于高效实现已训练好的深度学习模型的**推理过程**的 SDK
- 内含**推理优化器**和**运行时环境**
- 使 DL 模型能以**更高吞吐量和更低的延迟**运行
- 有 C++ 和 python 的 API, 完全等价可以混用

➤ 链接

- <https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html> (TensorRT 文档)
- https://docs.nvidia.com/deeplearning/tensorrt/api/c_api (C++ API 文档)
- https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/ (python API 文档)
- <https://developer.nvidia.com/nvidia-tensorrt-download> (TensorRT 下载)
- <https://github.com/NVIDIA/trt-samples-for-hackathon-cn/tree/master/cookbook> (本教程配套代码, 包含视频以外的更多范例代码)



• TensorRT 简介

➤ TensorRT 做的工作

➤ 构建期 (推理优化器)

➤ 模型解析 / 建立

加载 Onnx 等其他格式的模型 / 使用原生 API 搭建模型

➤ 计算图优化

横向层融合 (Conv) , 纵向层融合 (Conv+add+ReLU) ,

➤ 节点消除

去除无用层, 节点变换 (Pad, Slice, Concat, Shuffle) ,

➤ 多精度支持

FP32 / FP16 / INT8 / TF32 (可能插入 reformat 节点)

➤ 优选 kernel / format

硬件有关优化

➤ 导入 plugin

实现自定义操作

➤ 显存优化

显存池复用

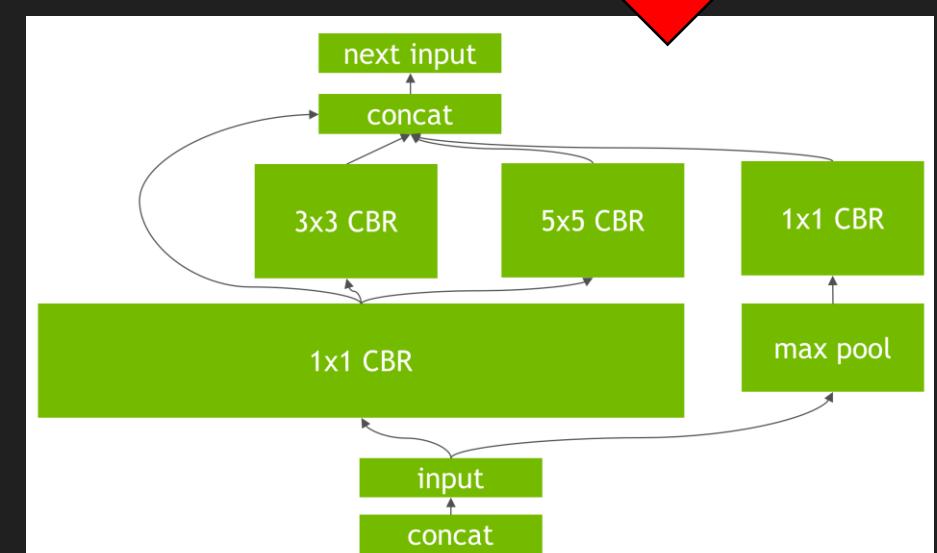
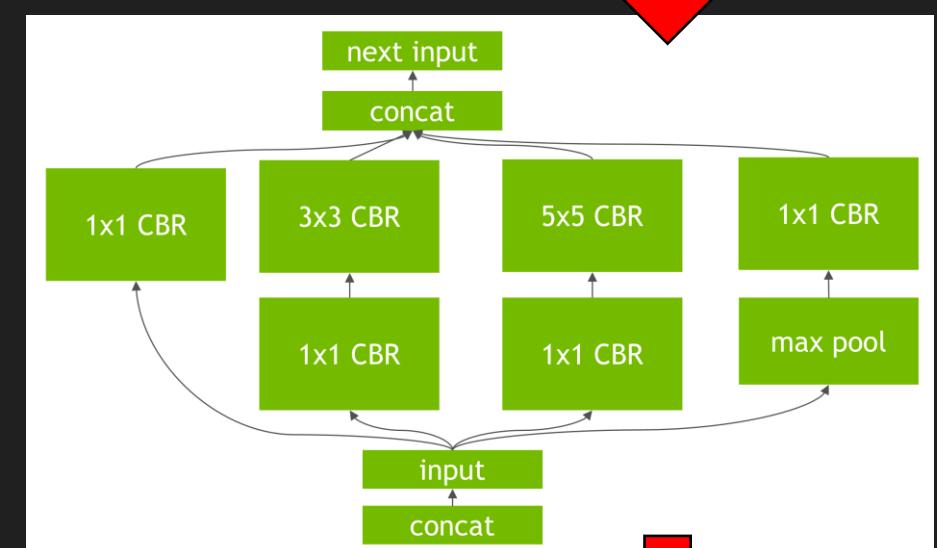
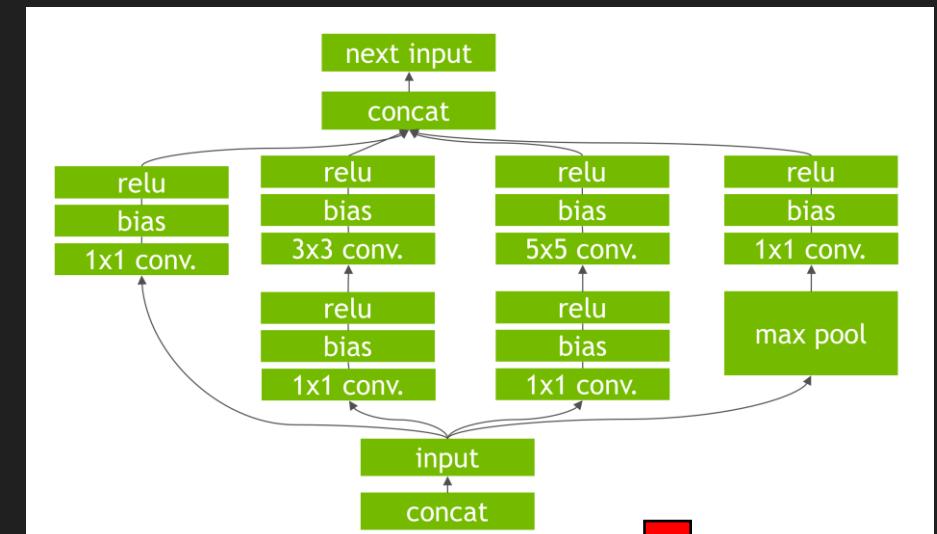
➤ 运行期 (运行时环境)

➤ 运行时环境

对象生命周期管理, 内存显存管理, 异常处理

➤ 序列化反序列化

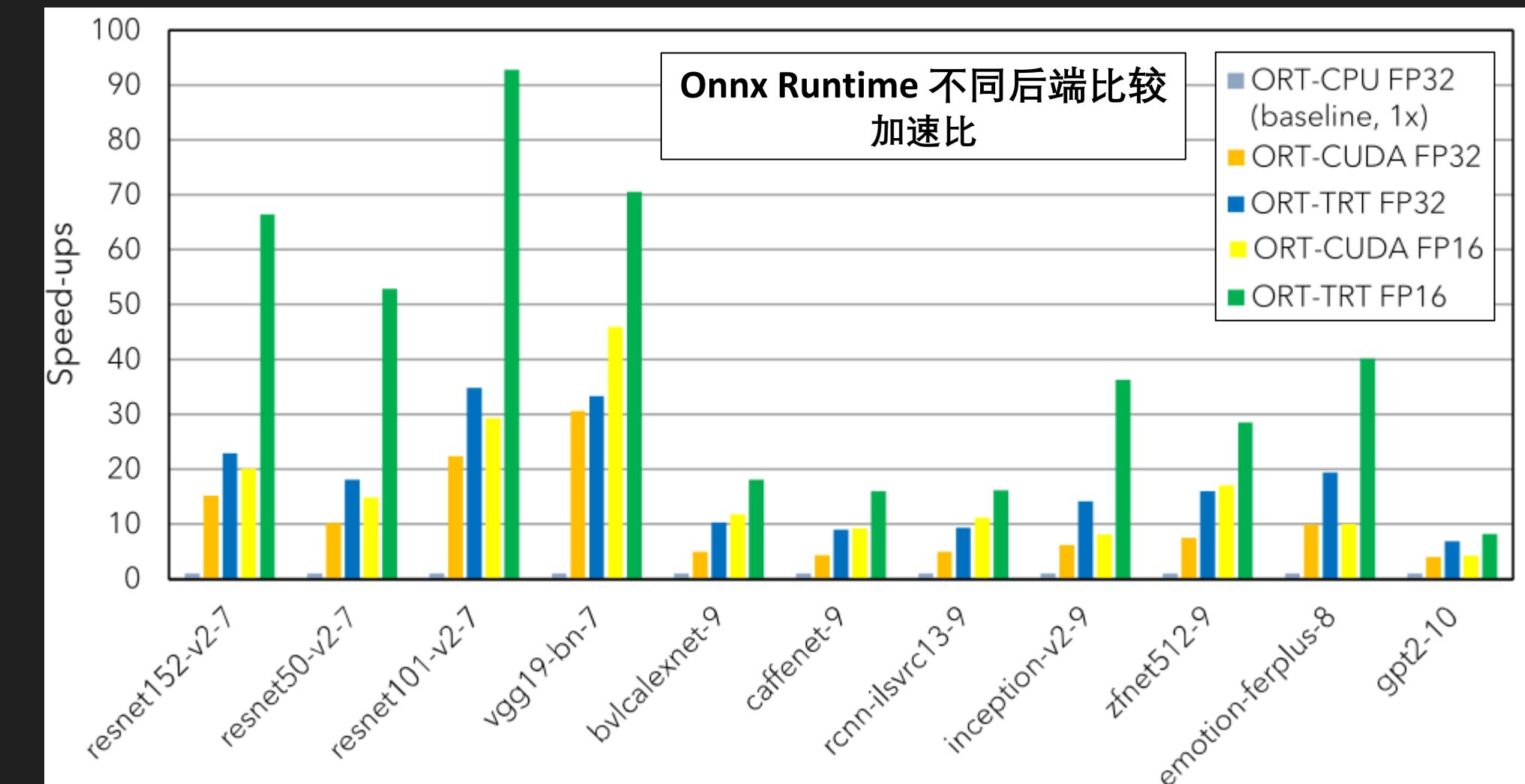
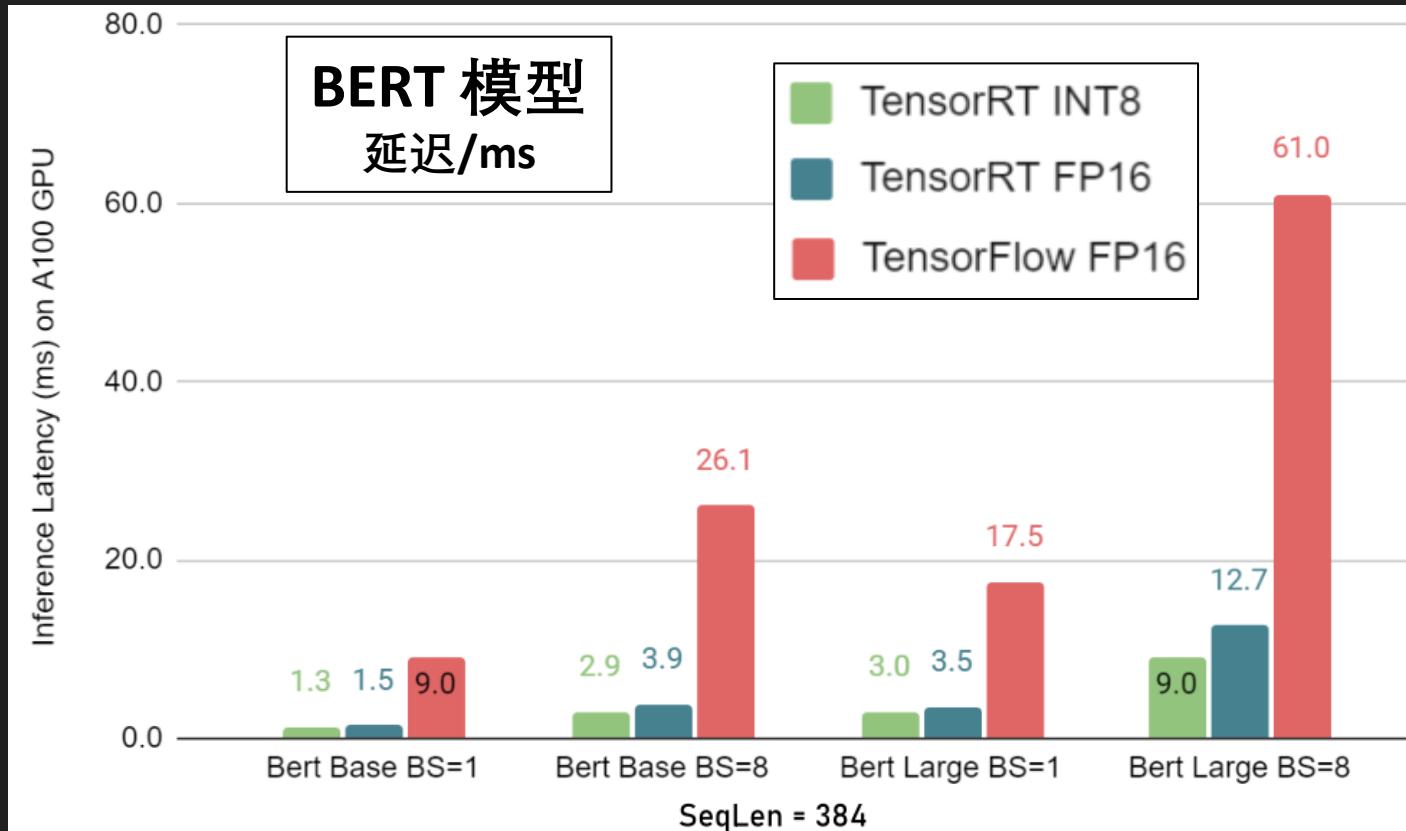
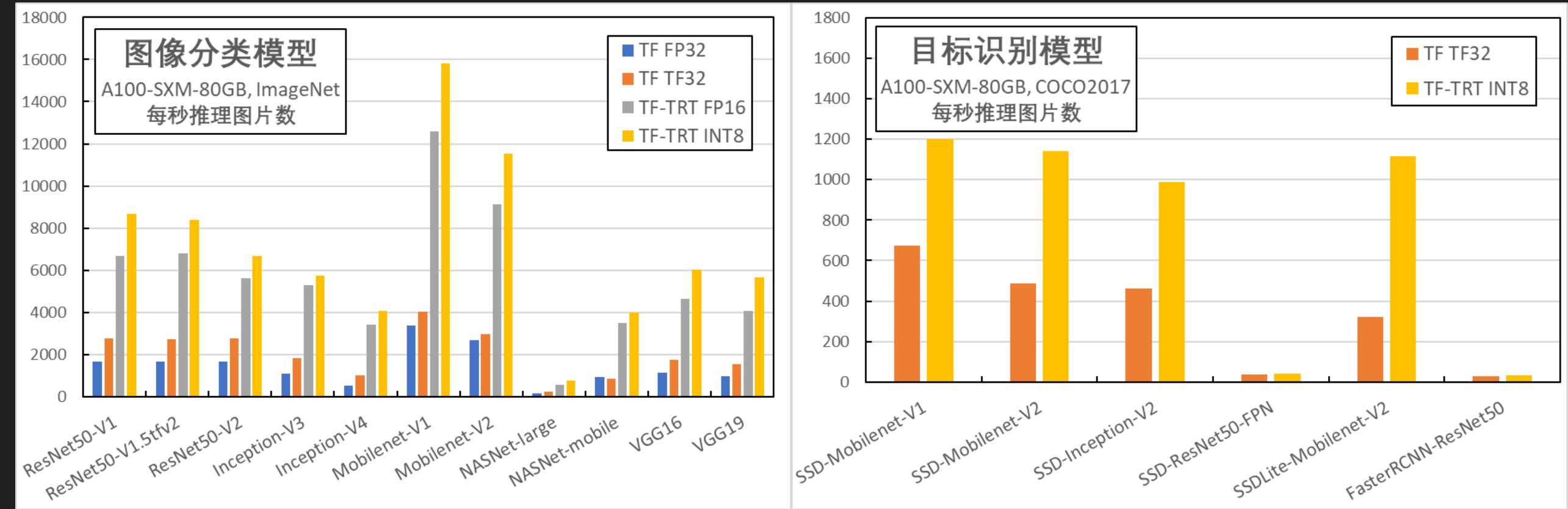
推理引擎保存为文件或从文件中加载



• TensorRT 简介

➤ TensorRT 的表现

- 不同模型加速效果不同
- 选用高效算子提升运算效率
- 算子融合减少访存数据、提高访问效率
- 使用低精度数据类型，节约时间空间



- S31732, Zheng(2021) Inference with Tensorflow2 Integrated with TensorRT
- S31876, Lee(2021) Accelerate Deep Learning Inference with TensorRT 8.0
- S32224, Li(2021) Accelerating Deep Learning Inference With Onnx-Runtime TensorRT



• TensorRT 简介

➤ TensorRT 的表现 ##

➤ Stable Diffusion on TensorRT8.5

Optimization	CLIP ms	UNETx50 ms	VAE ms	Pipeline ms
OOTB(out of the box)	3.01	2467.711	22.064	2492.8473
OOTB+Myelin	3.20	1374.108	29.199	1406.5756
OOTB+Myelin +Seq2Spatial	2.83	1259.683	27.844	1290.4158
OOTB+Myelin +Seq2Spatial+SplitGelu	2.77	1296.7014	27.877	1327.4083
OOTB+Myelin +Seq2Spatial+SplitGelu +LayerNorm	4.37	1353.2231	27.783	1385.4217
OOTB+Myelin +Seq2Spatial+SplitGelu +LayerNorm+GroupNorm	4.41	1211.409	17.314	1233.1823
OOTB+Myelin +Seq2Spatial+SplitGelu +LayerNorm+GroupNorm +FMHCA	4.21	1131.186	17.312	1152.7560
ALL Plugins	4.34	817.277	17.065	838.7314



• TensorRT 简介

- TensorRT 的表现 ##
- ChatGLM-6B on TensorRT8.6

pyTorch				
InputSequenceLength	ContextStep/ms	LastGeneration/ms	TimePerToken/ms	AccelerationRatio
4	44.881	51.474	51.148	
32	47.471	53.981	49.700	
64	46.986	52.197	55.493	
128	49.126	51.837	53.926	
256	53.694	46.243	48.577	
512	71.813	45.231	45.310	
1024	125.434	45.450	46.411	
TensorRT 8.6, BatchSize=1				
4	16.418	11.202	11.375	4.496
32	16.904	12.151	11.716	4.242
64	16.873	11.970	11.680	4.751
128	17.852	12.736	12.205	4.418
256	21.925	12.561	12.653	3.839
512	45.233	13.741	13.887	3.263
1024	90.592	15.794	16.375	2.834



• TensorRT 基本流程

➤ 范例代码

➤ 01-SimpleDemo/TensorRT8.5

main.py 或 main.cpp (python 和 C++ 等价版本)

➤ 基本流程

➤ 构建期

➤ 前期准备 (Logger, Builder, Config, Profile)

➤ 创建 Network (计算图内容)

➤ 生成序列化网络 (计算图 TRT 内部表示)

➤ 运行期

➤ 建立 Engine 和 Context

➤ Buffer 相关准备 (申请+拷贝)

➤ 执行推理 (Execute)

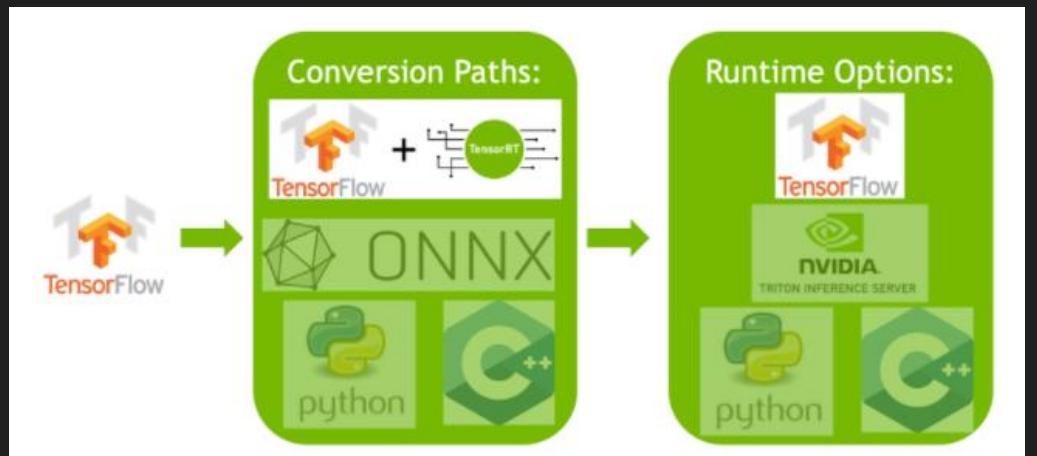
➤ 善后工作

```
1 import numpy as np
2 import tensorrt as trt
3 from cuda import cudart
4
5 logger = trt.Logger(trt.Logger.ERROR)
6 builder = trt.Builder(logger)
7 network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
8 profile = builder.create_optimization_profile()
9 config = builder.create_builder_config()
10 inputTensor = network.add_input("inputT0", trt.float32, [-1, -1, -1])
11 profile.set_shape(inputTensor.name, [1, 1, 1], [3, 4, 5], [6, 8, 10])
12 config.add_optimization_profile(profile)
13 identityLayer = network.add_identity(inputTensor)
14 network.mark_output(identityLayer.get_output(0))
15 engineString = builder.build_serialized_network(network, config) 构建期
16
17 engine = trt.Runtime(logger).deserialize_cuda_engine(engineString) 运行期
18 nIO = engine.num_io_tensors
19 lTensorName = [engine.get_tensor_name(i) for i in range(nIO)]
20 nInput = [engine.get_tensor_mode(lTensorName[i]) for i in range(nIO)].count(trt.TensorIOMode.INPUT)
21 context = engine.create_execution_context()
22 context.set_input_shape(lTensorName[0], [3, 4, 5])
23 bufferH = [np.ascontiguousarray(np.arange(3 * 4 * 5, dtype=np.float32).reshape(3, 4, 5))]
24 bufferH.extend([np.empty(context.get_tensor_shape(lTensorName[1]), dtype=trt.nptype(engine.get_tensor(
25 bufferD = [cudart.cudaMalloc(bufferH[i].nbytes)[1] for i in range(nIO)]
26 [context.set_tensor_address(lTensorName[i], int(bufferD[i])) for i in range(nIO)]
27 [cudart.cudaMemcpy(bufferD[i], bufferH[i].ctypes.data, bufferH[i].nbytes, cudart.cudaMemcpyKind.cudaM
28 context.execute_async_v3(0)
29 [cudart.cudaMemcpy(bufferH[i].ctypes.data, bufferD[i], bufferH[i].nbytes, cudart.cudaMemcpyKind.cudaM
30 [cudart.cudaFree(b) for b in bufferD]
```

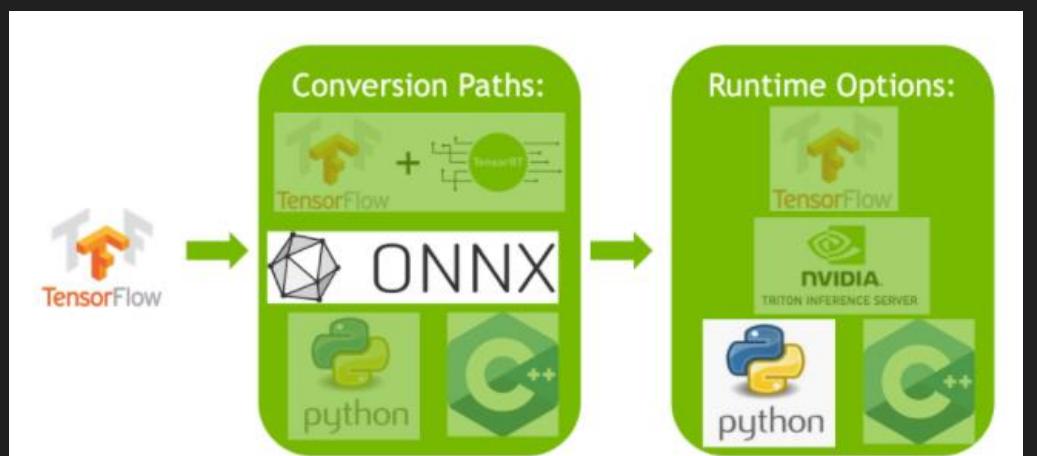


• Workflow

- 使用框架自带 TRT 接口 (TF-TRT, Torch-TensorRT)



- 使用 Parser (TF/Torch/... → ONNX → TensorRT)



- 使用 TensorRT 原生 API 搭建网络

方法	易用性	性能	兼容性	开发效率	遇到不支持的OP
框架自带接口	★★★	★	★★	★★★	返回原框架计算
使用Parser	★★	★★☆	★★☆	★★	改网/改Parser/写Plugin
API搭建	★	★★★	★★★	★	写Plugin



• Workflow: 使用 TensorRT API 搭建

➤ 问题

- 怎样从头开始写一个网络?
- 哪些代码是 API 搭建特有的, 哪些是所有 Workflow 通用的?
- 怎么让一个 Network 跑起来?
- 用于推理计算的输入输出内存显存怎么准备?
- 构建引擎需要时间, 怎么构建一次, 反复使用?
- TensorRT 的开发环境?



• Workflow: 使用 TensorRT API 搭建

➤ 使用 API 完整搭建一个 MNIST 手写识别模型的示例

➤ 范例代码

➤ 03-BuildEngineByTensorRTAPI/MNISTExample-*

➤ 基本流程:

➤ TensorFlow / pyTorch 中创建并训练一个网络

➤ 提取网络权重，保存为 para.npz

➤ TensorRT 中逐层重建该网络并加载 para.npz 中的权重

➤ 生成推理引擎

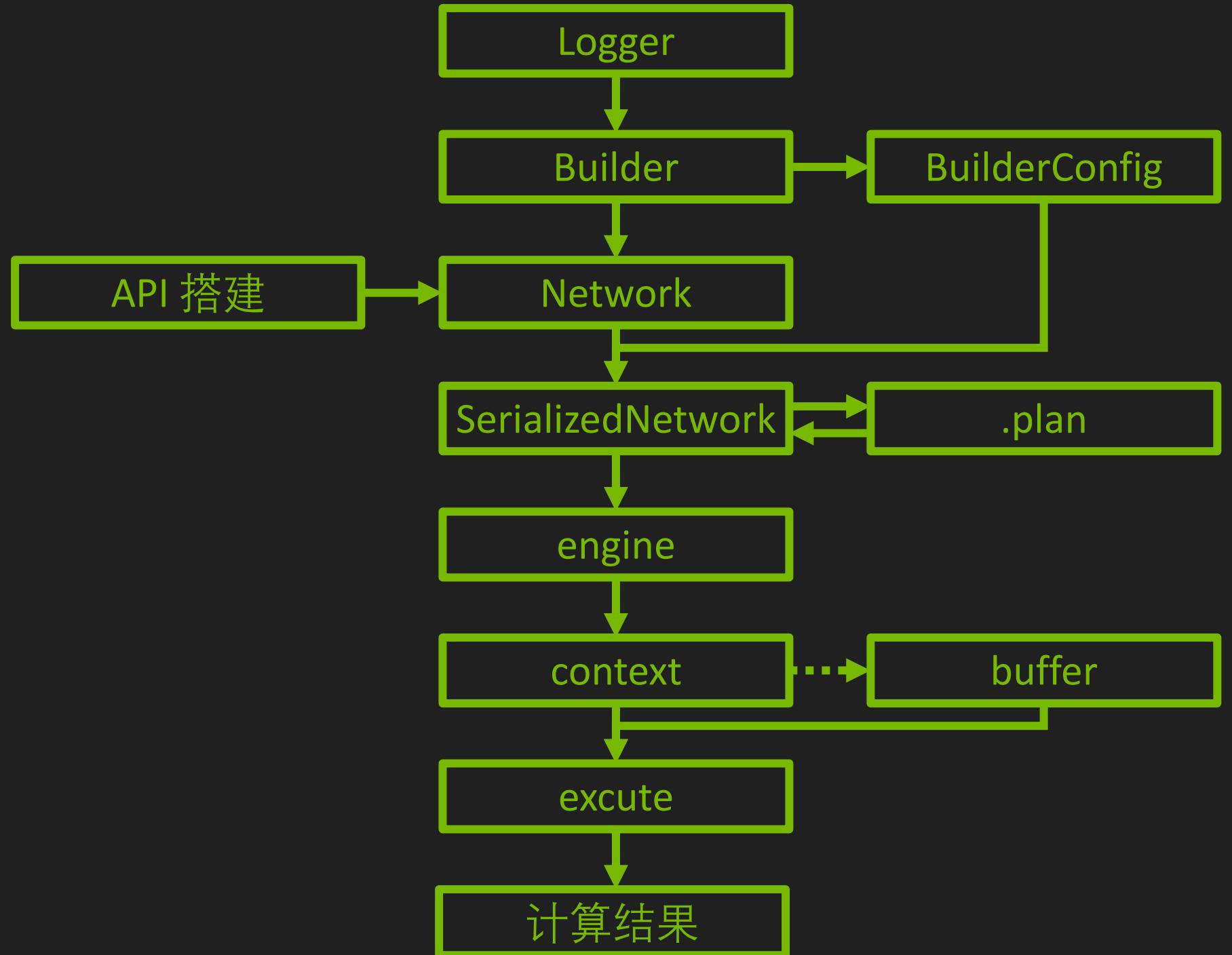
➤ 用引擎做实际推理

```
124     para = {} # save weight as file
125     for name, parameter in model.named_parameters():
126         #print(name, parameter.detach().cpu().numpy().shape)
127         para[name] = parameter.detach().cpu().numpy()
128     np.savez(paraFile, **para)
129
130     del para
131     print("Succeeded building model in pyTorch!")
132
133 # Rebuild network, load weights and do inference in TensorRT -----
134 logger = trt.Logger(trt.Logger.ERROR)
135 builder = trt.Builder(logger)
136 network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
137 profile = builder.create_optimization_profile()
138 config = builder.create_builder_config()
139 if bUseFP16Mode:
140     config.set_flag(trt.BuilderFlag.FP16)
141 if bUseINT8Mode:
142     config.set_flag(trt.BuilderFlag.INT8)
143     config.int8_calibrator = calibrator.MyCalibrator(calibrationDataPath, nCalibration, (1, 1))
144
145 inputTensor = network.add_input("inputT0", trt.float32, [-1, 1, nHeight, nWidth])
146 profile.set_shape(inputTensor.name, [1, 1, nHeight, nWidth], [4, 1, nHeight, nWidth], [8, 1,
147 config.add_optimization_profile(profile)
148
149 para = np.load(paraFile)
150
151 w = np.ascontiguousarray(para["conv1.weight"])
152 b = np.ascontiguousarray(para["conv1.bias"])
153 _0 = network.add_convolution_nd(inputTensor, 32, [5, 5], trt.Weights(w), trt.Weights(b))
154 _0.padding_nd = [2, 2]
155 _1 = network.add_activation(_0.get_output(0), trt.ActivationType.RELU)
156 _2 = network.add_pooling_nd(_1.get_output(0), trt.PoolingType.MAX, [2, 2])
157 _2.stride_nd = [2, 2]
```



• Workflow: 使用 TensorRT API 搭建

- 基本流程
 - 构建阶段
 - 建立 Logger (日志记录器)
 - 建立 Builder (网络元数据) 和BuilderConfig (网络元数据的选项)
 - 创建 Network (计算图内容)
 - 生成 SerializedNetwork (网络的 TRT 内部表示)
 - 运行阶段
 - 建立 Engine (可执行代码)
 - 创建 Context (GPU 进程)
 - Buffer 准备 (Host 端 + Device 端)
 - Buffer 拷贝 Host to Device
 - 执行推理 (Execute)
 - Buffer 拷贝 Device to Host
 - 善后工作



• Workflow: 使用 TensorRT API 搭建

➤ Logger 日志记录器

➤ `logger = trt.Logger(trt.Logger.VERBOSE)`

➤ 可选参数: VERBOSE, INFO, WARNING, ERROR, INTERNAL_ERROR, 产生不同等级的日志, 由详细到简略

➤ VERBOSE:

[TensorRT] VERBOSE: Graph construction and optimization completed in 0.000261295 seconds.

➤ INFO:

[TensorRT] INFO: Detected 1 inputs and 1 output network tensors.

➤ WARNING:

[TensorRT] WARNING: Tensor DataType is determined at build time for tensors not marked as input or output.

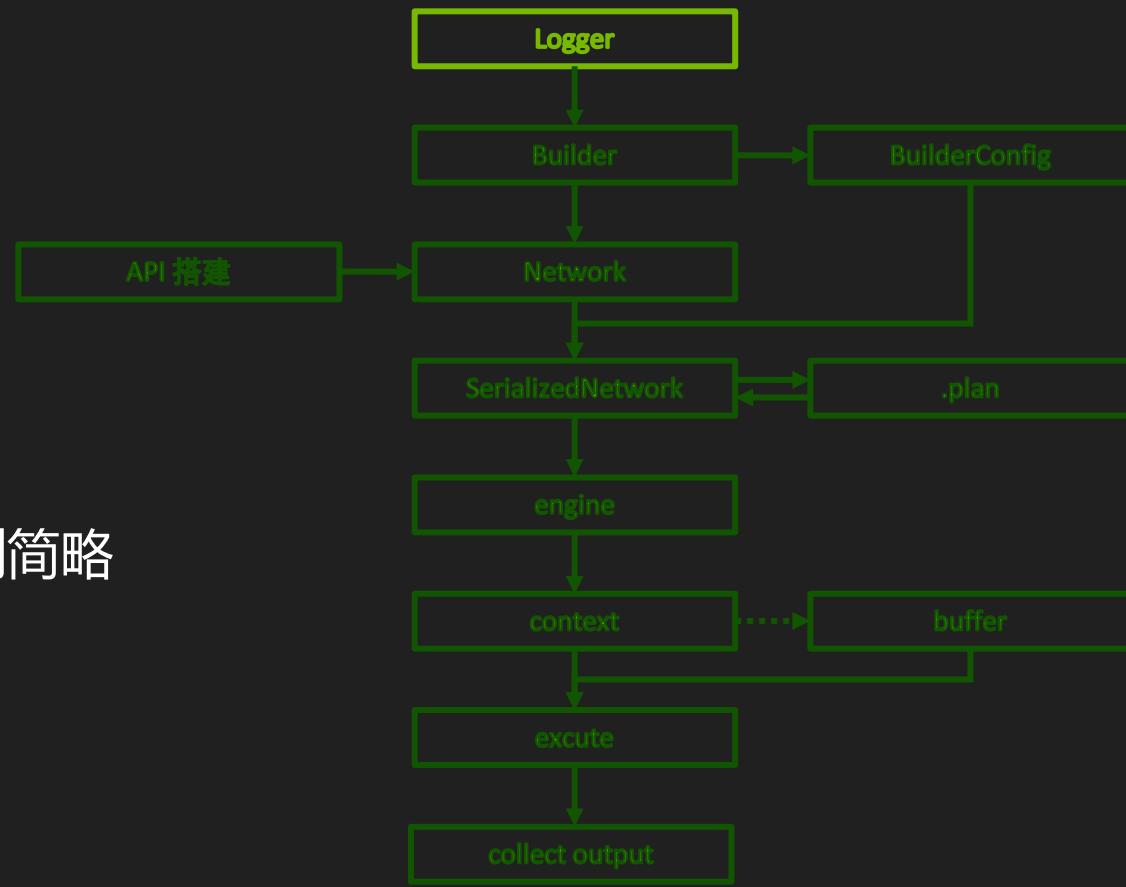
➤ ERROR:

[TensorRT] ERROR: INVALID_CONFIG: Deserialize the cuda engine failed.

➤ INTERNAL_ERROR:

[TensorRT] ERROR: ./builder/tacticOptimizer.cpp (1820) - TRTInternal Error in computeCosts: 0 (Could not find any implementation for node (Unnamed Layer 0) [TopK].)*

➤ 更多细节见 02-API/Logger



• Workflow: 使用 TensorRT API 搭建

➤ Builder 引擎构建器

➤ `builder = trt.Builder(logger)`

➤ 常用API

➤ `builder.create_network(...)`

创建 TensorRT 网络对象

➤ `builder.create_optimization_profile()`

创建用于 Dynamic Shape 输入的配置器

➤ 其他成员 (将被废弃, 全部不推荐使用) :

➤ `builder.max_batch_size = 256` 指定最大 Batch Size (Static Shape 模式下使用)

➤ `builder.max_workspace_size = 1<<30` 指定最大可用显存 (单位 Byte)

➤ `builder.fp16_mode = True/False` 开启/关闭 fp16 模式

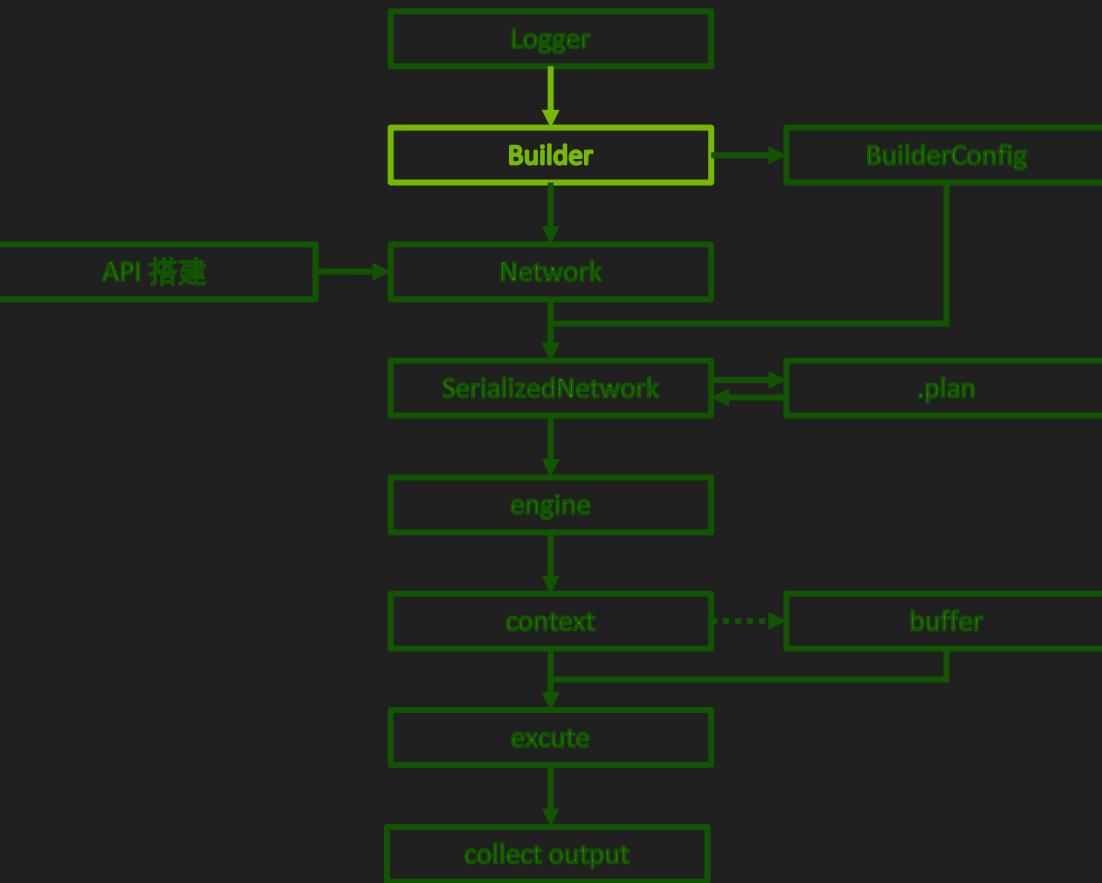
➤ `builder.int8_mode = True/False` 开启/关闭 int8 模式

➤ `builder.strict_type_constraints = True/False` 开启/关闭强制精度模式 (通常不用)

➤ `builder.refittable = True/False` 开启/关闭 refit 模式

➤ Dynamic Shape 模式必须改用 builderConfig 来进行这些设置

➤ 更多细节见 02-API/Builder



• Workflow: 使用 TensorRT API 搭建

➤ BuilderConfig 网络属性选项

➤ `config = builder.create_builder_config()`

➤ 常用成员：

➤ `config.config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 1 << 30)`

指定构建期可用显存 (单位: Byte)

➤ `config.flag = ...`

设置标志位开关, 如启闭 FP16/INT8 模式, Refit 模式, 手工数据类型限制等

➤ `config.int8_calibrator = ...`

指定 INT8-PTQ 的校正器

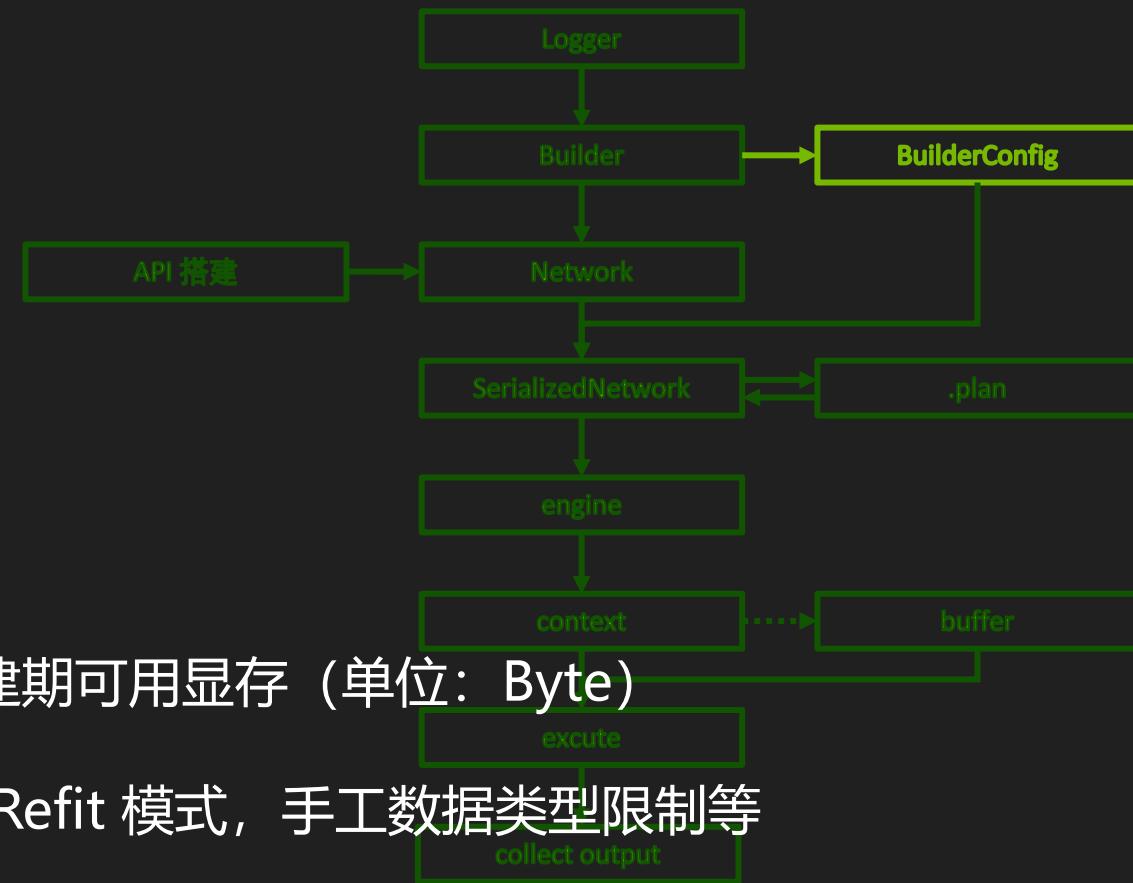
➤ `config.add_optimization_profile(...)`

添加用于 Dynamic Shape 输入的配置器

➤ `config.set_tactic_sources/set_timing_cache/set_preview_feature/ ...`

更多高级用法 (见教程第四部分)

➤ 更多细节见 02-API/BuilderConfig



• Workflow: 使用 TensorRT API 搭建

➤ Network 网络具体构造

➤ `network = builder.create_network()`

➤ 常用参数:

➤ `1 << int(tensorrt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)`, 使用 Explicit Batch 模式

➤ 常用方法:

➤ `network.add_input('oneTensor' ,trt.float32, (3,4,5))` 标记网络输入张量

➤ `convLayer = network.add_convolution_nd(XXX)` 添加各种网络层

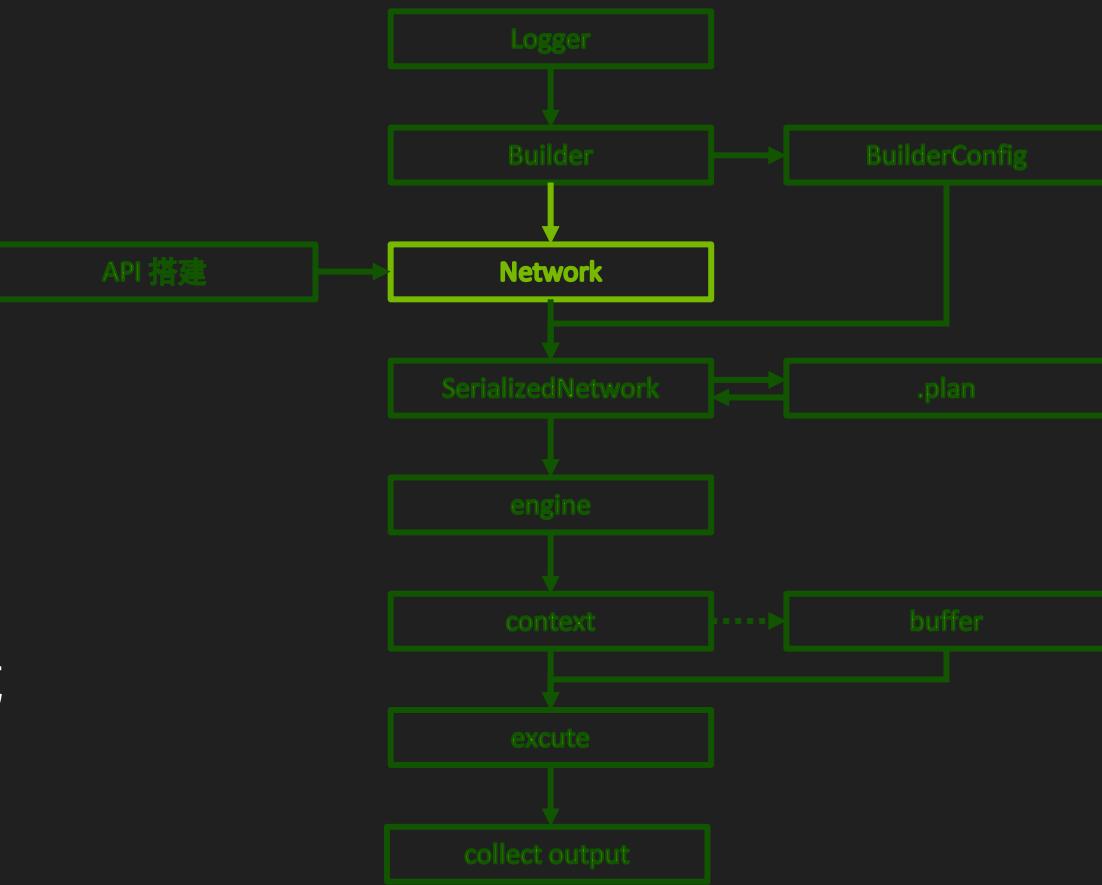
➤ `network.mark_output(convLayer.get_output(0))` 标记网络输出张量

➤ 常用获取网络信息的成员:

➤ `network.name / network.num_layers / network.num_inputs / network.num_outputs`

➤ `network.has_implicit_batch_dimension / network.has_explicit_precision`

➤ 其他细节见 02-API/Network



• Workflow: 使用 TensorRT API 搭建

- Explicit Batch 模式 v.s. Implicit Batch 模式
 - Explicit Batch 为 TensorRT 主流 Network 构建方法, Implicit Batch 模式 (`builder.create_network(0)`) 仅用作后向兼容
 - 所有张量显式包含 Batch 维度、比 Implicit Batch 模式多一维
 - 需要使用 `builder.create_network(1 << int(tensorrt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))`
 - Explicit Batch 模式能做、Implicit Batch 模式不能做的事情:
 - Batch Normalization (视频教程的录音中说成了 Layer Normalization)
 - Reshape/Transpose/Reduce over batch dimension
 - Dynamic shape 模式
 - Loop 结构
 - 一些 Layer 的高级用法 (如 `ShuffleLayer.set_input`)
 - 从 Onnx 导入的模型也默认使用 Explicit Batch 模式
 - 范例代码仅在 01-SimpleDemo 中 TensorRT6 和 TensorRT7 中保留了 Implicit Batch 的例子



• Workflow: 使用 TensorRT API 搭建

➤ Dynamic Shape 模式

- 适用于输入张量形状在推理时才决定网络
- 除了 Batch 维, 其他维度也可以推理时才决定
- 需要 Explicit Batch 模式
- 需要 Optimization Profile 帮助网络优化
- 需用 context.set_input_shape 绑定实际输入数据形状

```
1 network = builder.create_network(1<<int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
2 profile = builder.create_optimization_profile()
3 config = builder.create_builder_config()
4
5 inputTensor = network.add_input('inputT0', trt.DataType.FLOAT, [-1,-1,-1])
6 profile.set_shape(inputTensor.name, (1,1,1),(3,4,5),(6,8,10))
7 config.add_optimization_profile(profile)
8
9 # ...
10
11 context.set_binding_shape(0,[3,4,5])
12
13 # ...
14
15 outputH0 = np.empty(context.get_binding_shape(1),dtype = trt.nptype(engine.get_binding_dtype(1)))
16
17 # ...
18
19 context.execute_async_v2([int(inputD0), int(outputD0)], stream)
```

➤ Profile 指定输入张量大小范围

➤ profile = builder.create_optimization_profile()

➤ 常用方法:

➤ profile.set_shape(tensorName, minShape, commonShape, maxShape)

给定输入张量的最小、最常见、最大尺寸

➤ config.add_optimization_profile(profile)

将设置的 profile 传递给 config 以创建网络



• Workflow: 使用 TensorRT API 搭建

➤ Layer 和 Tensor

➤ 注意区别 Layer 和 Tensor

```
oneLayer = network.add_identity(inputTensor) # output is a layer  
  
oneTensor = oneLayer.get_output(0)           # get tensor from the layer  
  
nextLayer = network.add_identity(oneTensor)  # take the tensor into next layer
```

➤ Layer 的常用成员和方法:

➤ **oneLayer.name** = 'one'

获取或指定 Layer 的名字

➤ **oneLayer.type**

获取该层的种类

➤ **oneLayer.precision**

指定改层计算精度 (需配合 builder.strict_type_constraints)

➤ **oneLayer.get_output(i)**

获取该层第 i 个输出张量

➤ Tensor 的常用成员和方法:

➤ **oneTensor.name** = 'one'

获取或指定 tensor 的名字

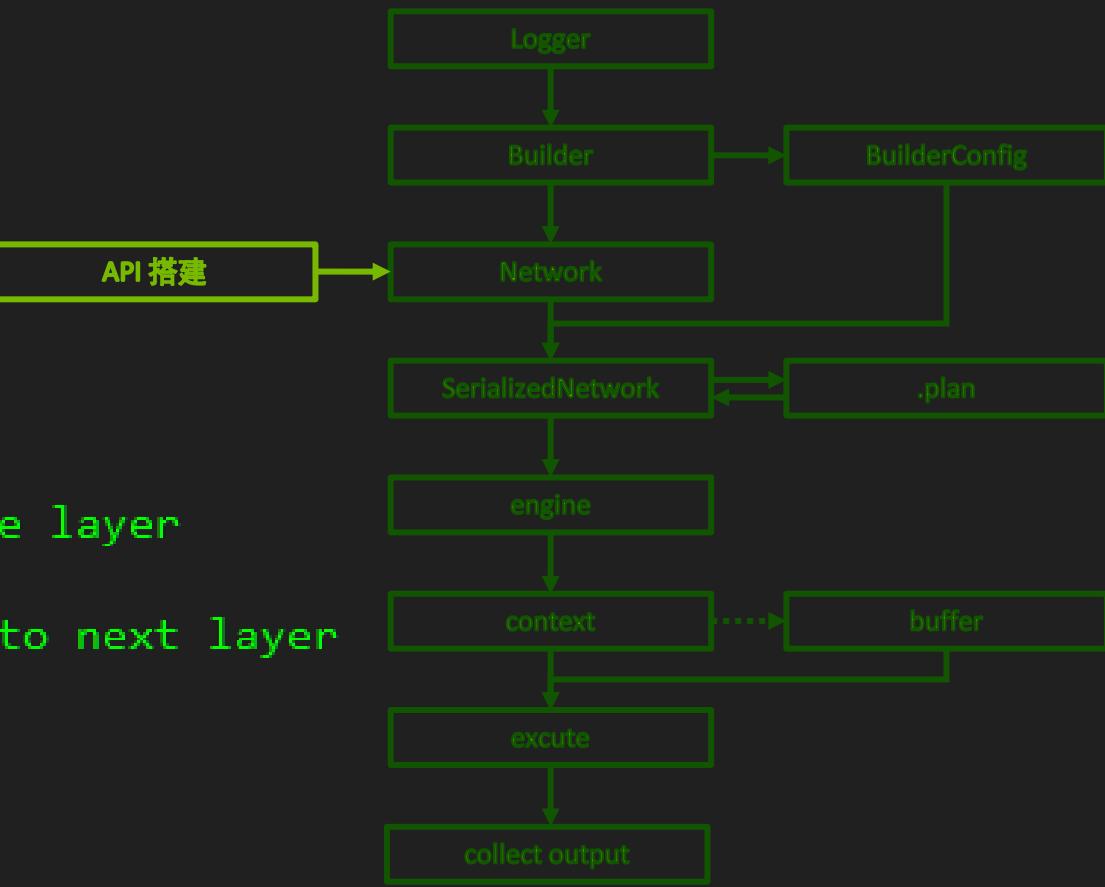
➤ **oneTensor.shape**

获取 tensor 的形状, 可用于 print 检查或作为后续层的参数

➤ **oneTensor.dtype**

获取或设定 tensor 的数据类型 (可用于配合 identity 层实现数据类型转换)

➤ 更多细节见 02-API/Tensor 和 02-API/Layer



• Workflow: 使用 TensorRT API 搭建

➤ 从 Network 中打印所有层和张量的信息

➤ 范例代码: 07-Tool/NetworkPrinter

➤ 外层循环遍历所有 Layer

➤ 内层循环遍历该 Layer 的的所有 input/output

```
1  for i in range(network.num_layers):
2      layer = network.get_layer(i)
3      print(i,"%s,in=%d,out=%d,%s"%(str(layer.type)[10:],layer.num_inputs,layer.num_outputs,layer.name))
4      for j in range(layer.num_inputs):
5          tensor = layer.get_input(j)
6          if tensor == None:
7              print("\tInput %2d:%%j,None")
8          else:
9              print("\tInput %2d:%s,%s,%s%(j,tensor.shape,str(tensor.dtype)[9:],tensor.name))
10     for j in range(layer.num_outputs):
11         tensor = layer.get_output(j)
12         if tensor == None:
13             print("\tOutput %2d:%%j,None")
14         else:
15             print("\tOutput %2d:%s,%s,%s%(j,tensor.shape,str(tensor.dtype)[9:],tensor.name))
```

```
1  0 IDENTITY,in=1,out=1,node_of_127
2      Input 0:(1,),INT32,1
3      Output 0:(1,),INT32,127
4  1 CONSTANT,in=0,out=1,129
5      Output 0:(),INT32,(Unnamed Layer* 1) [Constant]_output
6  2 SHUFFLE,in=1,out=1,node_of_131
7      Input 0:(),INT32,(Unnamed Layer* 1) [Constant]_output
8      Output 0:(1,),INT32,131
9
10 ...
11
12 600 SHUFFLE,in=1,out=1,(Unnamed Layer* 600) [Shuffle]
13      Input 0:(128,),FLOAT,(Unnamed Layer* 599) [Constant]_output
14      Output 0:(1, 1, 128),FLOAT,(Unnamed Layer* 600) [Shuffle]_output
15 601 ELEMENTWISE,in=2,out=1,node_of_586
16      Input 0:(16, 255, 128),FLOAT,585
17      Input 1:(1, 1, 128),FLOAT,(Unnamed Layer* 600) [Shuffle]_output
18      Output 0:(16, 255, 128),FLOAT,586
19 602 IDENTITY,in=1,out=1,2006-CastForOutput
20      Input 0:(1, 1, 255),INT32,176
21      Output 0:(1, 1, 255),INT32,2005-CastForOutput
```



• Workflow: 使用 TensorRT API 搭建

➤ 权重迁移

- 原模型中权重保存为 npz，方便 TensorRT 读取

➤ 范例代码

- 03-BuildEngineByTensorRTAPI/TypicalAPI-*

➤ 逐层搭建

- 注意算法一致性和权重的排列方式
- 举例：TensorFlow 中 LSTM 多种实现，各种实现导出权重的排列顺序不同

➤ 逐层检验输出

- FP32 模式相对误差均值 1×10^{-6} 量级，FP16 模式相对误差均值 1×10^{-3} 量级
- 保证 FP32 模式结果正确后，逐步尝试 FP16 和 INT8 模式

```
1 # TensorFlow -----
2 para = {}
3 for i in tf.compat.v1.get_collection(tf.compat.v1.GraphKeys.GLOBAL_VARIABLES):
4     name,value = i.name,sess.run(i)
5     tfPara[name] = value
6 np.savez("paraTF.npz",**para)
7
8
9 # pyTorch -----
10 para = {}
11 for name,para in model.named_parameters():
12     para[name] = para
13 np.savez("paraPT.npz",**para)
14
15
16 # Paddlepaddle -----
17 file_path = "../model/model_state"
18 program_state = fluid.load_program_state(file_path + "/temp")
19 fluid.set_program_state(prog, program_state)
20 para = {}
21 for block in prog.blocks:
22     for var in block.vars:
23         if 'tmp' in var:
24             continue
25         param = block.vars[var].name
26         para[param] = np.array( fluid.global_scope().find_var(param).get_tensor() )
27 np.savez('./paraPP.npz',**para)
```



• Workflow: 使用 TensorRT API 搭建

➤ 常见 Layer 的使用范例

➤ 范例代码 02-API/Layer/*/*.md, 41 种 Layer 示例

➤ 各 Layer 目录中示例代码可以直接运行

➤ 遇到 TensorRT 不原生支持的节点

➤ 自己实现 Plugin (见教程第三部分讲)



• Workflow: 使用 TensorRT API 搭建

➤ FP16 模式

- 范例代码: 03-BuildEngineByONNXParser/pyTorch-ONNX-TensorRT
- config.flags = 1<<int(trt.BuilderFlag.FP16)
- 建立 engine 时间比 FP32 模式更长 (更多 kernel 选择, 需要插入 Reformat 节点)
- Timeline 中出现 nchwToNchw 等 kernel 调用
- 部分层可能精度下降导致较大误差
 - 找到误差较大的层 (用 polygraphy等工具, 见教程第二部分)
 - 强制该层使用 FP32 进行计算
 - config.set_flag(trt.BuilderFlag.OBEY_PRECISION_CONSTRAINTS)
 - layer.precision = trt.float32



• Workflow: 使用 TensorRT API 搭建

➤ Int8 模式 —— PTQ

- 范例代码: 03-BuildEngineByONNXParser/pyTorch-ONNX-TensorRT-PTQ
- 需要有校准集 (输入范例数据)
- 自己实现 calibrator (如右图)
- config.set_flag(trt.BuilderFlag.INT8)
- config.int8_calibrator = ...

➤ Int8 模式 —— QAT

- 范例代码: 03-BuildEngineByONNXParser/pyTorch-ONNX-TensorRT-QAT
- config.set_flag(trt.BuilderFlag.INT8)
- 在 pyTorch 网络中插入 Quantize/Dequantize 层

```
1  class MyCalibrator(trt.IInt8EntropyCalibrator2):  
2  
3      def __init__(self, calibrationDataPath, calibrationCount, inputShape, cacheFile):  
4          trt.IInt8EntropyCalibrator2.__init__(self)  
5          ...  
6          self.oneBatch = self.batchGenerator()  
7  
8      def __del__(self):  
9          cudart.cudaFree(self.dIn)  
10  
11     def batchGenerator(self):  
12         yield np.ascontiguousarray(imageList(subImageList))  
13  
14     def get_batch_size(self): # do NOT change name  
15         return self.shape[0]  
16  
17     def get_batch(self, nameList=None, inputNodeName=None): # do NOT change name  
18         try:  
19             data = next(self.oneBatch)  
20             cudart.cudaMemcpy(self.dIn, data.ctypes.data, self.bufferSize,  
21                               cudart.cudaMemcpyKind.cudaMemcpyHostToDevice)  
22             return [int(self.dIn)]  
23         except StopIteration:  
24             return None  
25  
26     def read_calibration_cache(self): # do NOT change name  
27         if os.path.exists(self.cacheFile):  
28             print("Succeed finding cache file: %s" % (self.cacheFile))  
29             with open(self.cacheFile, "rb") as f:  
30                 cache = f.read()  
31             return cache  
32         else:  
33             print("Failed finding int8 cache!")  
34             return  
35  
36     def write_calibration_cache(self, cache): # do NOT change name  
37         with open(self.cacheFile, "wb") as f:  
38             f.write(cache)  
39             print("Succeed saving int8 cache!")
```



• Workflow：使用 TensorRT API 搭建

➤ 问题

- 怎样从头开始写一个可以跑的网络？
- API搭建比较复杂，那么哪些代码是API搭建特有的，哪些是所有 Workflow 通用的？
- 怎么让一个 Network 跑起来？
- 用于推理计算的输入输出数据怎么准备？
- 构建引擎需要时间，怎么构建一次，反复使用？
- TensorRT 的开发环境？



• TensorRT 运行期 (Runtime)

➤ 生成 TRT 内部表示

```
➤ serializedNetwork = builder.build_serialized_network(network, config)
```

➤ 生成 Engine

```
➤ engine = trt.Runtime(logger).deserialize_cuda_engine(serializedNetwork)
```

```
➤ lTensorName = [engine.get_tensor_name(i) for i in range(engine.num_io_tensors)]
```

➤ 创建 Context

```
➤ context = engine.create_execution_context()
```

➤ 绑定输入输出 (Dynamic Shape 模式必须)

```
➤ context.set_input_shape(lTensorName[0], [3, 4, 5]) ## TensorRT 8.5 开始 binding 系列 API 全部 deprecated, 换成 tensor 系列 API
```

➤ 准备 Buffer

```
➤ inputHost = np.ascontiguousarray(inputData.reshape(-1))
```

```
➤ outputHost = np.empty(context.get_tensor_shape(iTensorName[1]), trt.nptype(engine.get_tensor_dtype(iTensorName[1])))
```

```
➤ inputDevice = cudart.cudaMalloc(inputHost.nbytes)[1]
```

```
➤ outputDevice = cudart.cudaMalloc(outputHost.nbytes)[1]
```

```
➤ context.set_tensor_address(iTensorName[0], inputDevice) ## 用到的 GPU 指针提前在这里设置, 不再传入 execute_v3 函数
```

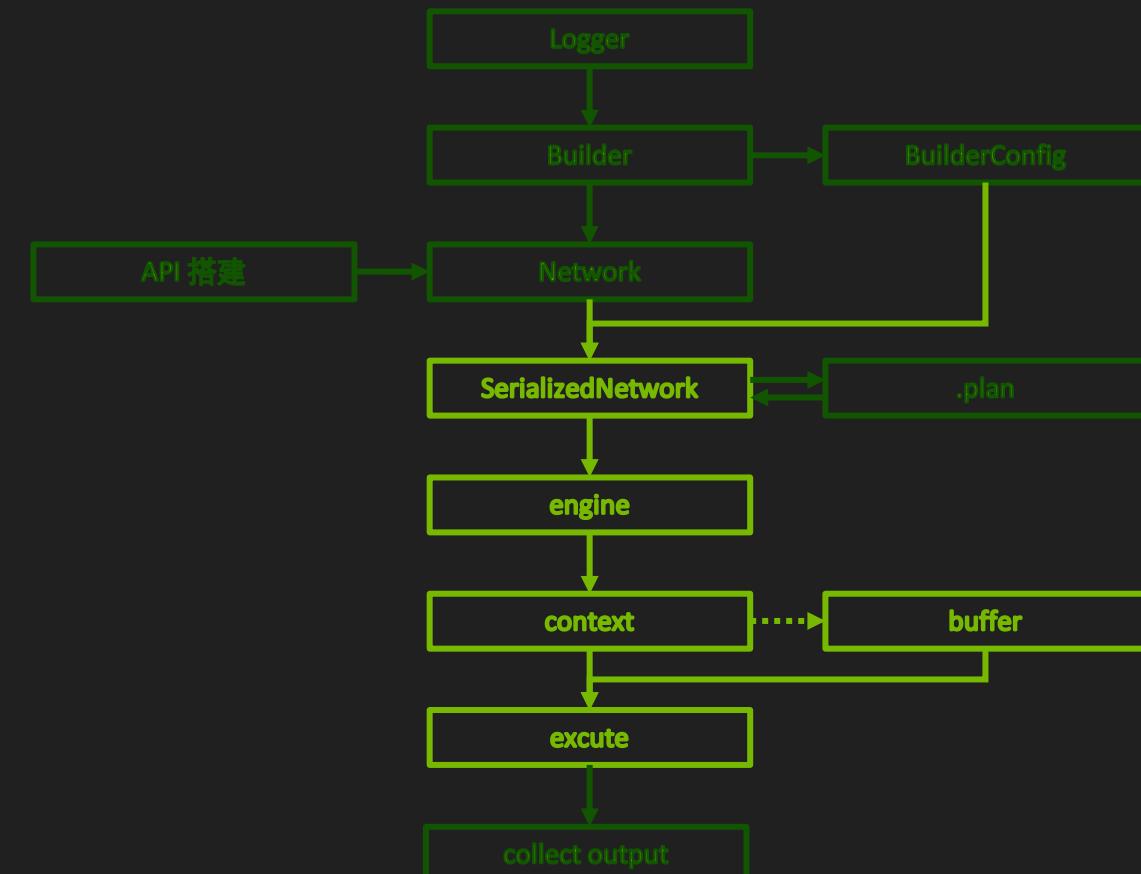
```
➤ context.set_tensor_address(iTensorName[1], outputDevice)
```

➤ 执行计算

```
➤ cudart.cudaMemcpy(inputDevice, inputHost.ctypes.data, inputHost.nbytes, cudart.cudaMemcpyKind.cudaMemcpyHostToDevice)
```

```
➤ context.execute_async_v3(0)
```

```
➤ cudart.cudaMemcpy(outputHost.ctypes.data, outputDevice, outputHost.nbytes, cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost)
```



• Workflow: 使用 TensorRT API 搭建

➤ Engine 计算引擎

➤ `serializedNetwork = builder. build_serialized_network(network, config)`

➤ `engine = trt.Runtime(logger).deserialize_cuda_engine(serializedNetwork)`

➤ 常用成员:

➤ `engine. num_io_tensors` 获取 engine 绑定的输入输出张量总数, $n + m$

➤ `engine.num_layers` 获取 engine (自动优化后) 总层数

➤ 常用方法: ## TensorRT 8.5 开始 binding 系列 API 全部 deprecated, 换成 tensor 系列 API

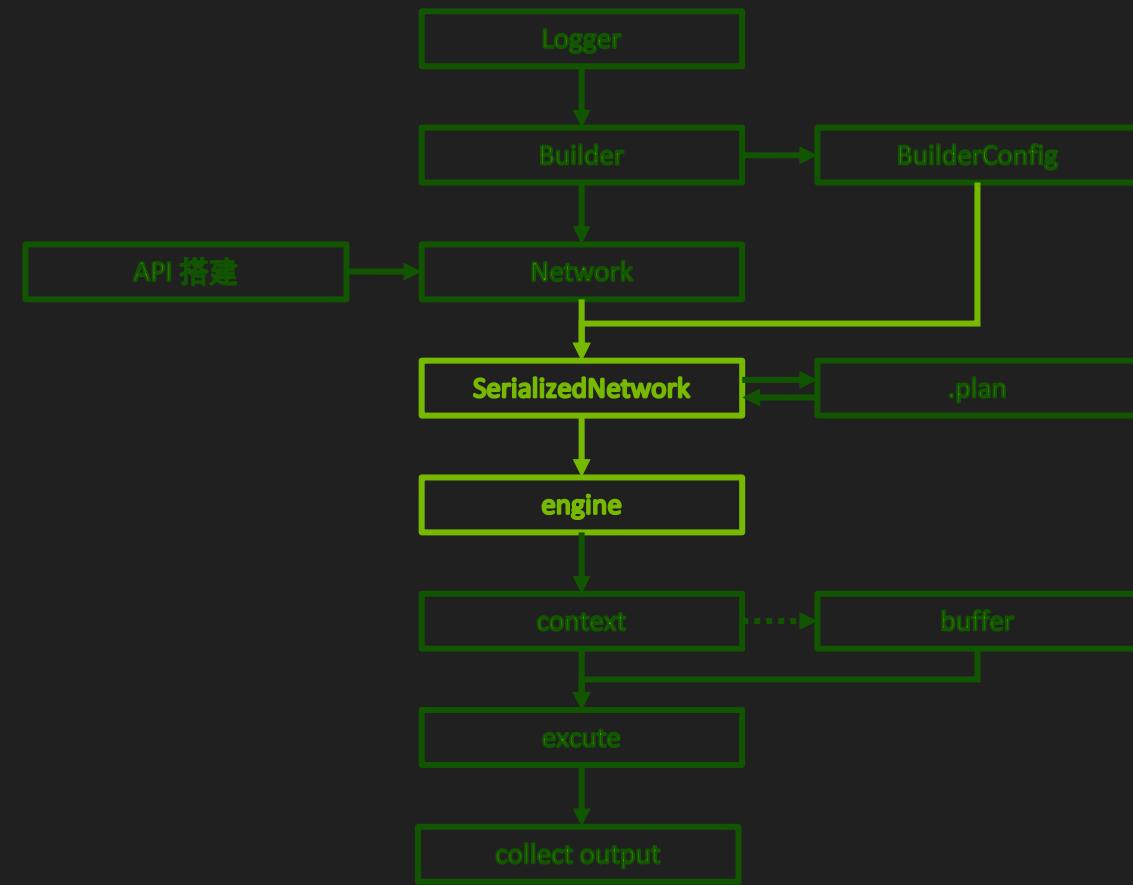
➤ `Engine.get_tensor_name(i)` 第 i 个张量的名字

➤ `engine.get_tensor_dtype(iTensorName[i])` 第 i 个张量的数据类型, 传入张量名字而非索引

➤ `engine.get_tensor_shape(iTensorName[i])` 第 i 个张量的张量形状, 传入张量名字而非索引, Dynamic Shape 模式下结果可能含 -1

➤ `engine. engine.get_tensor_mode(iTensorName[i])` 第 i 个张量属于是输入还是输出张量

➤ 更多细节见 02-API/CudaEngine



• Workflow: 使用 TensorRT API 搭建

➤ Context 推理进程

➤ `context = engine.create_execution_context()`

➤ 常用方法: ## TensorRT 8.5 开始 binding 系列 API 全部 deprecated, 换成 tensor 系列 API

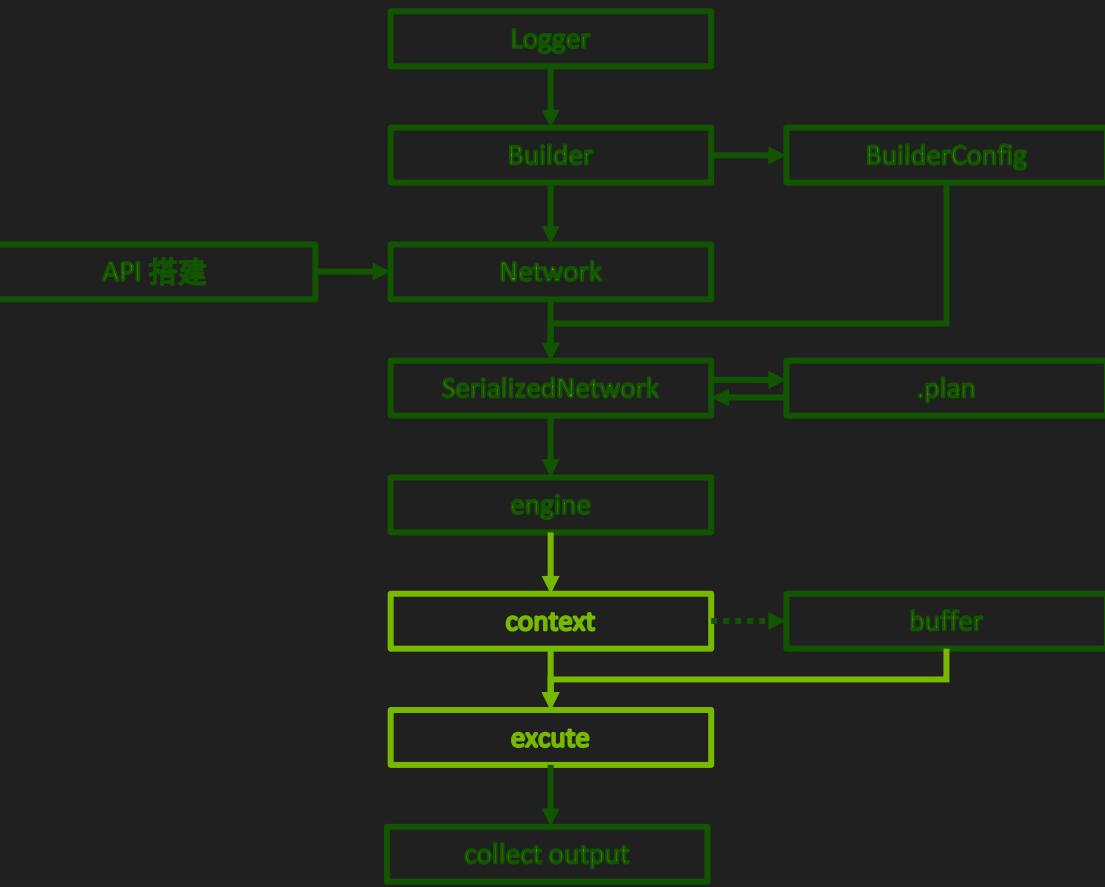
➤ `context.set_input_shape(iTensorName[i], shapeOfInputTensor)` 设定第 i 个张量的形状 (Dynamic Shape 模式中使用)

➤ `context.get_tensor_shape(iTensorName[i])` 获取第 i 个张量的形状

➤ `context.set_tensor_address(iTensorName[i], address)` 设定输入输出张量的指针

➤ `context.execute_async_v3(srteam)` Explicit batch 模式的异步执行

➤ 更多细节见 02-API/ExecutionContext



• Workflow: 使用 TensorRT API 搭建

➤ 问题

- 怎样从头开始写一个可以跑的网络?
- API搭建比较复杂，那么哪些代码是API搭建特有的，哪些是所有 Workflow 通用的?
- 怎么让一个 Network 跑起来?
- 用于推理计算的输入输出数据怎么准备?
- 构建引擎需要时间，怎么构建一次，反复使用?
- TensorRT 的开发环境?



• Workflow: 使用 TensorRT API 搭建

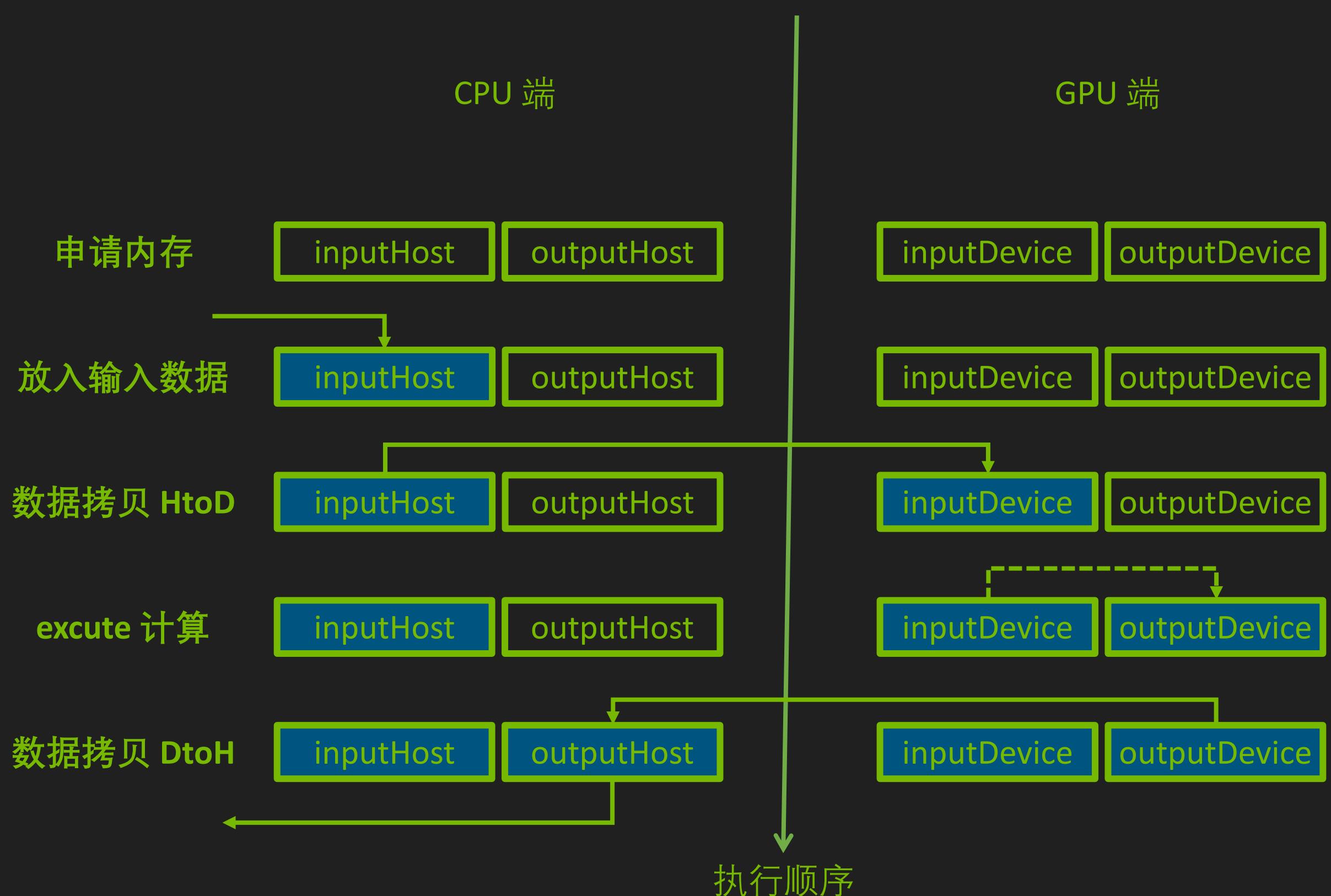
➤ CUDA 异构计算

➤ 同时准备 CPU 端内存和 GPU 端显存

➤ 开始计算前把数据从内存拷贝到显存中

➤ 计算过程的输入输出数据均在 GPU 端读写

➤ 计算完成后要把结果拷贝回内存才能使用



• Workflow: 使用 TensorRT API 搭建

➤ Buffer

➤ 内存和显存的申请

```
➤ inputHost = np.ascontiguousarray(inputData) # 不要忘了 ascontiguousarray!
```

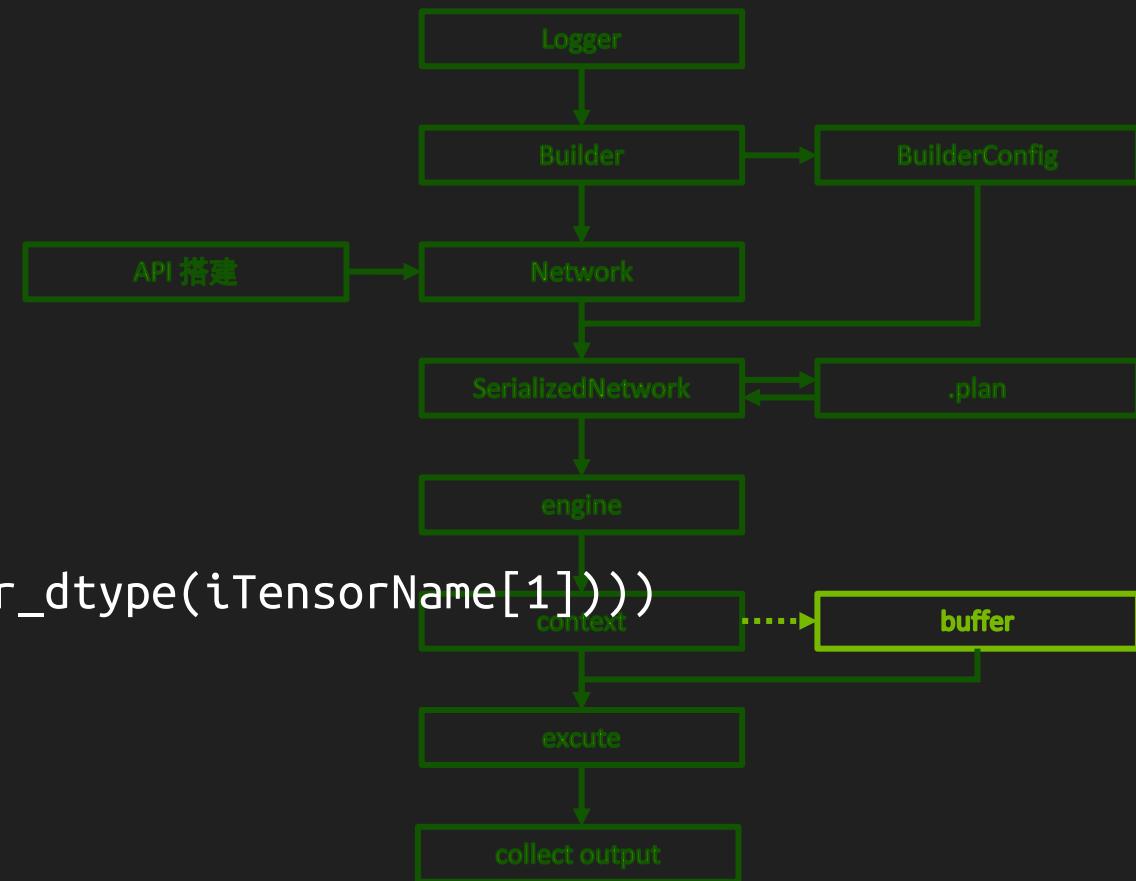
```
➤ outputHost = np.empty(context.get_tensor_shape(iTensorName[1]), trt.nptype(engine.get_tensor_dtype(iTensorName[1])))
```

```
➤ inputDevice = cudart.cudaMalloc(inputHost.nbytes)[1]
```

```
➤ outputDevice = cudart.cudaMalloc(outputHost.nbytes)[1]
```

```
➤ context.set_tensor_address(iTensorName[0], inputDevice)
```

```
➤ context.set_tensor_address(iTensorName[1], outputDevice)
```



➤ 内存和显存之间的拷贝

```
➤ cudart.cudaMemcpy(inputDevice, inputHost.ctypes.data, inputHost.nbytes, cudart.cudaMemcpyKind.cudaMemcpyHostToDevice)
```

```
➤ cudart.cudaMemcpy(outputHost.ctypes.data, outputDevice, outputHost.nbytes, cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost)
```

➤ 推理完成后释放显存

```
➤ cudart.cudaFree(inputDevice)
```

```
➤ cudart.cudaFree(outputDevice)
```



• Workflow: 使用 TensorRT API 搭建

➤ 问题

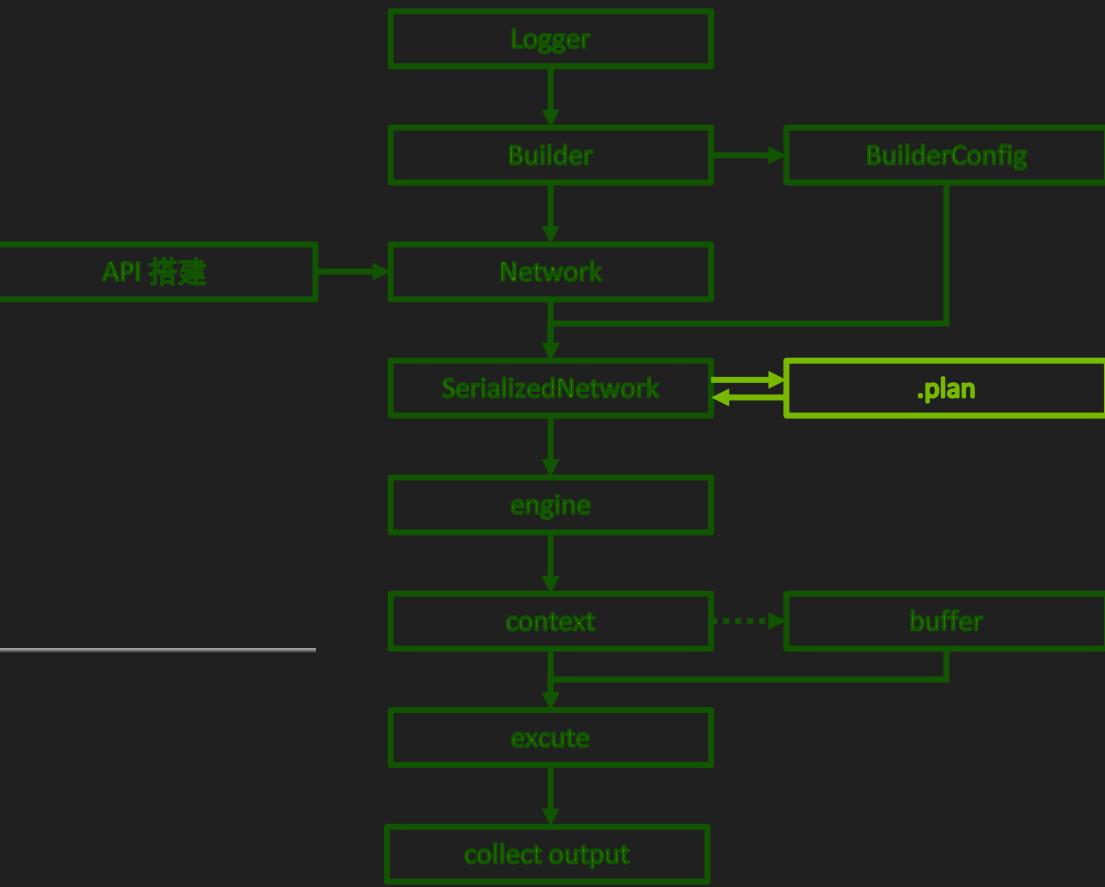
- 怎样从头开始写一个可以跑的网络?
- API搭建比较复杂，那么哪些代码是API搭建特有的，哪些是所有 Workflow 通用的?
- 怎么让一个 Network 跑起来?
- 用于推理计算的输入输出数据怎么准备?
- 构建引擎需要时间，怎么构建一次，反复使用?
- TensorRT 的开发环境?



• Workflow: 使用 TensorRT API 搭建

- 序列化与反序列化
- 将 SerializedNetwork 保存为文件，下次跳过构建直接使用
- 注意环境统一（硬件环境 + CUDA/cuDNN/TensorRT 环境）
 - Engine 包含硬件有关优化，不能跨硬件平台使用
 - 不同版本 TensorRT 生成的 engine 不能相互兼容
 - 同平台同环境多次生成的 engine 可能不同

```
1 if os.path.isfile(trtFile):  
2     with open(trtFile, 'rb') as f:  
3         serializedNetwork = f.read()  
4     else:  
5         ... # 构建网络  
6  
8     serializedNetwork = builder.build_serialized_network(network, config)  
9     with open(trtFile, 'wb') as f:  
10        f.write(serializedNetwork)  
11        print("Succeeded saving .plan file!")  
12  
13    engine = trt.Runtime(logger).deserialize_cuda_engine(serializedNetwork)
```



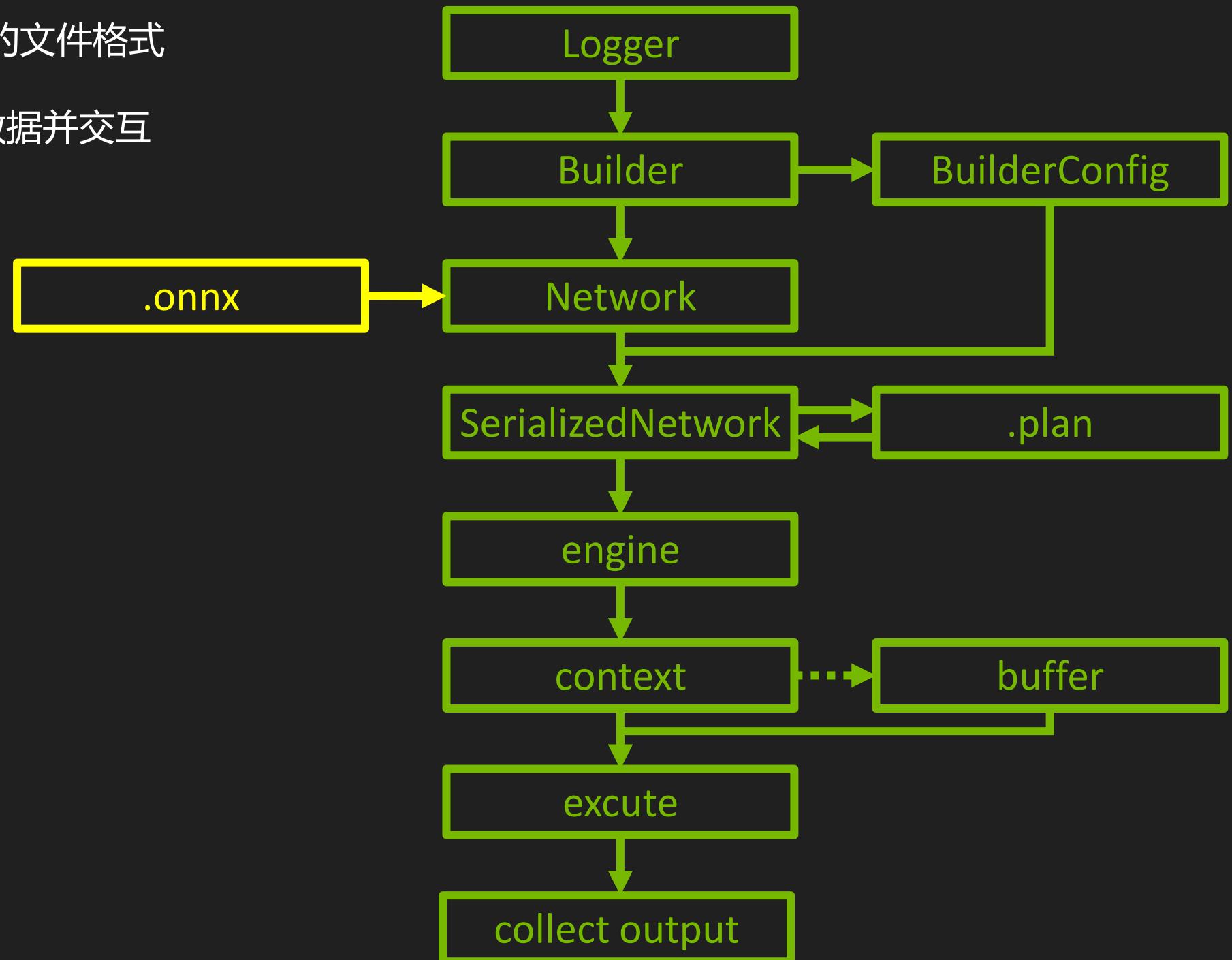
- TensorRT runtime 版本与 engine 版本不同时的报错信息
 - [TensorRT] ERROR: INVALID_CONFIG: The engine plan file is not compatible with this version of TensorRT, expecting library version 8.6.1.5 got 8.6.1.6, please rebuild.
 - [TensorRT] ERROR: engine.cpp (1646) - Serialization Error in deserialize: 0 (Core engine deserialization failure)
- 高级话题（见教程第四部分）：
 - 利用 AlgorithmSelector 或 TimingCache 多次生成一模一样的 engine
 - ## 利用 TensorRT 8.6 的 Hardware compatibility 和 Version compatibility 特性创建跨设备、跨 TensorRT 版本的 engine



• Workflow: 使用 Parser

➤ ONNX

- 简介 <https://onnx.ai/>
- Open Neural Network Exchange, 针对机器学习所设计的开放式的文件格式
- 用于存储训练好的模型, 使得不同框架可以采用相同格式存储模型数据并交互
- TensorFlow / pyTorch 模型转 TensorRT 的中间表示
- 当前 TensorRT 导入模型的主要途径



➤ Onnxruntime

- 简介 <https://onnxruntime.ai/>
- 利用 onnx 格式模型进行推理计算的框架
- 兼容多硬件、多操作系统, 支持多深度学习框架
- 可用于检查 TensorFlow / Torch 模型导出到 Onnx 的正确性



• Workflow: 使用 Parser

- pyTorch 的例子
- 范例代码 04-BuildEngineByONNXParser/pyTorch-ONNX-TensorRT

- 基本流程:
 - pyTorch 中创建网络并保存为 .pt 文件
 - 使用 pyTorch 内部 API 将 .pt 转化为 .onnx
 - TensorRT 中读取 .onnx 构建 engine 并做推理

- 本例在 TensorRT 中开启了 Int8 模式
 - 需要自己实现 calibrator 类 (calibrator.py 可作为 Int8 通用样例)

```
1 # pyTorch 中创建网络并保存为 .pt 文件 -----
2 # ...
3
4 t.save(net, ptFile)
5 print("Succeeded building model in pyTorch!")
6
7 # 将 .pt 文件转换为 .onnx 文件 -----
8 t.onnx.export(net,
9               t.randn(1, 1, imageHeight, imageWidth, device="cuda"),
10              "./model.onnx",
11              example_outputs=[t.randn(1,10,device="cuda"),t.randn(1,device="cuda")],
12              input_names=['x'],
13              output_names=['y', 'z'],
14              do_constant_folding=True,
15              verbose=True,
16              keep_initializer_as_inputs=True,
17              opset_version=12,
18              dynamic_axes={"x":{0:"nBatchSize"}, "z":{0: "nBatchSize"}})
19 print("Succeeded converting model into onnx!")
20
21 # TensorRT 中加载 .onnx 创建 engine -----
22 logger = trt.Logger(trt.Logger.ERROR)
23
24 # ...
25
26 # 关键部分，用 Parser 加载 .onnx
27 with open(onnxFile, 'rb') as model:
28     if not parser.parse( model.read() ):
29         print ("Failed parsing ONNX file!")
30         for error in range(parser.num_errors):
31             print (parser.get_error(error))
32             exit()
33         print ("Succeeded parsing ONNX file!")
34
35 # ...
36
37 # 准备 TensorRT runtime 和 buffer，并进行推理
38 context = engine.create_execution_context()
39
40 #...
41
42 print("Succeeded running model in TensorRT!")
```



• Workflow: 使用 Parser

➤ TensorFlow 的例子

➤ 范例代码 04-BuildEngineByONNXParser/
TensorFlow*-ONNX-TensorRT

➤ 基本流程:

➤ 中创建网络并保存为 .pb 文件

➤ 使用 tf2onnx 将 .pb 转化为 ONNX 文件

➤ TensorRT 中读取 ONNX 构建 engine 并做推理

```
133 # Export model as ONNX file -----
134 if bSinglePbFile:
135     os.system("python3 -m tf2onnx.convert --input      %s --output %s --inputs-as-nchw 'Input'
136     else:
137         os.system("python3 -m tf2onnx.convert --saved-model %s --output %s --inputs-as-nchw 'Input'
138     print("Succeeded converting model into ONNX!")
139
140 # Parse network, rebuild network and do inference in TensorRT -----
141 logger = trt.Logger(trt.Logger.VERBOSE)
142 builder = trt.Builder(logger)
143 network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
144 profile = builder.create_optimization_profile()
145 config = builder.create_builder_config()
146 if bUseFP16Mode:
147     config.set_flag(trt.BuilderFlag.FP16)
148 if bUseINT8Mode:
149     config.set_flag(trt.BuilderFlag.INT8)
150     config.int8_calibrator = calibrator.MyCalibrator(calibrationDataPath, nCalibration, (1, 1))
151
152 parser = trt.OnnxParser(network, logger)
153 if not os.path.exists(onnxFile):
154     print("Failed finding ONNX file!")
155     exit()
156 print("Succeeded finding ONNX file!")
157 with open(onnxFile, "rb") as model:
158     if not parser.parse(model.read()):
159         print("Failed parsing .onnx file!")
160         for error in range(parser.num_errors):
161             print(parser.get_error(error))
162         exit()
163     print("Succeeded parsing .onnx file!")
```



• Workflow: 使用 Parser

- 如果不用 Int8 模式, onnx parser 的代码几乎是通用的
 - 有命令行工具可用, 基本等价于脚本中的 API (见教程第二部分 trtexec)
- 遇到 TensorRT 不支持的节点
 - 修改源模型
 - 修改 Onnx 计算图 (见教程第二部分 onnx-surgeon)
 - TensorRT 中实现 Plugin (见教程第三部分)
 - 修改 Parser: 修改 TRT 源码并重新编译 TRT (见 <https://github.com/NVIDIA/TensorRT> 项目)
- trtexec, onnx-graphsurgeon 和 plugin 都是使用 parser 的必备知识, 将在后面的教程展开



• Workflow: 使用框架内 TensorRT 接口

- TF-TRT
 - 范例代码 06-UseFrameworkTRT/TensorFlow*-TFTRT
- Torch-TensorRT (旧名 TRTorch)
 - 范例代码 06-UseFrameworkTRT/Torch-TensorRT



• Workflow：使用 TensorRT API 搭建

➤ 问题

- 怎样从头开始写一个可以跑的网络？
- API搭建比较复杂，那么哪些代码是API搭建特有的，哪些是所有 Workflow 通用的？
- 怎么让一个 Network 跑起来？
- 用于推理计算的输入输出数据怎么准备？
- 构建引擎需要时间，怎么构建一次，反复使用？
- TensorRT 的开发环境？



• TensorRT 环境

- 推荐使用 NVIDIA-optimized Docker
 - 安装步骤: <https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#docker>
 - 镜像列表: <https://docs.nvidia.com/deeplearning/frameworks/support-matrix/index.html>
 - 推荐镜像: nvcr.io/nvidia/pytorch:23.03-py3 (pyTorch1.14+TRT8.5.2) , nvcr.io/nvidia/pytorch:23.04-py3 (pyTorch2.0+TRT8.6.1)
- python 库 (见 cookbook requirements.txt)
 - nvidia-pyindex, cuda-python (python \geq 3.7) , pycuda, onnx, onnx-surgeon, onnxruntime-gpu, opencv-python, polygraphy
- 推荐使用:
 - 最新的 TensorRT8, 图优化内容更多, 优化过程和推理过程显存使用量更少
 - builderConfig API, 功能覆盖旧版 builder API, 旧版 builder API 将被废弃
 - explicit batch 模式 + dynamic shape 模式, ONNX 格式默认模式, 灵活性和应用场景更广, 使模型适应性更好
 - cuda-python 库, 完整的 CUDA API 支持, 修复 pycuda 库可能存在的问题 (如与其他框架交互使用时的同步操作等)



Part 2

• 开发辅助工具

- trtexec TensorRT 命令行工具, 主要的 End2End 性能测试工具
- Netron 网络可视化
- onnx-graphsurgeon onnx 计算图编辑
- polygraphy 结果验证与定位, 图优化
- Nsight Systems 性能分析



• 开发辅助工具

➤ 希望解决的问题

- | | |
|--------------------------------|----------------------|
| ➤ 不想写脚本来跑 TensorRT | 用命令行搞定 |
| ➤ 怎么进行简单的推理性能测试? | 测量延迟、吞吐量等 |
| ➤ 网络结构可视化? | |
| ➤ 计算图上有些节点阻碍 TensorRT 自动优化 | 手工调整以便 TensorRT 能够处理 |
| ➤ 怎么处理 TensorRT 不支持的网络结构? | 手工调整以便 TensorRT 能够处理 |
| ➤ 怎么检验 TensorRT 上计算结果正确性 / 精度? | 同时在原框架和 TensorRT 上运行 |
| ➤ 怎么找出计算错误 / 精度不足的层? | 模型逐层比较 |
| ➤ 怎么进行简单的计算图优化? | 手工调整 |
| ➤ 怎样找出最耗时的层? | 找到热点集中优化 |



• trtexec

➤ TensorRT 的命令行工具,

➤ 随 TensorRT 安装, 位于 tensorrt-XX/bin/trtexec

➤ docker 中位于: /opt/tensorrt/bin/

➤ 功能

➤ 由 ONNX 文件生成 TensorRT 引擎并序列化为 Plan 文件

➤ 查看 ONNX 文件或 Plan 文件的网络逐层信息

➤ 模型性能测试 (测试 TensorRT 引擎基于随机输入或给定输入下的性能)

➤ 范例代码

➤ 07-Tool/trtexec, 运行 ./command.sh

```
1  clear
2
3  rm ./*.pb ./*.onnx ./*.plan ./result-*.*txt
4
5  # 从 TensorFlow 创建一个 .onnx 用来做 trtexec 的输入文件
6  python3 getOnnxModel.py
7
8  # 用上面的 .onnx 构建一个 TensorRT 引擎并作推理
9  trtexec \
10   --onnx=model.onnx \
11   --minShapes=x:0:1x1x28x28 \
12   --optShapes=x:0:4x1x28x28 \
13   --maxShapes=x:0:16x1x28x28 \
14   --workspace=1024 \
15   --saveEngine=model-FP32.plan \
16   --shapes=x:0:4x1x28x28 \
17   --verbose \
18   > result-FP32.txt
19
20  # 用上面的 .onnx 构建一个 TensorRT 引擎并作推理, 使用 FP16 模式
21  trtexec \
22   --onnx=model.onnx \
23   --minShapes=x:0:1x1x28x28 \
24   --optShapes=x:0:4x1x28x28 \
25   --maxShapes=x:0:16x1x28x28 \
26   --workspace=1024 \
27   --saveEngine=model-FP16.plan \
28   --shapes=x:0:4x1x28x28 \
29   --verbose \
30   --fp16 \
31   > result-FP16.txt
32
33  # 读取上面构建的 result-FP32.plan 并作推理
34  trtexec \
35   --loadEngine=./model-FP32.plan \
36   --shapes=x:0:4x1x28x28 \
37   --verbose \
38   > result-load-FP32.txt
```



• trtexec

➤ 常用选项，构建阶段

- `--onnx=./model-NCHW.onnx` 指定输入模型文件名
- `--output=y:0` 指定输出张量名（使用 ONNX 时该选项无效）
- `--minShapes=x:0:1x1x28x28 --optShapes=x:0:4x1x28x28 --maxShapes=x:0:16x1x28x28` 指定输入形状的范围最小值、最常见值、最大值
- `--memPoolSize=workspace:1024MiB` 优化过程可使用显存最大值 ##
- `--fp16, --int8, --noTF32, --best, --sparsity=...` 指定引擎精度和稀疏性等属性
- `--saveEngine=./model.plan` 指定输出引擎文件名
- `--skipInference` 只创建引擎不运行
- `--verbose` 打印详细日志
- `--preview=profileSharing0806` 启用某些 preview 功能 ##
- `--builderOptimizationLevel=5` 设置优化等级（默认值 2） ##
- `--timingCacheFile=timing.cache` 指定输出优化计时缓存文件名^[1]
- `--profilingVerbosity=detailed` 构建期保留更多的逐层信息
- `--dumpLayerInfo, --exportLayerInfo=layerInfo.txt` 导出引擎逐层信息，可与 profilingVerbosity 合用^[2]



- **trtexec**

- 常用选项，运行阶段

- `--loadEngine=model.plan` 读取 engine 文件，而不是输入 ONNX 文件
- `--shapes=x:0:1x1x28x28` 指定输入张量形状
- `--warmUp=1000` 热身阶段最短运行时间（单位：ms）
- `--duration=10` 测试阶段最短运行时间（单位：s）
- `--iterations=100` 指定测试阶段运行的最小迭代次数
- `--useCudaGraph` 使用 CUDAGraph 来捕获和执行推理过程^[1]
- `--noDataTransfers` 关闭 Host 和 Device 之间的数据传输
- `--streams=2` 使用多个 stream 来运行推理^[2]
- `--verbose` 打印详细日志
- `--dumpProfile, --exportProfile=layerProfile.txt` 保存逐层性能数据信息

➤ [1][2] 教程第四部分有单独的使用介绍



• trtexec

➤ 性能测试结果

```

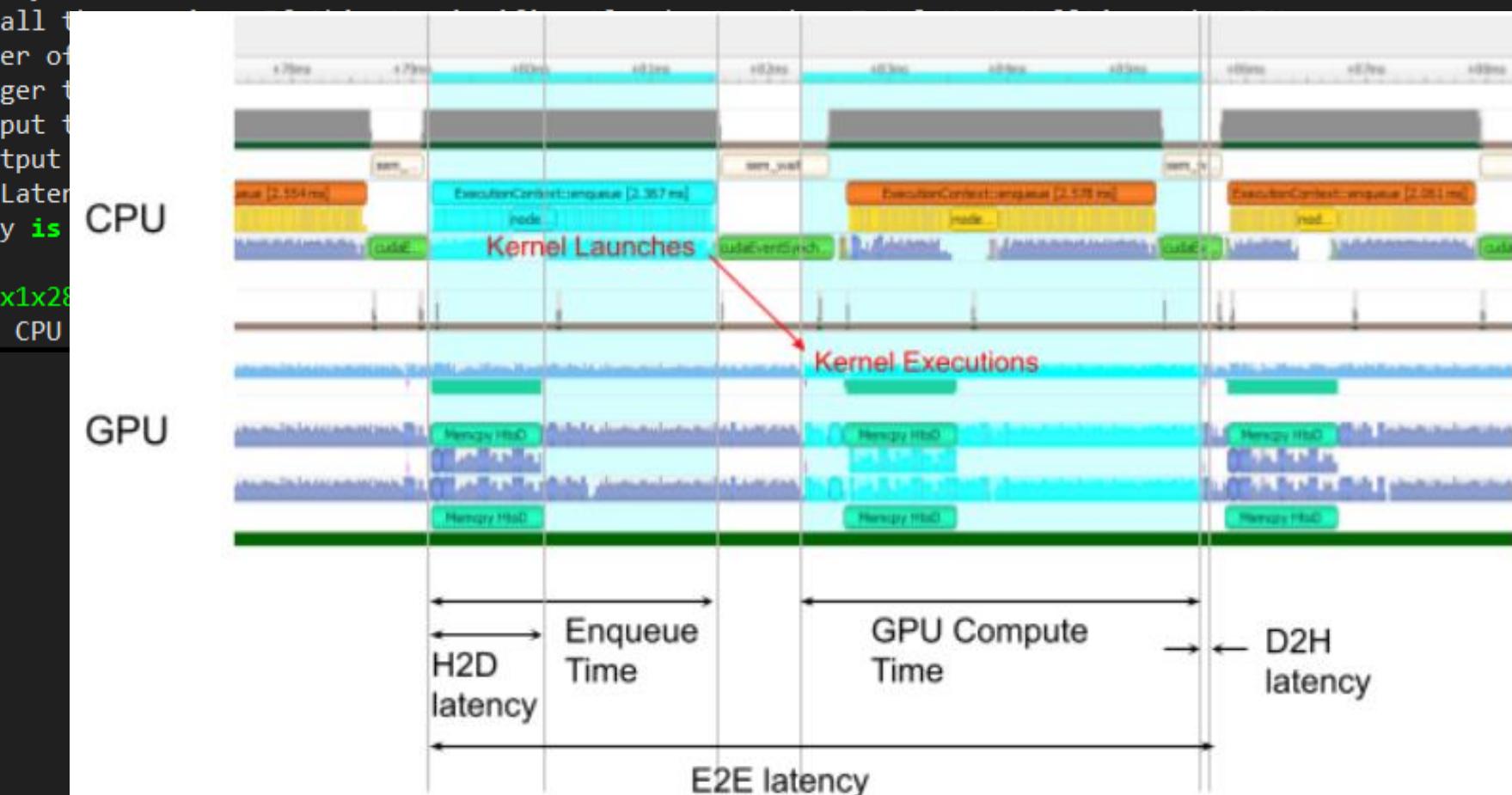
1 [03/07/2022-05:05:52] [I] === Performance summary ===
2 [03/07/2022-05:05:52] [I] Throughput: 6430.89 qps
3 [03/07/2022-05:05:52] [I] Latency: min = 0.156982 ms, max = 0.384766 ms, mean = 0.160129 ms, median = 0.159668 ms, percentile(99%) = 0.165771 ms
4 [03/07/2022-05:05:52] [I] End-to-End Host Latency: min = 0.171387 ms, max = 0.504395 ms, mean = 0.284758 ms, median = 0.289673 ms, percentile(99%) = 0.298828 ms
5 [03/07/2022-05:05:52] [I] Enqueue Time: min = 0.0300293 ms, max = 0.126465 ms, mean = 0.0396717 ms, median = 0.0333252 ms, percentile(99%) = 0.0698242 ms
6 [03/07/2022-05:05:52] [I] H2D Latency: min = 0.00317383 ms, max = 0.0455933 ms, mean = 0.00408154 ms, median = 0.00366211 ms, percentile(99%) = 0.00854492 ms
7 [03/07/2022-05:05:52] [I] GPU Compute Time: min = 0.151367 ms, max = 0.373779 ms, mean = 0.15368 ms, median = 0.153564 ms, percentile(99%) = 0.156677 ms
8 [03/07/2022-05:05:52] [I] D2H Latency: min = 0.00170898 ms, max = 0.0136719 ms, mean = 0.00237128 ms, median = 0.00219727 ms, percentile(99%) = 0.00341797 ms
9 [03/07/2022-05:05:52] [I] Total Host Walltime: 3.00036 s
10 [03/07/2022-05:05:52] [I] Total GPU Compute Time: 2.96525 s
11 [03/07/2022-05:05:52] [I] Explanations of the performance metrics are printed below.
12 [03/07/2022-05:05:52] [V]
13 [03/07/2022-05:05:52] [V] === Explanations of the performance metrics ===
14 [03/07/2022-05:05:52] [V] Total Host Walltime: the host walltime from when the first query (after warmups) is enqueued to when the last query is completed.
15 [03/07/2022-05:05:52] [V] GPU Compute Time: the GPU latency to execute the kernels for a query.
16 [03/07/2022-05:05:52] [V] Total GPU Compute Time: the summation of the GPU Compute Time of all the queries.
17 [03/07/2022-05:05:52] [V] Throughput: the observed throughput computed by dividing the number of queries by the total host walltime.
18 [03/07/2022-05:05:52] [V] Enqueue Time: the host latency to enqueue a query. If this is longer than the enqueue time limit, the enqueue will fail.
19 [03/07/2022-05:05:52] [V] H2D Latency: the latency for host-to-device data transfers for input tensors.
20 [03/07/2022-05:05:52] [V] D2H Latency: the latency for device-to-host data transfers for output tensors.
21 [03/07/2022-05:05:52] [V] Latency: the summation of H2D Latency, GPU Compute Time, and D2H Latency.
22 [03/07/2022-05:05:52] [V] End-to-End Host Latency: the duration from when the H2D of a query is enqueued to when the D2H of the same query is completed.
23 [03/07/2022-05:05:52] [I]
24 &&& PASSED TensorRT.trtexec [TensorRT v8003] # trtexec --onnx=model.onnx --minShapes=x:0:1x1x28
25 [03/07/2022-05:05:52] [I] [TRT] [MemUsageChange] Init cuBLAS/cuBLASLt: CPU +0, GPU +0, now: CPU +0, GPU +0

```

吞吐量: 每秒完成 query 数量

各种延迟的统计结果

各种延迟的解释



$$\text{Host latency} = (\text{H2D latency}) + (\text{GPU Compute latency}) + (\text{D2H latency})$$

$$\text{Throughput} = (\text{Total host wall time for N queries}) / N$$



• 开发辅助工具

- | 希望解决的问题 | 方法 |
|------------------------------|----------------------|
| 不想写脚本来跑 TensorRT | 用命令行搞定 |
| 怎么进行简单的推理性能测试？ | 测量延迟、吞吐量等 |
| 网络结构可视化？ | |
| 计算图上有些节点阻碍 TensorRT 自动优化 | 手工调整以便 TensorRT 能够处理 |
| 怎么处理 TensorRT 不支持的网络结构？ | 手工调整以便 TensorRT 能够处理 |
| 怎么检验 TensorRT 上计算结果正确性 / 精度？ | 同时在原框架和 TensorRT 上运行 |
| 怎么找出计算错误 / 精度不足的层？ | 模型逐层比较 |
| 怎么进行简单的计算图优化？ | 手工调整 |
| 怎样找出最耗时的层？ | 找到热点集中优化 |



• Netron

➤ 模型网络可视化工具

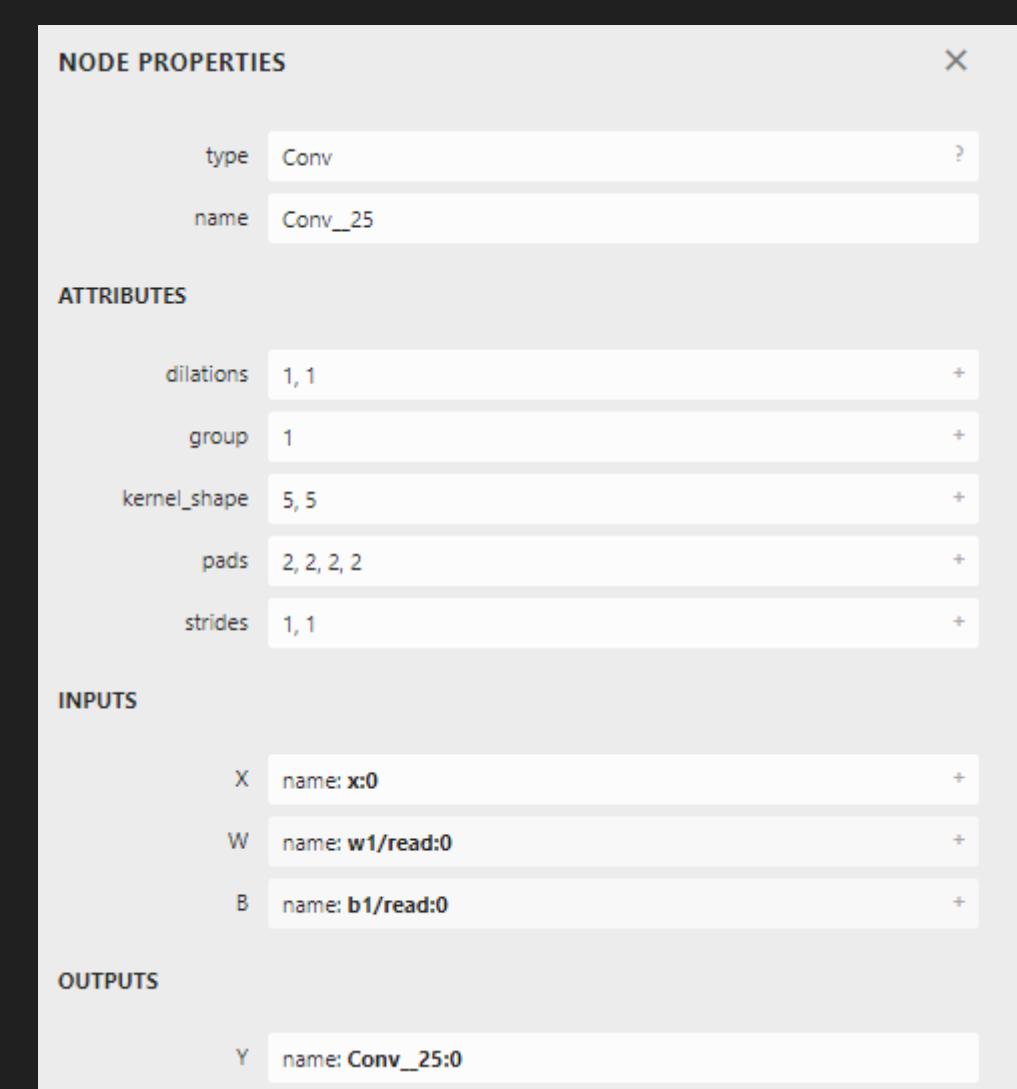
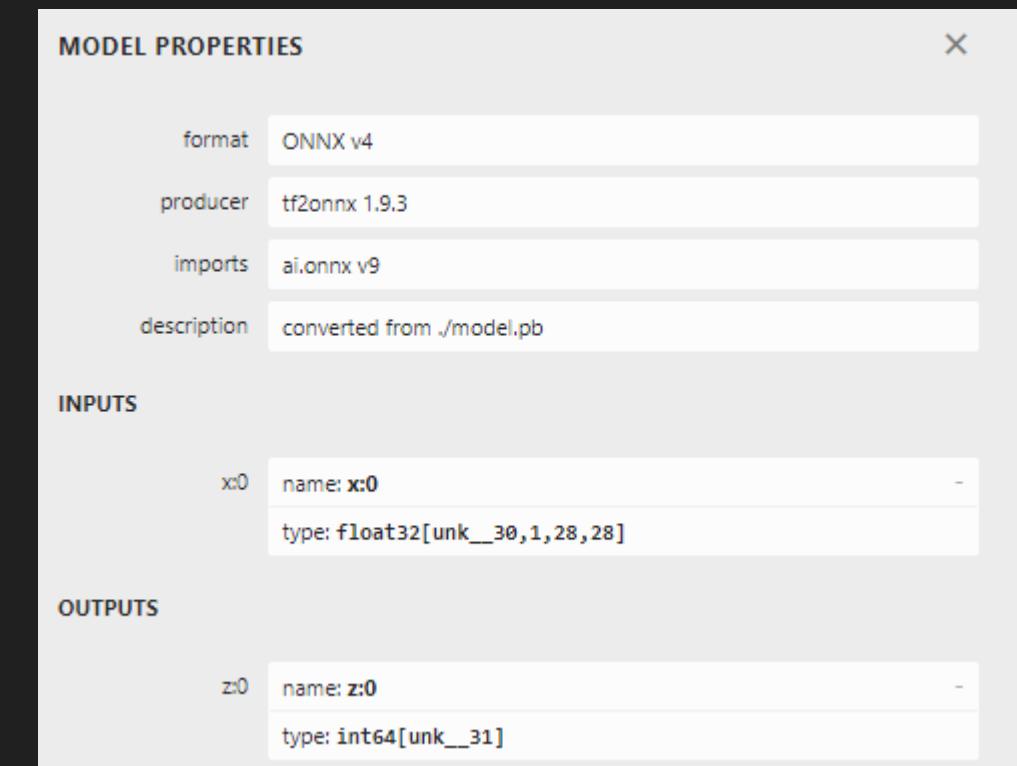
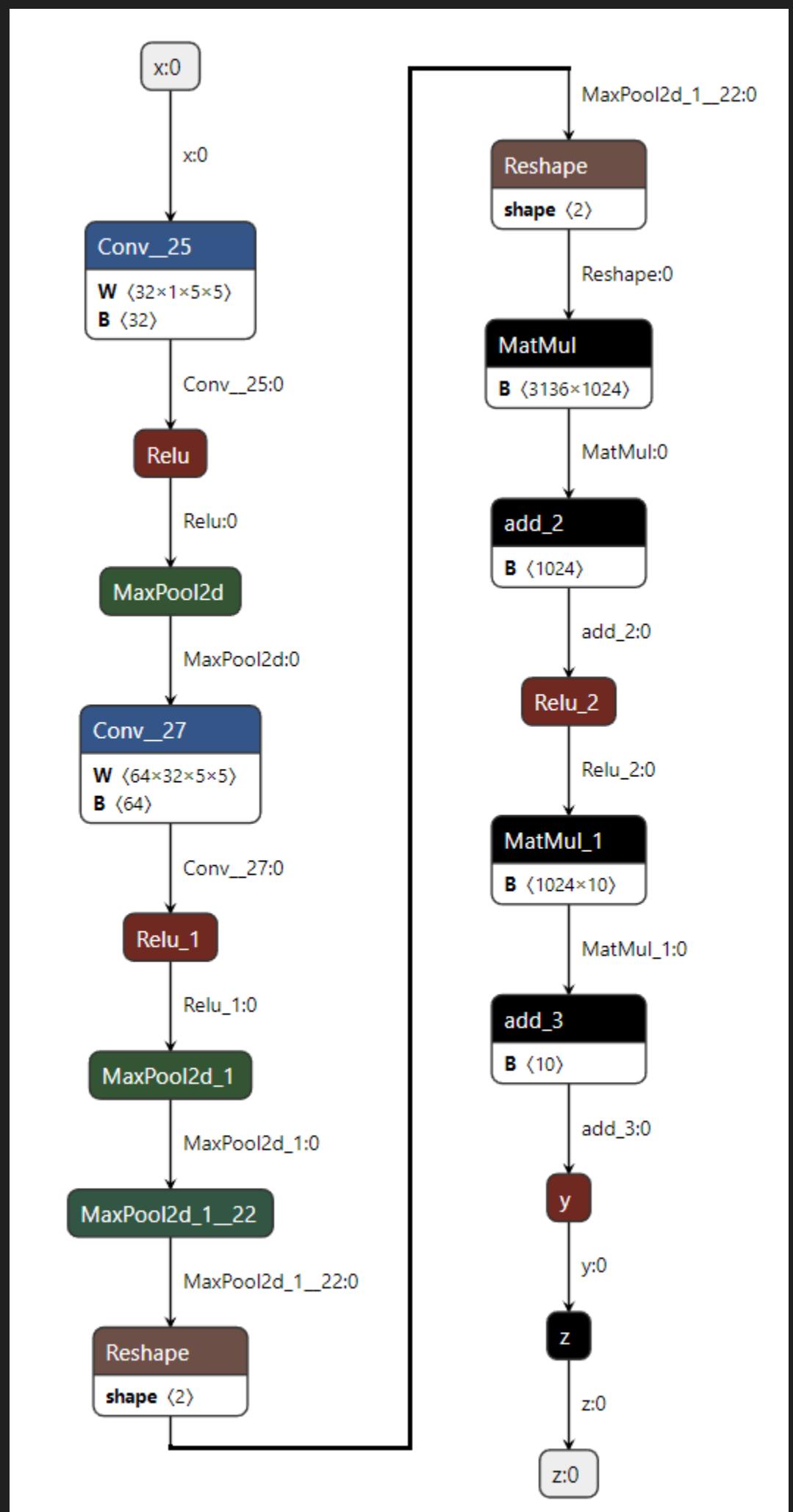
➤ 下载安装

➤ <https://github.com/lutzroeder/Netron>

➤ 查看网络结构

➤ 查看计算图信息

➤ 查看节点信息



• 开发辅助工具

➤ 希望解决的问题

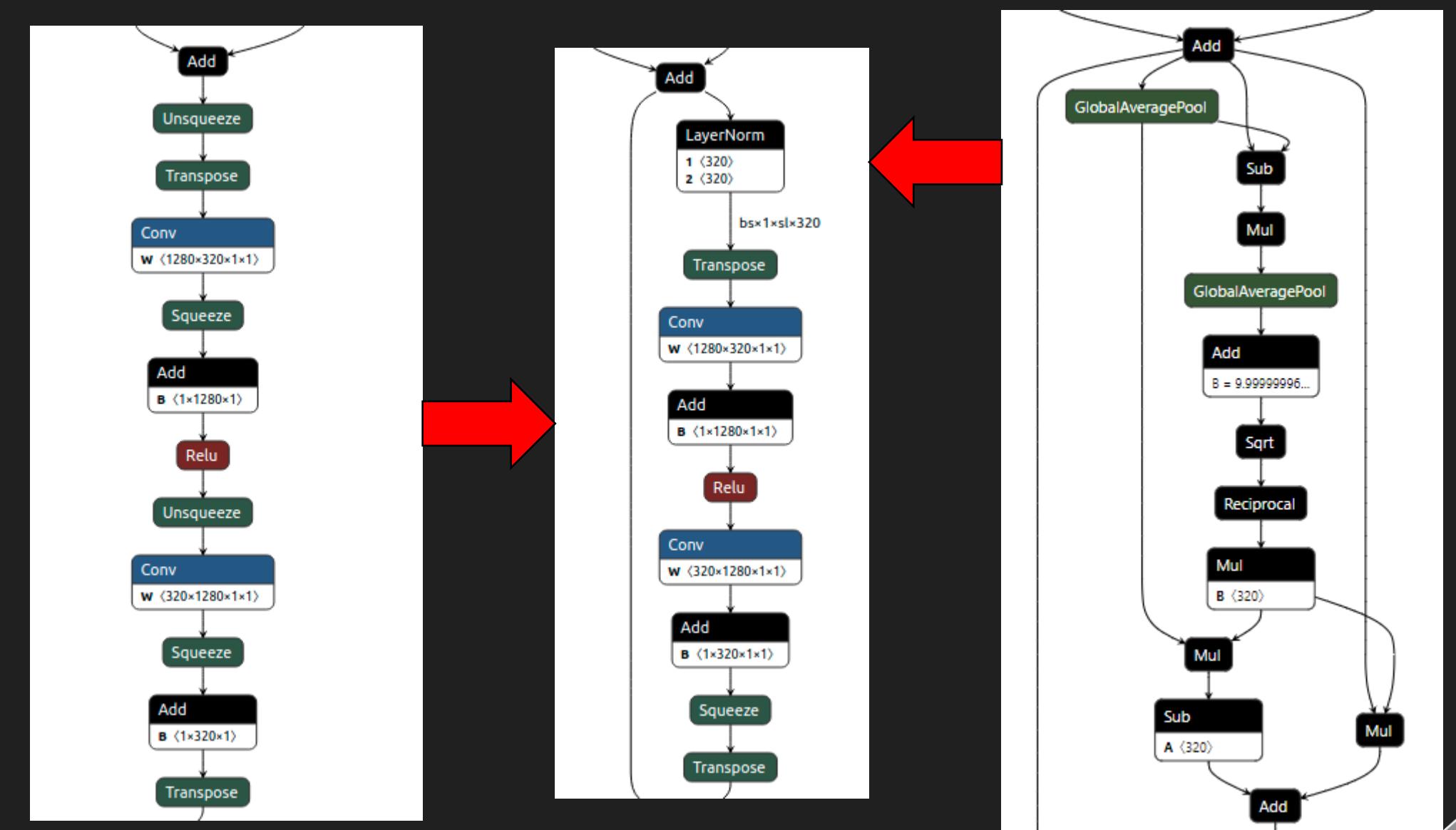
- | | |
|--------------------------------|----------------------|
| ➤ 不想写脚本来跑 TensorRT | 用命令行搞定 |
| ➤ 怎么进行简单的推理性能测试? | 测量延迟、吞吐量等 |
| ➤ 网络结构可视化? | |
| ➤ 计算图上有些节点阻碍 TensorRT 自动优化 | 手工调整以便 TensorRT 能够处理 |
| ➤ 怎么处理 TensorRT 不支持的网络结构? | 手工调整以便 TensorRT 能够处理 |
| ➤ 怎么检验 TensorRT 上计算结果正确性 / 精度? | 同时在原框架和 TensorRT 上运行 |
| ➤ 怎么找出计算错误 / 精度不足的层? | 模型逐层比较 |
| ➤ 怎么进行简单的计算图优化? | 手工调整 |
| ➤ 怎样找出最耗时的层? | 找到热点集中优化 |



• onnx-graphsurgeon

- 需要手工修改网络的情形?
- 冗余节点
- 阻碍 TensorRT 融合的节点组合
- 可以手工模块化的节点

➤ 更多范例: 09-BestPractice



• **onnx-graphsurgeon**

➤ ONNX 模型的编辑器，包含 python API (下面简称 ogs)

➤ 功能：

➤ 修改计算图：图属性 / 节点 / 张量 / 节点和张量的连接 / 权重

➤ 修改子图：添加 / 删除 / 替换 / 隔离

➤ 优化计算图：常量折叠 / 拓扑排序 / 去除无用层

➤ 功能和 API 上有别于 onnx 库

➤ 下载和参考文档

➤ pip install nvidia-pyindex onnx-graphsurgeon

➤ <https://github.com/NVIDIA/TensorRT/tree/master/tools/onnx-graphsurgeon/examples>

➤ <https://docs.nvidia.com/deeplearning/tensorrt/onnx-graphsurgeon/docs/index.html>



- **onnx-graphsurgeon**

- 范例代码

- 07-Tool/OnnxGraphSurgeon，运行 test.sh
 - 共有 9 个例子，包含创建模型、隔离子图、替换节点、常量折叠、删除节点、**shape** 操作

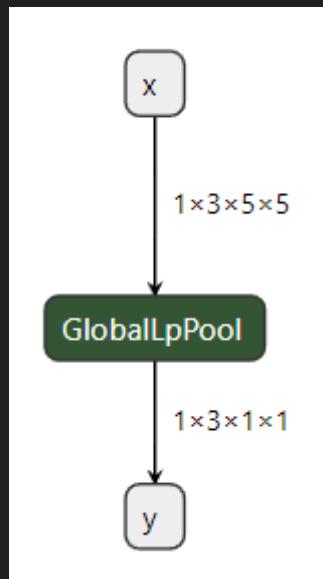


• onnx-graphsurgeon

➤ 01-CreateModel

➤ 使用 ogs 创建 onnx 格式的模型

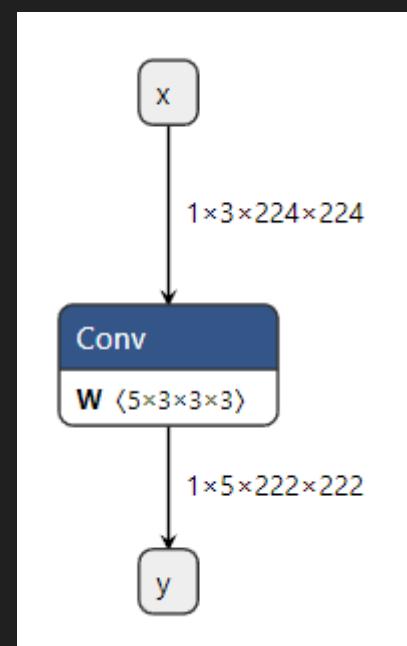
```
1 import onnx
2 import onnx_graphsurgeon as gs
3 import numpy as np
4
5 x = gs.Variable(name="x", dtype=np.float32, shape=[1, 3, 5, 5])
6 y = gs.Variable(name="y", dtype=np.float32, shape=[1, 3, 1, 1])
7 node = gs.Node(op="GlobalLpPool", attrs={"p": 2}, inputs=[x], outputs=[y])
8
9 graph = gs.Graph(nodes=[node], inputs=[x], outputs=[y])
10 onnx.save(gs.export_onnx(graph), "01-CreateModel.onnx")
```



➤ 02-CreateModelWithInitializer

➤ 使用 ogs 创建 onnx 格式的模型，具有带权重的节点

```
1 import onnx
2 import onnx_graphsurgeon as gs
3 import numpy as np
4
5 x = gs.Variable(name="x", dtype=np.float32, shape=(1, 3, 224, 224))
6 y = gs.Variable(name="y", dtype=np.float32, shape=(1, 5, 222, 222))
7 w = gs.Constant(name="w", values=np.ones(shape=(5, 3, 3, 3), dtype=np.float32))
8
9 # w 将被当做 initializer 合并到 Conv 节点中
10 node = gs.Node(op="Conv", inputs=[x, w], outputs=[y])
11
12 graph = gs.Graph(nodes=[node], inputs=[x], outputs=[y])
13 onnx.save(gs.export_onnx(graph), "02-CeateModelWithInitializer.onnx")
```

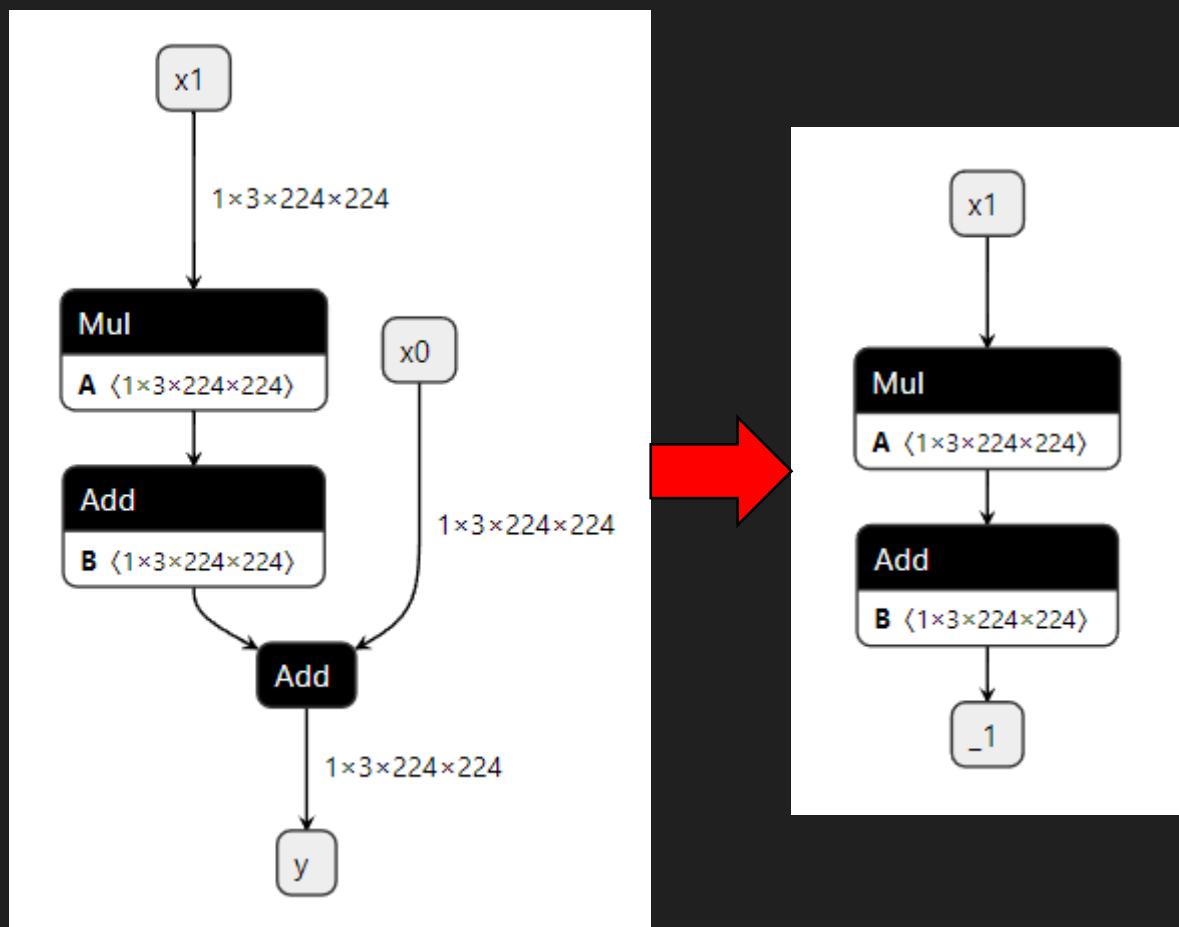


• onnx-graphsurgeon

➤ 03-IsolateSubgraph

- 从计算图中取一个子图出来

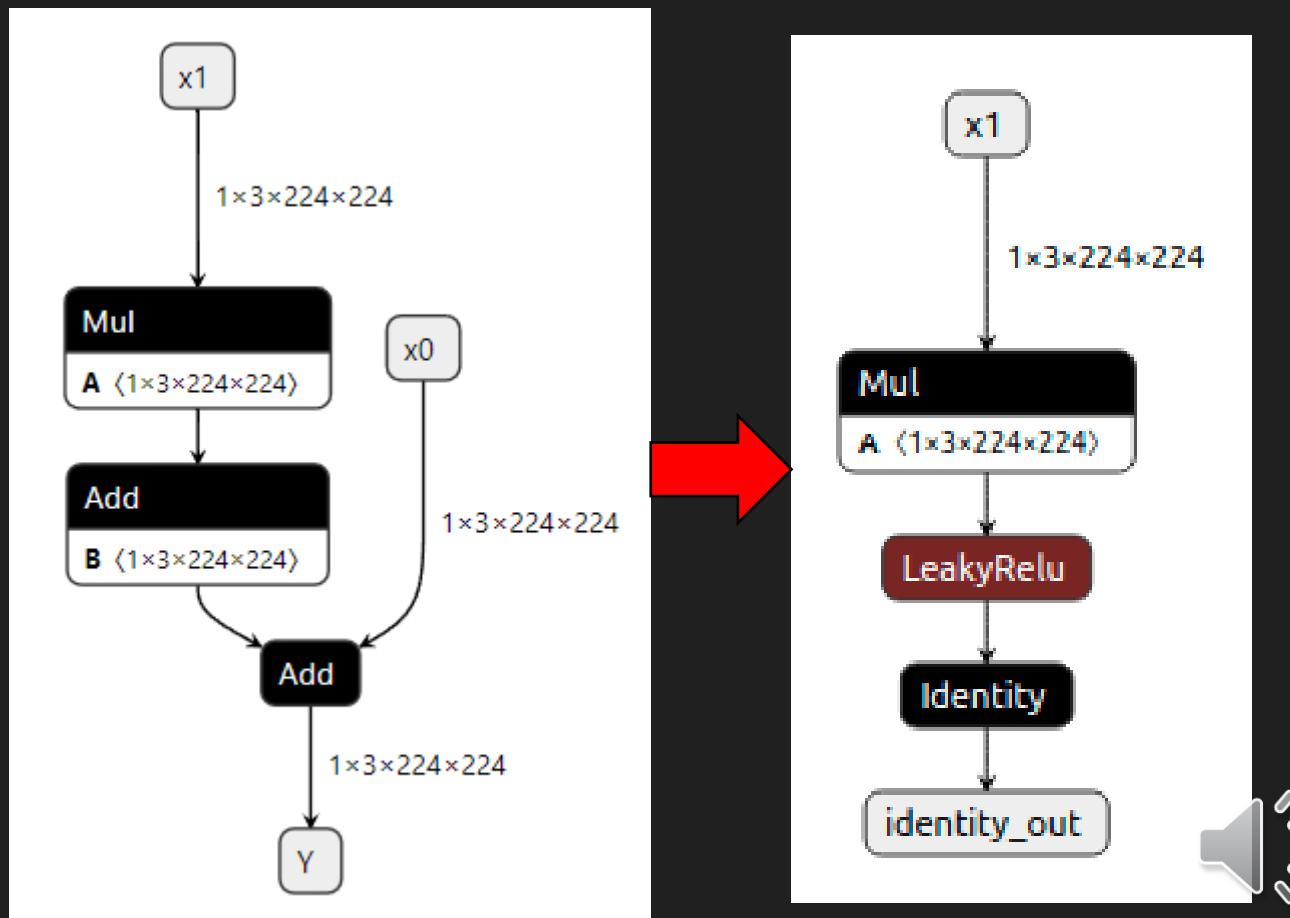
```
1 graph = gs.import_onnx(onnx.load("03-IsolateSubgraph_0.onnx"))
2 tensors = graph.tensors()
3
4 graph.inputs = [tensors["x1"].to_variable(dtype=np.float32)]
5 graph.outputs = [tensors["_1"].to_variable(dtype=np.float32)]
6
7 graph.cleanup()
8 onnx.save(gs.export_onnx(graph), "03-IsolateSubgraph_1.onnx")
```



➤ 04-ModifyModel

- 替换并插入一个新节点

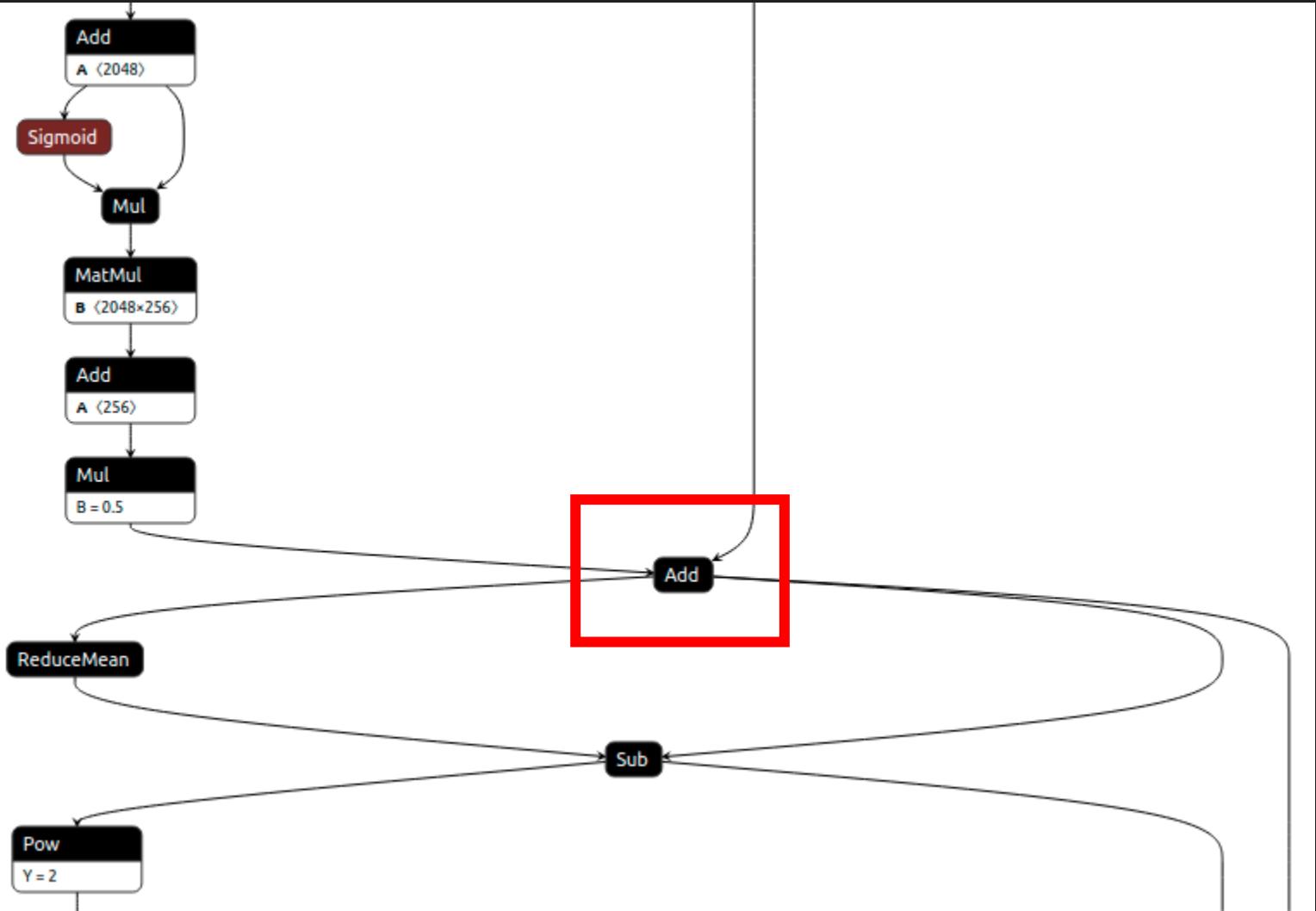
```
1 graph = gs.import_onnx(onnx.load("04-ModifyModel_0.onnx"))
2
3 # 找到第一个 Add 节点, 去掉加数 "b"
4 first_add = [node for node in graph.nodes if node.op == "Add"][0]
5 first_add.inputs = [inp for inp in first_add.inputs if inp.name != "b"]
6
7 # 将第一个 Add 节点替换为 LeakyRelu 节点
8 first_add.op = "LeakyRelu"
9 first_add.attrs["alpha"] = 0.02
10
11 # 在第一个 Add 节点后插入一个 Identity 节点
12 identity_out = gs.Variable("identity_out", dtype=np.float32)
13 identity = gs.Node(op="Identity", inputs=first_add.outputs, outputs=[identity_out])
14 graph.nodes.append(identity)
15
16 # 修改计算图输入输出
17 graph.inputs = [graph.inputs[1]]
18 graph.outputs = [identity_out]
19
20 graph.cleanup().toposort()
21 onnx.save(gs.export_onnx(graph), "04-ModifyModel_1.onnx")
```



• onnx-graphsurgeon

➤ Ogs 的 Node 和 Variable

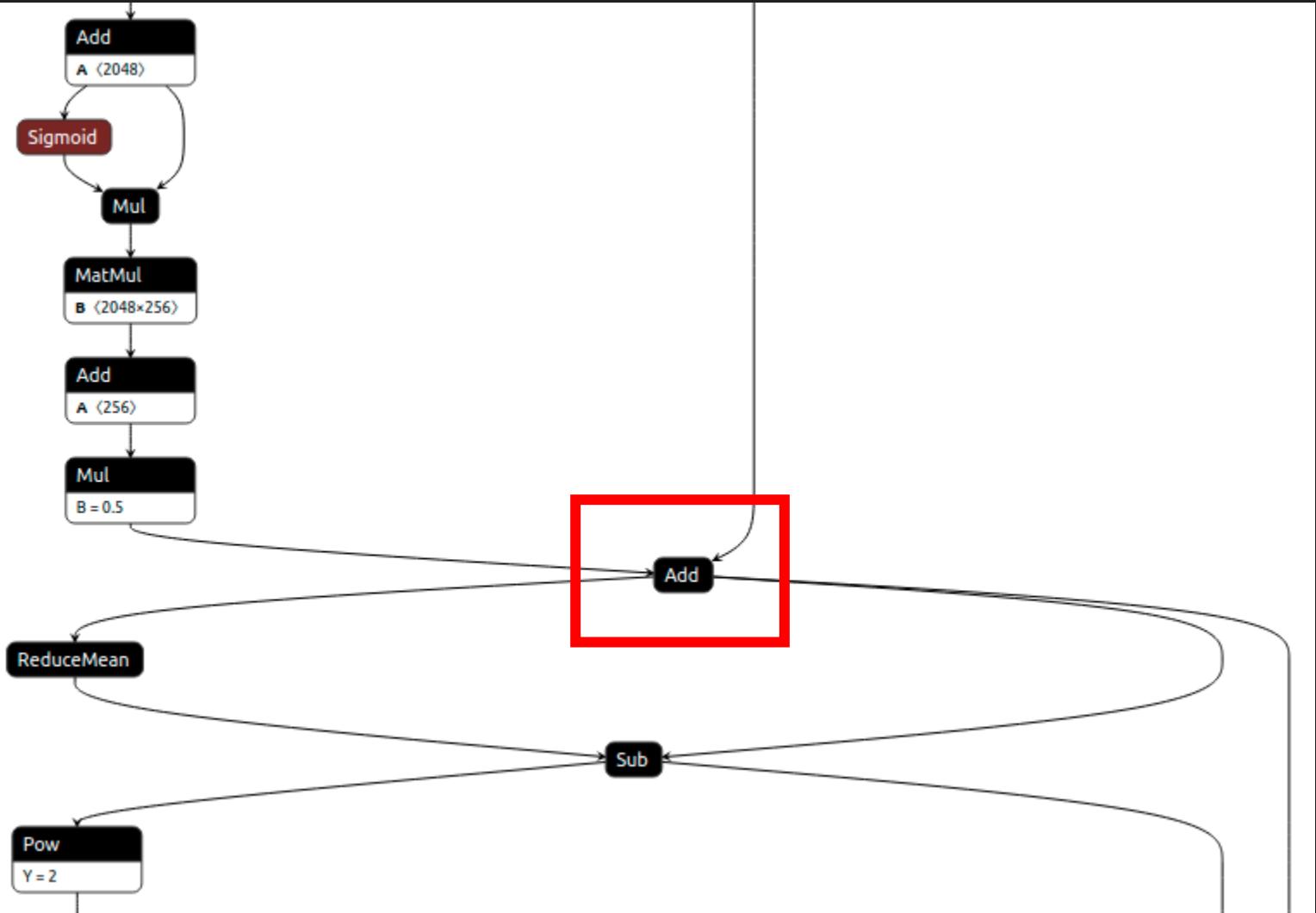
```
1  node
2  └─Add_104 (Add)
3    Inputs: [Variable (587): (shape=None, dtype=float32), Variable (634): (shape=None, dtype=float32)]
4    Outputs:[Variable (635): (shape=None, dtype=float32)]
5
6  node.inputs
7  [Variable (587): (shape=None, dtype=float32), Variable (634): (shape=None, dtype=float32)]
8
9  node.outputs
10 [Variable (635): (shape=None, dtype=float32)]
11
12 node.i(0)
13 └─Mul_64 (Mul)
14   Inputs: [Variable (585): (shape=None, dtype=float32)Variable (586): (shape=[], dtype=float32)]
15   Outputs:[Variable (587): (shape=None, dtype=float32)]
16
17 node.i(1)
18 └─Mul_103 (Mul)
19   Inputs: [Variable (632): (shape=None, dtype=float32), Variable (633): (shape=[], dtype=float32)]
20   Outputs:[Variable (634): (shape=None, dtype=float32)]
21
22 node.o(0)
23 └─ReduceMean_105 (ReduceMean)
24   Inputs: [Variable (635): (shape=None, dtype=float32)]
25   Outputs:[Variable (636): (shape=None, dtype=float32)]
26   Attributes: OrderedDict([('axes', [-1])])
27
28 node.o(1)
29 └─Sub_106 (Sub)
30   Inputs: [Variable (635): (shape=None, dtype=float32), Variable (636): (shape=None, dtype=float32)]
31   Outputs:[Variable (637): (shape=None, dtype=float32)]
32
33 node.o(2)
34 └─Add_180 (Add)
35   Inputs: [Variable (635): (shape=None, dtype=float32), Variable (757): (shape=None, dtype=float32)]
36   Outputs: [Variable (758): (shape=None, dtype=float32)]
```



• onnx-graphsurgeon

➤ Ogs 的 Node 和 Variable

```
38     node.inputs[0].inputs
39     □ [Mul_64 (Mul)
40         Inputs: [Variable (585): (shape=None, dtype=float32), Variable (586): (shape=[], dtype=float32)]
41         Outputs: [Variable (587): (shape=None, dtype=float32)]
42     ]
43
44     node.outputs[0].outputs
45     □ [
46         ReduceMean_105 (ReduceMean)
47             Inputs: [Variable (635): (shape=None, dtype=float32)]
48             Outputs: [Variable (636): (shape=None, dtype=float32)]
49             Attributes: OrderedDict([('axes', [-1])]),
50
51         Sub_106 (Sub)
52             Inputs: [Variable (635): (shape=None, dtype=float32), Variable (636): (shape=None, dtype=float32)]
53             Outputs: [Variable (637): (shape=None, dtype=float32)],
54
55         Add_180 (Add)
56             Inputs: [Variable (635): (shape=None, dtype=float32), Variable (757): (shape=None, dtype=float32)]
57             Outputs: [Variable (758): (shape=None, dtype=float32)]
58     ]
59
60     node.inputs[0].i(0)
61     Variable (585): (shape=None, dtype=float32)
62
63     node.inputs[0].i(1)
64     Variable (586): (shape=[], dtype=float32)
65
66     node.outputs[0].o(0)
67     Variable (636): (shape=None, dtype=float32)
68
69     node.outputs[0].o(1)
70     Variable (637): (shape=None, dtype=float32)
71
72     node.outputs[0].o(2)
73     Variable (758): (shape=None, dtype=float32)
```

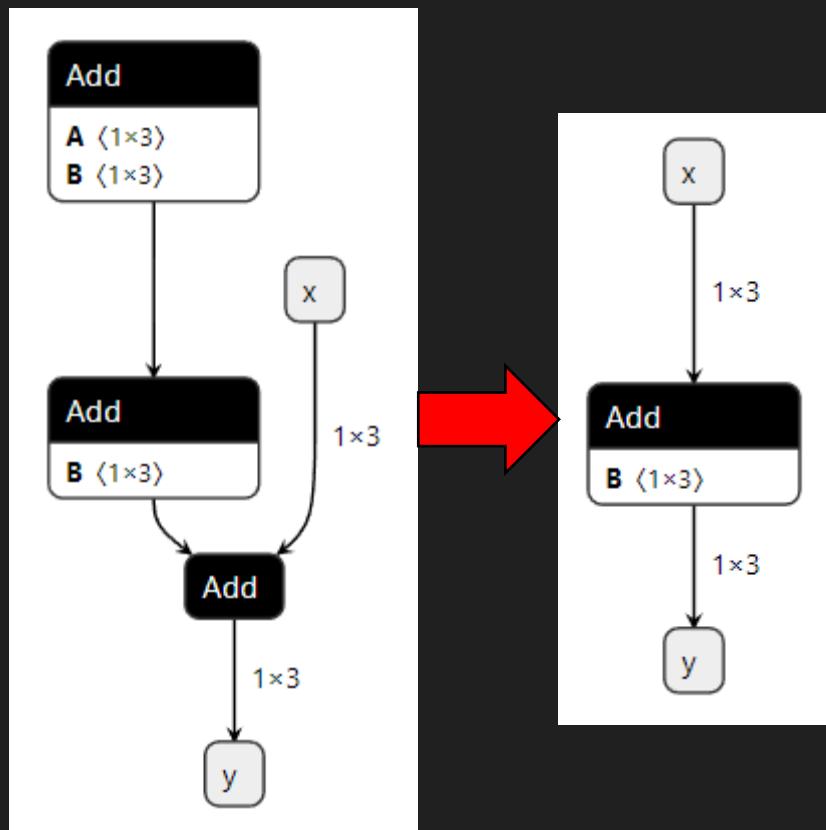


• onnx-graphsurgeon

➤ 05-FoldModel

➤ 常量折叠 (用 onnx-simplifier 或 polygraphy 也可以做到)

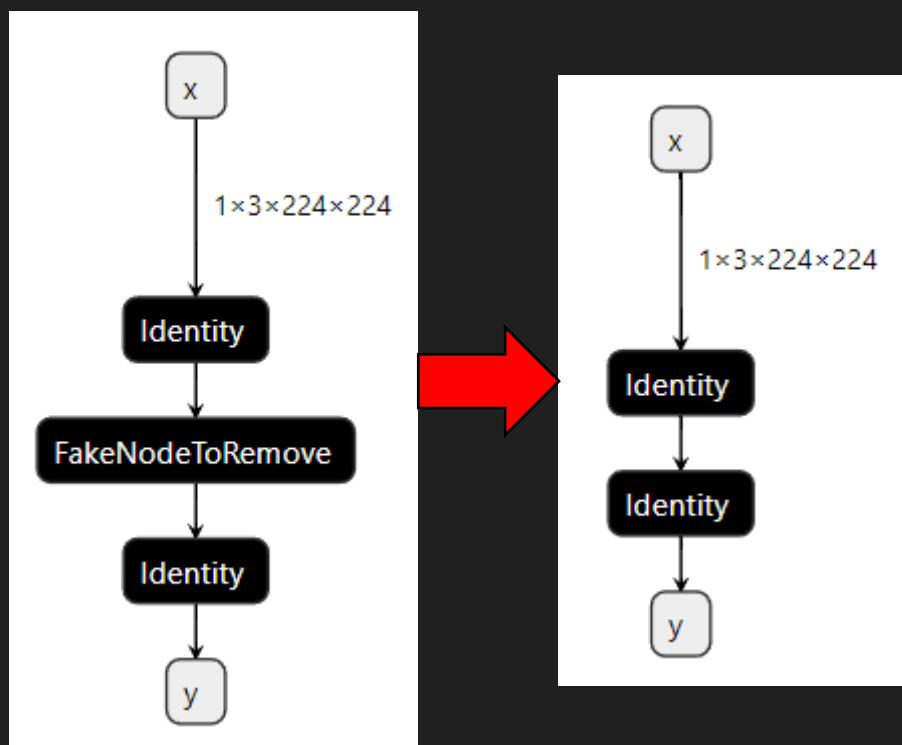
```
1 model = onnx.shape_inference.infer_shapes(onnx.load("model.onnx")) # 带有形状推理的计算图
2
3 graph = gs.import_onnx(model)
4
5 graph.fold_constants().cleanup()
6 onnx.save(gs.export_onnx(graph), "05-FoldModel_1.onnx")
```



➤ 06-RemoveNode

➤ 删除节点

```
1 graph = gs.import_onnx(onnx.load("06-RemoveNode_0.onnx"))
2 # 找到欲删除的节点
3 fake_node = [node for node in graph.nodes if node.op == "FakeNodeToRemove"][0]
4 # 获取欲删除节点的输入节点（母节点）
5 inp_node = fake_node.i()
6 # 将母节点的输出张量赋为欲删除节点的输出张量
7 inp_node.outputs = fake_node.outputs
8 fake_node.outputs.clear()
9
10 graph.cleanup()
11 onnx.save(gs.export_onnx(graph), "06-RemoveNode_1.onnx")
```



• onnx-graphsurgeon

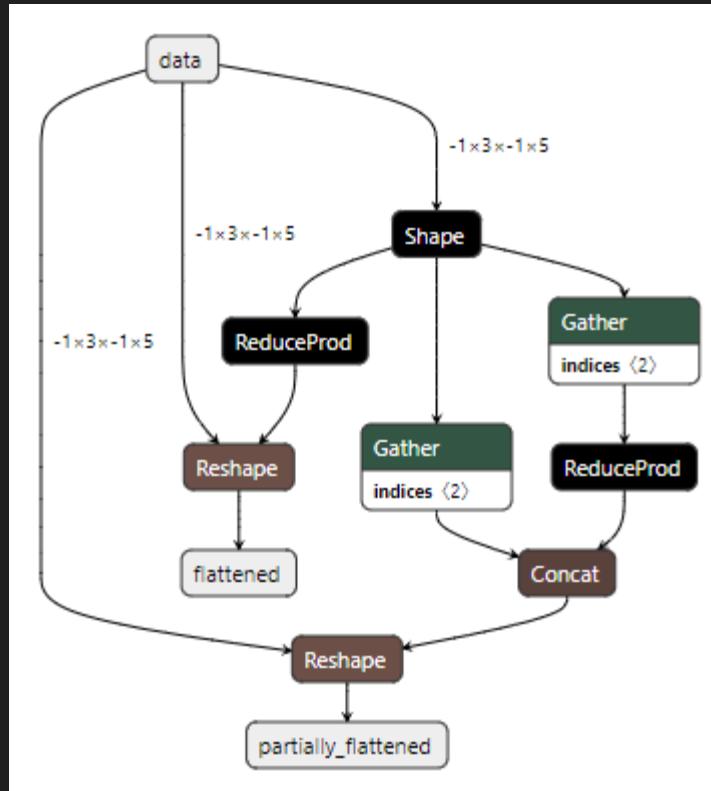
➤ 07-BuildModelWithAPI 和 08-RepaceModel

➤ 使用 `@gs.Graph.register` 来搭建、包装和调整计算图 (用得不多)

➤ 09-ShapeOperation

➤ 与 Dynamic Shape 有关的操作

```
1 graph = gs.Graph()
2
3 graph.inputs = [gs.Variable(name="data", dtype=np.float32, \
4                             shape=(gs.Tensor.DYNAMIC, 3, gs.Tensor.DYNAMIC, 5))] # shape=(A,3,B,5)
5
6 input_shape = graph.shape(graph.inputs[0]) # value=(A,3,B,5), shape=(4,)
7
8 volume = graph.reduce_prod(input_shape, axes=[0]) # value=(A*B*5), shape=()
9 flattened = graph.reshape(graph.inputs[0], volume) # shape=(A*B*5, )
10
11 NC = graph.gather(input_shape, indices=[0, 1]) # value=(A,3), shape=(2, )
12 HW = graph.gather(input_shape, indices=[2, 3]) # value=(B,5), shape=(2, )
13 new_shape = graph.concat([NC, graph.reduce_prod(HW, axes=[0])]) # value=(A,3,B*5), shape=(3, )
14 partially_flattened = graph.reshape(graph.inputs[0], new_shape) # shape=(A,3,B*5)
15
16 flattened.name = "flattened"
17 flattened.dtype = np.float32
18 partially_flattened.name = "partially_flattened"
19 partially_flattened.dtype = np.float32
20
21 graph.outputs = [flattened, partially_flattened]
22 onnx.save(gs.export_onnx(graph), "09-ShapeOperation.onnx")
```



• 开发辅助工具

➤ 希望解决的问题

- | | |
|--------------------------------|----------------------|
| ➤ 不想写脚本来跑 TensorRT | 用命令行搞定 |
| ➤ 怎么进行简单的推理性能测试? | 测量延迟、吞吐量等 |
| ➤ 网络结构可视化? | |
| ➤ 计算图上有些节点阻碍 TensorRT 自动优化 | 手工调整以便 TensorRT 能够处理 |
| ➤ 怎么处理 TensorRT 不支持的网络结构? | 手工调整以便 TensorRT 能够处理 |
| ➤ 怎么检验 TensorRT 上计算结果正确性 / 精度? | 同时在原框架和 TensorRT 上运行 |
| ➤ 怎么找出计算错误 / 精度不足的层? | 模型逐层比较 |
| ➤ 怎么进行简单的计算图优化? | 手工调整 |
| ➤ 怎样找出最耗时的层? | 找到热点集中优化 |



• polygraphy

➤ 深度学习模型调试器，包含 python API 和命令行工具（这里只介绍命令行）

➤ 功能：

- 使用多种后端运行推理计算，包括 TensorRT, onnxruntime, TensorFlow
- 比较不同后端的逐层计算结果
- 由模型文件生成 TensorRT 引擎并序列化为 .plan
- 查看模型网络的逐层信息
- 修改 Onnx 模型，如提取子图，计算图化简
- 分析 Onnx 转 TensorRT 失败原因，将原计算图中可以 / 不可以转 TensorRT 的子图分割保存
- 隔离 TensorRT 中的错误 tactic

➤ 下载和参考文档：

- pip install polygraphy
- <https://docs.nvidia.com/deeplearning/tensorrt/polygraphy/docs/index.html>
- <https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31695/> (onnx-graphsurgeon 和 polygraphy 的一个视频教程)



• polygraphy

➤ Run 模式

➤ 范例代码

07-Tool/polygraphy/runExample，运行 ./command.sh

- 首先生成一个 .onnx 文件（同前面基于 MNIST 的模型）
- 其次使用 polygraphy 生成一个 FP16 的 TRT 引擎，并对比使用 onnxruntime 和 TensorRT 的计算结果
- 然后使用 polygraphy 生成一个 FP32 的 TRT 引擎，将网络中所有层都标记为输出，并对比使用 onnxruntime 和 TensorRT 的计算结果（逐层结果对比）
- 最后（右图中没有）使用 polygraphy 生成一个“用于生成 FP32 的 TRT 引擎”的 python 脚本，可用于理解 polygraphy 的 API 和原理，这里不展开

```
1  clear
2
3  rm ./*.pb ./*.onnx ./*.plan ./result-*.*.txt
4
5  # 从 TensorFlow 创建一个 .onnx 用来做 polygraphy 的输入文件
6  python getOnnxModel.py
7
8  # 01 用上面的 .onnx 构建一个 TensorRT 引擎，使用 FP16精度，同时在 onnxruntime 和 TensorRT 中运行，对比结果
9  polygraphy run model.onnx \
10   --onnxrt --trt \
11   --workspace 1000000000 \
12   --save-engine=model-FP16.plan \
13   --atol 1e-3 --rtol 1e-3 \
14   --fp16 \
15   --verbose \
16   --trt-min-shapes 'x:0:[1,1,28,28]' \
17   --trt-opt-shapes 'x:0:[4,1,28,28]' \
18   --trt-max-shapes 'x:0:[16,1,28,28]' \
19   --input-shapes 'x:0:[4,1,28,28]' \
20   > result-run-FP16.txt
21
22  # 注意参数名和格式跟 trtexec 不一样，多个形状之间用空格分隔，如：
23  # --trt-max-shapes 'input0:[16,320,256]' 'input1:[16, 320]' 'input2:[16]'
24
25  # 02 用上面的 .onnx 构建一个 TensorRT 引擎，输出所有层的计算结果作对比
26  polygraphy run model.onnx \
27   --onnxrt --trt \
28   --workspace 1000000000 \
29   --save-engine=model-FP32-MarkAll.plan \
30   --atol 1e-3 --rtol 1e-3 \
31   --verbose \
32   --onnx-outputs mark all \
33   --trt-outputs mark all \
34   --trt-min-shapes 'x:0:[1,1,28,28]' \
35   --trt-opt-shapes 'x:0:[4,1,28,28]' \
36   --trt-max-shapes 'x:0:[16,1,28,28]' \
37   --input-shapes 'x:0:[4,1,28,28]' \
38   > result-run-FP32-MarkAll.txt
```



• polygraphy

➤ Run 模式

➤ 输出结果示例

```
1 [I] Comparing Output: 'z:0' (dtype=int32, shape=(4,)) with 'z:0' (dtype=int64, shape=(4,)) | Tolerance: [abs=0.001, rel=0.001] | Checking elemwise error
2 [I] trt-runner-N0-03/14/22-08:20:47: z:0 | Stats: mean=6.25, std-dev=3.0311, var=9.1875, median=8, min=1 at (2,), max=8 at (0,), avg-magnitude=6.25
3 [V] ---- Histogram ----
4 Bin Range | Num Elems | Visualization
5 (1 , 1.7) | 1 | #####
6 (1.7, 2.4) | 0 |
7 (2.4, 3.1) | 0 |
8 (3.1, 3.8) | 0 |
9 (3.8, 4.5) | 0 |
10 (4.5, 5.2) | 0 |
11 (5.2, 5.9) | 0 |
12 (5.9, 6.6) | 0 |
13 (6.6, 7.3) | 0 |
14 (7.3, 8 ) | 3 | #####
15 [I] onnxrt-runner-N0-03/14/22-08:20:47: z:0 | Stats: mean=6.25, std-dev=3.0311, var=9.1875, median=8, min=1 at (2,), max=8 at (0,), avg-magnitude=6.25
16 [V] ---- Histogram ----
17 Bin Range | Num Elems | Visualization
18 (1 , 1.7) | 1 | #####
19 (1.7, 2.4) | 0 |
20 (2.4, 3.1) | 0 |
21 (3.1, 3.8) | 0 |
22 (3.8, 4.5) | 0 |
23 (4.5, 5.2) | 0 |
24 (5.2, 5.9) | 0 |
25 (5.9, 6.6) | 0 |
26 (6.6, 7.3) | 0 |
27 (7.3, 8 ) | 3 | #####
28 [I] Error Metrics: z:0
29 [I] Minimum Required Tolerance: elemwise error | [abs=0] OR [rel=0] (requirements may be lower if both abs/rel tolerances are set)
30 [I] Absolute Difference | Stats: mean=0, std-dev=0, var=0, median=0, min=0 at (0,), max=0 at (0,), avg-magnitude=0
```

两者结果之差的统计



- polygraphy

➤ Run 模式常用选项

- | | |
|--|---|
| ➤ --model-type | 输入模型文件类型，可用 frozen, keras, ckpt, onnx, engine 等 |
| ➤ --verbose | 打印详细日志 |
| ➤ --trt --tf -onnxrt | 选用的后端（可以指定多个） |
| ➤ --input-shapes "x:[4,1,28,28]" "y:[4,8]" | 做推理时的实际数据形状，注意跟 trtexec 格式不同 |
| ➤ --ckpt "./model.ckpt" --save-pb "./model.pb" --freeze-graph | TensorFlow相关的模型在载入和保存 |
| ➤ --save-onnx ".model.onnx" --opset 13 | 指定导出 Onnx 文件名和算子集编号 |
| ➤ --shape-inference | 启用 Onnx 的形状推理功能 |
| ➤ --trt-min-shapes 'x:[1,1,28,28]' 'y:[1,8]' --trt-opt-shapes 'x:[4,1,28,28]' 'y:[4,8]' --trt-max-shapes 'x:[16,1,28,28]' 'y:[16,8]' | 设定 TensorRT Dynamic Shape 的输入性状范围 |
| ➤ --fp16 --int8 --tf32 --sparse-weights | 指定 TensorRT 中网络计算精度和稀疏性 |
| ➤ --workspace 1G | 指定 TensorRT 工作空间大小，可用 M 或者 G 做后缀 |
| ➤ --save-engine "./model.plan" | 指定 TensorRT 导出的 .plan 文件名 |
| ➤ --rtol 1e-6 --atol 1e-6 --validate | 比较计算结果时的相对误差/绝对误差上限，并检查 NaN 和 INF |



• polygraphy

➤ inspect 模式

➤ 范例代码

07-Tool/polygraphy/inspectExample，运行 ./command.sh

- 首先生成一个 .onnx 文件（同前面基于 MNIST 的模型）
- 其次使用 polygraphy 导出 .onnx 详细信息（计算图属性、逐层属性、权重信息等）
- 然后生成一个 FP32 的 TRT 引擎，使用 polygraphy 导出 .plan 详细信息（网络属性、引擎binding信息，显存需求等）
- 使用 polygraphy 判断一个 .onnx 是否完整被 TensorRT 支持
- 故意生成一个 TensorRT 不完整支持的 .onnx（含一个 NonZero 节点），再次做上述判断

```
1  clear
2
3  rm /*.pb /*.onnx /*.plan ./result-*.txt
4
5  # 从 TensorFlow 创建一个 .onnx 用来做 polygraphy 的输入文件
6  python getOnnxModel.py
7
8  # 导出上面 .onnx 的详细信息
9  polygraphy inspect model model.onnx \
10   --mode=full \
11   > result-inspectOnnxModel.txt
12
13 # 用上面 .onnx 生成一个 .plan 及其相应的 tactics 用于后续分析
14 polygraphy run model.onnx \
15   --trt \
16   --workspace 1000000000 \
17   --save-engine="./model.plan" \
18   --save-inputs="./model-input.txt" \
19   --save-outputs="./model-output.txt" \
20   --silent \
21   --trt-min-shapes 'x:0:[1,1,28,28]' \
22   --trt-opt-shapes 'x:0:[4,1,28,28]' \
23   --trt-max-shapes 'x:0:[16,1,28,28]' \
24   --input-shapes 'x:0:[4,1,28,28]'
25
26 # 导出上面 .plan 的详细信息（要求 TensorRT >= 8.2）
27 polygraphy inspect model model.plan \
28   --mode=full \
29   > result-inspectPlanModel.txt
30
31 # 确认 TensorRT 是否完全原生支持该 .onnx
32 polygraphy inspect capability model.onnx
33 # 输出信息: [I] Graph is fully supported by TensorRT; Will not generate subgraphs.
34
35 # 生成一个含有 TensorRT 不原生支持的 .onnx，再次用 inspect capability 来确认
36 python getOnnxModel-NonZero.py
37
38 polygraphy inspect capability model-NonZero.onnx > result-NonZero.txt
39 # 产生目录 .results，包含网络分析信息和支持的子图(.onnx)、不支持的子图(.onnx)
```



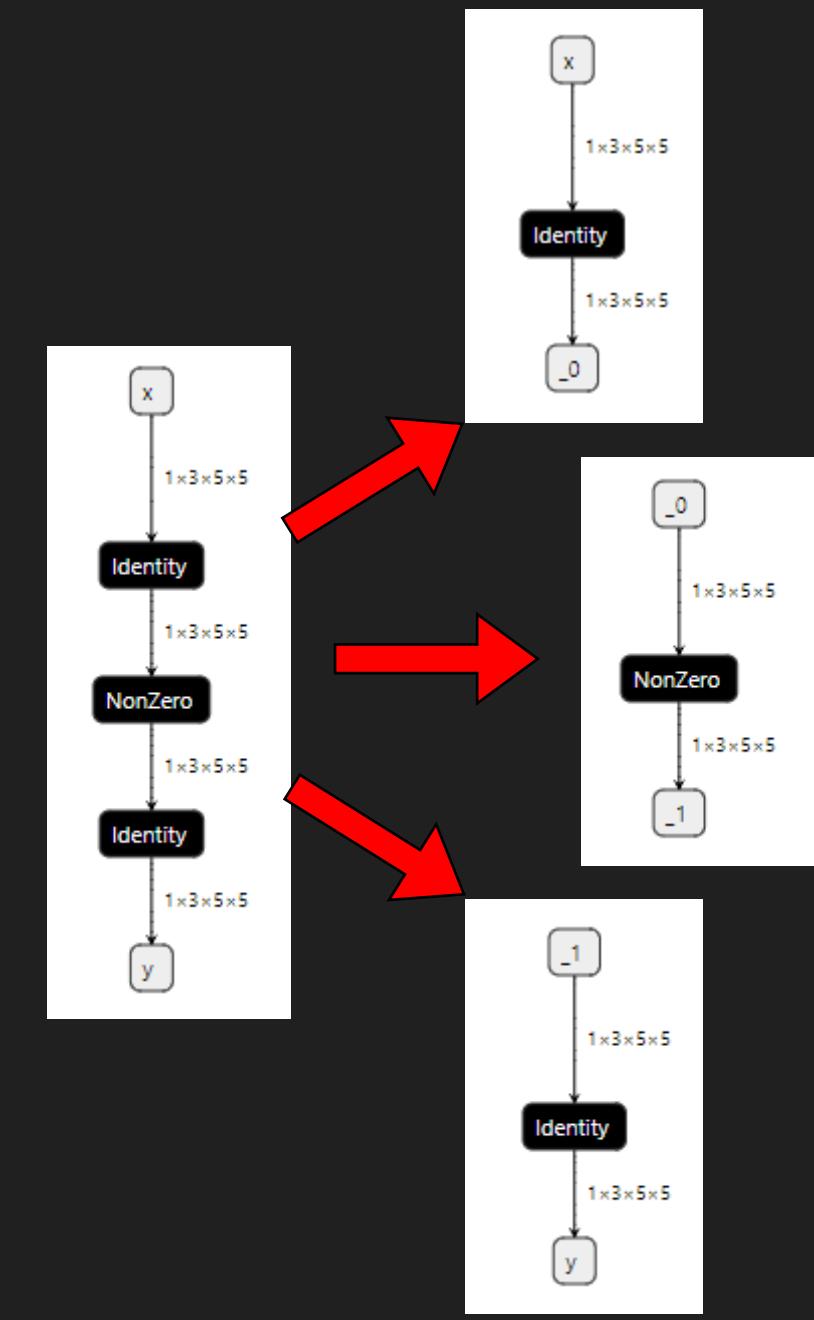
• polygraphy

➤ inspect 模式

➤ 输出结果示例

```
1  └[I] === ONNX Model ===
2      Name: tf2onnx | Opset: 9
3
4      ---- 1 Graph Input(s) ----
5      {x:0 [dtype=float32, shape=('unk_30', 1, 28, 28)]}
6
7      ---- 1 Graph Output(s) ----
8      {z:0 [dtype=int64, shape=('unk_31',)]}
9
10     ---- 9 Initializer(s) ----
11     Initializer | w4/read:0 [dtype=float32, shape=[1024, 10]] | Values:
12     [[ 0.03731047 -0.05782586 -0.12556368 ... -0.08612765 -0.06061816
13       -0.08246689]
14     [ 0.19088742 -0.06377193 -0.12876913 ... 0.06157729 -0.05597464
15       -0.04897792]
16     [-0.08788037 0.04984109 -0.03226187 ... 0.09335365 0.03903114
17       0.00912305]
18     ...
19     [-0.07114442 -0.18008882 0.1481347 ... -0.12616785 0.13713098
20       -0.05504219]
21     [-0.04465632 -0.07094491 -0.04796916 ... 0.02845419 -0.03283151
22       0.07002489]
23     [-0.12010998 0.02466527 0.00236313 ... -0.09452797 0.07460572
24       0.02989185]]
25
26     ...
27
28     ---- 15 Node(s) ----
29     Node 0 | Conv_25 [Op: Conv]
30     {x:0 [dtype=float32, shape=('unk_30', 1, 28, 28)],
31       Initializer | w1/read:0 [dtype=float32, shape=(32, 1, 5, 5)],
32       Initializer | b1/read:0 [dtype=float32, shape=(32,)]}
33     -> {Conv_25:0}
34     ---- Attributes ----
35     Conv_25.dilations = [1, 1]
36     Conv_25.strides = [1, 1]
37     Conv_25.kernel_shape = [5, 5]
38     Conv_25.pads = [2, 2, 2, 2]
39     Conv_25.group = 1
40
41     ...
```

名称	修改日期	类型
results.txt	2022/3/19 20:12	文本文档
supported_subgraph-nodes-0-0.onnx	2022/3/19 20:12	ONNX 文件
supported_subgraph-nodes-2-2.onnx	2022/3/19 20:12	ONNX 文件
unsupported_subgraph-nodes-1-1.on...	2022/3/19 20:12	ONNX 文件



• polygraphy

➤ surgeon 模式

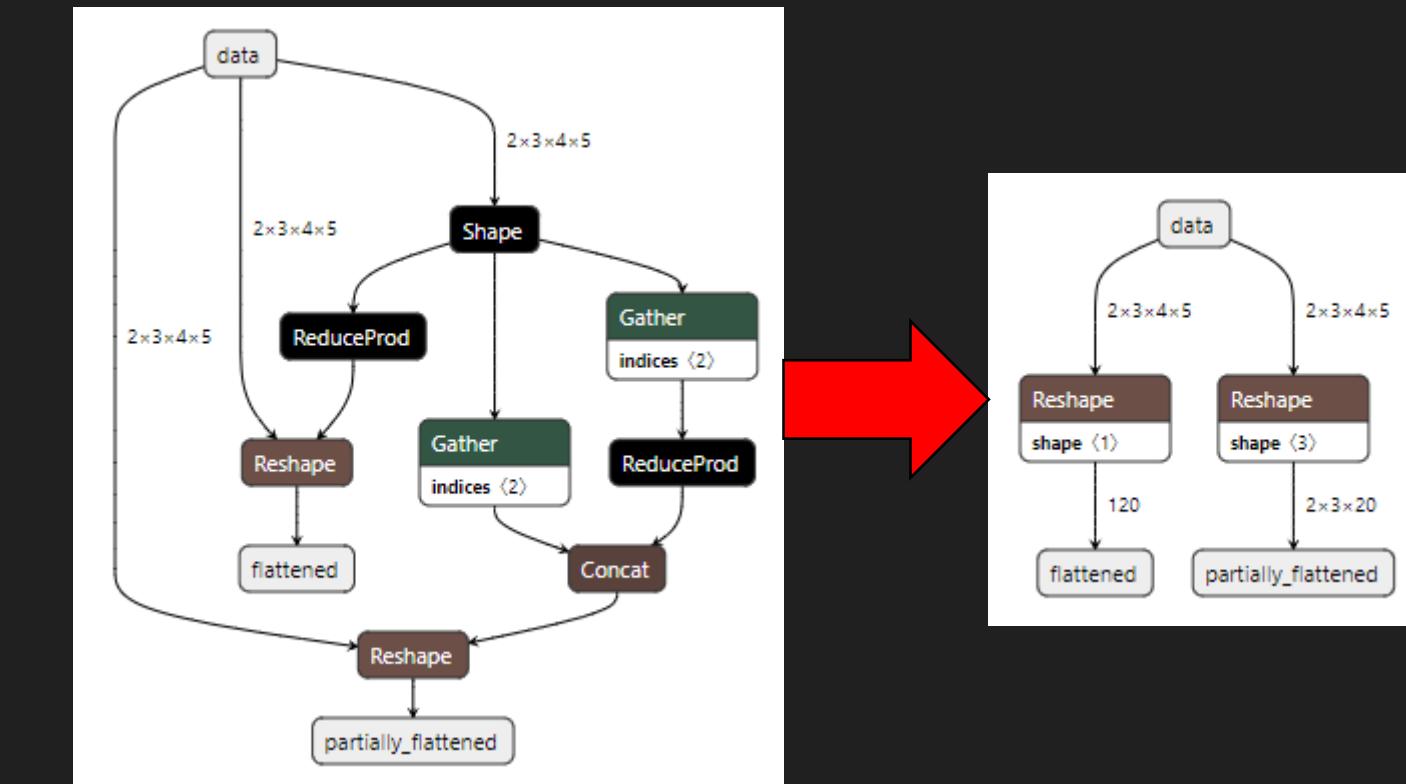
➤ 范例代码

08-Tool/polygraphy/surgeonExample, 运行 ./command.sh

➤ 首先生成一个 .onnx 文件 (一个 shape 操作相关的网络)

➤ 其次使用 polygraphy 优化其计算图

```
1  clear
2
3  rm /*.pb /*.onnx /*.plan ./result-*.*txt
4
5  # 从 TensorFlow 创建一个 .onnx 用来做 polygraphy 的输入文件
6  python getShapeOperateOnnxModel.py
7
8  # 01 对该模型做图优化
9  polygraphy surgeon sanitize model.onnx \
10   --fold-constant \
11   -o model-foldConstant.onnx \
12   > result-surgeon.txt
```



• polygraphy

➤ 其他模式

You can never have too many tools

➤ convert 模式

➤ 基本同 run 模式

➤ 范例代码: 08-Tool\Polygraphy\convertExample

➤ debug 模式

➤ 检查模型转 TensorRT 时的错误并分离出一个可以运行的最大子图

➤ 范例代码: 08-Tool\Polygraphy\debugExample

➤ data 模式

➤ 调整和分析用于运行模型的输入输出、权重数据

➤ template 模式

➤ 用于生成 polygraphy 的 python 脚本，使用脚本对模型进行调整

➤ 范例代码: 08-Tool\Polygraphy\templateExample



• 开发辅助工具

➤ 希望解决的问题

- | | |
|--------------------------------|----------------------|
| ➤ 不想写脚本来跑 TensorRT | 用命令行搞定 |
| ➤ 怎么进行简单的推理性能测试? | 测量延迟、吞吐量等 |
| ➤ 网络结构可视化? | |
| ➤ 计算图上有些节点阻碍 TensorRT 自动优化 | 手工调整以便 TensorRT 能够处理 |
| ➤ 怎么处理 TensorRT 不支持的网络结构? | 手工调整以便 TensorRT 能够处理 |
| ➤ 怎么检验 TensorRT 上计算结果正确性 / 精度? | 同时在原框架和 TensorRT 上运行 |
| ➤ 怎么找出计算错误 / 精度不足的层? | 模型逐层比较 |
| ➤ 怎么进行简单的计算图优化? | 手工调整 |
| ➤ 怎样找出最耗时的层? | 找到热点集中优化 |



• nsight systems

➤ 性能调试器

➤ 随 CUDA 安装或独立安装，位于 /usr/local/cuda/bin/ 下的 nsys 和 nsys-ui

➤ 替代旧工具 nvprof 和 nvvp

➤ 首先命令行运行 nsys profile XXX，获得 .qdrep 或 .qdrep-nsys 文件

➤ 然后打开 nsys-ui，将上述文件拖入即可观察 timeline

➤ 建议

➤ 只计量运行阶段，而不分析构建期

➤ 构建期打开 profiling 以便获得关于 Layer 的更多信息

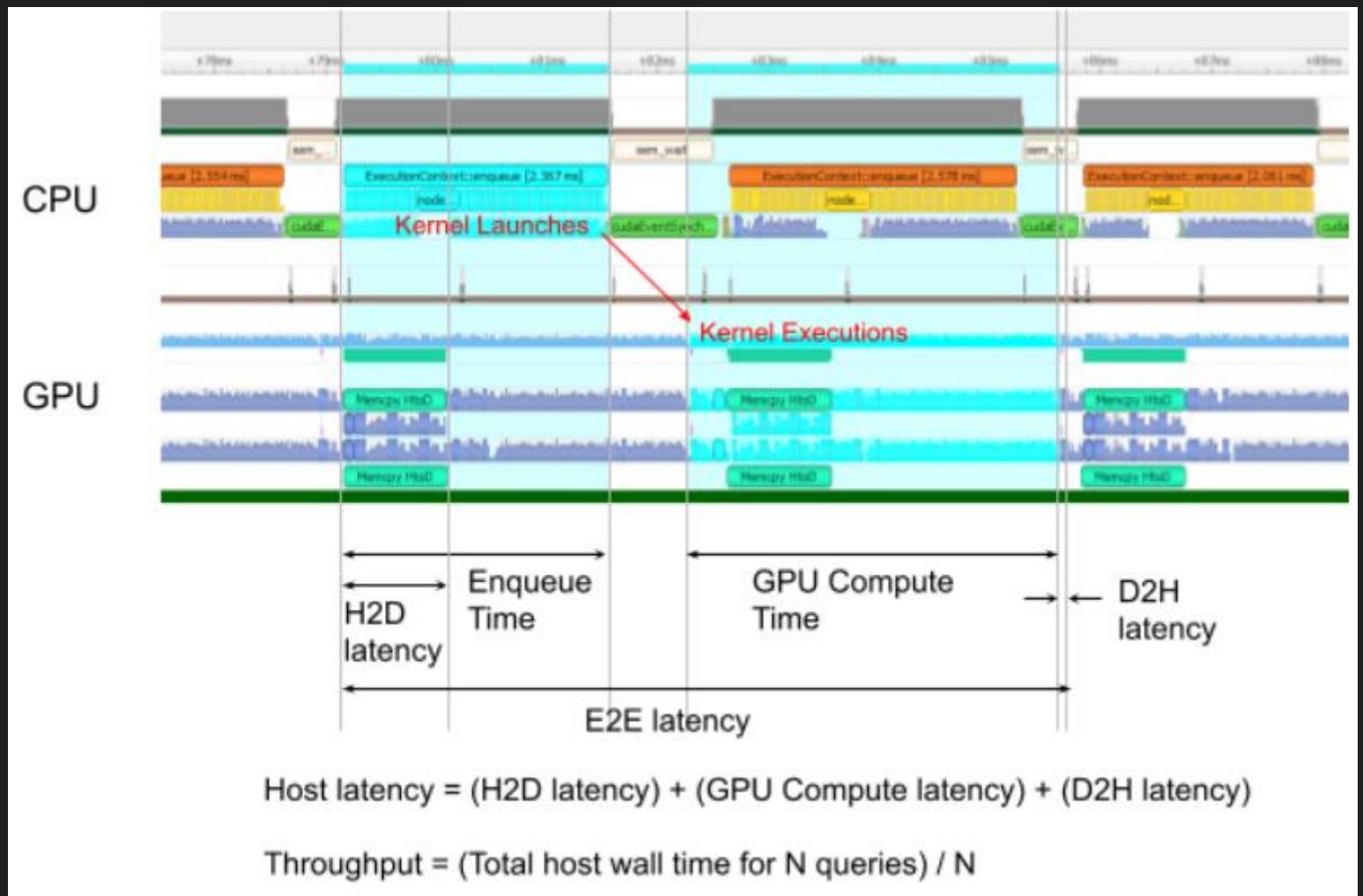
➤ builder_config.profiling_verbosity = trt.ProfilingVerbosity.DETAILED

➤ 范例代码 08-Advance\ProfilingVerbosity

➤ 可以搭配 trtexec 使用： nsys profile -o myProfile trtexec --loadEngine=model.plan --warmUp=0 --duration=0 --iterations=50

➤ 也可以配合自己的 script 使用： nsys profile -o myProfile python myScript.py

➤ 务必将 nsys 更新到最新版， nsys 不能前向兼容



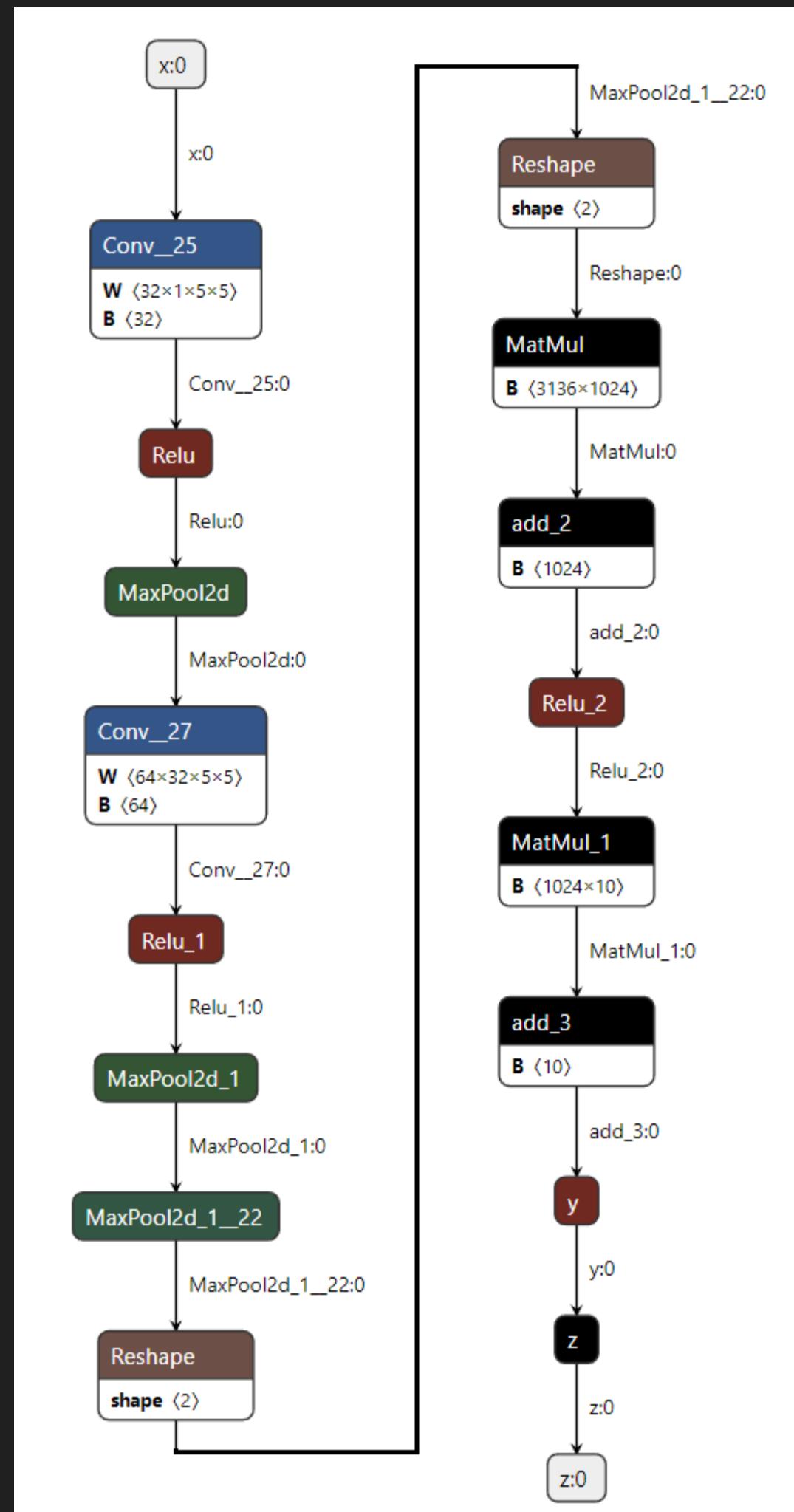
• nsight systems

➤ 范例代码

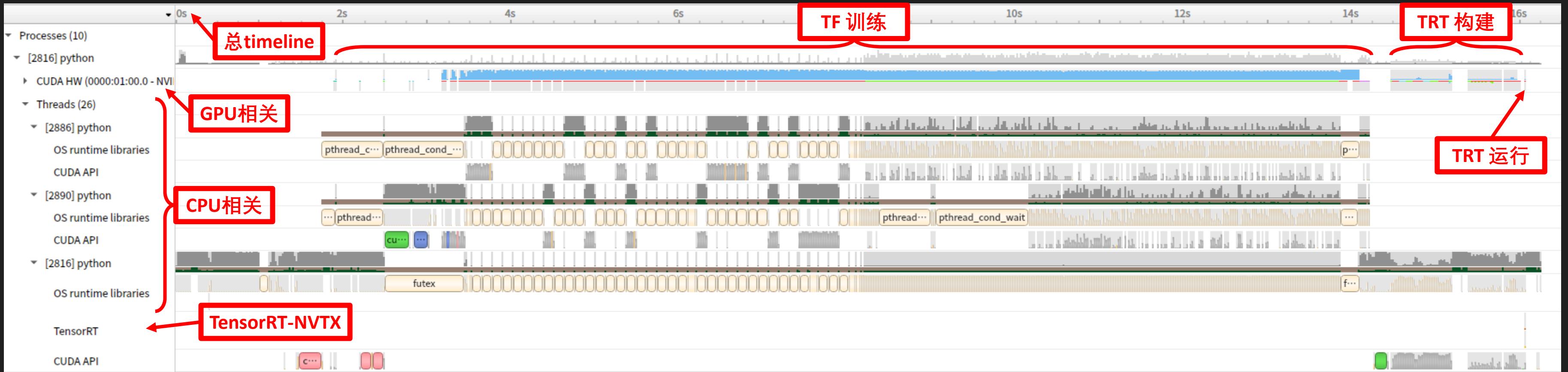
- 08-Tool\NsightSystem，运行 ./command.sh
- 范例在 TensorFlow 上训练一个 MNIST 手写识别模型，然后在 TensorRT 中用 API 重新搭建该模型，并加载训练好的权重，最后进行 30 次推理

➤ 下载和参考文档

- <https://developer.nvidia.com/nsight-systems>
- <https://docs.nvidia.com/nsight-systems/UserGuide/index.html>



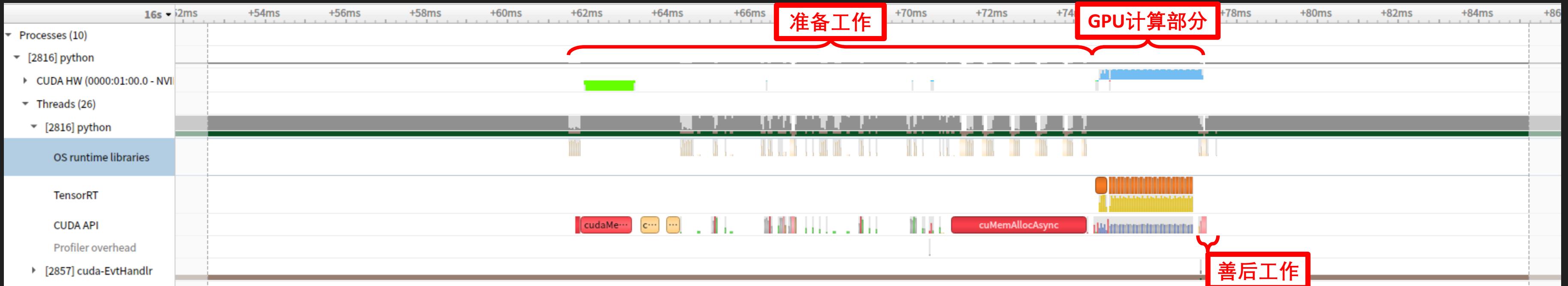
• nsys systems



- 整个 timeline
- 建议使用 nsys 只计量 TensorRT 运行部分的时间（上面 timeline 包含 TF 训练、TRT 构建和 TRT 运行）



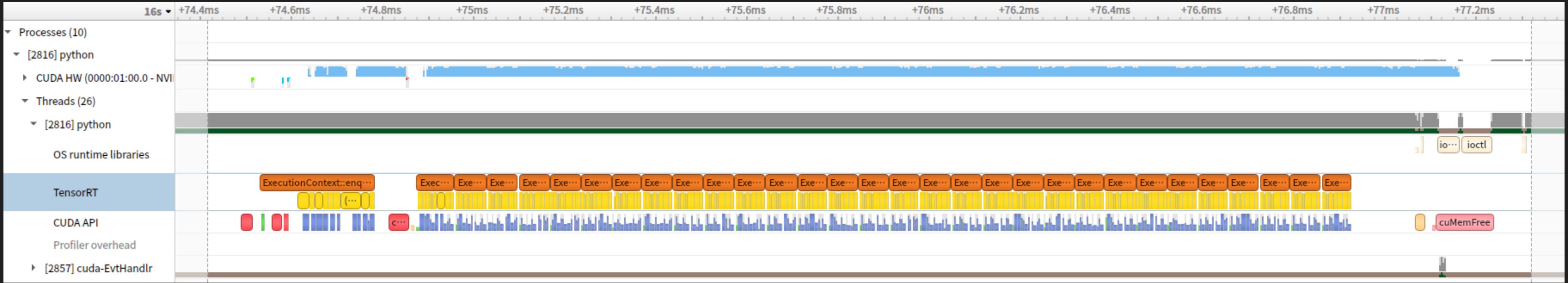
• nsight systems



➤ TensorRT 运行阶段的 timeline



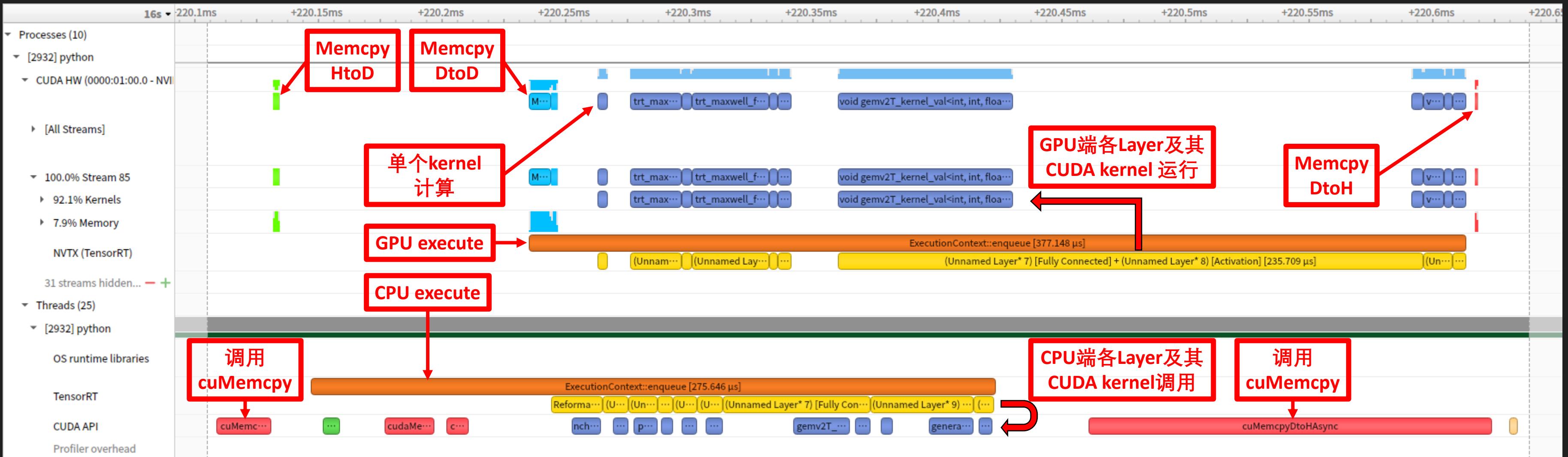
• nsight systems



- 计算部分的 timeline (**GPU timeline** 要从左边 CUDA HW 行展开, Threads 下面的是 CPU 端的情况)
- 一共有 31 次 ExecutionContext::enqueue (橘黄色) , 对应 31 次 context.execute 调用
- 下面的黄色矩形对应网络中各层, 蓝色矩形为相应的 cuda kernel 的调用
- 首次推理 (warming up) 为完整推理过程, 前后包含一些 memcpy, 后续 30 次只做计算不做 memcpy



• nsight systems



➤ 单次 inference 的 timeline, 展开 GPU部分

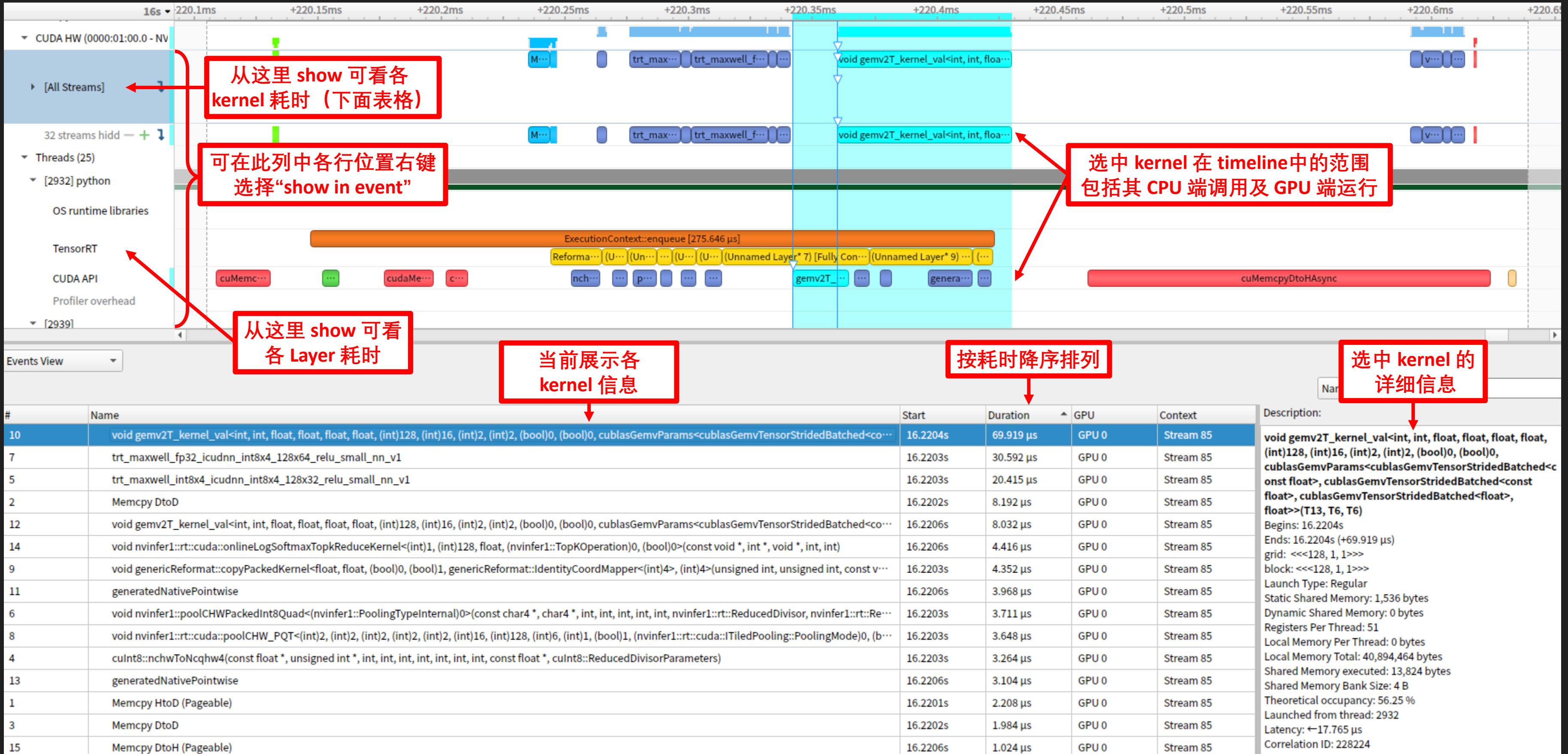
➤ 9 个黄矩形代表网络的 9 个层:

➤ Reformat、Conv+ReLU、Pooling、Conv+ReLU、Pooling、Shuffle、FullyConnected+ReLU、FullyConnected+ReLU、softmax+TopK

➤ CPU 端调用 cuda API 与 GPU 端实际执行 kernel 之间存在滞后



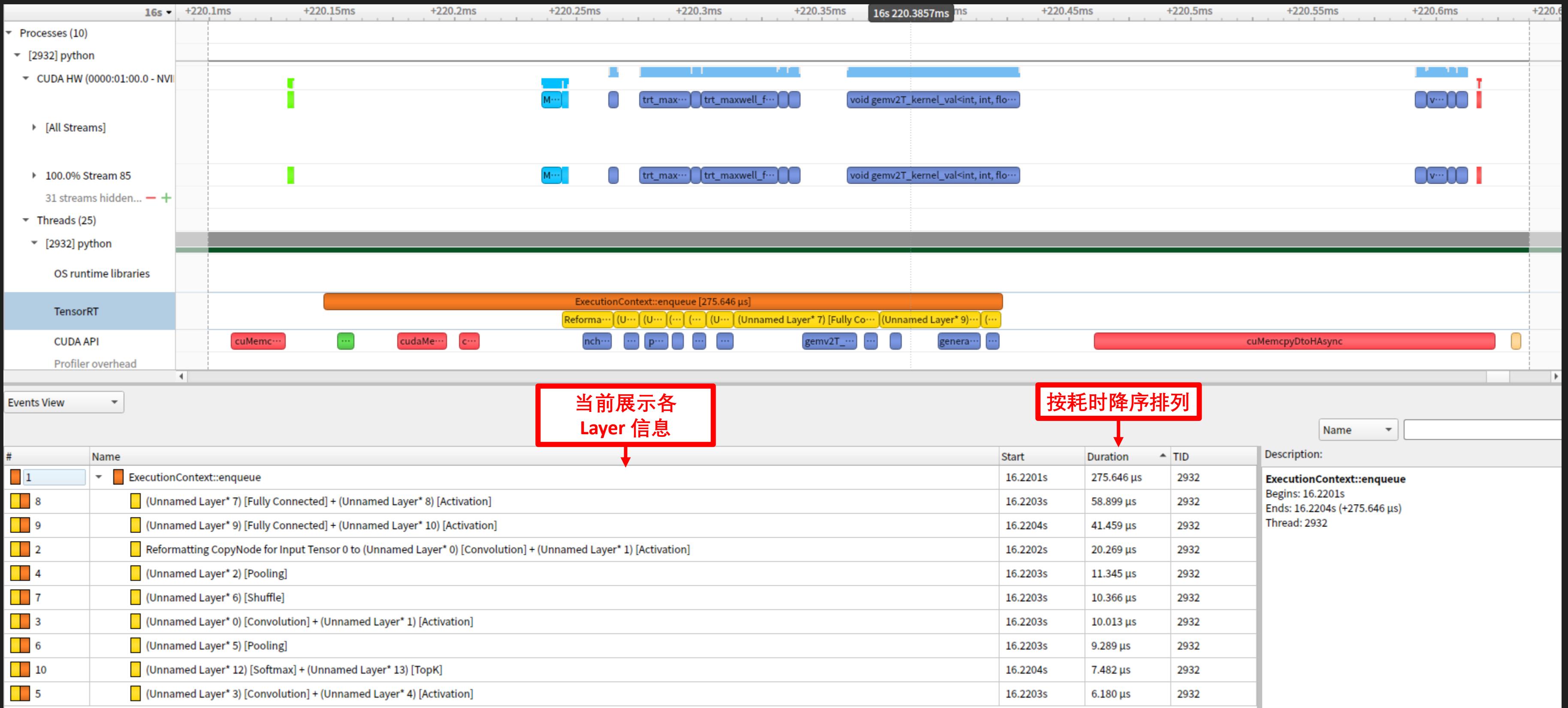
• nsight systems



➤ 列出 GPU kernel 运行情况 (在 [All Streams] 那一行右键选择 “show in events”)



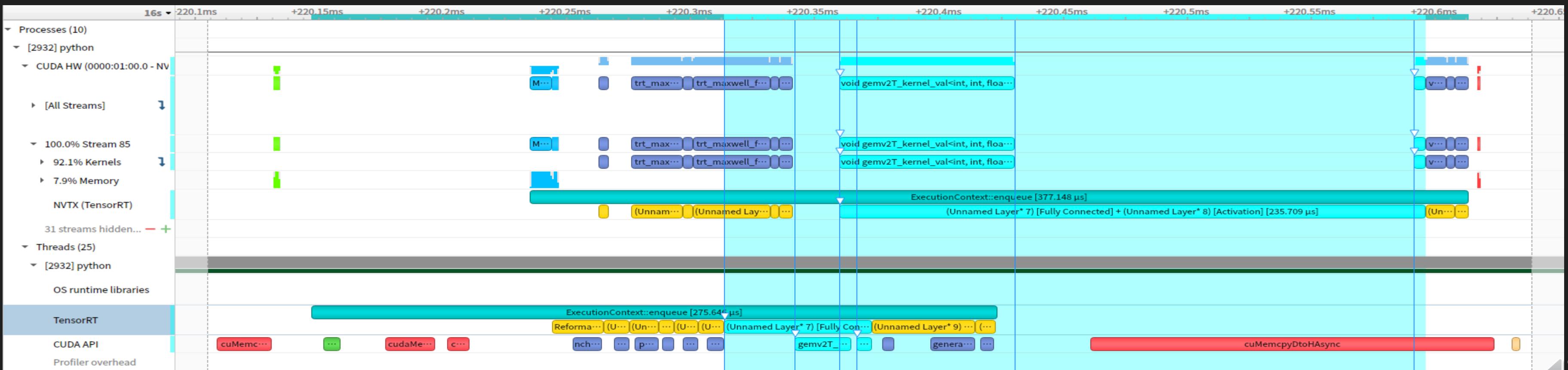
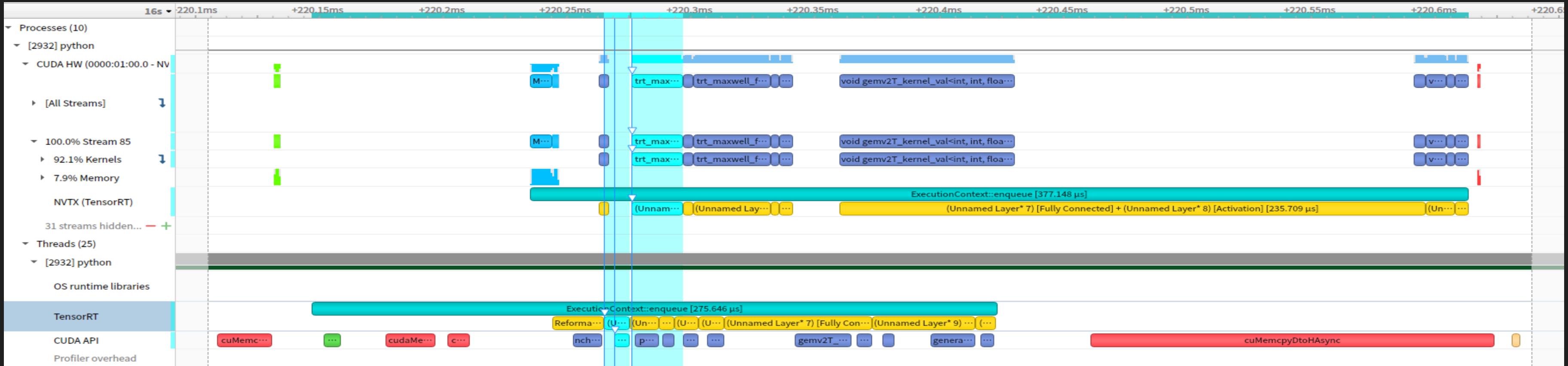
• nsight systems



➤ 列出 TensorRT 的 NVTX (在 TensorRT 那一行右键选择 “show in events”)



• nsight systems



Part 3

• TensorRT Plugin

- Plugin 简介
- 使用 Plugin 的简单例子
- TensorRT 与 Plugin 的交互
- 关键 API
- 结合使用 Parser 和 Plugin
- 高级话题
- 使用 Plugin 的案例



• Plugin 简介

➤ 功能

- 以 .so 的形式插入到网络中实现某些算子
- 实现 TensorRT 不原生支持的层或结构
- 替换性能不足的层或结构
- 手动合并 TensorRT 没有自动融合的层或结构

➤ 限制条件

- 自己实现 CUDA C++ kernel，为结果精度和性能负责
- Plugin 与其他 Layer 之间无法 fusing
- 可能在 Plugin 节点前后插入 reformatting 节点，增加开销

➤ 建议

- 先尝试原生 Layer 的组合保证计算正确性
- 尝试 TensorRT 自带的 Plugin 是否满足要求
- 还是不满意，自己写！

➤ TensorRT 自带的 Plugin 及使用说明

- <https://github.com/NVIDIA/TensorRT/tree/main/plugin>
- TensorRT 8.6 中部分 Plugin 已加入 TensorRT，不再需要 Plugin ##

Plugin	Name	Plugin	Name
batchTilePlugin	BatchTilePlugin_TRT	leakyReluPlugin	LReLU_TRT
batchedNMSPlugin	BatchedNMS_TRT	multilevelCropAndResizePlugin	MultilevelCropAndResize_TRT
batchedNMSDynamicPlugin	BatchedNMSDynamic_TRT	multilevelProposeROI	MultilevelProposeROI_TRT
bertQKVToContextPlugin	CustomQKVToContextPluginDynamic	nmsPlugin	NMS_TRT
coordConvACPlugin	CoordConvAC	normalizePlugin	Normalize_TRT
cropAndResizePlugin	CropAndResize	nvFasterRCNN	RPN_TRT
decodeBbox3DPlugin	DecodeBbox3DPlugin	pillarScatterPlugin	PillarScatterPlugin
detectionLayerPlugin	DetectionLayer_TRT	priorBoxPlugin	PriorBox_TRT
efficientNMSPlugin	EfficientNMS_TRT	proposalLayerPlugin	ProposalLayer_TRT
efficientNMSONNXPlugin	EfficientNMS_ONNX_TRT	proposalPlugin	Proposal
embLayerNormPlugin	CustomEmbLayerNormPluginDynamic	pyramidROIAlignPlugin	PyramidROIAlign_TRT
fcPlugin	CustomFCPluginDynamic	regionPlugin	Region_TRT
flattenConcat	FlattenConcat_TRT	reorgPlugin	Reorg_TRT
geluPlugin	CustomGeluPluginDynamic	resizeNearestPlugin	ResizeNearest_TRT
generateDetectionPlugin	GenerateDetection_TRT	scatterPlugin	ScatterND
gridAnchorPlugin	GridAnchor_TRT	skipLayerNormPlugin	CustomSkipLayerNormPluginDynamic
gridAnchorRectPlugin	GridAnchorRect_TRT	specialSlicePlugin	SpecialSlice_TRT
groupNormalizationPlugin	GroupNormalizationPlugin	splitPlugin	Split
instanceNormalizationPlugin	InstanceNormalization_TRT	voxelGeneratorPlugin	VoxelGeneratorPlugin



• Plugin 关键问题

- 怎么从头开始实现一个 Plugin? 要写哪些类和函数
- 怎么把 Plugin 接到 TensorRT 网络中去? 要怎么包装 kernel 以便 TensorRT 识别
- TensorRT 怎么参与 Plugin 的资源管理? 两者之间要交换些什么信息
- Plugin 能不能序列化到 .plan 中去?
- Plugin 有些什么扩展性? FP16/INT8, Dynamic Shape, data-dependent-shape, ...
- Plugin 与原生 Layer 相比性能怎么样?



• Plugin 的简单例子

➤ 范例代码

➤ 05-Plugin\usePluginV2DynamicExt

➤ 功能：给输入张量所有元素加上同一个值

➤ 实现步骤

➤ 继承 IPluginV2DynamicExt 类实现一个Plugin 类

➤ 继承 IPluginCreator 类实现一个 PluginCreator 类

➤ 实现用于计算的 CUDA C++ kernel

➤ 将 Plugin 编译为 .so 保存

➤ 在 TensorRT 中加载和使用 Plugin

```
1  namespace nvinfer1
2  {
3      class AddScalarPlugin : public IPluginV2DynamicExt
4  {
5      private:
6          // ... // 一些自定义的数据
7
8      public:
9          // ... // 重写部分成员函数
10     };
11
12     class AddScalarPluginCreator : public IPluginCreator
13  {
14      private:
15          // ... // 一些自定义的数据
16
17      public:
18          // ... // 重写部分成员函数
19
20     };
21
22 } // namespace nvinfer1
```

```
1 __global__ void addScalarKernel(const float *input,
2                                 float *output,
3                                 const float scalar,
4                                 const int nElement)
5 {
6     const int index = blockIdx.x * blockDim.x + threadIdx.x;
7     if (index >= nElement)
8         return;
9
10    float _1      = input[index];
11    float _2      = _1 + scalar;
12    output[index] = _2;
13 }
```

```
1 trt.init_libnvinfer_plugins(logger, '')
2 ctypes.cdll.LoadLibrary('AddScalar.so')
3
4 for creator in trt.get_plugin_registry().plugin_creator_list:
5     if creator.name == 'AddScalar':
6         parameterList = []
7         parameterList.append(trt.PluginField("scalar", np.float32(scalar), trt.PluginFieldType.FLOAT32))
8         plugin = creator.create_plugin(creator.name, trt.PluginFieldCollection(parameterList))
9
10    pluginLayer = network.add_plugin_v2([inputT0], plugin)
```



• Plugin 与 TensorRT 的交互

- 构建期
 - TensorRT 向 Plugin 传输参数和权重
 - Plugin 向 TensorRT 报告其输入输出张量信息，包括数量、形状（Shape）、数据类型（DataType）和数据排布（Layout）组合
 - Plugin 向 TensorRT 报告其需要的 workspace 大小
 - TensorRT 尝试各种允许的组合，选择性能最佳的输入输出组合（可能在 Plugin 前后插入 reformat 节点）
 - Plugin 不参与层 fusing
- 运行期
 - TensorRT 为 Plugin 提供输入输出张量的地址，workspace 的地址，以及所在的 stream



• Plugin 的类型

➤ IPluginV1

➤ IPluginV2

(支持单一 input / output 格式)

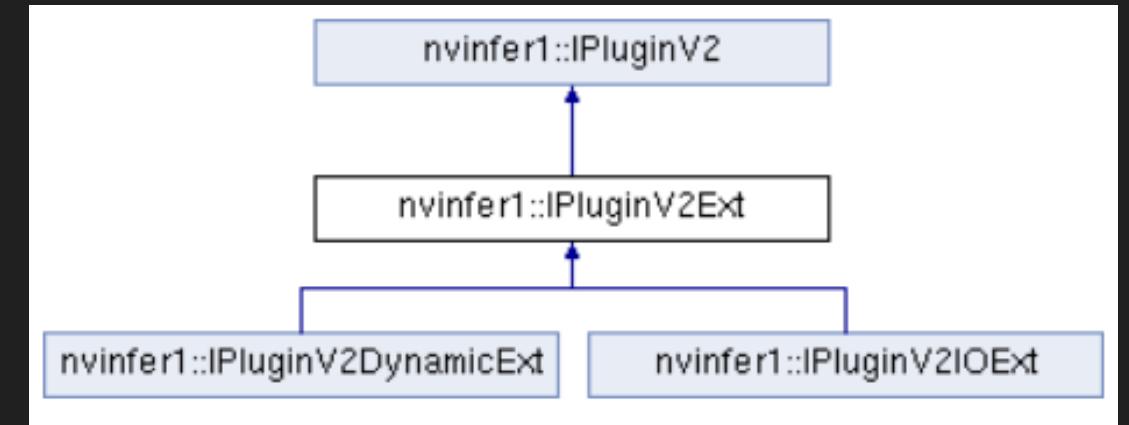
➤ IPluginV2Ext

(支持单一 input 格式和混合 output 格式)

➤ IPluginV2IOExt

(支持混合 input / output 格式, Implicit Batch 模式)

➤ **IPluginV2DynamicExt** (支持混合 input / output 格式, Dynamic Shape 模式)



➤ 范例代码

➤ 05-Plugin/usePluginV2IOExt 和 05-Plugin/usePluginV2DynamicExt, 使用两种类实现同一个 Plugin, 并搭建网络进行推理



• Plugin 的关键 API

- 以 IPluginV2DynamicExt 类为例

➤ 构造函数

- 接受来自 PluginCreator 的参数
- 做一些构建期的初始化工作

```
1 CuBLASGemmPlugin::CuBLASGemmPlugin(const std::string &name,
2                                         Weights weight,
3                                         int k,
4                                         int n,
5                                         bool needDeepCopy = false):// 传入各种参数
6     name_(name),
7     bOwnWeight_(needDeepCopy),
8     nK_(k), nN_(n)
9 {
10    WHERE_AM_I()
11    assert(weight.type == DataType::kFLOAT); // 检查传入的参数
12    assert(weight.values != nullptr);
13    assert(weight.count == k * n);
14
15    weight_.type = DataType::kFLOAT;           // 设置 Plugin 内部变量
16    weight_.count = weight.count;
17    if (needDeepCopy)
18    {
19        size_t size = sizeof(float) * weight.count;
20        weight_.values = malloc(size);
21        memcpy(reinterpret_cast<char*>(const_cast<void*>(weight_.values)), weight.values, size);
22    }
23    else
24    {
25        weight_.values = weight.values;
26    }
27
28    CHECK(cublasCreate(&handle_));           // 创建某些需要的构件
29 }
```



• Plugin 的关键 API

➤ getOutputDimensions

- 向 TensorRT 报告每个输出张量的形状
 - Dynamic Shape 模式输入输出张量形状不一定是构建期常量，需要用表达式来计算
 - outputIndex 只索引输出张量
-
- 范例代码
 - 设该 Plugin 输入两个张量，输出三个张量
 - 某次 inference 输入张量形状分别为 [2,3,5] 和 [7,9]
 - 输出张量形状分别为 [2,11,3,5+7] 和 [7,9] 和 []

```
1  DimsExprs getOutputDimensions(int32_t outputIndex,
2                                const DimsExprs *inputs,
3                                int32_t nbInputs,
4                                IExprBuilder &exprBuilder) noexcept
5  {
6      DimsExprs ret;
7
8      switch (outputIndex)
9      {
10         case 0:
11             ret.nbDims = 4;
12             ret.d[0] = inputs[0].d[0];
13             ret.d[1] = exprBuilder.constant(11);
14             ret.d[2] = inputs[0].d[1];
15             ret.d[3] = exprBuilder.operation(DimensionOperation::kSUM,
16                                              *inputs[0].d[2],
17                                              *inputs[1].d[0]);
18
19             return ret;
20
21         case 1:
22             return inputs[1];
23
24         case 2:
25             ret.nbDims = 0;
26             return ret;
27
28         default: // should NOT be here!
29             return inputs[0];
30     }
31 }
```



• Plugin 的关键 API

➤ supportsFormatCombination

- Plugin 可以同时支持多种数据类型 (DataType) 和数据排布 (LayerOut) 的输入输出张量组合
- TensorRT 深度优先遍历各 pos 尝试各种组合，该函数返回 “Plugin 是否支持当前尝试的组合”
- pos 同时索引输入和输出张量
- 第 k 号张量的条件可以基于 0~k-1 号张量的条件
- 打开 FP16 模式时才会尝试含 kHALF 的组合
- 打开 INT8 模式时才会尝试含 kINT8 的组合

➤ 范例代码

- 设该 Plugin 输入两个张量，输出一个张量
- 要求两个输入张量同时为 Int 或 Float，输出张量总是 Int

➤ 建议

- 保证性能的前提下，多实现一些 format 的组合，消除 TensorRT 自动插入 reformat 节点的开销

```
1  bool supportsFormatCombination(int32_t pos,
2                                   const PluginTensorDesc *inOut,
3                                   int32_t nbInputs,
4                                   int32_t nbOutputs) noexcept
5  {
6      WHERE_AM_I()
7      if (inOut[pos].format != TensorFormat::kLINEAR)
8          return false;
9
10     switch (pos)
11     {
12         case 0:
13             return inOut[0].type == DataType::kFLOAT || inOut[0].type == DataType::kINT32;
14         case 1:
15             return inOut[1].type == inOut[0].type;
16         case 2:
17             return inOut[2].type == DataType::kINT32;
18         default: // should NOT be here!
19             return false;
20     }
21     return false;
22 }
```



• Plugin 的关键 API

➤ configurePlugin

- 在推理前将调用该成员函数
- Dynamic Shape 模式中，每当输入数据形状发生变化（调用 context.set_binding_shape）时，该成员函数被调用
- 构建期调用时 in/out 张量形状中含有 -1
- 运行期调用时 in/out 张量形状为真实绑定的形状

```
1  configurePlugin(const DynamicPluginTensorDesc *in,
2                   int32_t nbInputs,
3                   const DynamicPluginTensorDesc *out,
4                   int32_t nbOutputs) noexcept override;
5  {
6      m_.height = in[0].desc.dims.d[1];
7      m_.width  = in[0].desc.dims.d[2];
8  }
```

➤ getWorkspaceSize

- 向 TensorRT 报告中间计算结果的存储空间
- 由 TensorRT 管理，参与显存优化

```
1  #define ALIGN_SIZE    256
2  #define ALIGN1024(x)   (((x) + (ALIGN_SIZE) - 1) / (ALIGN_SIZE) * (ALIGN_SIZE))
3
4  size_t getWorkspaceSize(const PluginTensorDesc *inputs,
5                          int32_t nbInputs,
6                          const PluginTensorDesc *outputs,
7                          int32_t nbOutputs) const noexcept override;
8  {
9      return ALIGN(sizeof(int) * inputs[0].dims.d[0] * (m_.height + 2) * (m_.width + 2)) +
10         ALIGN(sizeof(char8) * input[0].dims.d[0] * (m_.height + 2) * (m_.width + 2)) +
11         ALIGN(sizeof(int) * input[0].dims.d[0]) +
12         ALIGN(sizeof(int) * input[0].dims.d[0]);
13 }
```



• Plugin 的关键 API

➤ enqueue

- 调用 CUDA C++ kernel 计算的地方
- 可以根据输入张量的不同形状、数据类型等条件选择不同 kernel 执行计算
- 不要在 enqueue 中使用 cudaMalloc* 等函数

```
1 int32_t enqueue(const PluginTensorDesc* inputDesc,
2                  const PluginTensorDesc* outputDesc,
3                  const void* const* inputs,
4                  void* const* outputs,
5                  void* workspace,
6                  cudaStream_t stream) noexcept
7 {
8     const int nBlock = inputDesc[0].dims.d[0] * inputDesc[0].dims.d[1];
9     const int nValuePerBlock = inputDesc[0].dims.d[2];
10    if (inputDesc[0].type == DataType::kFLOAT)
11    {
12        (layerNormKernel<float,256>) <<<nBlock, nValuePerBlock, 0, stream>>>
13        ((float *)inputs[0],
14         (float *)inputs[1],
15         (float *)inputs[2],
16         (float *)outputs[0]);
17    }
18    else
19    {
20        (layerNormKernel<half,256>) <<<nBlock, nValuePerBlock, 0, stream>>>
21        ((half *)inputs[0],
22         (float *)inputs[1],
23         (float *)inputs[2],
24         (half *)outputs[0]);
25    }
26    return 0;
27 }
```

- enqueue 在 TensorRT7 和 TensorRT8 中函数声明不同，升级 TRT 版本时注意调整

```
1 int32_t enqueue(int32_t batchSize, const void * const * inputs, void *      * outputs, void * workspace, cudaStream_t stream)      override; // TensorRT7
2 int32_t enqueue(int32_t batchSize, const void * const * inputs, void * const * outputs, void * workspace, cudaStream_t stream) noexcept override; // TensorRT8
```



• Plugin 的关键 API

➤ 资源管理函数

- initialize (engine 创建时被调用，用于初始化 Plugin 层)
- terminate (engine 销毁时被调用，用于释放 initialize 函数申请的资源)
- clone (创建多个 context，可以与源对象共享本 engine 的资源)
- attachToContext (申请使用 context 独占的 cudnn 或 cublas 资源)
- detachFromContext (销毁 context 独占的 cudnn 或 cublas 资源)
- destroy (当 context/engine 销毁时被调用)

➤ 范例代码

- 05-Plugin/cuBLASPlugin，在 Plugin 中使用 cuBLAS 并通过引用计数来共享权重

```
1 CuBLASGemmPlugin(..., bool needDeepCopy = false): ...
2 {
3     ...
4     if (needDeepCopy)
5     {
6         size_t size = sizeof(float) * weight_.count;
7         weight_.values = malloc(size);
8         memcpy(reinterpret_cast<char*>(const_cast<void*>(weight_.values)), weight.values, size);
9     }
10    else
11    {
12        weight_.values = weight.values;
13    }
14 }
15 CHECK(cublasCreate(&handle_));
16 }

17 IPluginV2DynamicExt *CuBLASGemmPlugin::clone() const noexcept
18 {
19     CuBLASGemmPlugin *p = new CuBLASGemmPlugin(name_, weight_, nK_, nN_, false);
20     p->setPluginNamespace(namespace_.c_str());
21     p->pGPUWeight_ = this->pGPUWeight_;
22     return p;
23 }

24 int32_t CuBLASGemmPlugin::initialize() noexcept
25 {
26     size_t size = sizeof(float) * weight_.count;
27     CHECK(cudaMalloc((void**)&pGPUWeight_, size));
28     CHECK(cudaMemcpy(pGPUWeight_, weight_.values, size, cudaMemcpyHostToDevice));
29     return 0;
30 }

31 void CuBLASGemmPlugin::terminate() noexcept
32 {
33     CHECK(cudaFree(pGPUWeight_));
34 }

35 void CuBLASGemmPlugin::destroy() noexcept
36 {
37     if (bOwnWeight_)
38     {
39         free(const_cast<void*>(weight_.values));
40     }
41     CHECK(cublasDestroy(handle_));
42 }
```



• Plugin 的关键 API

➤ 序列化和反序列化

➤ 序列化 (Plugin 负责)

- getSerializationSize (报告序列化需要的空间大小, 单位 Byte)
- serialize (将Plugin 数据序列化到给定的 buffer 中)

➤ 反序列化 (PluginCreator 负责)

- deserializePlugin (把序列化的 buffer 传给 Plugin 的构造函数)
- Plugin 构造函数 (从 buffer 中读取数据并完成 Plugin 构造)

```
1  size_t CuBLASGemmPlugin::getSerializationSize() const noexcept
2  {
3      return sizeof(nK_) + sizeof(nN_) + sizeof(float) * weight_.count;
4  }
5
6  void CuBLASGemmPlugin::serialize(void *buffer) const noexcept
7  {
8      char * data   = reinterpret_cast<char *>(buffer);
9      size_t offset = 0;
10     memcpy(data + offset, &nK_, sizeof(nK_));
11     offset += sizeof(nK_);
12     memcpy(data + offset, &nN_, sizeof(nN_));
13     offset += sizeof(nN_);
14     size_t size = sizeof(float) * nK_ * nN_;
15     memcpy(data + offset, weight_.values, size);
16 }
17
18 IPluginV2 *CuBLASGemmPluginCreator::deserializePlugin(const char *name,
19                                                       const void *serialData,
20                                                       size_t serialLength) noexcept
21 {
22     return new CuBLASGemmPlugin(name, serialData, serialLength); }
```



```
1  CuBLASGemmPlugin::CuBLASGemmPlugin(const std::string &name,
2  const void *buffer,
3  size_t length): ...
4  {
5      const char *data   = reinterpret_cast<const char *>(buffer);
6      size_t      offset = 0;
7      memcpy(&nK_, data + offset, sizeof(nK_));
8      offset += sizeof(nK_);
9      memcpy(&nN_, data + offset, sizeof(nN_));
10     offset += sizeof(nN_);
11
12     weight_.type    = DataType::kFLOAT;
13     weight_.count   = nK_ * nN_;
14     size_t size    = sizeof(float) * nK_ * nN_;
15     weight_.values = malloc(size);
16     memcpy(reinterpret_cast<char *>(&const_cast<void *>(weight_.values)), data + offset, size);
17 }
```



• Plugin 的关键 API

➤ PluginCreator 部分

- createPlugin (依照传入参数调用 Plugin 构造函数)

```
1  IPluginV2 *createPlugin(const char *name, const PluginFieldCollection *fc) noexcept
2  {
3      int k, n;
4      Weights w;
5      for (int i = 0; i < fc->nbFields; i++)
6      {
7          PluginField field = fc->fields[i];
8          std::string field_name(field.name);
9
10         if (field_name.compare("weight") == 0)
11         {
12             w.values = field.data;
13             w.count = field.length;
14             w.type = DataType::kFLOAT;
15             continue;
16         }
17         if (field_name.compare("k") == 0)
18         {
19             k = *reinterpret_cast<const int *>(field.data);
20         }
21         if (field_name.compare("n") == 0)
22         {
23             n = *reinterpret_cast<const int *>(field.data);
24         }
25     }
26     return new CuBLASGemmPlugin(name, w, k, n, true);
27 }
```

➤ 注册 PluginCreator

- REGISTER_TENSORRT_PLUGIN(CuBLASGemmPluginCreator);



• Plugin 的关键 API

➤ Version 和 namespace 相关

➤ 序列化时 TensorRT 会将 Plugin 的名字 (Name) 、类型 (Type) 、版本号 (Version) 、命名空间 (Namespace) 信息写入 engine

➤ 通常不用修改

```
1 static const char *PLUGIN_NAME {"CuBLASGemm"};
2 static const char *PLUGIN_VERSION {"1"};
3
4 void CuBLASGemmPlugin::setPluginNamespace(const char *pluginNamespace) noexcept
5 {
6     namespace_ = pluginNamespace;
7 }
8 const char *CuBLASGemmPlugin::getPluginNamespace() const noexcept
9 {
10    return namespace_.c_str();
11 }
12
13 const char *CuBLASGemmPlugin::getPluginType() const noexcept
14 {
15    return PLUGIN_NAME;
16 }
17
18 const char *CuBLASGemmPlugin::getPluginVersion() const noexcept
19 {
20    return PLUGIN_VERSION;
21 }
22
23 void CuBLASGemmPluginCreator::setPluginNamespace(const char *pluginNamespace) noexcept
24 {
25     WHERE_AM_I()
26     namespace_ = pluginNamespace;
27 }
28
29 const char *CuBLASGemmPluginCreator::getPluginNamespace() const noexcept
30 {
31     WHERE_AM_I()
32     return namespace_.c_str();
33 }
34
35 const char *CuBLASGemmPluginCreator::getPluginName() const noexcept
36 {
37     WHERE_AM_I()
38     return PLUGIN_NAME;
39 }
40 const char *CuBLASGemmPluginCreator::getPluginVersion() const noexcept
41 {
42     WHERE_AM_I()
43     return PLUGIN_VERSION;
44 }
```



• TensorRT 中导入 Plugin

- 基本步骤
- 加载编译好的 .so
- 在 Plugin Registry 里找到需要的 Plugin
- 通过 Plugin Creator 构造需要的 Plugin
- 将 Plugin 插入网络中 (搭建网络情景)
- Parser 自动识别 (通过 Parser 加载模型场景)

```
1 logger = trt.Logger(trt.Logger.ERROR)
2 trt.init_libnvinfer_plugins(logger, '')
3 ctypes.cdll.LoadLibrary("./CuBLASGemmPlugin.so")
4 ...
5 ...
6
7 weight = np.random.rand(k * n).astype(np.float32).reshape(k, n) * 2 - 1
8 plugin = None
9
10 for c in trt.get_plugin_registry().plugin_creator_list:
11     if c.name == 'CuBLASGemm':
12         parameterList = []
13         parameterList.append(trt.PluginField("weight", np.float32(weight), trt.PluginFieldType.FLOAT32))
14         parameterList.append(trt.PluginField("k", np.int32(weight.shape[0]), trt.PluginFieldType.INT32))
15         parameterList.append(trt.PluginField("n", np.int32(weight.shape[1]), trt.PluginFieldType.INT32))
16         plugin = c.create_plugin(c.name, trt.PluginFieldCollection(parameterList))
17         break
18
19 pluginLayer = network.add_plugin_v2([inputTensor], plugin)
```

```
1 logger = trt.Logger(trt.Logger.ERROR)
2 trt.init_libnvinfer_plugins(logger, '')
3 ctypes.cdll.LoadLibrary(soFile)
4 ...
5 ...
6
7 parser = trt.OnnxParser(network, logger)
8 with open(onnxFile, 'rb') as model:
9     parser.parse(model.read())
```



• 结合使用 Parser 和 Plugin

➤ 范例代码

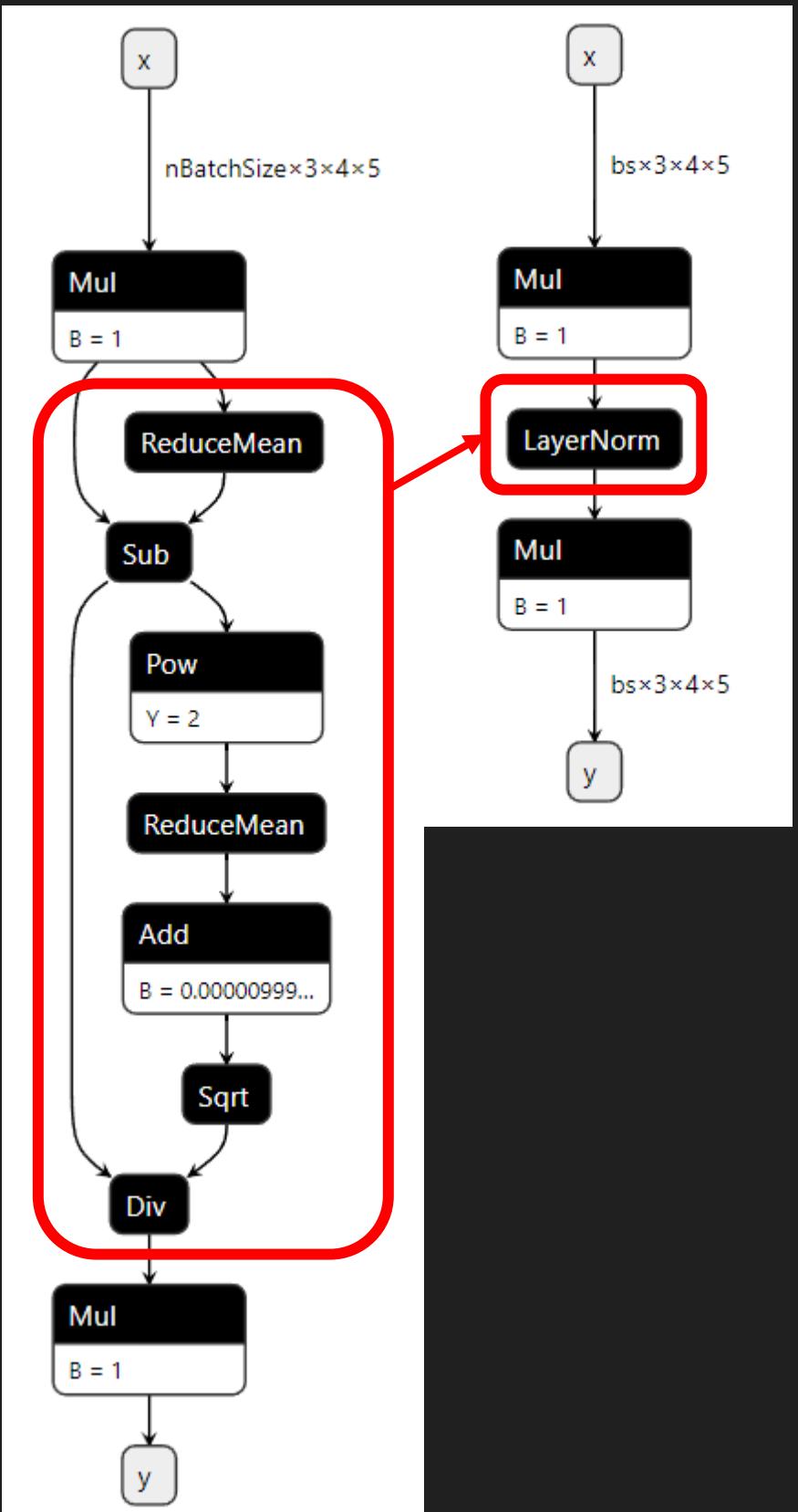
- 06-PluginAndParser 目录中 TensorFlow-addScalar 和 pyTorch-LayerNorm
- 实现 AddScalar Plugin / LayerNormalization Plugin 并替换原本 .onnx 中的相应部分

➤ 基本流程

- Netron 分析原始 .onnx 文件中需要替换的模块
- 使用 onnx-graphsurgeon 修改 .onnx 替换新节点
- 实现相应的 Plugin 并做好单元测试
- 在 TensorRT 中加载修改后的 .onnx 和 Plugin
- 对比加载前后的计算精度和性能表现

➤ 加载 Plugin 失败时的报错信息

- In node 2 (importFallbackPluginImporter): UNSUPPORTED_NODE: Assertion failed: creator
&& "Plugin not found, are the plugin name, version, and namespace correct?"



• 高级话题——Plugin 的扩展性

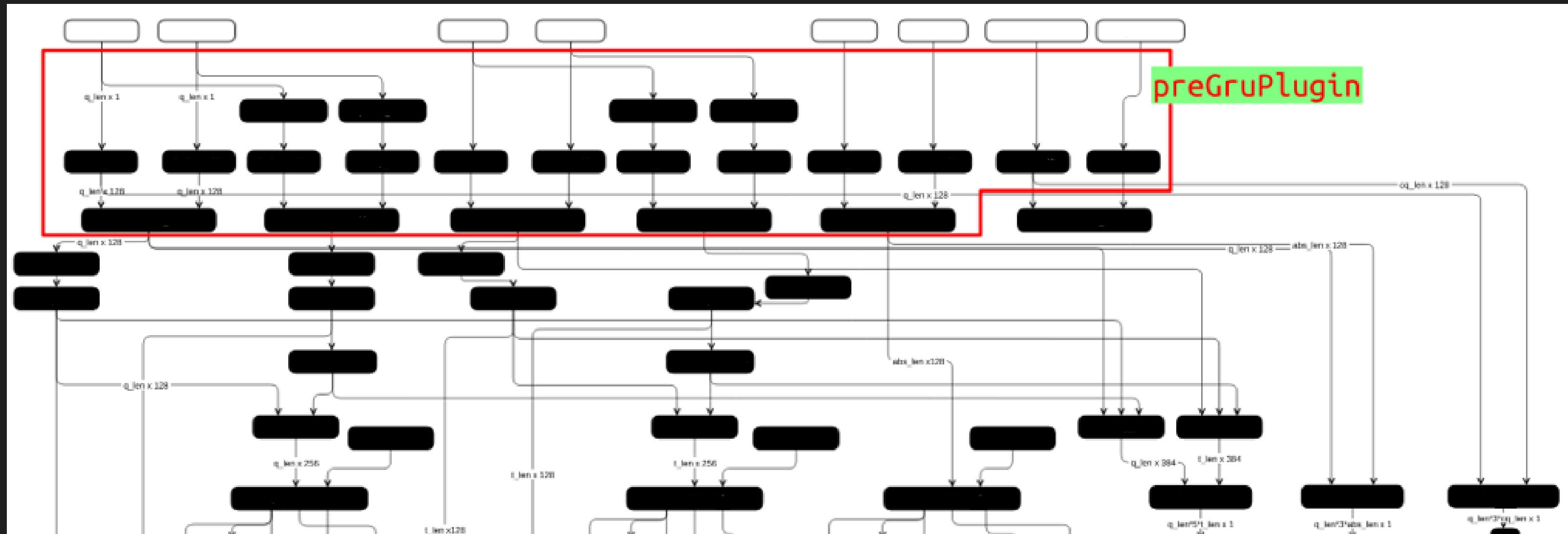
➤ Plugin 与 FP16/Int8

- 为了支持 FP16 模式，Plugin 要允许 float16 的输入输出张量类型，并实现 float16 的 CUDA C++ kernel
- 为了支持 Int8 模式下的 calibration 过程，Plugin 需要实现 FP32 的支持，否则要手工指定所有输入输出张量的 dynamic_range
- Int8 模式下，Plugin 内部张量的 dynamic_range 也要手工指定



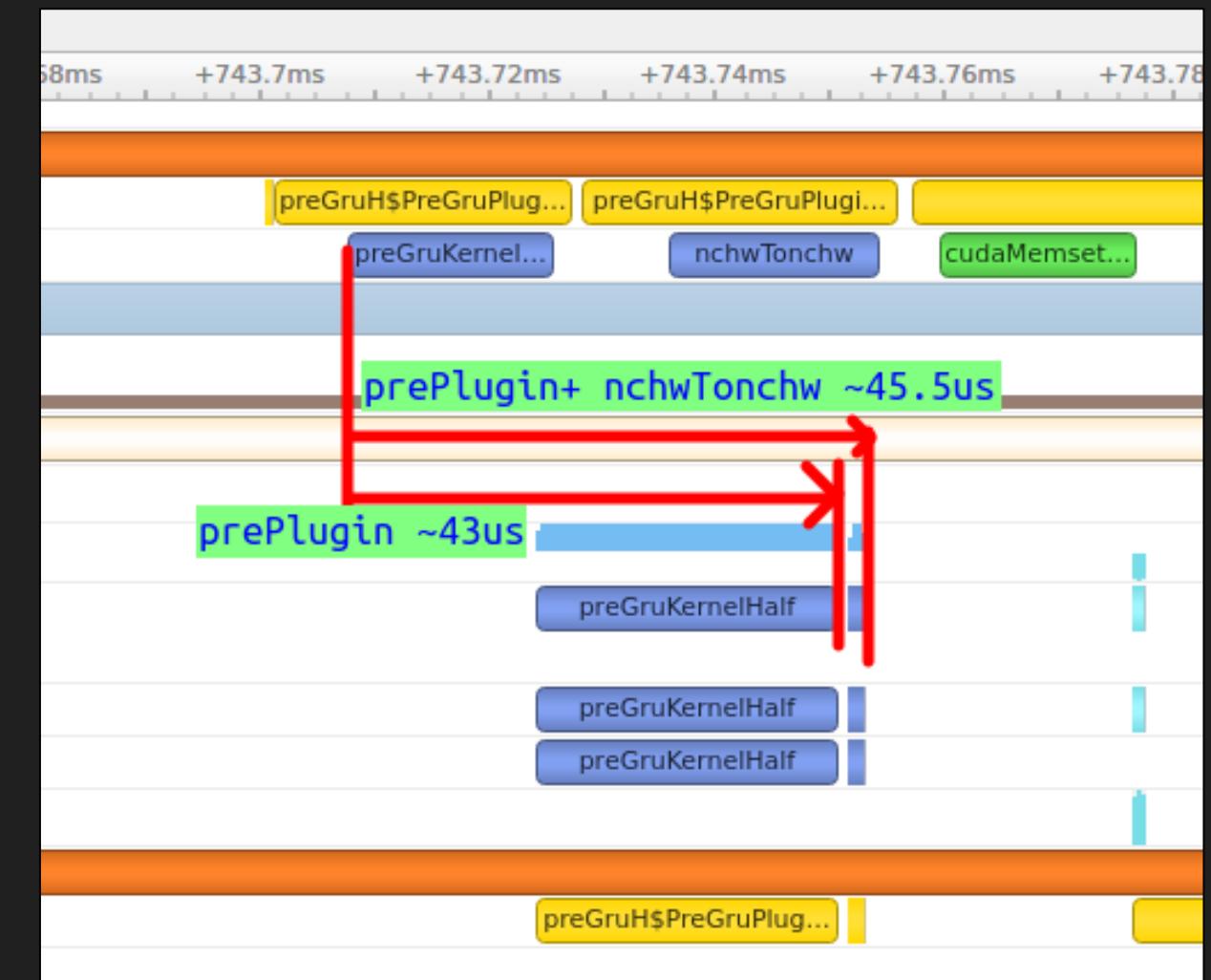
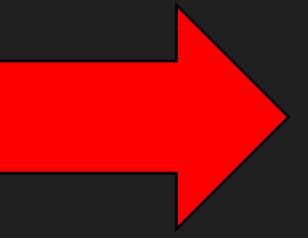
• 使用 Plugin 的例子

➤ 1、整合零散算子和 memory bound 的操作



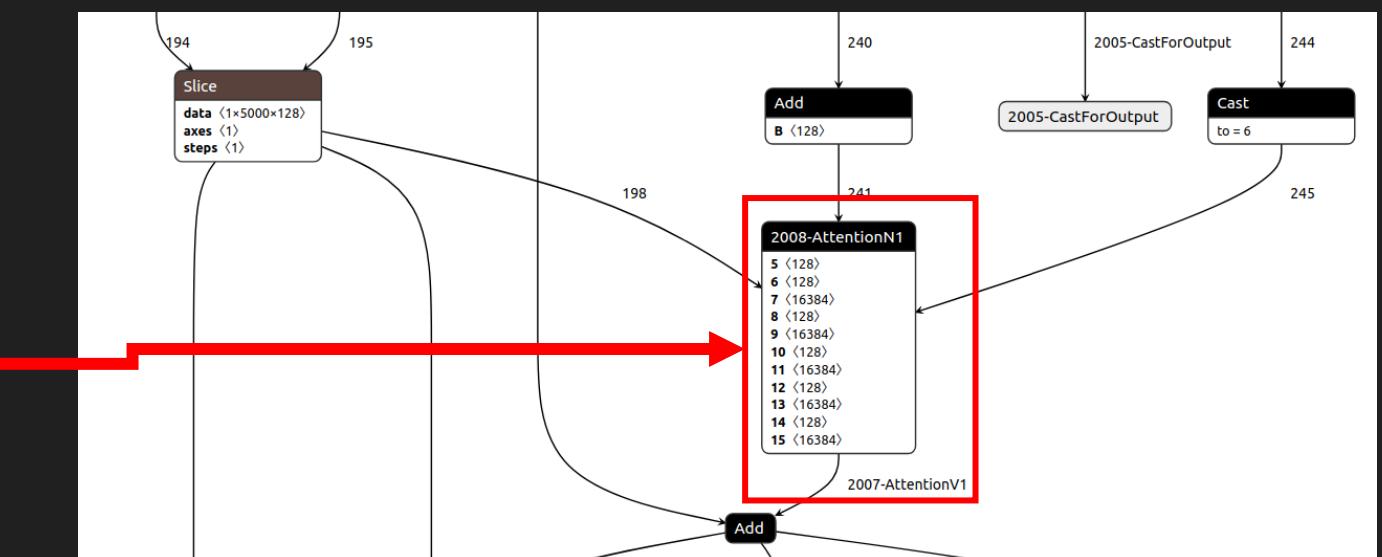
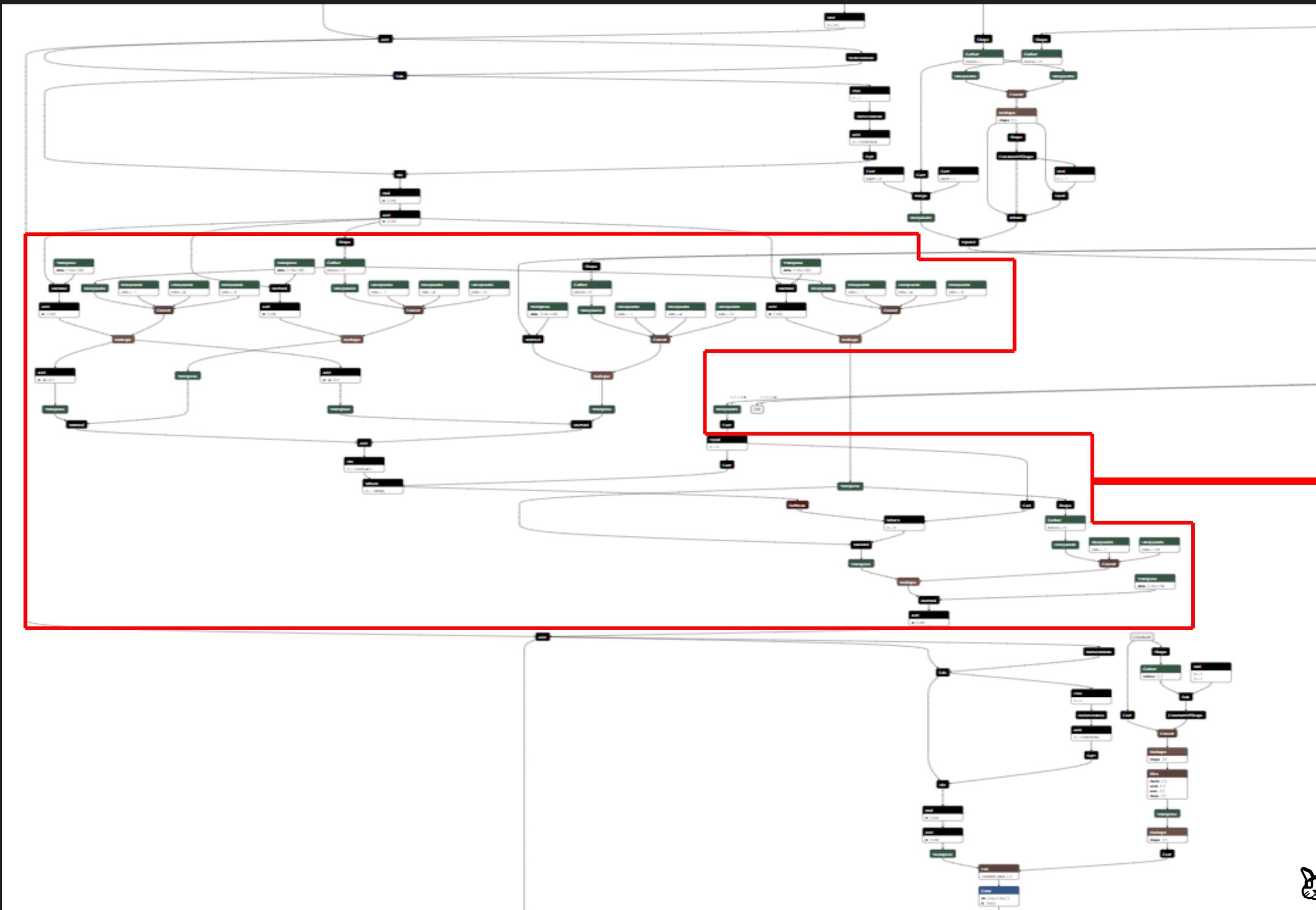
• 使用 Plugin 的例子

➤ 1、整合零散算子和 memory bound 的操作：330us v.s. 46us



• 使用 Plugin 的例子

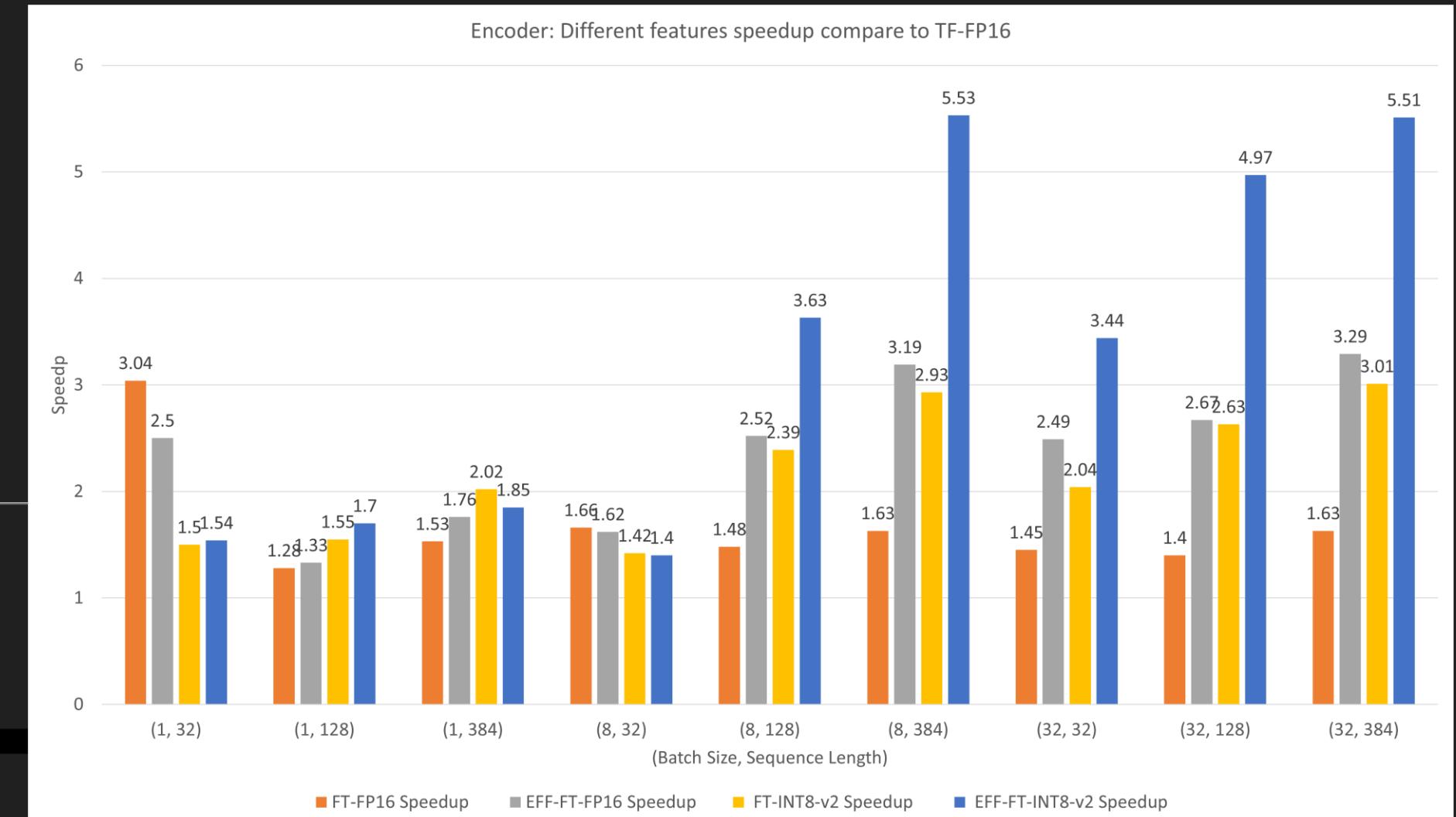
➤ 2、整合 Self-Attention 结构



• 使用 Plugin 的例子

➤ 3、融入其他高度优化的 CUDA kernel

```
1 int T5EncoderPlugin::enqueue(const PluginTensorDesc *inputDesc,
2                             const PluginTensorDesc *outputDesc,
3                             const void *const *inputs,
4                             const void *const *outputs,
5                             void *workspace,
6                             cudaStream_t stream) noexcept
7 {
8     m_.batch_size = inputDesc[0].dims.d[0];
9     m_.seq_len = inputDesc[0].dims.d[1];
10    cublasSetStream(cublasHandle_, stream);
11    pCublasWrapper_->setStream(stream);
12
13    std::unordered_map<std::string, Tensor> inputTensor {
14        {"input_ids", Tensor {MEMORY_GPU, TYPE_INT32, std::vector<size_t> {m_.batch_size, m_.seq_len}, (int *)inputs[0]}},
15        {"sequence_length", Tensor {MEMORY_GPU, TYPE_INT32, std::vector<size_t> {m_.batch_size}, (int *)inputs[1]}};
16    }
17    if (m_.useFP16)
18    {
19        std::unordered_map<std::string, Tensor> outputTensor {
20            {"output_hidden_state",
21             Tensor {MEMORY_GPU, TYPE_FP16, std::vector<size_t> {m_.batch_size, m_.seq_len, (size_t)(m_.head_num * m_.size_per_head)}, (half *)outputs[0]}}};
22        pT5EncoderHalf_->setStream(stream);
23        pT5EncoderHalf_->forward(&outputTensor, &inputTensor, pT5EncoderWeightHalf_);
24    }
25    else
26    {
27        std::unordered_map<std::string, Tensor> outputTensor {
28            {"output_hidden_state",
29             Tensor {MEMORY_GPU, TYPE_FP32, std::vector<size_t> {m_.batch_size, m_.seq_len, (size_t)(m_.head_num * m_.size_per_head)}, (float *)outputs[0]}}};
30        pT5EncoderFloat_->setStream(stream);
31        pT5EncoderFloat_->forward(&outputTensor, &inputTensor, pT5EncoderWeightFloat_);
32    }
33    return 0;
34 }
```



Part 4

• 高级话题

- 多 Optimization Profile
- 多 Context 与多 Stream
- CUDA Graph
- Timing Cache
- Algorithm Selector
- Refit
- Tactic Source
- Hardware compatibility 和 Version compatibility
- 更多工具



• 高级话题

➤ 希望解决的问题

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降?
- 怎样重叠计算和数据拷贝的时间，增加GPU利用率?
- 怎样使一个 engine 供多个线程使用?
- 怎么样优化 Kernel 的调用，减少 Launch Bound 的发生?
- engine 构建时间太长，怎么节约多次构建的时间?
- 某些 Layer 的算法导致较大的误差，能不能屏蔽掉该选择?
- 想更新模型的权重，但又不想重新构建 engine?
- 构建期/运行期显存占用过大，怎么减少?
- 能不能跨硬件或 TensorRT 版本运行 engine?



• 高级话题：多 OptimizationProfile

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降
- 解决方法：造多个 OptimizationProfile
- 范例代码
 - 08-Advance\MultiProfile
- 要点
 - 缩小每个 Profile 范围，方便 TensorRT 自动优化
 - 推理时，根据数据形状选择相应 Profile
 - 注意输入输出数据的绑定位置
- 间接作用
 - 多 Context 的基础工作
 - 增加显存占用、引擎尺寸和 .plan 尺寸

```
1 profile0 = builder.create_optimization_profile()
2 profile1 = builder.create_optimization_profile()
3
4 inputT0 = network.add_input('inputT0', trt.DataType.FLOAT, [-1, -1, -1, -1])
5
6 profile0.set_shape(inputT0.name, (1, 1, 32, 48), (4, 2, 120, 180), (8, 4, 320, 480))
7 profile1.set_shape(inputT0.name, (1, 1, 560, 840), (4, 3, 640, 960), (16, 16, 1024, 1536))
8 config.add_optimization_profile(profile0)
9 config.add_optimization_profile(profile1)
10 ...
11 ...
12 # 使用 Profile 0
13 context.set_optimization_profile_async(0, stream)
14 cudart.cudaStreamSynchronize(stream)
15 #context.active_optimization_profile = 0 # 等价方法, deprecated
16 context.set_binding_shape(0, [nIn, cIn, hIn, wIn])
17 print("Context binding all? %s" % ("Yes" if context.all_binding_shapes_specified else "No"))
18 for i in range(engine.num_bindings):
19     print(i,
20           "Input " if engine.binding_is_input(i) else "Output",
21           engine.get_binding_shape(i),
22           context.get_binding_shape(i),
23           engine.get_binding_name(i))
24 ...
25 ...
26 context.execute_v2([int(inputD0), int(outputD0), int(0), int(0)]) # buffer 一共 4 个, 放入前两个位置
27
28 # 使用 Profile 1
29 context.set_optimization_profile_async(1, stream)
30 cudart.cudaStreamSynchronize(stream)
31 context.set_binding_shape(2, [nIn, cIn, hIn, wIn])
32 print("Context binding all? %s" % ("Yes" if context.all_binding_shapes_specified else "No"))
33 for i in range(engine.num_bindings):
34     print(i,
35           "Input " if engine.binding_is_input(i) else "Output",
36           engine.get_binding_shape(i),
37           context.get_binding_shape(i),
38           engine.get_binding_name(i))
39 ...
40 context.execute_v2([int(0), int(0), int(inputD1), int(outputD1)]) # buffer 一共 4 个, 放入后两个位置
```



• 高级话题：多 OptimizationProfile

- 输入输出绑定位置
- 多出的 Profile 在绑定名末尾有 “[Profile X]”
- 推理时只需设定一组 profile 内的所有 input binding 就够了

```
1 Use Profile 0
2 Context binding all? Yes
3 0 Input (-1, -1, -1, -1) (4, 2, 120, 180) inputT0
4 1 Output (-1, -1, -1, -1) (4, 2, 120, 180) outputT0
5 2 Input (-1, -1, -1, -1) (-1, -1, -1, -1) inputT0 [profile 1]
6 3 Output (-1, -1, -1, -1) (-1, -1, -1, -1) outputT0 [profile 1]
7 check result: True
8
9 Use Profile 1
10 Context binding all? Yes
11 0 Input (-1, -1, -1, -1) (-1, -1, -1, -1) inputT0
12 1 Output (-1, -1, -1, -1) (-1, -1, -1, -1) outputT0
13 2 Input (-1, -1, -1, -1) (4, 3, 640, 960) inputT0 [profile 1]
14 3 Output (-1, -1, -1, -1) (4, 3, 640, 960) outputT0 [profile 1]
15 check result: True
```

```
1 profile0 = builder.create_optimization_profile()
2 profile1 = builder.create_optimization_profile()
3
4 inputT0 = network.add_input('inputT0', trt.DataType.FLOAT, [-1, -1, -1, -1])
5
6 profile0.set_shape(inputT0.name, (1, 1, 32, 48), (4, 2, 120, 180), (8, 4, 320, 480))
7 profile1.set_shape(inputT0.name, (1, 1, 560, 840), (4, 3, 640, 960), (16, 16, 1024, 1536))
8 config.add_optimization_profile(profile0)
9 config.add_optimization_profile(profile1)
10 ...
11 ...
12 # 使用 Profile 0
13 context.set_optimization_profile_async(0, stream)
14 cudart.cudaStreamSynchronize(stream)
15 #context.active_optimization_profile = 0 # 等价方法, deprecated
16 context.set_binding_shape(0, [nIn, cIn, hIn, wIn])
17 print("Context binding all? %s" % ("Yes" if context.all_binding_shapes_specified else "No"))
18 for i in range(engine.num_bindings):
19     print(i,
20           "Input " if engine.binding_is_input(i) else "Output",
21           engine.get_binding_shape(i),
22           context.get_binding_shape(i),
23           engine.get_binding_name(i))
24 ...
25 ...
26 context.execute_v2([int(inputD0), int(outputD0), int(0), int(0)]) # buffer 一共 4 个, 放入前两个位置
27
28 # 使用 Profile 1
29 context.set_optimization_profile_async(1, stream)
30 cudart.cudaStreamSynchronize(stream)
31 context.set_binding_shape(2, [nIn, cIn, hIn, wIn])
32 print("Context binding all? %s" % ("Yes" if context.all_binding_shapes_specified else "No"))
33 for i in range(engine.num_bindings):
34     print(i,
35           "Input " if engine.binding_is_input(i) else "Output",
36           engine.get_binding_shape(i),
37           context.get_binding_shape(i),
38           engine.get_binding_name(i))
39 ...
40 context.execute_v2([int(0), int(0), int(inputD1), int(outputD1)]) # buffer 一共 4 个, 放入后两个位置
```



• 高级话题

➤ 希望解决的问题

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降?
- 怎样重叠计算和数据拷贝的时间，增加GPU利用率?
- 怎样使一个 engine 供多个线程使用?
- 怎么样优化 Kernel 的调用，减少 Launch Bound 的发生?
- engine 构建时间太长，怎么节约多次构建时的时间?
- 某些 Layer 的算法导致较大的误差，能不能屏蔽掉该选择?
- 想更新模型的权重，但又不想重新构建 engine?
- 构建期/运行期显存占用过大，怎么减少?
- 能不能跨硬件或 TensorRT 版本运行 engine?



• 高级话题：多 Stream

➤ CUDA 编程经典话题：重叠计算和数据拷贝

➤ 解决方法：恰当使用 Stream

➤ 范例代码：

➤ 08-Advance\MultiStream

➤ 要点

➤ 使用 CUDA event 和 CUDA stream

➤ 使用 pinned-memory

➤ 使用 *Async 函数和 context.execute_async_v2 方法



```
1  context = engine.create_execution_context()
2  context.set_binding_shape(0, [nIn, cIn, hIn, wIn])
3  _, stream0 = cudart.cudaStreamCreate()
4  _, stream1 = cudart.cudaStreamCreate()
5  _, event0 = cudart.cudaEventCreate()
6  _, event1 = cudart.cudaEventCreate()
7
8  ...
9
10 cudart.cudaEventRecord(event1, stream1)
11 for i in range(30//2):
12     cudart.cudaMemcpyAsync(inputD0, inputH0, inputSize,
13                           cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream0)
14     cudart.cudaStreamWaitEvent(stream0, event1, cudart.cudaEventWaitDefault)
15     context.execute_async_v2([int(inputD0), int(outputD0)], stream0)
16     cudart.cudaEventRecord(event0, stream0)
17     cudart.cudaMemcpyAsync(outputH0, outputD0, outputSize,
18                           cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream0)
19
20     cudart.cudaMemcpyAsync(inputD1, inputH1, inputSize,
21                           cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream1)
22     cudart.cudaStreamWaitEvent(stream1, event0, cudart.cudaEventWaitDefault)
23     context.execute_async_v2([int(inputD1), int(outputD1)], stream1)
24     cudart.cudaEventRecord(event1, stream1)
25     cudart.cudaMemcpyAsync(outputH1, outputD1, outputSize,
26                           cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream1)
27
28 cudart.cudaEventSynchronize(event1)
```

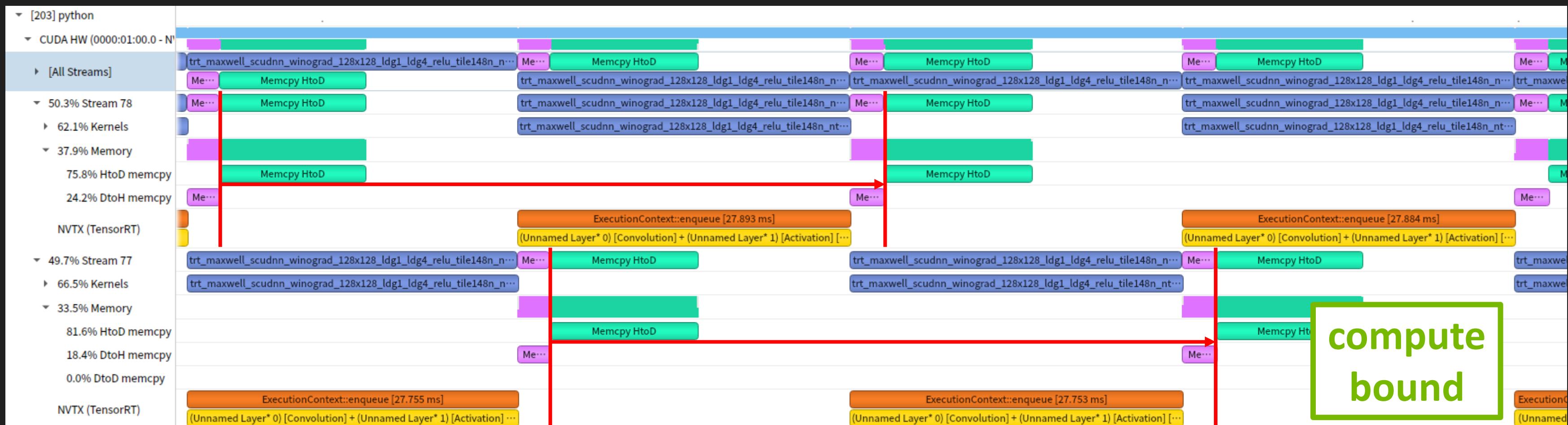


• 高级话题：多 Stream

➤ 使用单 Stream 和同步函数调用



➤ 使用双 Stream 和异步函数调用



高级话题：多 Stream



• 高级话题

➤ 希望解决的问题

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降?
- 怎样重叠计算和数据拷贝的时间，增加GPU利用率?
- 怎样使一个 engine 供多个线程使用?
- 怎么样优化 Kernel 的调用，减少 Launch Bound 的发生?
- engine 构建时间太长，怎么节约多次构建时的时间?
- 某些 Layer 的算法导致较大的误差，能不能屏蔽掉该选择?
- 想更新模型的权重，但又不想重新构建 engine?
- 构建期/运行期显存占用过大，怎么减少?
- 能不能跨硬件或 TensorRT 版本运行 engine?



• 高级话题：多 Context

➤ 一个 Engine 供多个 Context 共用

➤ 范例代码

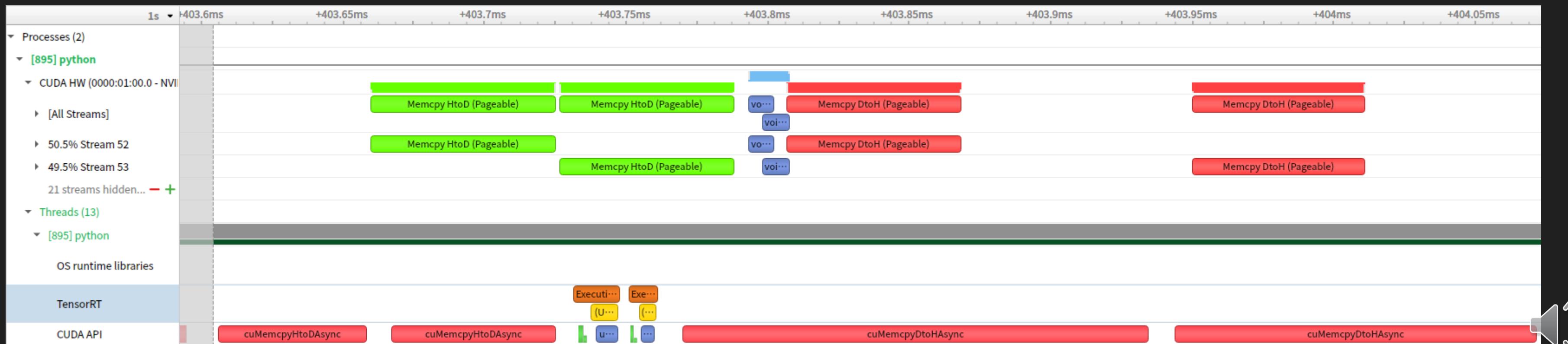
➤ 08-Advance\MultiContext

➤ 要点

➤ 从 engine 中创建多个 context

➤ `contextList = [engine.create_execution_context() for index in range(nContext)]`

➤ 不再需要使用多个 OptimizationProfile ##



• 高级话题

➤ 希望解决的问题

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降?
- 怎样重叠计算和数据拷贝的时间，增加GPU利用率?
- 怎样使一个 engine 供多个线程使用?
- 怎么样优化 Kernel 的调用，减少 Launch Bound 的发生?
- engine 构建时间太长，怎么节约多次构建时的时间?
- 某些 Layer 的算法导致较大的误差，能不能屏蔽掉该选择?
- 想更新模型的权重，但又不想重新构建 engine?
- 构建期/运行期显存占用过大，怎么减少?
- 能不能跨硬件或 TensorRT 版本运行 engine?



• 高级话题：CUDA Graph

➤ 利用 CUDA Graph

➤ <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs>

➤ 范例代码

➤ 08-Advance\CudaGraph

➤ 优点：

➤ 降低 CPU Launch cost

➤ CUDA 工作流优化

➤ 缓解大量 kernel 调用时的 Launch Bound

➤ 要点

➤ 步骤：Graph 定义，Graph 实例化，Graph 执行

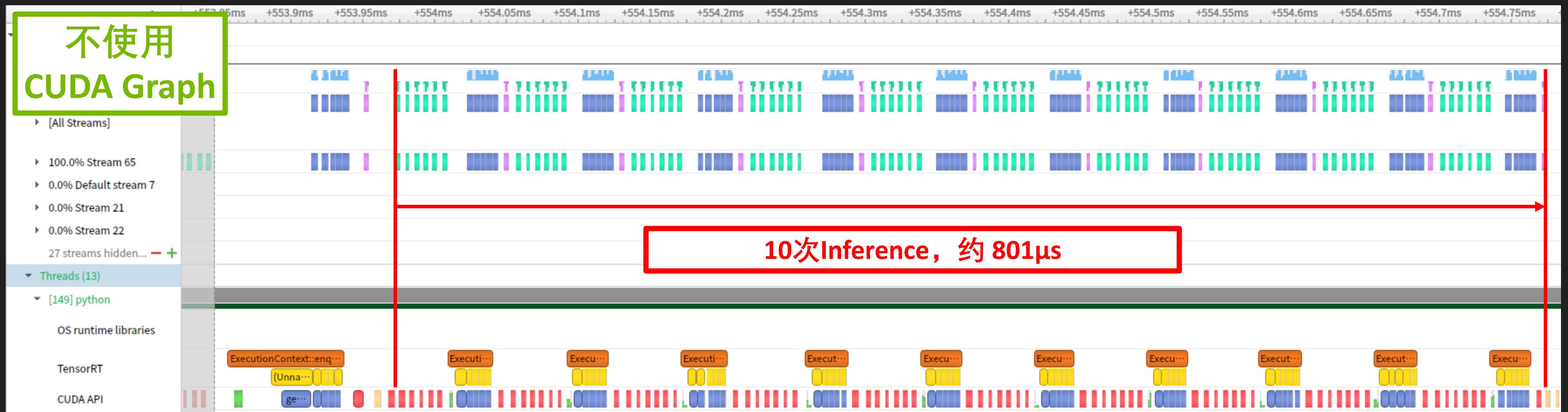
➤ Dynamic Shape 模式中，实际数据形状发生改变时（调用 context.set_binding_shape），

要先跑一遍 context.execute 再重新捕获 Graph，最后再实例化和执行 Graph

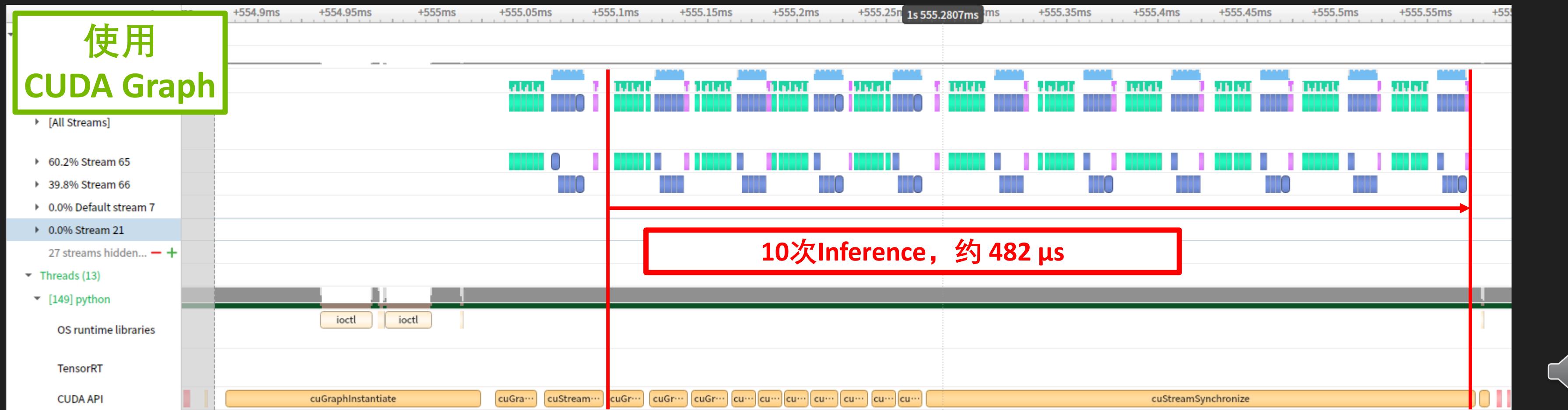
```
1 # 首次捕获 CUDA Graph 并运行
2 cudart.cudaStreamBeginCapture(stream, cudart.cudaStreamCaptureMode.cudaStreamCaptureModeGlobal)
3 cudart.cudaMemcpyAsync(inputD0, inputH0.ctypes.data, inputH0.nbytes,
4 cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
5 context.execute_async_v2([int(inputD0), int(outputD0)], stream)
6 cudart.cudaMemcpyAsync(outputH0.ctypes.data, outputD0, outputH0.nbytes,
7 cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
8 #cudart.cudaStreamSynchronize(stream) # 不用在 graph 内同步
9 _, graph = cudart.cudaStreamEndCapture(stream)
10 _, graphExe, _ = cudart.cudaGraphInstantiate(graph, b"", 0)
11
12 cudart.cudaGraphLaunch(graphExe, stream) # 每次准备好 buffer 后用 cudaGraphLaunch 运行
13 cudart.cudaStreamSynchronize(stream)
14
15 # 输入尺寸改变后，需要首先运行一次推理，然后重新捕获 CUDA Graph，最后再运行
16 context.set_binding_shape(0, [2, 3, 4])
17 ...
18 context.execute_async_v2([int(inputD0), int(outputD0)], stream)
19 ...
20 cudart.cudaStreamSynchronize(stream)
21
22 cudart.cudaStreamBeginCapture(stream, cudart.cudaStreamCaptureMode.cudaStreamCaptureModeGlobal)
23 cudart.cudaMemcpyAsync(inputD0, inputH0.ctypes.data, inputH0.nbytes,
24 cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
25 context.execute_async_v2([int(inputD0), int(outputD0)], stream)
26 cudart.cudaMemcpyAsync(outputH0.ctypes.data, outputD0, outputH0.nbytes,
27 cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
28 _, graph = cudart.cudaStreamEndCapture(stream)
29 _, graphExe, _ = cudart.cudaGraphInstantiate(graph, b"", 0)
30
31 cudart.cudaGraphLaunch(graphExe, stream)
32 cudart.cudaStreamSynchronize(stream)
```



• 高级话题：CUDA Graph



10次Inference，约 801μs



10次Inference, 约 482 μs



• 高级话题

➤ 希望解决的问题

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降?
- 怎样重叠计算和数据拷贝的时间，增加GPU利用率?
- 怎样使一个 engine 供多个线程使用?
- 怎么样优化 Kernel 的调用，减少 Launch Bound 的发生?
- engine 构建时间太长，怎么节约多次构建时的时间?
- 某些 Layer 的算法导致较大的误差，能不能屏蔽掉该选择?
- 想更新模型的权重，但又不想重新构建 engine?
- 构建期/运行期显存占用过大，怎么减少?
- 能不能跨硬件或 TensorRT 版本运行 engine?



• 高级话题：Timing Cache

➤ 范例代码：

➤ 08-Advance\TimingCache

➤ 优点：

- 优化单次引擎构建时间（模型内多个同参数的算子）
- 优化多次引擎构建时间（debug、参数更新后重新构建）
- 优化同环境下多个引擎构建时间（跨 builder 可用）
- 用于反复生成一模一样的引擎

➤ 要点

- 类似引擎序列化反序列化，将 Timing Cache 保存出来下次用
- 类似 .plan，不可跨平台和开发环境使用

```
1  timeCache = b""
2  if useTimeCache and os.path.isfile(timeCacheFile):
3      with open(timeCacheFile, 'rb') as f:
4          timeCache = f.read()
5
6      ... # 建立 Builder 和 BbuilderConfig
7
8  if useTimeCache:
9      cache = config.create_timing_cache(timeCache)
10     config.set_timing_cache(cache, False)
11
12     ... # 建立 Network 和 SerializedNetwork
13
14 if useTimeCache and not os.path.isfile(timeCacheFile):
15     timeCache = config.get_timing_cache()
16     timeCacheString = timeCache.serialize()
17     with open(timeCacheFile, 'wb') as f:
18         f.write(timeCacheString)
```

```
1  run(0) # 不使用 Timing Cache
2  run(1) # 创建并保存 Timing Cache
3  run(1) # 读取并使用 Timing Cache
4
5  Without timing cache, 1707.987785 ms
6  With timing cache, 724.868298 ms
7  Succeeded saving .cache file!
8  Succeeded getting serialized timing cache!
9  With timing cache, 32.098293 ms
```



• 高级话题：Timing Cache

创建并保存 Timing Cache

```
1 [V] Applying generic optimizations to the graph for inference.
2 ...
3 ...
4 ...
5 ...
6 [V] ===== Computing reformatting costs
7 [V] ***** Autotuning Reformat: Float(784,784,28,1) -> Float(784,1,28,1) *
8 [V] ----- Timing Runner: Optimizer Reformat(inputT0 -> <out>) (Reformat)
9 [V] Tactic: 1002 Time: 0.00512
10 [V] Tactic: 0 Time: 0.003072
11 [V] Fastest Tactic: 0 Time: 0.003072
12 [V] ===== Computing reformatting costs
13 [V] ***** Autotuning Reformat: Float(25088,1,896,32) -> Float(25088,784,2
14 [V] ----- Timing Runner: Optimizer Reformat(Unnamed Layer* 1) [Activatio
15 [V] Tactic: 1002 Time: 0.01536
16 [V] Tactic: 0 Time: 0.008896
17 [V] Fastest Tactic: 0 Time: 0.008896
18 ...
19 ...
20 ...
21 [V] Formats and tactics selection completed in 1.28525 seconds.
22 ...
23 ...
24 ...
25 Layer(CaskConvolution): (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1) [Acti
26 Layer(TiledPooling): (Unnamed Layer* 2) [Pooling], Tactic: 6357249, (Unnamed Layer*
27 Layer(Reformat): Reformatting CopyNode for Input Tensor 0 to (Unnamed Layer* 3) [Co
28 Layer(CaskConvolution): (Unnamed Layer* 3) [Convolution] + (Unnamed Layer* 4) [Acti
29 Layer(Reformat): Reformatting CopyNode for Input Tensor 0 to (Unnamed Layer* 5) [Po
30 Layer(TiledPooling): (Unnamed Layer* 5) [Pooling], Tactic: 6357249, Reformatted Inp
31 Layer(Shuffle): (Unnamed Layer* 6) [Shuffle], Tactic: 0, (Unnamed Layer* 5) [Poolin
32 Layer(CublasConvolution): (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
33 Layer(CublasConvolution): (Unnamed Layer* 9) [Fully Connected] + (Unnamed Layer* 10
34 Layer(NoOp): (Unnamed Layer* 11) [Shuffle], Tactic: 0, (Unnamed Layer* 10) [Activat
35 Layer(LogSoftmaxTopK): (Unnamed Layer* 12) [Softmax] + (Unnamed Layer* 13) [TopK],
36 ...
37 ...
38 ...
39 [V] Deleting timing cache: 31 entries, 0 hits
40 Succeeded getting serialized engine!
41 Succeeded saving .cache file!
42 Succeeded building engine!
```

读取并使用 Timing Cache

```
1 [V] Loaded 31 timing cache entries.
2 [V] Applying generic optimizations to the graph for inference.
3 ...
4 ...
5 ...
6 [V] ===== Computing reformatting costs
7 [V] ***** Autotuning Reformat: Float(784,784,28,1) -> Float(784,1,28,1) *
8 [V] ----- Timing Runner: Optimizer Reformat(inputT0 -> <out>) (Reformat)
9 ...
10 ...
11 ...
12 [V] ===== Computing reformatting costs
13 [V] ***** Autotuning Reformat: Float(25088,1,896,32) -> Float(25088,784,2
14 ...
15 ...
16 ...
17 ...
18 ...
19 ...
20 ...
21 [V] Formats and tactics selection completed in 0.014933 seconds.
22 ...
23 ...
24 ...
25 Layer(CaskConvolution): (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1) [Acti
26 Layer(TiledPooling): (Unnamed Layer* 2) [Pooling], Tactic: 6226177, (Unnamed Layer*
27 Layer(Reformat): Reformatting CopyNode for Input Tensor 0 to (Unnamed Layer* 3) [Co
28 Layer(CaskConvolution): (Unnamed Layer* 3) [Convolution] + (Unnamed Layer* 4) [Acti
29 Layer(Reformat): Reformatting CopyNode for Input Tensor 0 to (Unnamed Layer* 5) [Po
30 Layer(TiledPooling): (Unnamed Layer* 5) [Pooling], Tactic: 6488321, Reformatted Inp
31 Layer(Shuffle): (Unnamed Layer* 6) [Shuffle], Tactic: 0, (Unnamed Layer* 5) [Poolin
32 Layer(CublasConvolution): (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
33 Layer(CublasConvolution): (Unnamed Layer* 9) [Fully Connected] + (Unnamed Layer* 10
34 Layer(NoOp): (Unnamed Layer* 11) [Shuffle], Tactic: 0, (Unnamed Layer* 10) [Activat
35 Layer(LogSoftmaxTopK): (Unnamed Layer* 12) [Softmax] + (Unnamed Layer* 13) [TopK],
36 ...
37 ...
38 ...
39 [V] Deleting timing cache: 31 entries, 31 hits
40 Succeeded getting serialized timing cache!
41 Succeeded getting serialized engine!
42 Succeeded building engine!
```



• 高级话题

➤ 希望解决的问题

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降?
- 怎样重叠计算和数据拷贝的时间，增加GPU利用率?
- 怎样使一个 engine 供多个线程使用?
- 怎么样优化 Kernel 的调用，减少 Launch Bound 的发生?
- engine 构建时间太长，怎么节约多次构建时的时间?
- 某些 Layer 的算法导致较大的误差，能不能精确选择算法?
- 想更新模型的权重，但又不想重新构建 engine?
- 构建期/运行期显存占用过大，怎么减少?



• 高级话题：Algorithm Selector

➤ 样例代码

➤ 08-Advance\AlgorithmSelector

➤ 要点

- 自己实现一个 MyAlgorithmSelector 类
- 关键是实现两个成员函数
 - 一个用来挑选特定层的算法
 - 一个用来报告所有层的挑选结果
- 构建网络时交给 BuilderConfig

```
1  class MyAlgorithmSelector(trt.IAlgorithmSelector):  
2  
3      def __init__(self, keepAll=True):  
4          super(MyAlgorithmSelector, self).__init__()  
5          self.keepAll = keepAll  
6  
7      def select_algorithms(self, layerAlgorithmContext, layerAlgorithmList):  
8  
9          # 构建 SerializedNetwork 时被调用  
10         # 输入一个 Layer (layerAlgorithmContext) 和一个算法列表 (layerAlgorithmList)  
11         # 返回一个从 0 开始的索引列表, 表示上述算法列表中哪些可以保留  
12  
13     def report_algorithms(self, modelAlgorithmContext, modelAlgorithmList):  
14  
15         # 构建 SerializedNetwork 完成后被调用  
16         # 输入网络所有 Layer (modelAlgorithmContext) 和各 Layer 保留的算法列表 (modelAlgorithmList)  
17         # 没有返回  
18  
19     ... # 创建 Builder 和BuilderConfig  
20     config.algorithm_selector = MyAlgorithmSelector(True) # 设置算法选择器
```

➤ 实际工作流程：

- 先通过 polygraphy 等工具发现某层的个 Tactic 结果不理想,
- 构造 Algorithm Selector 屏蔽掉盖层的该 tactic
- 构建引擎



• 高级话题：Algorithm Selector

```
1  class MyAlgorithmSelector(trt.IAlgorithmSelector):
2
3      def __init__(self, keepAll=True):
4          super(MyAlgorithmSelector, self).__init__()
5          self.keepAll = keepAll
6
7      def select_algorithms(self, layerAlgorithmContext, layerAlgorithmList):
8          if self.keepAll: # 保留全部选择
9              return list(range(len(layerAlgorithmList)))
10
11     else: # 手工筛选算法
12         # 选择计算时间最长的算法
13         timeList = [algorithm.timing_msec for algorithm in layerAlgorithmList]
14         return list(np.argmax(timeList))
15
16     # 选择 workspace 最小的算法
17     workspaceSizeList = [algorithm.workspace_size for algorithm in layerAlgorithmList]
18     return list(np.argmin(workspaceSizeList))
19
20     # 让特定层选择特定算法（用于多次构建一模一样的引擎）
21     if layerAlgorithmContext.name == "(Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1) [Activation]":
22         # 最后的数字来自 VERBOSE 日志中信息，代表该层的一种实现
23         return [index for index, algorithm in enumerate(layerAlgorithmList)
24                 if algorithm.algorithm_varient.implementation == 2147483648]
25
26     return []
27
28     def report_algorithms(self, modelAlgorithmContext, modelAlgorithmList):
29         for i in range(len(modelAlgorithmContext)):
30             context = modelAlgorithmContext[i]
31             algorithm = modelAlgorithmList[i]
32
33             print("Layer%4d:%s" % (i, context.name))
34             nInput = context.num_inputs
35             nOutput = context.num_outputs
36             for j in range(nInput):
37                 ioInfo = algorithm.get_algorithm_io_info(j)
38                 print("      Input [%2d]:%s,%s,%s,%s" % (j, context.get_shape(j), ioInfo.dtype, ioInfo.strides, ioInfo.tensor_format))
39             for j in range(nOutput):
40                 ioInfo = algorithm.get_algorithm_io_info(j + nInput)
41                 print("      Output[%2d]:%s,%s,%s,%s" % (j, context.get_shape(j + nInput), ioInfo.dtype, ioInfo.strides, ioInfo.tensor_format))
42             print("      algorithm:[implementation:%d,tactic:%d,timing:%fms,workspace:%dMB]"% \
43                   (algorithm.algorithm_varient.implementation,
44                    algorithm.algorithm_varient.tactic,
45                    algorithm.timing_msec,
46                    algorithm.workspace_size))
```



• 高级话题：Algorithm Selector

```
1  ↳ Layer  0:(Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1) [Activation]
2      Input [ 0]:[(1, 1, 28, 28), (1, 1, 28, 28), (2, 1, 28, 28)], DataType.FLOAT, (784, 784, 28, 1), TensorFormat.LINEAR
3      Output[ 0]:[(1, 32, 28, 28), (1, 32, 28, 28), (2, 32, 28, 28)], DataType.FLOAT, (25088, 784, 28, 1), TensorFormat.LINEAR
4      algorithm:[implementation:2147483649,tactic:9306111,timing:0.000000ms,workspace:0MB]
5  ↳ Layer  1:(Unnamed Layer* 2) [Pooling]
6      Input [ 0]:[(1, 32, 28, 28), (1, 32, 28, 28), (2, 32, 28, 28)], DataType.FLOAT, (25088, 784, 28, 1), TensorFormat.LINEAR
7      Output[ 0]:[(1, 32, 14, 14), (1, 32, 14, 14), (2, 32, 14, 14)], DataType.FLOAT, (6272, 196, 14, 1), TensorFormat.LINEAR
8      algorithm:[implementation:2147483679,tactic:6095105,timing:0.000000ms,workspace:0MB]
9  ↳ Layer  2:(Unnamed Layer* 3) [Convolution] + (Unnamed Layer* 4) [Activation]
10     Input [ 0]:[(1, 32, 14, 14), (1, 32, 14, 14), (2, 32, 14, 14)], DataType.FLOAT, (6272, 196, 14, 1), TensorFormat.LINEAR
11     Output[ 0]:[(1, 64, 14, 14), (1, 64, 14, 14), (2, 64, 14, 14)], DataType.FLOAT, (12544, 196, 14, 1), TensorFormat.LINEAR
12     algorithm:[implementation:2147483649,tactic:8060927,timing:0.000000ms,workspace:0MB]
13  ↳ Layer  3:(Unnamed Layer* 5) [Pooling]
14     Input [ 0]:[(1, 64, 14, 14), (1, 64, 14, 14), (2, 64, 14, 14)], DataType.FLOAT, (12544, 196, 14, 1), TensorFormat.LINEAR
15     Output[ 0]:[(1, 64, 7, 7), (1, 64, 7, 7), (2, 64, 7, 7)], DataType.FLOAT, (3136, 49, 7, 1), TensorFormat.LINEAR
16     algorithm:[implementation:2147483679,tactic:6357249,timing:0.000000ms,workspace:0MB]
17  ↳ Layer  4:(Unnamed Layer* 6) [Shuffle]
18     Input [ 0]:[(1, 64, 7, 7), (1, 64, 7, 7), (2, 64, 7, 7)], DataType.FLOAT, (3136, 49, 7, 1), TensorFormat.LINEAR
19     Output[ 0]:[(1, 3136, 1, 1), (1, 3136, 1, 1), (2, 3136, 1, 1)], DataType.FLOAT, (3136, 1, 1, 1), TensorFormat.LINEAR
20     algorithm:[implementation:2147483661,tactic:0,timing:0.000000ms,workspace:0MB]
21  ↳ Layer  5:(Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8) [Activation]
22     Input [ 0]:[(1, 3136, 1, 1), (1, 3136, 1, 1), (2, 3136, 1, 1)], DataType.FLOAT, (3136, 1, 1, 1), TensorFormat.LINEAR
23     Output[ 0]:[(1, 1024, 1, 1), (1, 1024, 1, 1), (2, 1024, 1, 1)], DataType.FLOAT, (1024, 1, 1, 1), TensorFormat.LINEAR
24     algorithm:[implementation:2147483692,tactic:0,timing:0.000000ms,workspace:0MB]
25  ↳ Layer  6:(Unnamed Layer* 9) [Fully Connected] + (Unnamed Layer* 10) [Activation]
26     Input [ 0]:[(1, 1024, 1, 1), (1, 1024, 1, 1), (2, 1024, 1, 1)], DataType.FLOAT, (1024, 1, 1, 1), TensorFormat.LINEAR
27     Output[ 0]:[(1, 10, 1, 1), (1, 10, 1, 1), (2, 10, 1, 1)], DataType.FLOAT, (10, 1, 1, 1), TensorFormat.LINEAR
28     algorithm:[implementation:2147483692,tactic:1,timing:0.000000ms,workspace:0MB]
29  ↳ Layer  7:(Unnamed Layer* 11) [Shuffle]
30     Input [ 0]:[(1, 10, 1, 1), (1, 10, 1, 1), (2, 10, 1, 1)], DataType.FLOAT, (10, 1, 1, 1), TensorFormat.LINEAR
31     Output[ 0]:[(1, 10), (1, 10), (2, 10)], DataType.FLOAT, (10, 1), TensorFormat.LINEAR
32     algorithm:[implementation:2147483661,tactic:0,timing:0.000000ms,workspace:0MB]
33  ↳ Layer  8:(Unnamed Layer* 12) [Softmax] + (Unnamed Layer* 13) [TopK]
34     Input [ 0]:[(1, 10), (1, 10), (2, 10)], DataType.FLOAT, (10, 1), TensorFormat.LINEAR
35     Output[ 0]:[(1, 1), (1, 1), (2, 1)], DataType.FLOAT, (1, 1), TensorFormat.LINEAR
36     Output[ 1]:[(1, 1), (1, 1), (2, 1)], DataType.INT32, (1, 1), TensorFormat.LINEAR
37     algorithm:[implementation:2147483685,tactic:1001,timing:0.000000ms,workspace:0MB]
38     Succeeded getting serialized engine!
39     Succeeded building engine!
40     Bind[ 0]:i[ 0]-> DataType.FLOAT (-1, 1, 28, 28) (1, 1, 28, 28) inputT0
41     Bind[ 1]:o[ 0]-> DataType.INT32 (-1, 1) (1, 1) (Unnamed Layer* 13) [TopK]_output_2
42     inputT0
43     (Unnamed Layer* 13) [TopK]_output_2
```



• 高级话题

➤ 希望解决的问题

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降?
- 怎样重叠计算和数据拷贝的时间，增加GPU利用率?
- 怎样使一个 engine 供多个线程使用?
- 怎么样优化 Kernel 的调用，减少 Launch Bound 的发生?
- engine 构建时间太长，怎么节约多次构建时的时间?
- 某些 Layer 的算法导致较大的误差，能不能精确选择算法?
- 想更新模型的权重，但又不想重新构建 engine?
- 构建期/运行期显存占用过大，怎么减少?
- 能不能跨硬件或 TensorRT 版本运行 engine?



• 高级话题：Refit

➤ 优点

➤ 节约反复构建引擎的时间

➤ 强化学习必备

➤ 要点

➤ BuilderConfig 中设置相应 Flag

➤ 在构建好 engine 的基础上更新权重

➤ 更新某层权重后，邻近层可能也需要更新（尽管其值可能不变），如 Convolution 层中的 kernel 和 bias

[TensorRT] ERROR: 4: [refit.cpp::refitCudaEngine::1769] Error Code 4: Internal Error (missing 1 needed Weights. Call IRefitter::getMissing to get their layer names and roles or IRefitter::getMissingWeights to get their weights names.)

➤ 注意权重的排布方式

➤ Dynamic Shape 模式暂不支持（未来 TensorRT 版本将添加支持）

[TRT] [E] 4: [network.cpp::validate::2924] Error Code 4: Internal Error (Refittable networks with dynamic shapes is not supported.)

```
1 config.flags = 1 << int(trt.BuilderFlag.REFIT)
2 ...
3 ... # 已经构建好了 engine
4
5 refitter = trt.Refitter(engine, logger)
6 refitter.set_weights("conv", trt.WeightsRole.KERNEL, weight)
7 refitter.set_weights("conv", trt.WeightsRole.BIAS, bias)
8
9 [missingLayer, weightRole] = refitter.get_missing()
10 for layer, role in zip(missingLayer, weightRole):
11     print("[", layer, "-", role, "]")
12
13 if refitter.refit_cuda_engine() == False:
14     print("Failed Refitting engine!")
15     return
```



• 高级话题

➤ 希望解决的问题

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降?
- 怎样重叠计算和数据拷贝的时间，增加GPU利用率?
- 怎样使一个 engine 供多个线程使用?
- 怎么样优化 Kernel 的调用，减少 Launch Bound 的发生?
- engine 构建时间太长，怎么节约多次构建时的时间?
- 某些 Layer 的算法导致较大的误差，能不能精确选择算法?
- 想更新模型的权重，但又不想重新构建 engine?
- 构建期/运行期显存占用过大，怎么减少?
- 能不能跨硬件或 TensorRT 版本运行 engine?



• 高级话题：Tactic Source

➤ 要点

- BuilderConfig 中设置相应 Flag
- 可以开启或关闭 cuBLAS、cuBLASLt、cuDNN

➤ 优点

- 节约部分内存、显存，减少构建时间

➤ 缺点

- 不能使用某些优化，可能导致性能下降
- 可能导致构建失败
- 后续版本中，TensorRT 将彻底断开对外部 Library 依赖

```
1 config.set_tactic_sources(1 << int(trt.TacticSource.CUBLAS) | \
2                               1 << int(trt.TacticSource.CUBLAS_LT) | \
3                               1 << int(trt.TacticSource.CUDNN))
4 #config.set_tactic_sources(1 << int(trt.TacticSource.CUBLAS) | \
5 #                               1 << int(trt.TacticSource.CUBLAS_LT))
```



• 高级话题：Tactic Source

所有 Tactic

```
1 [V] Using cublas as a tactic source
2 [I] [MemUsageChange] Init cuBLAS/cuBLASLt: CPU +269, GPU +112, now: CPU 541, GPU 739 (MiB)
3- [V] Using cuDNN as a tactic source
4- [I] [MemUsageChange] Init cuDNN: CPU +112, GPU +46, now: CPU 653, GPU 785 (MiB)
5 [I] Local timing cache in use. Profiling results in this builder pass will not be stored.
6-
7 [V] Constructing optimization profile number 0 [1/1].
8
9 ...
10
11 [V] ----- Timing Runner: (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1)
    [Activation] (CudnnConvolution)
12- [V] Tactic: 0 Time: 0.013312
13- [V] Tactic: 1 Time: 0.01328
14- [V] Tactic: 2 Time: 0.0256
15- [V] Tactic: 4 Time: 0.031744
16- [V] Tactic: 5 Time: 0.03072
17- [V] Fastest Tactic: 1 Time: 0.01328
18 [V] ----- Timing Runner: (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1)
    [Activation] (CaskConvolution)
19 [V] (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1) [Activation] Set Tactic Name:
    maxwell_scudnn_128x32_relu_medium_nn_v1 Tactic: 1062367460111450758
20- [V] Tactic: 1062367460111450758 Time: 0.014272
21
22 ...
23
24 [V] (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1) [Activation] Set Tactic Name:
    maxwell_scudnn_128x128_relu_small_nn_v1 Tactic: -410470605513481746
25- [V] Tactic: -410470605513481746 Time: 0.012288
26- [V] Fastest Tactic: 7144526460361122478 Time: 0.009216
27
28 ...
29
30 [V] ----- Timing Runner: (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
    [Activation] (CudaDepthwiseConvolution)
31 [V] CudaDepthwiseConvolution has no valid tactics for this config, skipping
32 [V] ----- Timing Runner: (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
    [Activation] (FusedConvActConvolution)
33 [V] FusedConvActConvolution has no valid tactics for this config, skipping
34 [V] ----- Timing Runner: (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
    [Activation] (CudnnConvolution)
35- [V] Tactic: 0 Time: 0.32768
36- [V] Tactic: 1 Time: 0.072704
37- [V] Tactic: 2 Time: 0.400384
38- [V] Tactic: 4 skipped. Scratch requested: 7426048000, available: 6442450944
39- [V] Tactic: 5 Time: 6.10406
40- [I] Some tactics do not have sufficient workspace memory to run. Increasing workspace size may
    - increase performance, please check verbose output.
```

不用cuDNN

少了cuDNN的memory占用

```
1 [V] Using cublas as a tactic source
2 [I] [MemUsageChange] Init cuBLAS/cuBLASLt: CPU +269, GPU +112, now: CPU 541, GPU 739 (MiB)
3+
4+
5 [I] Local timing cache in use. Profiling results in this builder pass will not be stored.
6+ [V] CuDNN tactics have been disabled during engine build
7 [V] Constructing optimization profile number 0 [1/1].
8
9 ...
10
11 [V] ----- Timing Runner: (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1)
    [Activation] (CudnnConvolution)
12+ [V] CudnnConvolution has no valid tactics for this config, skipping
13+
14+
15+
16+
17+
18 [V] ----- Timing Runner: (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1)
    [Activation] (CaskConvolution)
19 [V] (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1) [Activation] Set Tactic Name:
    maxwell_scudnn_128x32_relu_medium_nn_v1 Tactic: 1062367460111450758
20+ [V] Tactic: 1062367460111450758 Time: 0.01424
21
22 ...
23
24 [V] (Unnamed Layer* 0) [Convolution] + (Unnamed Layer* 1) [Activation] Set Tactic Name:
    maxwell_scudnn_128x128_relu_small_nn_v1 Tactic: -410470605513481746
25+ [V] Tactic: -410470605513481746 Time: 0.013312
26+ [V] Fastest Tactic: 7144526460361122478 Time: 0.009184
27
28 ...
29
30 [V] ----- Timing Runner: (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
    [Activation] (CudaDepthwiseConvolution)
31 [V] CudaDepthwiseConvolution has no valid tactics for this config, skipping
32 [V] ----- Timing Runner: (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
    [Activation] (FusedConvActConvolution)
33 [V] FusedConvActConvolution has no valid tactics for this config, skipping
34 [V] ----- Timing Runner: (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
    [Activation] (CudnnConvolution)
35+ [V] CudnnConvolution has no valid tactics for this config, skipping
36+
37+
38+
39+
40+
41+
```



• 高级话题：Tactic Source

所有 Tactic

不用cuDNN

```
39- [V] Tactic: 5 Time: 6.10406
40- [I] Some tactics do not have sufficient workspace memory to run. Increasing workspace size may
   - increase performance, please check verbose output.
41- [V] Fastest Tactic: 1 Time: 0.072704
42- [V] Setting workspace to 7426048000enables more tactics for profiling
43 [V] ----- Timing Runner: (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
   [Activation] (CublasConvolution)
44 ...
45 ...
46 ...
47 [V] Fastest Tactic: 6629944304117643200 Time: 0.165888
48 [V] >>>>>>>>> Chose Runner Type: CaskConvolution Tactic: 6629944304117643200
49 ...
50 ...
51 ...
52- [V] Formats and tactics selection completed in 1.29186 seconds.
53 [V] After reformat layers: 9 layers
54 ...
55 ...
56 ...
57- [I] [MemUsageStats] Peak memory usage of TRT CPU/GPU memory allocators: CPU 12 MiB, GPU 430 MiB
58 ...
59 ...
60 ...
61- [I] [MemUsageChange] Init cuBLAS/cuBLASLt: CPU +0, GPU +8, now: CPU 953, GPU 927 (MiB)
62- [V] Using cuDNN as a tactic source
63- [I] [MemUsageChange] Init cuDNN: CPU +0, GPU +8, now: CPU 953, GPU 935 (MiB)
64- [V] Engine generation completed in 1.65861 seconds.
65 ...
66 ...
67 ...
68- [I] [MemUsageChange] TensorRT-managed allocation in building engine: CPU +0, GPU +13, now: CPU
   0, GPU 13 (MiB)
69- [I] [MemUsageChange] Init CUDA: CPU +0, GPU +0, now: CPU 965, GPU 887 (MiB)
70 [I] Loaded engine size: 12 MiB
71 [V] Using cublas as a tactic source
72- [I] [MemUsageChange] Init cuBLAS/cuBLASLt: CPU +1, GPU +10, now: CPU 966, GPU 911 (MiB)
73- [V] Using cuDNN as a tactic source
74- [I] [MemUsageChange] Init cuDNN: CPU +0, GPU +8, now: CPU 966, GPU 919 (MiB)
75- [V] Deserialization required 4159 microseconds.
76- [I] [MemUsageChange] TensorRT-managed allocation in engine deserialization: CPU +0, GPU +12,
   now: CPU 0, GPU 12 (MiB)
77 [V] Using cublas as a tactic source
78- [I] [MemUsageChange] Init cuBLAS/cuBLASLt: CPU +1, GPU +10, now: CPU 966, GPU 911 (MiB)
79- [V] Using cuDNN as a tactic source
80- [I] [MemUsageChange] Init cuDNN: CPU +0, GPU +8, now: CPU 966, GPU 919 (MiB)
81 ...
82 ...
39+
40+
41+
42+
43+
44 [V] ----- Timing Runner: (Unnamed Layer* 7) [Fully Connected] + (Unnamed Layer* 8)
   [Activation] (CublasConvolution)
45 ...
46 ...
47 ...
48 [V] Fastest Tactic: 6629944304117643200 Time: 0.165888
49 [V] >>>>>>>>> Chose Runner Type: CaskConvolution Tactic: 6629944304117643200
50 ...
51 ...
52 ...
53+ [V] Formats and tactics selection completed in 1.03523 seconds.
54 [V] After reformat layers: 9 layers
55 ...
56 ...
57 ...
58+ [I] [MemUsageStats] Peak memory usage of TRT CPU/GPU memory allocators: CPU 12 MiB, GPU 25 MiB
59 ...
60 ...
61 ...
62+ [I] [MemUsageChange] Init cuBLAS/cuBLASLt: CPU +0, GPU +8, now: CPU 542, GPU 761 (MiB)
63+
64+
65+ [V] Engine generation completed in 1.28971 seconds.
66 ...
67 ...
68 ...
69 [I] [MemUsageChange] TensorRT-managed allocation in building engine: CPU +0, GPU +13, now: CPU
   0, GPU 13 (MiB)
70+ [I] [MemUsageChange] Init CUDA: CPU +0, GPU +0, now: CPU 554, GPU 729 (MiB)
71 [I] Loaded engine size: 12 MiB
72 [V] Using cublas as a tactic source
73+ [I] [MemUsageChange] Init cuBLAS/cuBLASLt: CPU +1, GPU +10, now: CPU 555, GPU 753 (MiB)
74+
75+
76+ [V] Deserialization required 2721 microseconds.
77 [I] [MemUsageChange] TensorRT-managed allocation in engine deserialization: CPU +0, GPU +12,
   now: CPU 0, GPU 12 (MiB)
78 [V] Using cublas as a tactic source
79+ [I] [MemUsageChange] Init cuBLAS/cuBLASLt: CPU +0, GPU +10, now: CPU 555, GPU 753 (MiB)
80+
81+
82 ...
83 ...
```

优化耗时减少

构建期 memory 需求减少

运行期 memory 需求减少



• 高级话题

➤ 希望解决的问题

- Dynamic Shape 模式在 min-opt-max 跨度较大时性能下降?
- 怎样重叠计算和数据拷贝的时间，增加GPU利用率?
- 怎样使一个 engine 供多个线程使用?
- 怎么样优化 Kernel 的调用，减少 Launch Bound 的发生?
- engine 构建时间太长，怎么节约多次构建时的时间?
- 某些 Layer 的算法导致较大的误差，能不能精确选择算法?
- 想更新模型的权重，但又不想重新构建 engine?
- 构建期/运行期显存占用过大，怎么减少?
- 能不能跨硬件或 TensorRT 版本运行 engine?



• 高级话题：Hardware compatibility 和 Version compatibility

➤ TensorRT 8.6 新功能

- 硬件兼容只支持 Ampere 及更新架构的 GPU ($sm \geq 8.0$)，版本兼容要求 TensorRT 版本 ≥ 8.6

➤ 范例代码

- 08-Advance/Hardware compatibility 和 08-Advance/ Version compatibility

➤ 硬件兼容

- `config.hardware_compatibility_level = trt.HardwareCompatibilityLevel.AMPERE_PLUS`
- 前向后向均可 (A100 构建 A10 运行✓, A10 构建 A100 运行✓)
- 可能会有少许性能损失

➤ 版本兼容

- `config.set_flag(trt.BuilderFlag.VERSION_COMPATIBLE)`
- `runtime.engine_host_code_allowed = True`
- 前向后向均可 (TRT8.6 构建 TRT 8.7 运行✓, TRT8.6 构建 TRT 8.7 运行✓)
- 可能会有少许性能损失



• 高级话题：更多功能

- Profiling
- Profiling Verbosity
- Error Recorder
- GPU Allocator
- Engine Inspector
- Empty Tensor
- ...
- 范例代码均在 cookbook 的 08-Advance 中，以后逐步更新完善.....



Part 5

Something here

