

# TensorRT Hackathon 2022

## 初赛总结

NVIDIA DevTech. Wei Li

2022年5月20日





# 目录

- 比赛总体介绍
  - 初赛成果
  - 复赛安排
- 初赛中用到的优化技术



# • 比赛总体介绍

## ➤ 比赛目标

- 借助比赛形式，提高同学们开发 TensorRT 应用的能力，积累开发经验
- 选拔部分同学与英伟达专家进行沟通交流
- 扩大 TensorRT 在业界的应用规模，积累同学们的作品和经验，引导更多同学用好 TensorRT

## ➤ 比赛时间表

- 报名开放：2022年4月2日
- 初赛（选手选拔）阶段：2022年4月2日-5月20日
- 复赛（导师面对面）阶段：2022年5月23日-6月27日
- 名次公布：2022年7月6日



# 初赛情况

➤ 时间：5 月 23 日至 6 月 27 日

## ➤ 初赛现场

➤ <https://tianchi.aliyun.com/competition/entrance/531953/rankingList>

## ➤ 比赛方式

- 使用统一的 docker image、.onnx 模型、性能测试脚本
- 选手本地撰写代码，提交至指定仓库
- 在线服务器运行选手代码，构建 .plan 并测试
- 根据结果精度和速度计算得分，并以此进行排名
- 排名前 40 名的队伍进入复赛

## ➤ 赛题

- ASR 模型 Wenet（含 encoder 和 decoder 两部分）
- 要求从 .onnx 转为 .plan 并实现性能优化

阿里云 | TIANCHI 天池

首页

天池学习

天池大赛

数据集

天池实验室

在线编程

技术圈

其他

消息

willinvidia

LVT

退出

语言

首页>天池大赛>英伟达TensorRT加速AI推理 Hackathon 2022 —— Transformer模型优化赛

	状态	举办方	赛季1	奖金	参赛队伍
英伟达TensorRT加速AI推理 Hackathon 20...	已结束	Alibaba Cloud TIANCHI 天池 NVIDIA	2022-05-20	¥ 49000	666

赛制

赛题与数据

排行榜

论坛

提交结果

我的成绩

我的团队

排名	参与者	组织	score	最优成绩提交日
1	ching	?	10096.08	2022-05-19
2	lamei	oppo	3798.56	2022-05-20
3	摇阿摇	华南理工大学	3634.71	2022-05-19
4	Good Luck To You!	西电、AVIC	3403.89	2022-05-19
5	willinvidia	?	2167.05	2022-05-02
6	错误代码114	西安电子科技大学	2164.04	2022-05-19
7	云上浪	2	2034.84	2022-05-16
8	Q_RT	--	2032.21	2022-05-17
9	TRTRush	TX	2017.34	2022-05-08
10	杭师大的饭真好吃	江南大学	1999.37	2022-05-17
11	摸鱼小组	University of Illinois at Urb...	1984.40	2022-05-01
12	摸鱼小分队	上海交通大学	1944.17	2022-05-11




# • 比赛成果

- 截至 5 月 20 日,
  - **666** 支队伍报名
  - **85** 支队伍完成模型转换和性能调优
  - **65** 支队伍进行手工计算图优化并尝试 FP16 / INT8 模式
  - **1002** 次代码提交和测评 (56% 机时)
  - 前 **40** 名选手基本完成 Layer Normalization Plugin 的撰写
  - 至少 **4** 支队伍实现了 Attention Plugin 并超过了TensorRT 默认实现的性能
  - 第 **1** 名选手使用 Faster Transformer 优化模型并取得极优性能




- 比赛成果
  - 同学们知识共享和趟坑记录
  - 有经验开发者间的经验交流





@所有人 <https://note.youdao.com/s/TJo0XXJk> 这是我今天写的关于encoder图优化的笔记，以及后续优化方向(附代码)。大家有兴趣可以看看。先声明不一定对，就是想和大家一起学习trt。


有道云笔记


<https://note.youdao.com/s/TJo0XXJk>




 test01, yj, 美迪康AI Lab-徐静, 李争尔, TripleMu, 田海滨

 志远-银河飞车, 如何变胖\_马特兰博, wili



 收到

回复

 9条回复

亚鹏

啊对对对-御坂美琴

牛批

尧哲

将个烂就 宋尧哲

膜拜!

海滨

请问合并qkv计算得是不是合matmul然后split呀?

 1条回复

 ching

是的，不能保证，但nv没开源的qkv速度是真的快，之前在swin3d上测的比调cublas快了快一倍了

 2条回复

ly

爱笔-lyon

lyon

laynorm的话是不是可以变成  $n*(c*h*w)*1*1$

写错了，应该是 $1*n*(c*h*w)*1?$

 TensorRT\_Tutorial

ching

是的，不能保证，但nv没开源的qkv速度是真的快，之前在swin3d上测的比调cublas快了快一倍了

@ching

那就没办法了，你看他的文件名，是根据固定输入大小来做特定优化的。并不是支持所有维度的。这么做肯定会快很多。



# • 复赛说明

➤ 时间：5 月 23 日至 6 月 27 日

➤ 比赛介绍 <https://github.com/NVIDIA/trt-samples-for-hackathon-cn/blob/master/hackathon/TRT-Hackathon-2022-final.md>

➤ 内容介绍

- 开放赛题，各选手可**自由选择公开的 transformer 模型**，在 TensorRT 上优化运行，**选手最终代码也必须开源**
- 提交代码：从原始模型出发直到运行 TensorRT 模型全过程所有脚本及代码，发布到公共代码托管平台（建议用 github）
- 讨论学习：每个小组将分配到**至少两名英伟达专家**一同参与模型优化和测试工作
- 最终报告：用固定的模板、以 markdown 的形式发布在代码仓库根目录的 README.md 里（报告模板在上述链接中）
- 专家评审：综合模型价值、技术难度、优化效果、代码可读性、repo 学习价值等方面打分，进行排名



# • 比赛复盘

## ➤ 用到的技术手段：

- 使用 onnx-graphsurgeon 调整节点，解决 .onnx 不能被 TensorRT 解析的问题
- 使用一些现成的计算图优化库（Polygraphy, onnx-optimizer, onnx-onnx simplifier)
- 优化计算图，方便 TensorRT 融合或选择效率更高的 kernel
- 优化计算图，将部分运行期计算提前到构建期完成
- 使用 FP16 和 INT8 模式，并使用 strict type 显式控制某些 layer 计算精度以控制误差
- Layer Normalization Plugin
- Attention Plugin 以提高性能（超越 Myelin 自动融合效果)
- Mask Plugin + SkipSigmoid Plugin
- 尺寸对齐
- 名尺寸和精度问题
- 彩蛋部分





# • 比赛复盘

## ➤ 使用 onnx-graphsurgeon 修改 .onnx 计算图的一般方法:

➤ 读取 .onnx 转为 ogs 的 graph 对象 (Line12-13)

➤ 遍历计算图, 找到要修改的节点 (Line17)

➤ 调整计算图 (Line19)

➤ 计算图清理 (Line22)

➤ 导出编辑完成的计算图 (Line23)

## ➤ 一些建议

➤ 分趟 (Pass) 遍历完成不同类型的节点调整, 防止相互干扰

➤ 使用开关 (Line10,16) 以便导出不同优化的模型, 检查精度和对比性能

```
1  from collections import OrderedDict
2  from copy import deepcopy
3  import numpy as np
4  import onnx
5  import onnx_graphsurgeon as gs
6
7  sourceOnnx = "./V1.onnx"
8  destinationOnnx = "./V2.onnx"
9
10 bNot = True
11
12 model = onnx.load(sourceOnnx)
13 graph = gs.import_onnx(model)
14 #graph = gs.import_onnx(onnx.shape_inference.infer_shapes(model))
15
16 if bNot:
17     for node in graph.nodes:
18         if node.name == 'Not_30':
19             # do something
20             continue
21
22 graph.cleanup()
23 onnx.save(gs.export_onnx(graph), destinationOnnx)
24
25 print("%s -> %s"%(sourceOnnx,destinationOnnx))
```



# • 优化技术

## ➤ 调整节点，解决 .onnx 不能被 TensorRT 解析的问题

### ➤ 直接用 TensorRT 见解析模型遇到报错：

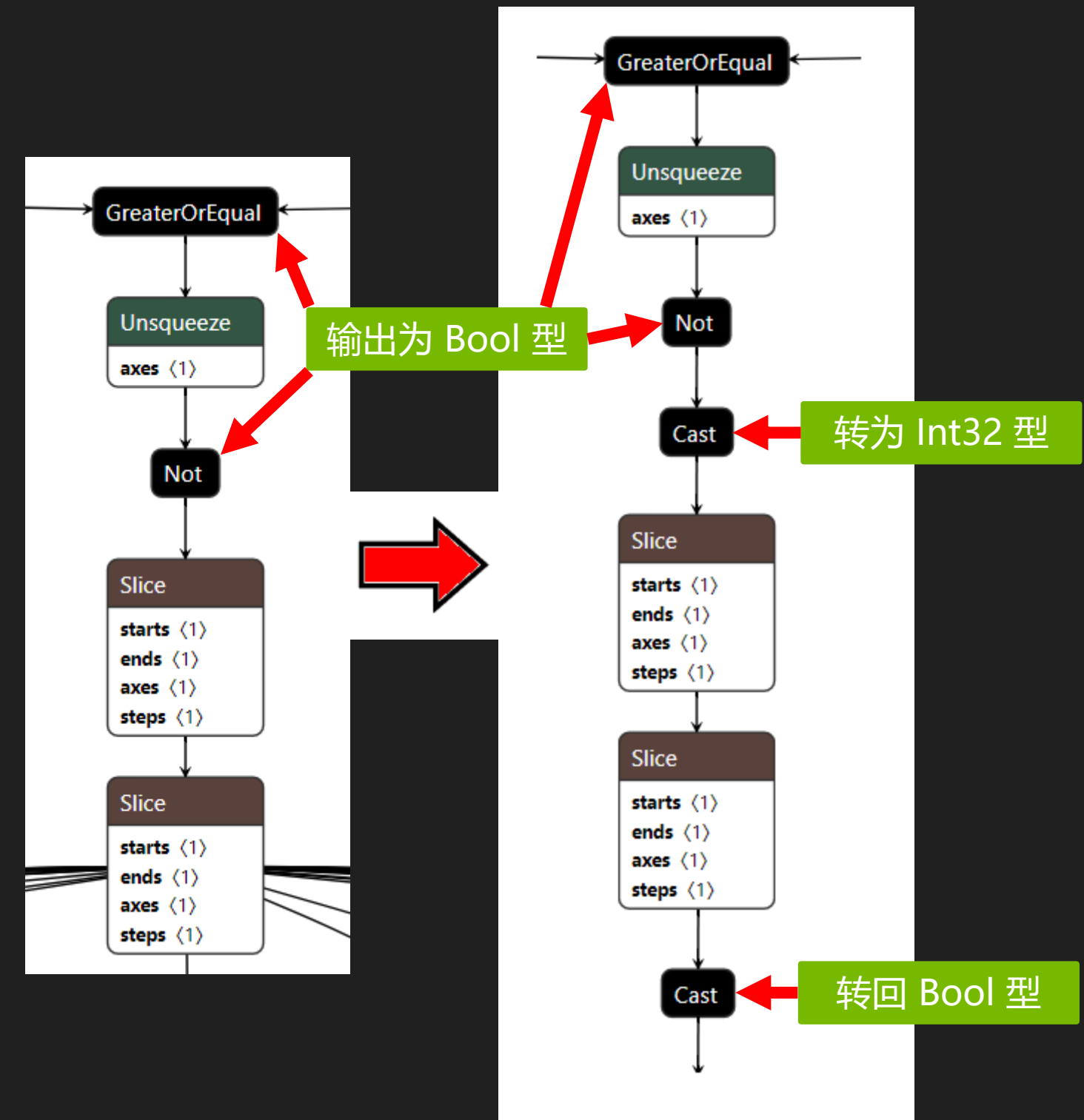
- [E] 2: [myelinBuilderUtils.cpp::operator()::293] Error Code 2: Internal Error  
(Slice\_79requires bool I/O but node can not be handled by Myelin.)

### ➤ 原因：TensorRT 的 Slice 层不支持 Bool 型输入

### ➤ TensorRT 支持列表 (*NVIDIA TensorRT Support Matrix*) :

<https://docs.nvidia.com/deeplearning/tensorrt/support-matrix/index.html#layers-precision-matrix>

### ➤ 解决办法：在 Slice 前后插入 cast 节点，将张量转为 int32 类型



# • 优化技术

## ➤ 调整节点，解决 .onnx 不能被 TensorRT 解析的问题

### ➤ 核心代码

### ➤ 要点：

- 插入张量 (Tensor) 和节点 (Node) 相互配合
- 记得向计算图中塞入新增的节点 (Line9,18)
- 张量的数据类型和形状不是必需的 (Line3,12)
- 添加属性 (attrs) 需要有序字典
  - `from collections import OrderedDict`
- 获取 Onnx 数据类型枚举值 (右下小图)

```
1  for node in graph.nodes:
2      if node.name == 'Not_30':
3          castV = gs.Variable("CastV1", np.dtype(bool), None)
4          castN = gs.Node("Cast",
5                          "CastN1",
6                          inputs=[node.i().outputs[0]],
7                          outputs=[castV],
8                          attrs=OrderedDict([('to', onnx.TensorProto.BOOL)]))
9          graph.nodes.append(castN)
10         node.inputs = [castV]
11
12         castV = gs.Variable("CastV2", np.dtype(np.int32), None)
13         castN = gs.Node("Cast",
14                         "CastN2",
15                         inputs=node.outputs,
16                         outputs=[castV],
17                         attrs=OrderedDict([('to', onnx.TensorProto.INT32)]))
18         graph.nodes.append(castN)
```

```
>>> print(onnx.TensorProto.BOOL)
9
>>> print(onnx.TensorProto.INT32)
6
>>> print(onnx.TensorProto.INT64)
7
>>> print(onnx.TensorProto.FLOAT)
1
>>> print(onnx.TensorProto.FLOAT16)
10
```





# • 优化技术

## ➤ 使用计算图优化库 (Polygraphy, onnx-optimizer, onnx-simplifier)

### ➤ 使用 polygraphy:

➤ `polygraphy surgeon sanitize --fold-constant /workspace/encoder.onnx -o /target/encoderV2.onnx`

### ➤ 使用 onnx-optimizer (脚本)

### ➤ 使用 onnxsim (命令或脚本)

### ➤ 作用: 计算图化简, 常量折叠, 部分算子消除

```
1  import onnx
2  import onnxoptimizer
3
4  sourceOnnx = "/workspace/encoder.onnx"
5  destinationOnnx = "./encoderOpt.onnx"
6
7  model0 = onnx.load(sourceOnnx)
8  model1 = onnxoptimizer.optimize(model0)
9  onnx.save(model1, destinationOnnx)
```

```
1  import onnx
2  import onnxsim
3
4  sourceOnnx = "/workspace/encoder.onnx"
5  destinationOnnx = "./encoderOpt.onnx"
6
7  shapeDict = {"speech": [16, 256, 80], "speech_lengths": [16,]}
8  model1, _ = onnxsim.simplify(sourceOnnx,
9                               dynamic_input_shape=True,
10                              input_shapes=shapeDict)
11  onnx.save(model1, destinationOnnx)
```



# • 优化技术

## ➤ 效果对比

- 对 encoder 部分使用三种工具进行加工
  - 用 ogs 统计计算图节点数和张量数，并构建 .plan 来评测性能
  - V0: 原始 encoder 模型文件
  - V1: 手工优化过的模型文件
  - Poly: 使用 polygraphy 进行加工
  - Opt: 使用 onnx-optimizer 进行加工
  - Sim: 使用 onnxsim 进行加工
- 结论：对于我们模型的 encoder 部分，
  - onnx-optimizer 基本没有效果；polygraphy 有一定化简效果；onnxsim 化简效果最明显，但节点改动比较大，后续手工优化时要注意调整
  - 对 TensorRT 来说，三种工具做的图优化没有显著改善最终性能 (<2%)
  - 若模型部署到 onnxruntime 等后端可能有一定性能改善 (这里没有测试)
  - 使用顺序建议：原模型 → polygraphy 加工 → ogs 手工优化 → TensorRT 构建

		V0	V1
节点数	优化前	1990	975
	Poly	1622	975
	Opt	1990	975
	Sim	1582	1034
张量数	优化前	2610	1716
	Poly	2600	1716
	Opt	2610	1716
	Sim	2051	1502
性能 word/s	优化前	1.444E+05	\
	Poly	1.428E+05	
	Opt	1.442E+05	
	Sim	1.430E+05	



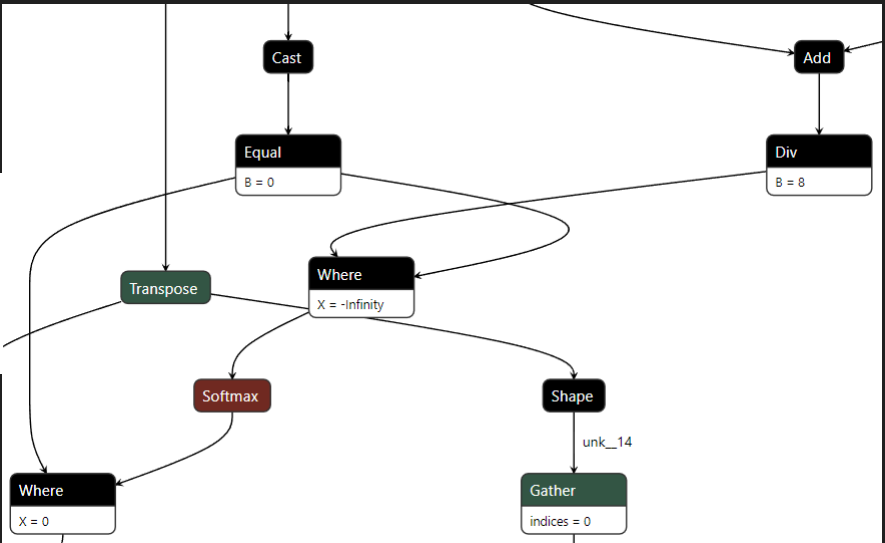
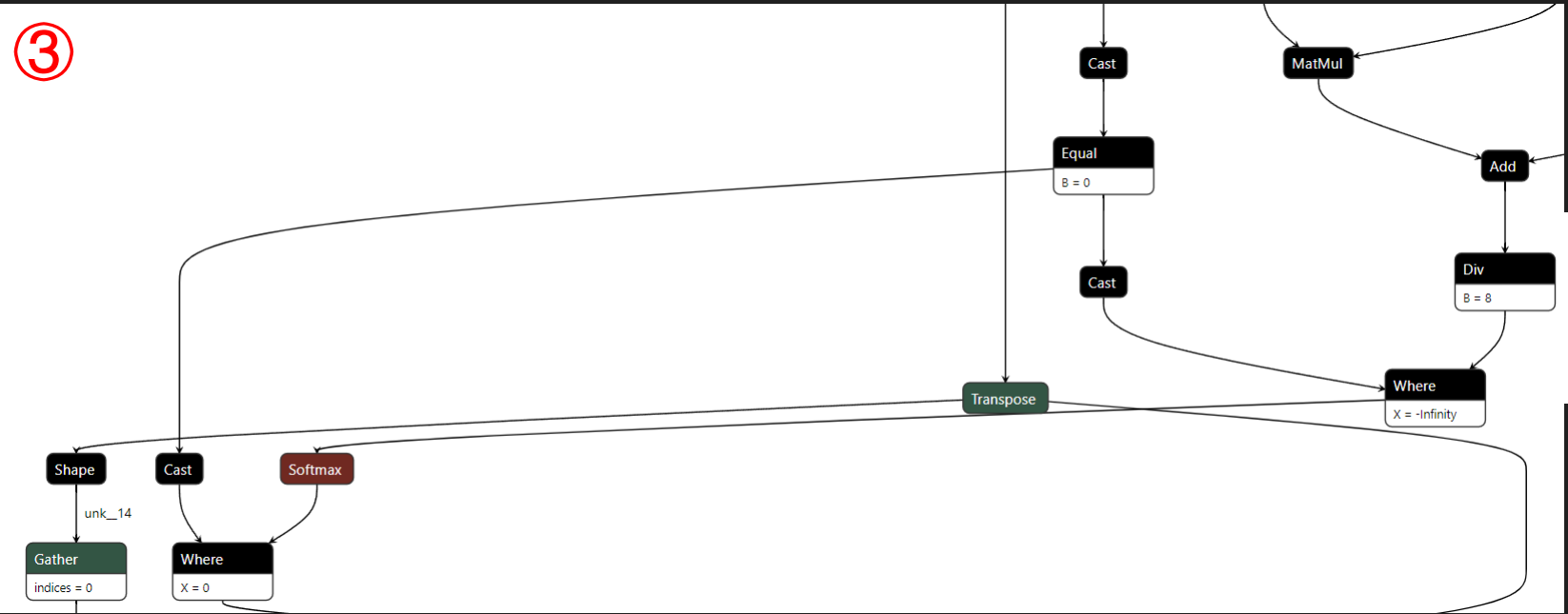
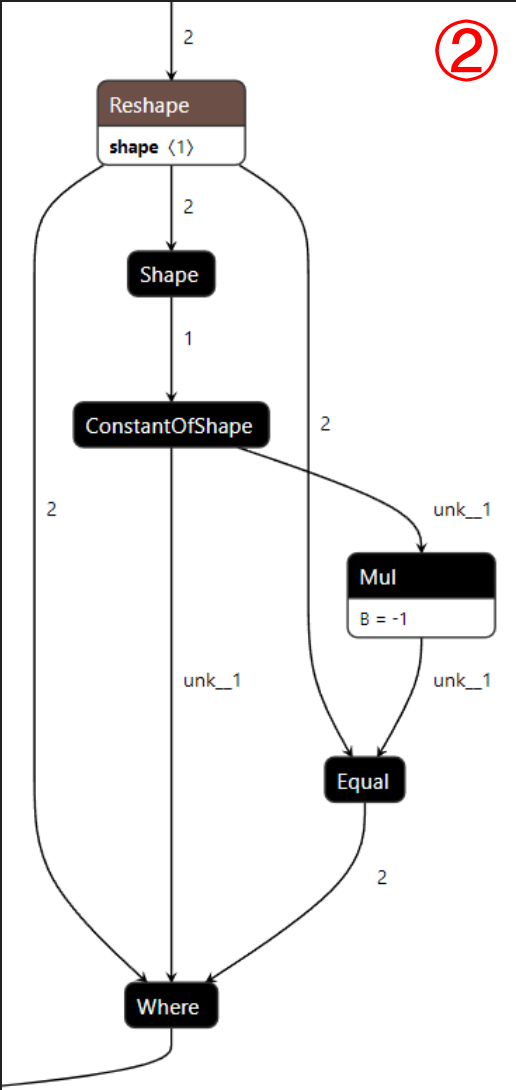
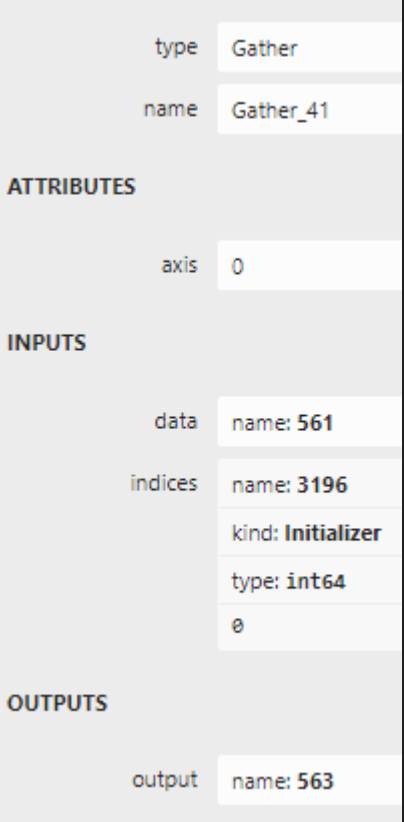
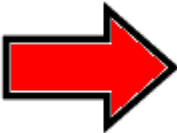
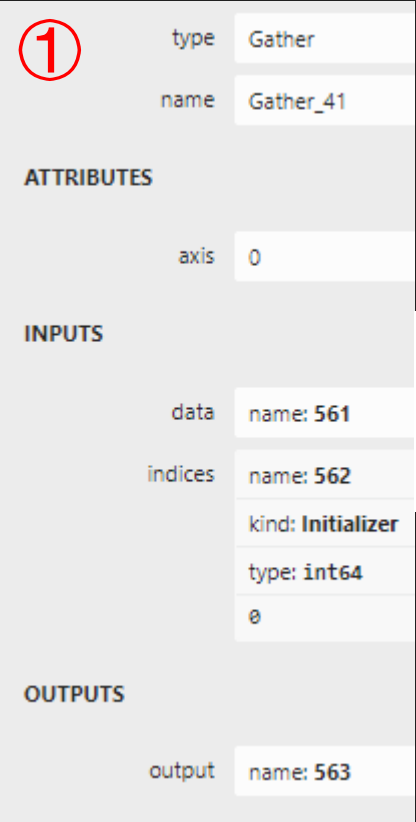
# • 优化技术

## ➤ 效果对比

➤ 仔细对比 polygraphy 和 onnxsim, onnxsim 多做的工作:

- 常量复用 ①, 包括使用使用 ConstantOfShape 节点 ②
- 删除 Cast (直接在父节点输出张量里指定数据类型, ③)

➤ TensorRT 优化器自带 shape 推理器, 构建期同样可以完成上述工作





# • 优化技术

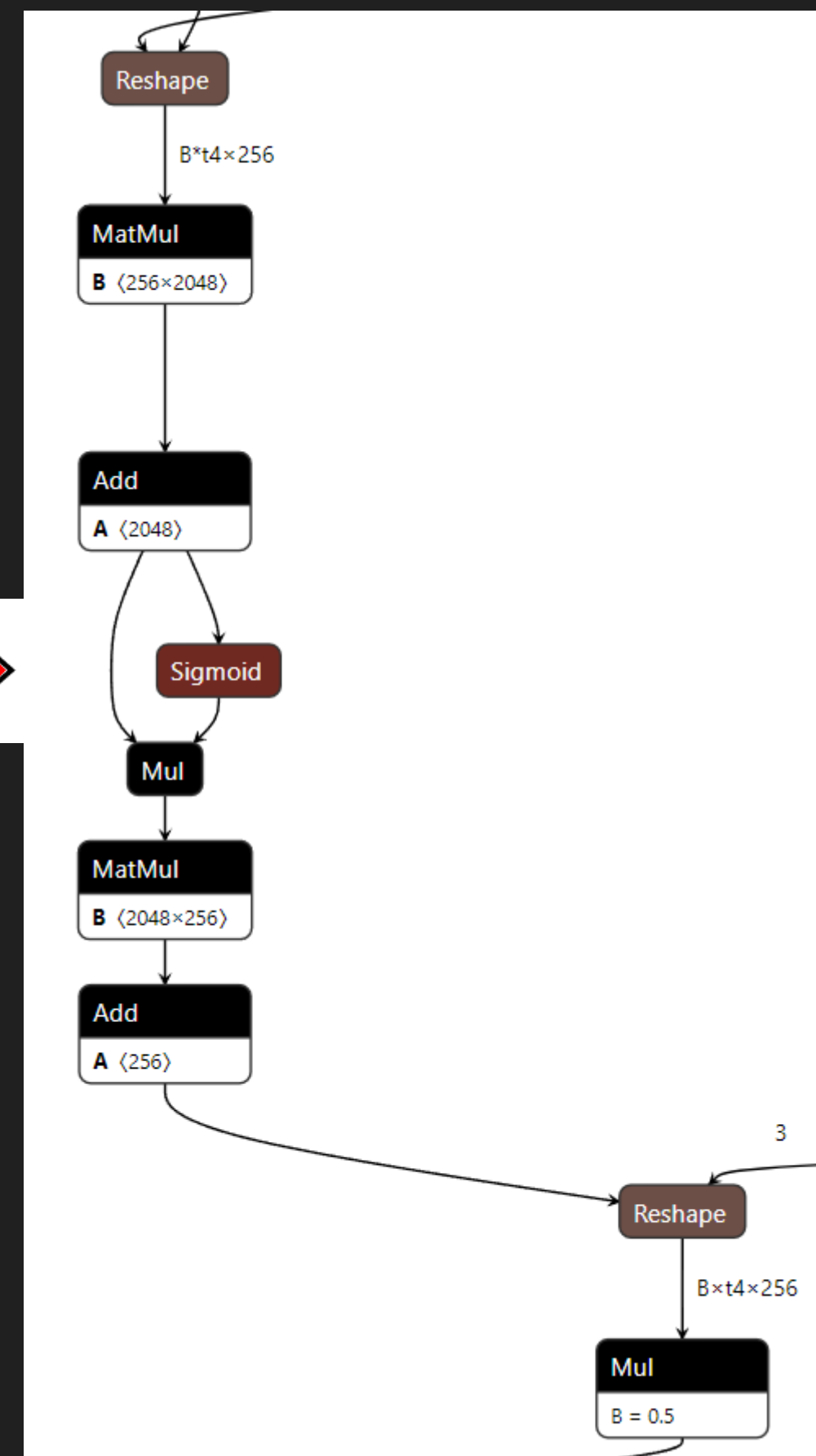
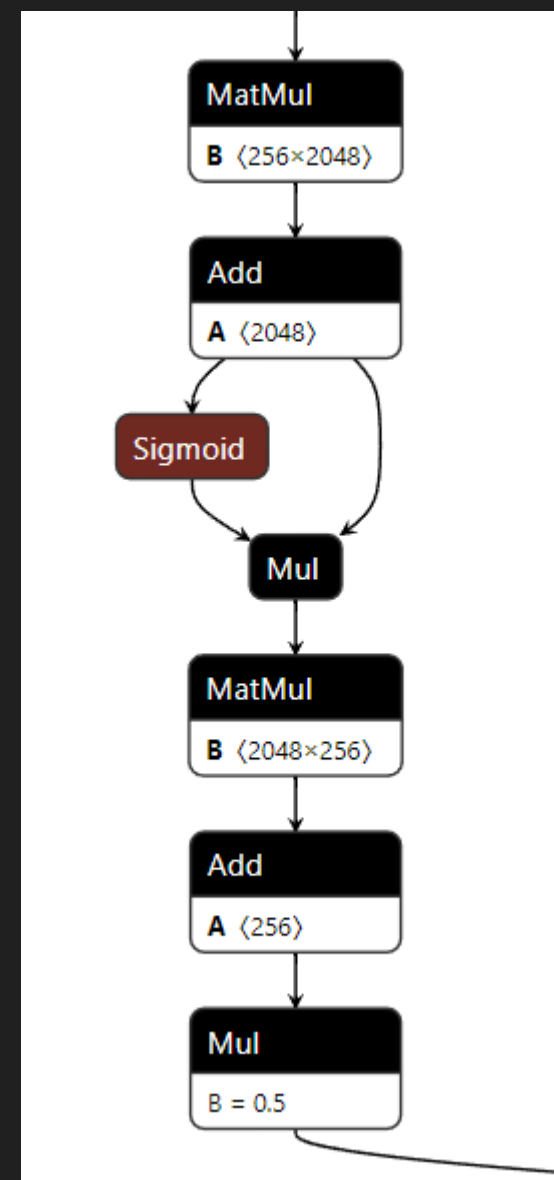
➤ 优化计算图，方便 TensorRT 融合或选择效率更高的 kernel

➤ 赛题中的例子：Layer Normalization 后的线性变换

➤ 原模型此处输入、输出形状均为  $[B, t4, 256]$

➤ 修改后将输入 Reshape 为  $[B * t4, 256]$ ，最后 Reshape 回  $[B, t4, 256]$

➤ 相当于 3D 矩阵乘法变为 2D 矩阵乘法



# • 优化技术

## ➤ 一个例子来说明 3DMM 和 2DMM 的性能差别

➤ [cookbook/10-BestPractice/Convert3DMMTo2DMM](#)

➤ 网络反复使用 MatMul+Add+ReLU 分别使用 3DMM 和 2DMM 计算

### ➤ 3DMM 最终网络:

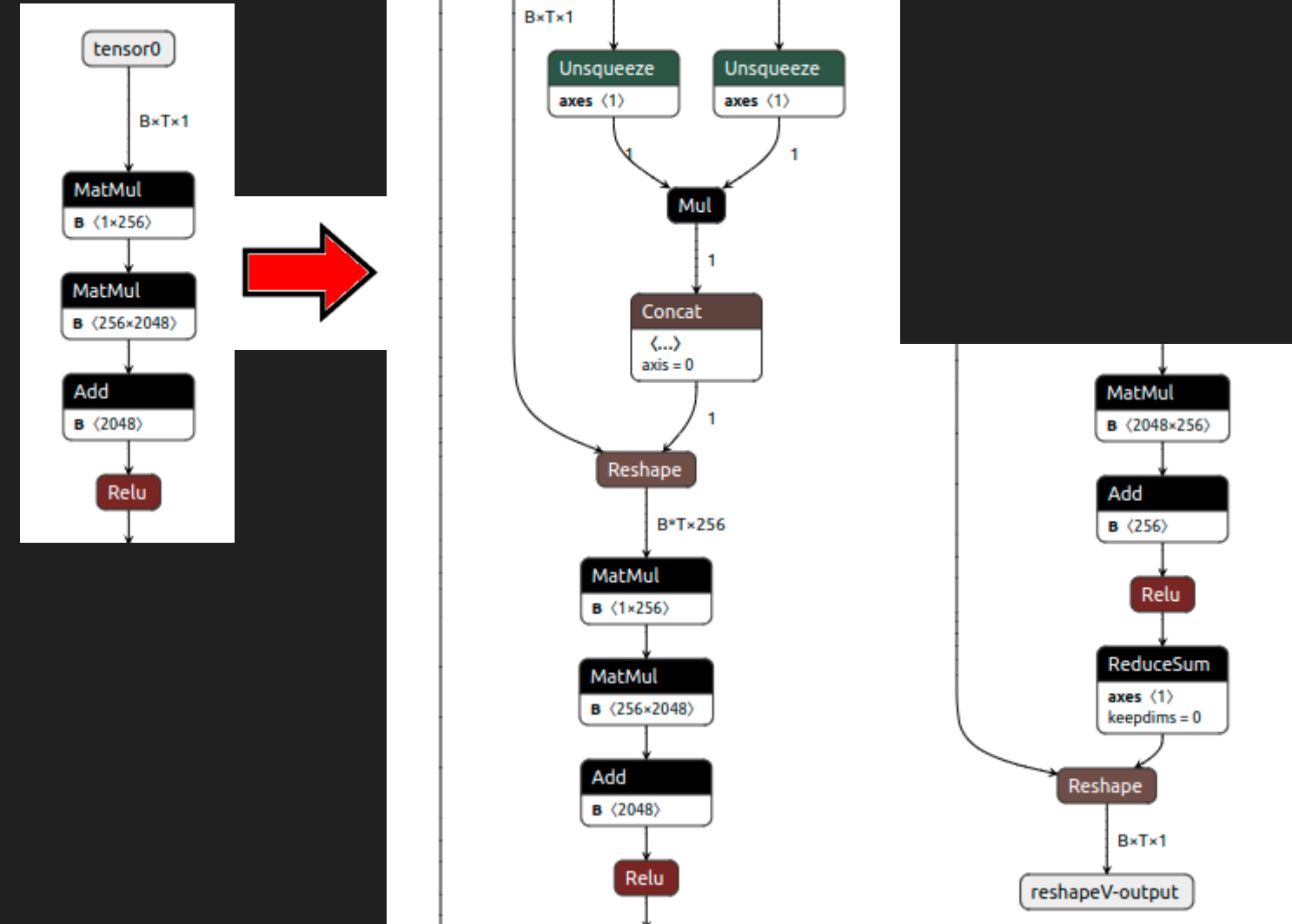
- Layer(Constant): ...
- Layer(MatrixMultiply): ...
- Layer(Constant): ...
- Layer(MatrixMultiply): ...
- ...

### ➤ 2DMM 最终网络:

- Layer(CaskConvolution): ...
- Layer(CaskConvolution): ...
- ...

### ➤ 性能比较

- 3DMM: 6.43479 ms
- 2DMM: 4.55748 ms



# • 优化技术

## ➤ 另一个例子（赛题中没有出现，但是比较常见）

### ➤ 10-BestPractice/EliminateSqueezeUnsqueezeTranspose

### ➤ 某些 Squeeze / Unsqueeze / Transpose 节点会影响融合

### ➤ 带 Squeeze/Unsqueeze 或带 Transpose 最终网络：

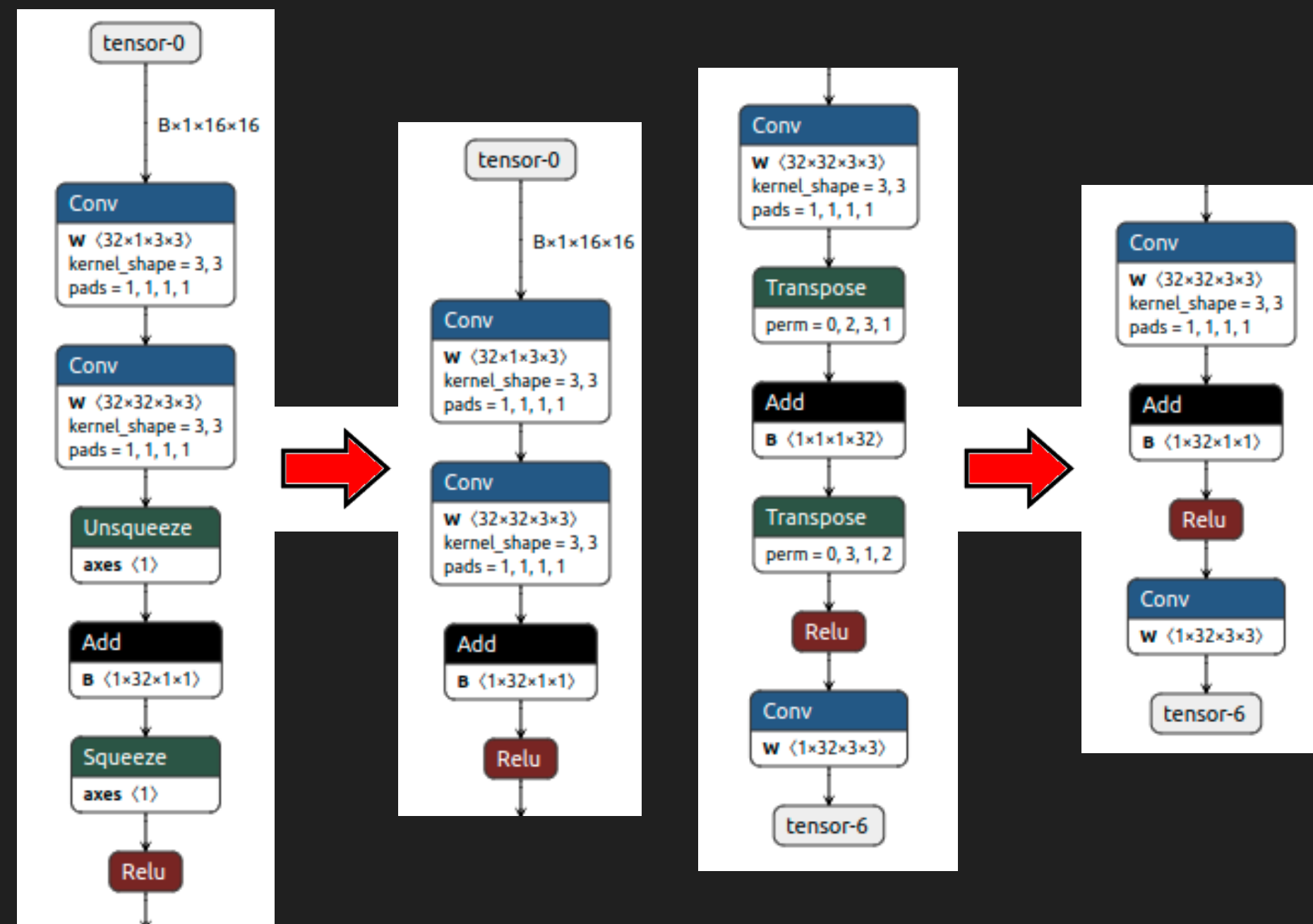
- Layer(FusedConvActConvolution): ...
- Layer(NoOp): ...
- Layer(Scale): ...
- Layer(NoOp): ...
- Layer(PointWiseV2): ...
- ...
- Layer(FusedConvActConvolution): ...
- Layer(Shuffle): ...
- Layer(Constant): ...
- Layer(ElementWise): ...
- Layer(Shuffle): ...
- Layer(PointWiseV2): ...
- ...

### ➤ 消除 SUT 后最终网络：

- Layer(FusedConvActConvolution): ...

### ➤ 性能比较

- 带 SUT 网络：0.244098 ms，消除SUT 的网络：0.109195 ms





# • 优化技术

## ➤ 优化计算图，将部分运行期计算提前到构建期完成

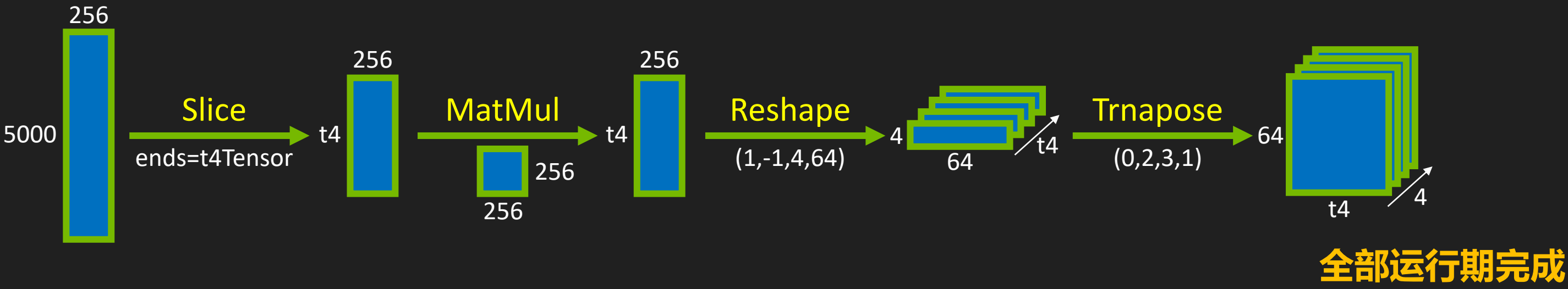
- 赛题中的例子：encoder 的 position embedding 部分
- 原模型此处先切片后计算矩阵乘，最后转置
- 修改后构建期算好矩阵乘和转置，运行期只做切片
- 减少 12 个  $m=5000, k=n=256$  的矩阵乘法
- 减少 12 个后续的 Transpose



# • 优化技术

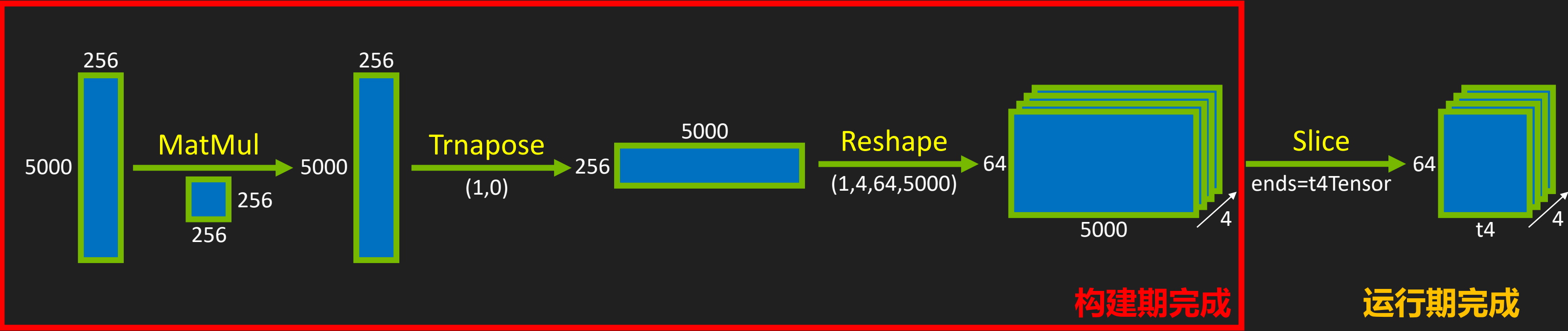
## ➤ 提前计算

### ➤ 原计算图



这里  $t4 = \text{Floor}(T/4) - 1$ ，为 Sequence Length 下采样后的长度（后面说）

### ➤ 优化后



# • 优化技术

## ➤ 提前计算

### ➤ 核心代码

```
1  for node in graph.nodes:
2      if node.op == 'Slice' and node.name == 'Slice_74':
3          node.inputs[2] = t4Tensor
4
5          table5000x256 = node.inputs[0].values[0]
6      for i in range(1, 24, 2):
7          trashNode = node.o(i).o().o() # Transpose Node
8          factor256x256 = node.o(i).inputs[1].values
9
10         newTable = np.matmul(table5000x256, factor256x256)
11         newTable = newTable.transpose().reshape(1, 4, 64, 5000)
12         constantData = gs.Constant("Data", np.ascontiguousarray(newTable))
13         sliceV = gs.Variable("sliceData", np.dtype(np.float32), [1, 4, 64, 't4'])
14         sliceN = gs.Node(
15             "Slice",
16             "SliceN",
17             inputs=[
18                 constantData, # data
19                 wiliConstant0, # start=0
20                 t4Tensor, # end
21                 wiliConstant3, # axes=3
22                 wiliConstant1, # step=1
23             ],
24             outputs=[sliceV]
25         )
26         graph.nodes.append(sliceN)
27         node.o(i).o().o().o().inputs[1] = sliceV
28         trashNode.outputs = []
```

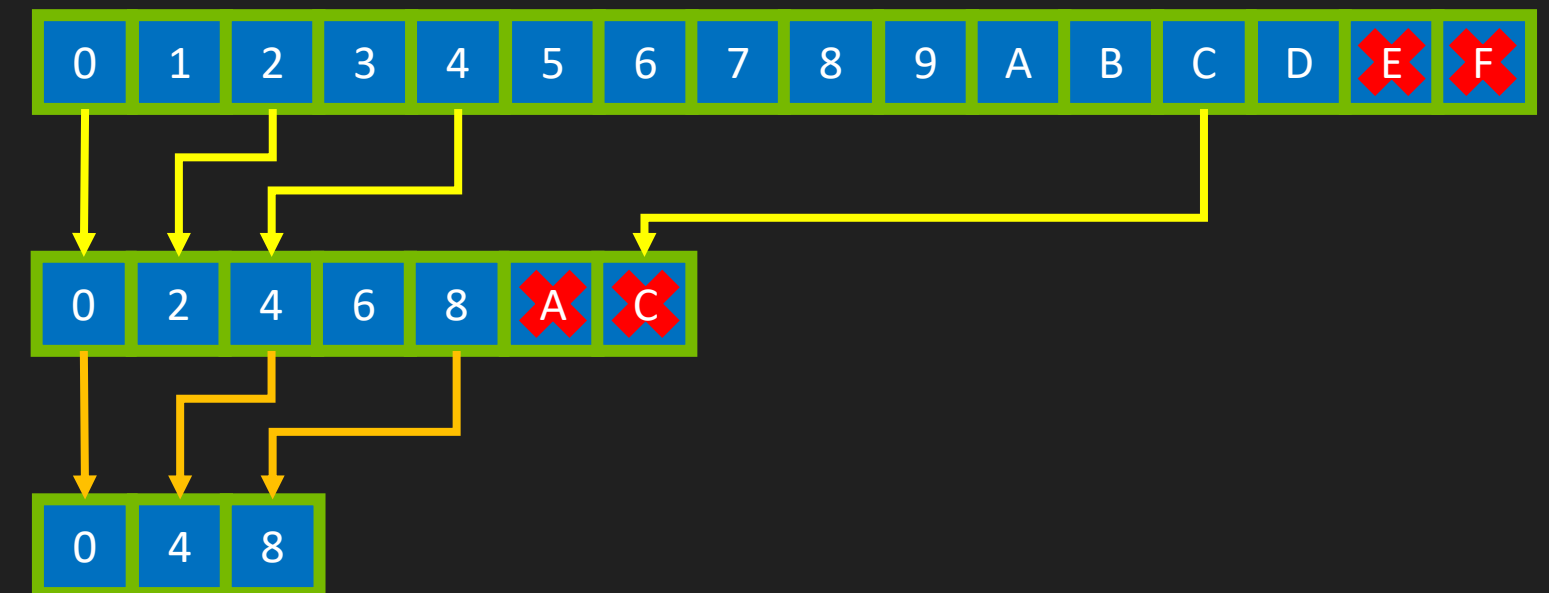
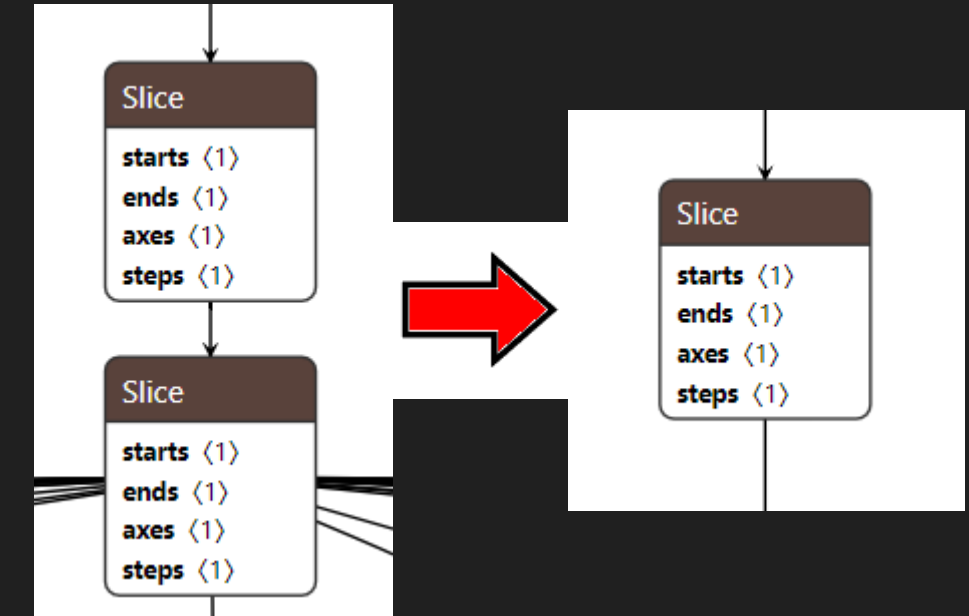




# • 优化技术

## ➤ 手工合并算子

- 赛题中的例子：encoder 的 Slice\_79 + Slice\_84
- 他们跟 encoder 的两次卷积下采样有关
- Slice\_79 与 Slice\_84 参数：starts=0, ends=-2, steps=2
- 合并后 Slice 参数：starts=0, ends=-4, steps=4



# • 优化技术

## ➤ 手工合并算子

➤ 手工算子变化（性能以测试为准）

➤ 感谢**宏伟**同学提供思路

➤ 原计算图为 ShapeOperation + MatMul + Add

➤ 修改后 Conv + Squeeze + Transpose

➤ 优化前最终网络：

- Layer(Shuffle): ...
- Layer(Constant): ...
- Layer(MatrixMultiply): ...
- Layer(Constant): ...
- Layer(ElementWise): ...

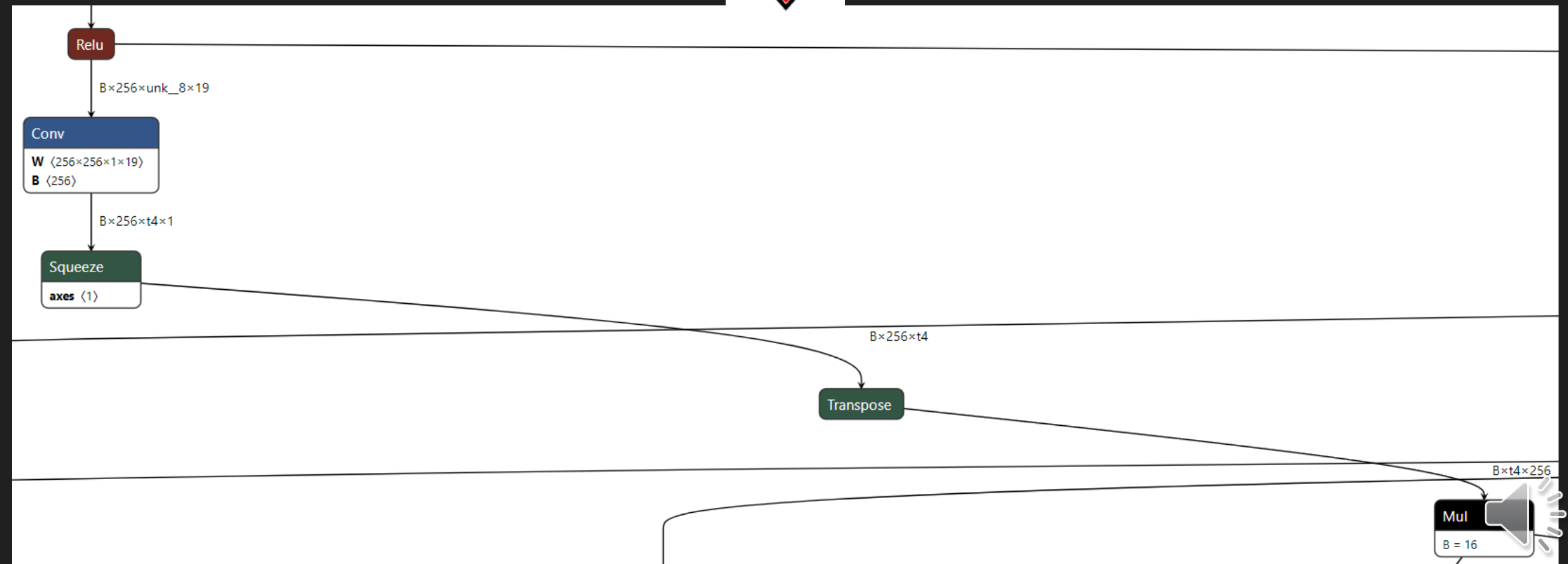
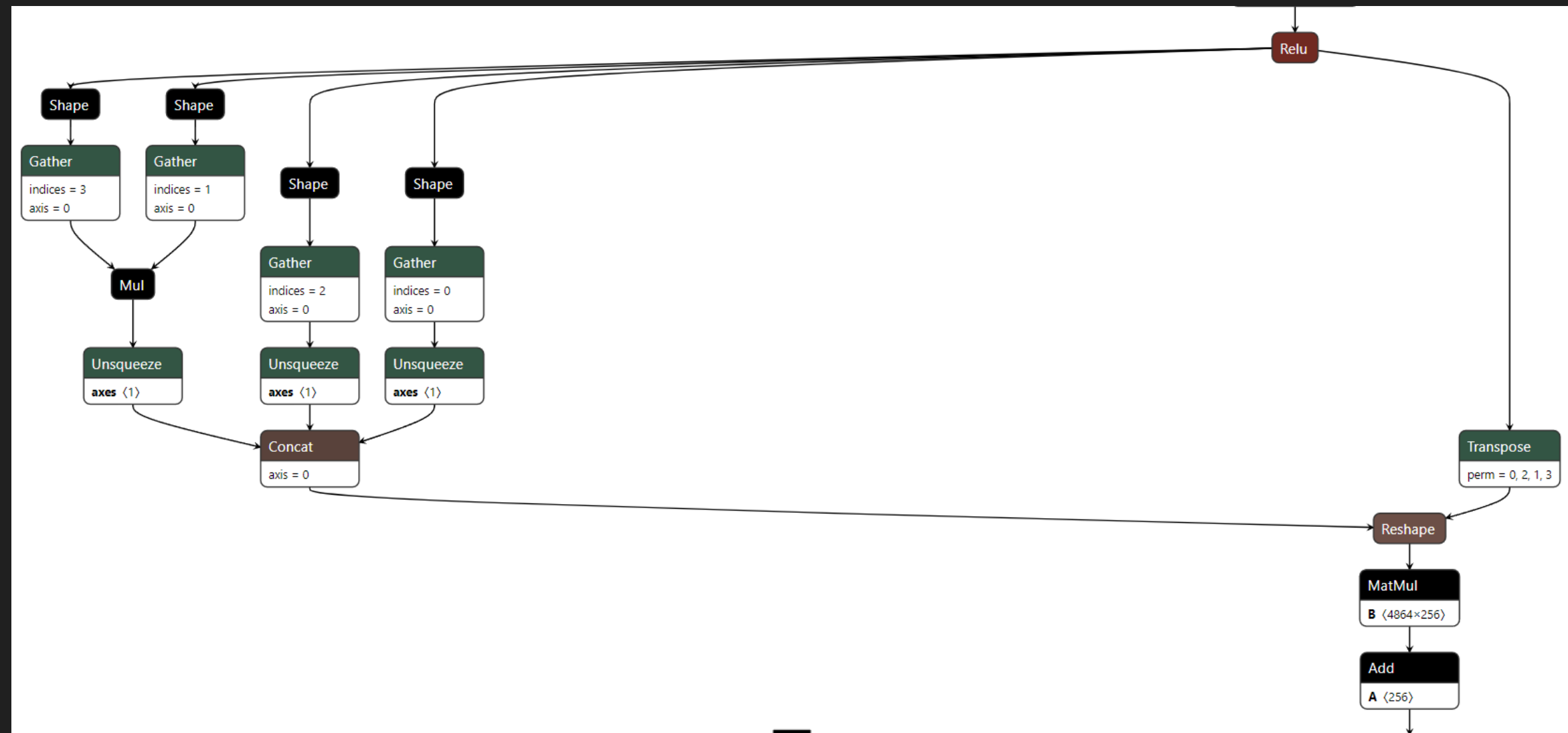
➤ 优化后最终网络

- Layer(CaskConvolution): ...
- Layer(Shuffle): ...

➤ 性能对比

➤ 优化前：2.09467 ms

➤ 优化后：1.83174 ms



# • 优化技术

## ➤ 手工合并算子

### ➤ 核心代码

### ➤ 要点

#### ➤ 安排卷积核权重排列

#### ➤ 补充 Squeeze 和 Transpose 节点

```
1  for node in graph.nodes:
2      if node.op == "Relu" and node.name == 'Relu_38':
3          matmulNode = node.o(2).o().o()
4          addNode = matmulNode.o()
5          mulNode = addNode.o()
6
7          convKernel = matmulNode.inputs[1].values
8          convKernel = convKernel.transpose(1, 0).reshape(256, 256, 1, 19).astype(np.float32)
9          convKernelV = gs.Constant("ConvKernelV", np.ascontiguousarray(convKernel))
10         convBias = addNode.inputs[0].values
11         convBiasV = gs.Constant("ConvBiasV", np.ascontiguousarray(convBias))
12         convV = gs.Variable("ConvV", np.dtype(np.float32), ['B', 256, 't4', 1])
13         convN = gs.Node("Conv",
14                         "ConvN",
15                         inputs=[node.outputs[0], convKernelV, convBiasV],
16                         outputs=[convV])
17         convN.attrs = OrderedDict([
18             ('dilations', [1, 1]),
19             ('kernel_shape', [1, 19]),
20             ('pads', [0, 0, 0, 0]),
21             ('strides', [1, 1]),
22         ])
23         graph.nodes.append(convN)
24
25         squeezeV = gs.Variable("SqueezeV", np.dtype(np.float32), ['B', 256, 't4'])
26         squeezeN = gs.Node("Squeeze",
27                             "SqueezeN",
28                             inputs=[convV, Constant3],
29                             outputs=[squeezeV])
30         graph.nodes.append(squeezeN)
31
32         transposeV = gs.Variable("TransposeV", np.dtype(np.float32), ['B', 't4', 256])
33         transposeN = gs.Node("Transpose",
34                             "TransposeN-",
35                             inputs=[squeezeV],
36                             outputs=[transposeV],
37                             attrs=OrderedDict([('perm', [0, 2, 1])]))
38         graph.nodes.append(transposeN)
39
40         mulNode.inputs[0] = transposeV
```



# • 优化技术

## ➤ 使用 strict type 显式控制某些 layer 计算精度以控制误差

### ➤ 关键操作

#### ➤ Builder 标志位 STRICT\_TYPES

#### ➤ layer 中设置 precision 和 输出张量的 dtype

### ➤ 自动化程度比较低，需要手工调整

```
1 config = builder.create_builder_config()
2 config.flags = config.config.flags | 1 << int(trt.BuilderFlag.STRICT_TYPES)
3
4 # ...
5
6 for i in range(network.num_layers):
7     if layer.name in targetLayerList:
8         layer.precision = trt.float32
9         layer.get_output(0).dtype = trt.float32
10
11
12 engineString = builder.build_serialized_network(network, config)
```





# • 优化技术

## ➤ 书写 Plugin 替换计算图中一些模块

➤ 基本流程：参考 [cookbook/05-Plugin/usePluginV2DynamicExt](#)

➤ 总结步骤：

➤ 实现一个Plugin 类和 PluginCreator 类

➤ 实现用于计算的 CUDA C++ kernel

➤ 将 Plugin 编译为 .so 保存

➤ 在 TenorRT 中加载和使用 Plugin

➤ Layer Normalization

➤ Attention

➤ Mask

➤ SkipSigmoid



# • 优化技术

## ➤ 书写 Plugin 替换计算图中一些模块

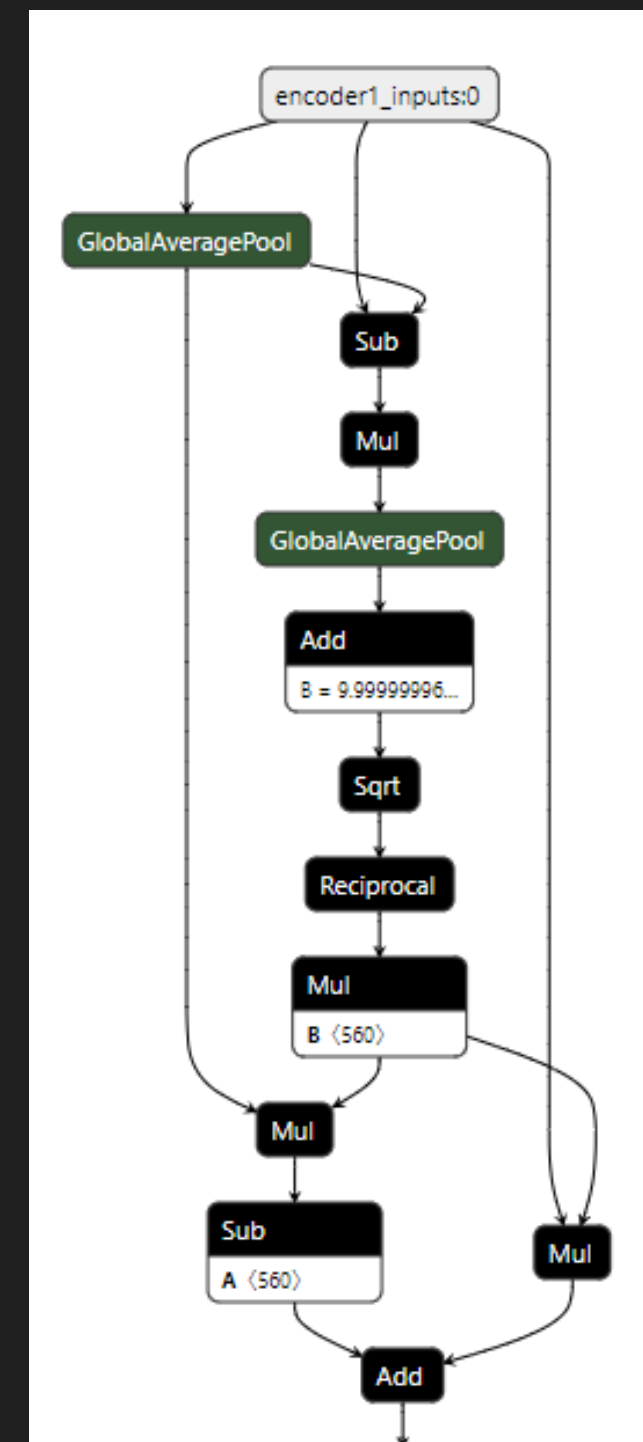
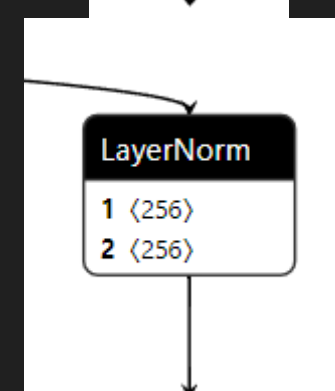
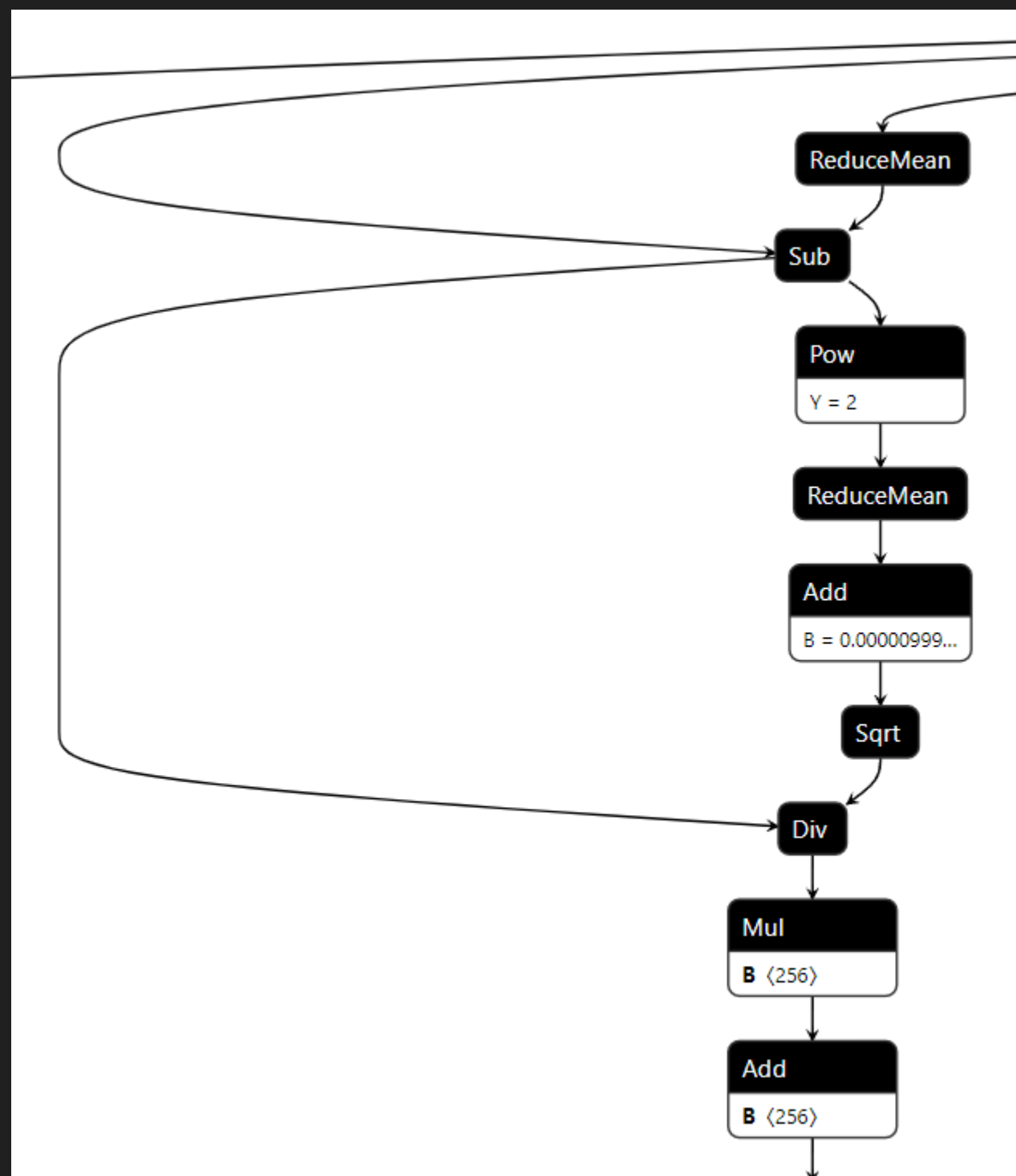
➤ Layer Normalization

➤ 替换计算图中的 Layer Norm 模块

➤ encoder 中有 61 个, decoder 中有 19 个

➤ 介绍 3 个版本: 基本版, CUB版, OneFlow 版

➤ **LayerNorm 形式不统一, TensorRT 暂时没有自动融合**



# • 优化技术

## ➤ 书写 Plugin 替换计算图中一些模块

### ➤ Layer Normalization

### ➤ 基本版：简单的 CUDA C++ 编程练习

#### ➤ 正确完成所需功能

#### ➤ 大幅改善原模型精度和性能

#### ➤ 只接受 Float32 输入输出

#### ➤ 不包含 LN 后续的线性变换

#### ➤ hidden size 绑定 launch parameter 不能扩展

#### ➤ 手工计算规约效率不足

#### ➤ 注意同步原语的使用

```
1  __global__ void layerNormKernel(float *pInput, float *pOutput)
2  {
3      const int tx = threadIdx.x, index = blockIdx.x * 256 + threadIdx.x;
4      __shared__ float temp[128];
5
6      float value0 = pInput[index], value1 = pInput[index + 128];
7      temp[tx] = value0 + value1;
8      __syncthreads();
9
10     for (int stride = 64; stride >= 1; stride /= 2)
11     {
12         if (tx < stride)
13             temp[tx] += temp[tx + stride];
14         __syncthreads();
15     }
16     float mean = temp[0] / 256;
17     __syncthreads();
18
19     temp[tx] = (value0 - mean) * (value0 - mean) + (value1 - mean) * (value1 - mean);
20     __syncthreads();
21
22     for (int stride = 64; stride >= 1; stride /= 2)
23     {
24         if (tx < stride)
25             temp[tx] += temp[tx + stride];
26         __syncthreads();
27     }
28     float var = temp[0] / 256;
29
30     pOutput[index] = (value0 - mean) * rsqrtf(var + 6e-6);
31     pOutput[index + 128] = (value1 - mean) * rsqrtf(var + 6e-6);
32 }
```



# • 优化技术


## ➤ 书写 Plugin 替换计算图中一些模块


### ➤ CUB 版


### ➤ CUB (<https://nvlabs.github.io/cub/index.html>)

### ➤ 提供大量并行算法的 CUDA C++ 实现

### ➤ 包含我们想要的 ReduceSum 操作

[Download CUB v1.16.0](#)

 NVIDIA Research

[Browse or fork CUB at GitHub](#)

## (1) What is CUB?

CUB provides state-of-the-art, reusable software components for every layer of the CUDA programming model:

- **Parallel primitives**
  - **Warp-wide "collective" primitives**
    - Cooperative warp-wide prefix scan, reduction, etc.
    - Safely specialized for each underlying CUDA architecture
  - **Block-wide "collective" primitives**
    - Cooperative I/O, sort, scan, reduction, histogram, etc.
    - Compatible with arbitrary thread block sizes and types
  - **Device-wide primitives**
    - Parallel sort, prefix scan, reduction, histogram, etc.
    - Compatible with CUDA dynamic parallelism
- **Utilities**
  - **Fancy iterators**
  - **Thread and thread block I/O**
  - **PTX intrinsics**
  - **Device, kernel, and storage management**

### Table of Contents

- ↓ (1) What is CUB?
- ↓ (2) CUB's collective primitives
- ↓ (3) An example (block-wide sorting)
- ↓ (4) Why do you need CUB?
- ↓ (5) How do CUB collectives work?
  - ↓ 5.1 Generic programming
  - ↓ 5.2 Reflective class interfaces
  - ↓ 5.3 Flexible data arrangement across threads
  - ↓ 5.4 Static tuning and co-tuning
- ↓ (6) How do I get started using CUB?
- ↓ (7) How is CUB different than Thrust and Modern GPU?
- ↓ (8) Stable releases
- ↓ (9) Contributors
- ↓ (10) Open Source License





# • 优化技术

## ➤ 书写 Plugin 替换计算图中一些模块

### ➤ CUB 版 V1

- 与基本版流程基本对应
- 只是 ReduceSum 操作用 CUB 实现
- 可以接受 Float32 和 Float16 输入输出
- 囊括 LN 后续的线性变换
  - 减少两个 elementwise 级别的算子
- hidden size 为编译期常量，不能动态变化
  - 可用 switch 来支持多种取值

```
1  template<typename T>
2  __global__ void layerNormKernel(T *pInput, float *pGamma, float *pBeta, T *pOutput)
3  {
4      const int n = 256;
5      const int tx = threadIdx.x, index = blockIdx.x * n + tx;
6      T _x = pInput[index], _b = (T)pGamma[tx], _a = (T)pBeta[tx];
7
8      __shared__ T mean_shared, var_shared;
9
10     typedef cub::BlockReduce<T, n> BlockReduce;
11     __shared__ typename BlockReduce::TempStorage temp;
12     T& ref0 = _x;
13     T sum = BlockReduce(temp).Sum(ref0);
14     //__syncthreads();
15     if(tx == 0)
16         mean_shared = sum / T(n);
17     __syncthreads();
18
19     T moment = _x - mean_shared, moment2 = moment * moment;
20     T& ref1 = moment2;
21     T var = BlockReduce(temp).Sum(ref1);
22     //__syncthreads();
23     if(tx == 0)
24         var_shared = var / T(n);
25     __syncthreads();
26
27     pOutput[index] = (moment) * (T)rsqrtf(var_shared + epsilon<T>()) * _b + _a;
28 }
```



# • 优化技术

## ➤ 书写 Plugin 替换计算图中一些模块

### ➤ CUB 版 V2

➤ <https://github.com/NVIDIA/TensorRT/blob/master/plugin/skipLayerNormPlugin/skipLayerNormKernel.cu>

➤ 注意到  $DX = EX^2 - (EX)^2$

➤ CUB 一趟归并同时计算上式右端两项

➤ 可接受 Float32/Float16 输入输出，但仅使用  
Float32 作为中间结果精度

➤ 使用自定义的内存拷贝函数来搬运输入输出数据

```
1  template <typename T, int TPB, int VPT>
2  __global__ void layerNormKernel(const T* input, const T* gamma, const T* beta, T* output)
3  {
4      const int idx = blockIdx.x * 256 + threadIdx.x * VPT;
5      T localX[VPT], localGamma[VPT], localBeta[VPT];
6
7      copy<sizeof(T) * VPT>(&input[idx], localX);
8      float2 localFloat2 = {0.f, 0.f};
9      const float rld = float(1) / float(256);
10     #pragma unroll
11     for (int it = 0; it < VPT; it++)
12     {
13         const float tmp = rld * (float)localX[it];
14         localFloat2.x += tmp;
15         localFloat2.y += tmp * (float)localX[it];
16     }
17
18     copy<sizeof(T) * VPT>(&beta[threadIdx.x * VPT], localBeta);
19     copy<sizeof(T) * VPT>(&gamma[threadIdx.x * VPT], localGamma);
20     using BlockReduce = cub::BlockReduce<float2, TPB>;
21     __shared__ typename BlockReduce::TempStorage temp_storage;
22     __shared__ float mu; // mean
23     __shared__ float rsigma; // 1 / std.dev.
24     const float2 sumKV = BlockReduce(temp_storage).Reduce(localFloat2, cub::Sum());
25
26     if (threadIdx.x == 0)
27     {
28         mu = sumKV.x;
29         rsigma = rsqrt(sumKV.y - mu * mu + 1e-6);
30     }
31     __syncthreads();
32     #pragma unroll
33     for (int it = 0; it < VPT; it++)
34     {
35         localX[it] = (float)localGamma[it] * ((float)localX[it] - mu) * rsigma + (float)localBeta[it];
36     }
37
38     copy<sizeof(T) * VPT>(localX, &output[idx]);
39 }
```



# • 优化技术

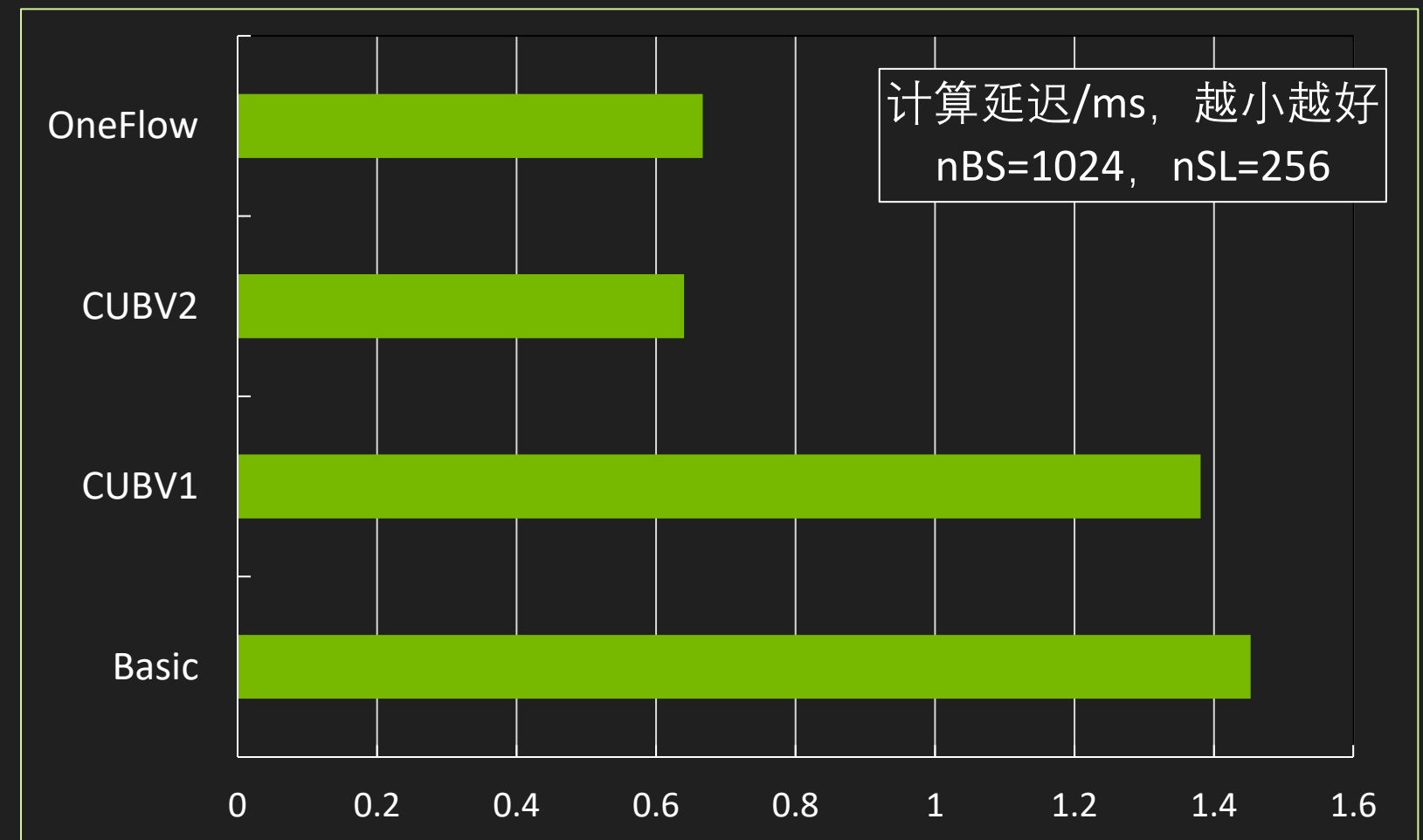
## ➤ 书写 Plugin 替换计算图中一些模块

### ➤ OneFlow 版

➤ [https://github.com/Oneflow-Inc/oneflow/blob/master/oneflow/core/cuda/layer\\_norm.cuh](https://github.com/Oneflow-Inc/oneflow/blob/master/oneflow/core/cuda/layer_norm.cuh)

➤ 并行 Welford 算法

## ➤ 几种 Layer Norm Plugin 的性能比较



# • 优化技术

## ➤ 书写 Plugin 替换计算图中一些模块

➤ Attention Plugin (代码以后开源)

➤ 介绍 3 个版本：基本版，TRTPlugin 版，FasterTransformer 版

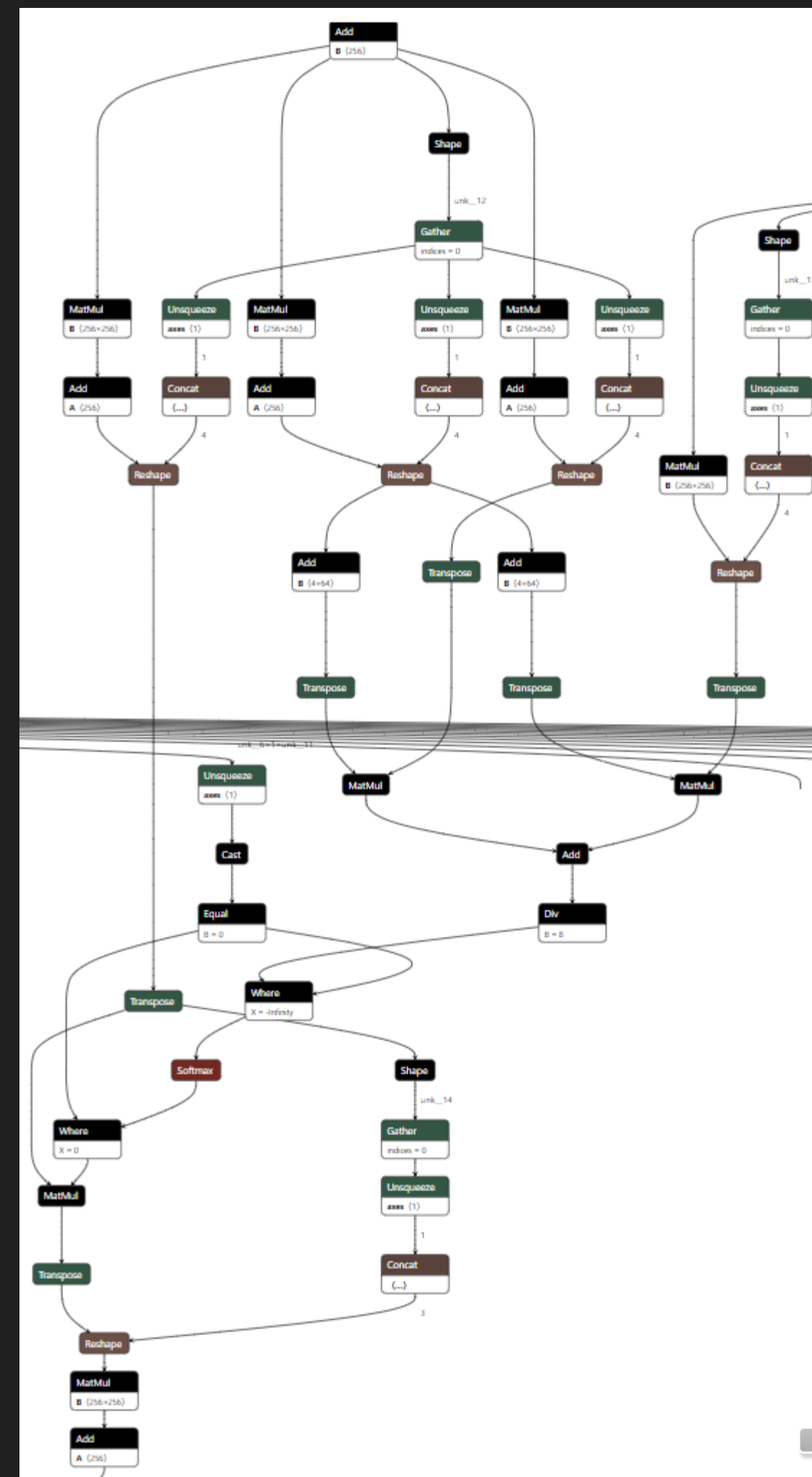
➤ 基本版：

➤ 使用 cuBLAS / cuBLASLt 并实现 Transpose 和 Softmax

➤ TRTPlugin 版 (**Good Luck To You!** 队的版本)：

➤ 使用 TensorRT 自带的 bertQKVToContextPlugin 加工

➤ <https://github.com/NVIDIA/TensorRT/tree/main/plugin/bertQKVToContextPlugin>





# • 优化技术

- 书写 Plugin 替换计算图中一些模块
  - Attention Plugin (代码以后开源)
  - Faster Transformer 版 (**ching** 大佬的版本)
    - <https://github.com/NVIDIA/FasterTransformer>
  - 一个FT 写成 TensorRT 的 Plugin 的例子
    - [https://github.com/NVIDIA/FasterTransformer/tree/main/src/fastertransformer/tensorrt\\_plugin/t5](https://github.com/NVIDIA/FasterTransformer/tree/main/src/fastertransformer/tensorrt_plugin/t5)

A100 GPU FP16 模式下 BERT-encoder 性能对比 (Torch v.s. FT)

Batch_size	Seq_len	Precision	TorchScript Latency (ms)	FT Latency (ms)	EFF-FT Latency (ms)	FT Speedup	EFF-FT Speedup
1	32	FP16	4.22	0.93	1.15	4.53	3.66
1	128	FP16	4.47	1.09	1.20	4.10	3.72
1	384	FP16	4.16	1.36	1.16	3.05	3.58
8	32	FP16	4.19	1.11	1.13	3.77	3.70
8	128	FP16	4.35	1.81	1.59	2.40	2.73
8	384	FP16	9.51	4.77	3.04	1.99	3.12
32	32	FP16	4.19	1.82	1.58	2.30	2.65
32	128	FP16	9.57	4.81	2.79	1.98	3.43
32	384	FP16	32.43	14.68	8.07	2.20	4.01

```
1 int T5EncoderPlugin::enqueue(...) noexcept
2 {
3     // ...
4     std::unordered_map<std::string, Tensor> inputTensor{
5     {
6         "input_ids",
7         Tensor{
8             MEMORY_GPU,
9             TYPE_INT32,
10            std::vector<size_t>{m_.batch_size, m_.seq_len},
11            (int*)inputs[0]
12        }
13    },
14    {
15        "sequence_length",
16        Tensor{
17            MEMORY_GPU,
18            TYPE_INT32, std::vector<size_t>{m_.batch_size},
19            (int*)inputs[1]
20        }
21    }
22 };
23 if (m_.useFP16)
24 {
25     std::unordered_map<std::string, Tensor> outputTensor{
26     {
27         "output_hidden_state",
28         Tensor{
29             MEMORY_GPU,
30             TYPE_FP16,
31             std::vector<size_t>{m_.batch_size, m_.seq_len, (size_t)(m_.head_num * m_.size_per_head)},
32             (half*)outputs[0]
33         }
34     }
35 };
36 pT5EncoderHalf->setStream(stream);
37 pT5EncoderHalf->forward(&outputTensor, &inputTensor, pT5EncoderWeightHalf_);
38 }
39 else
40 {
41     // ...
42 }
43 return 0;
44 }
```



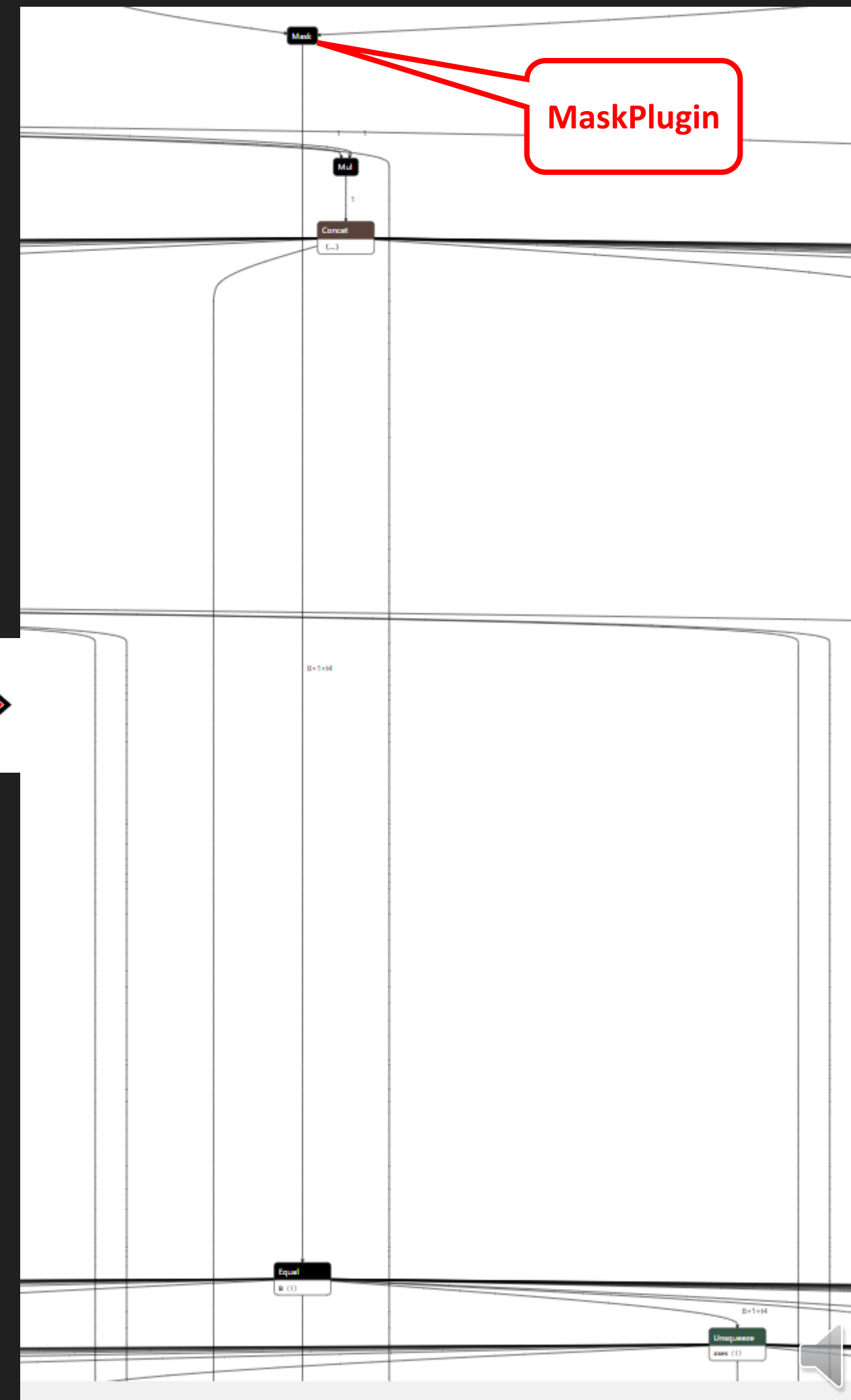
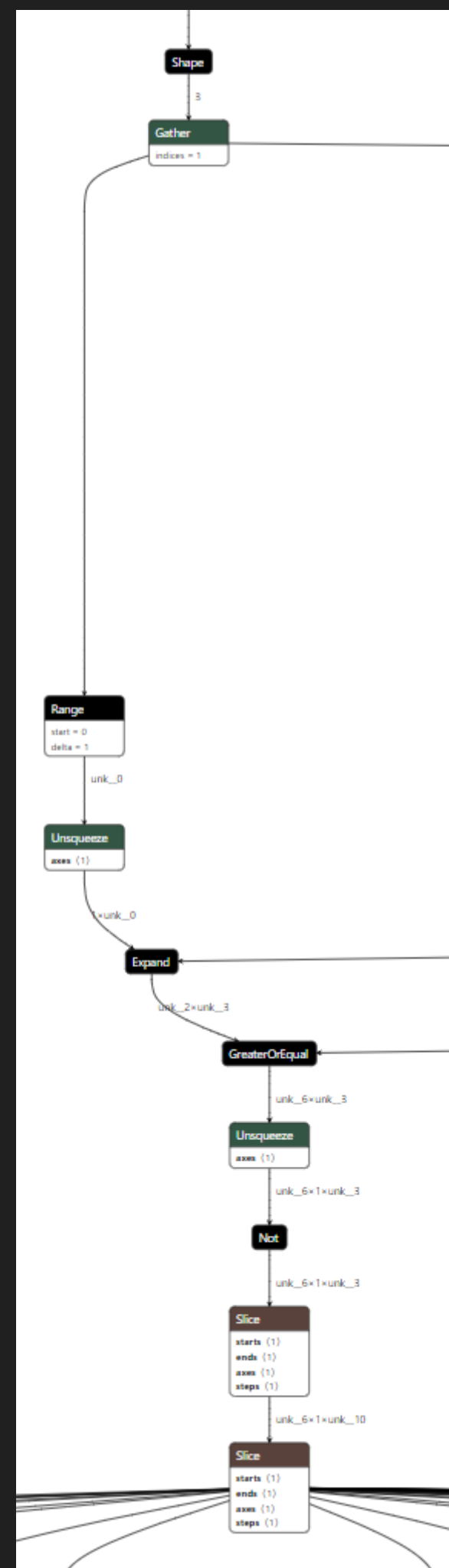
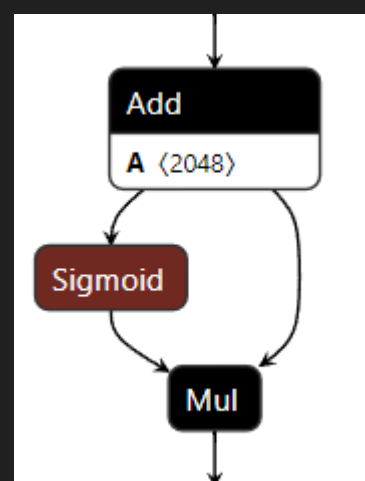
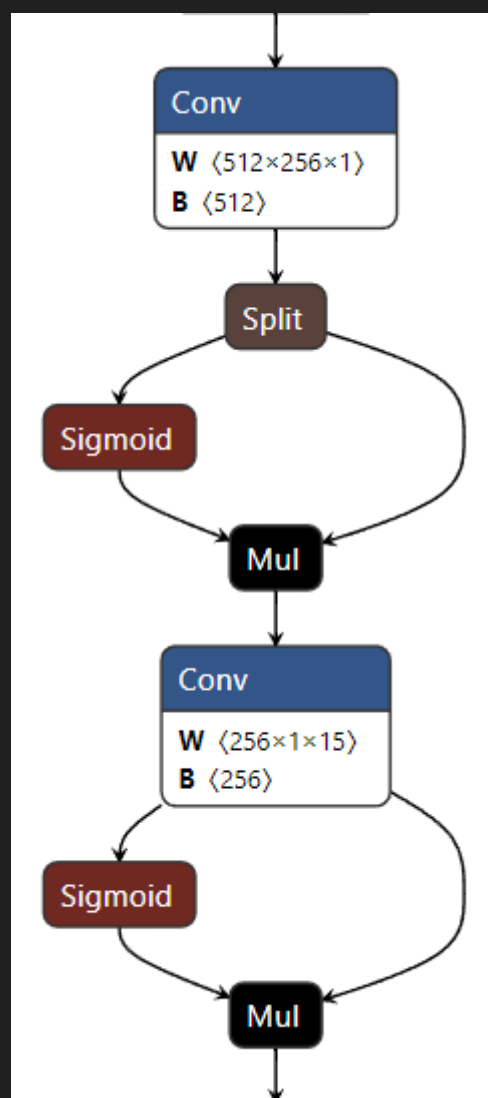
# • 优化技术

➤ 书写 Plugin 替换计算图中一些模块

➤ Mask Plugin

➤ SkipSigmoid Plugin

➤ 主要为 Elementwise 操作, 优化空间小

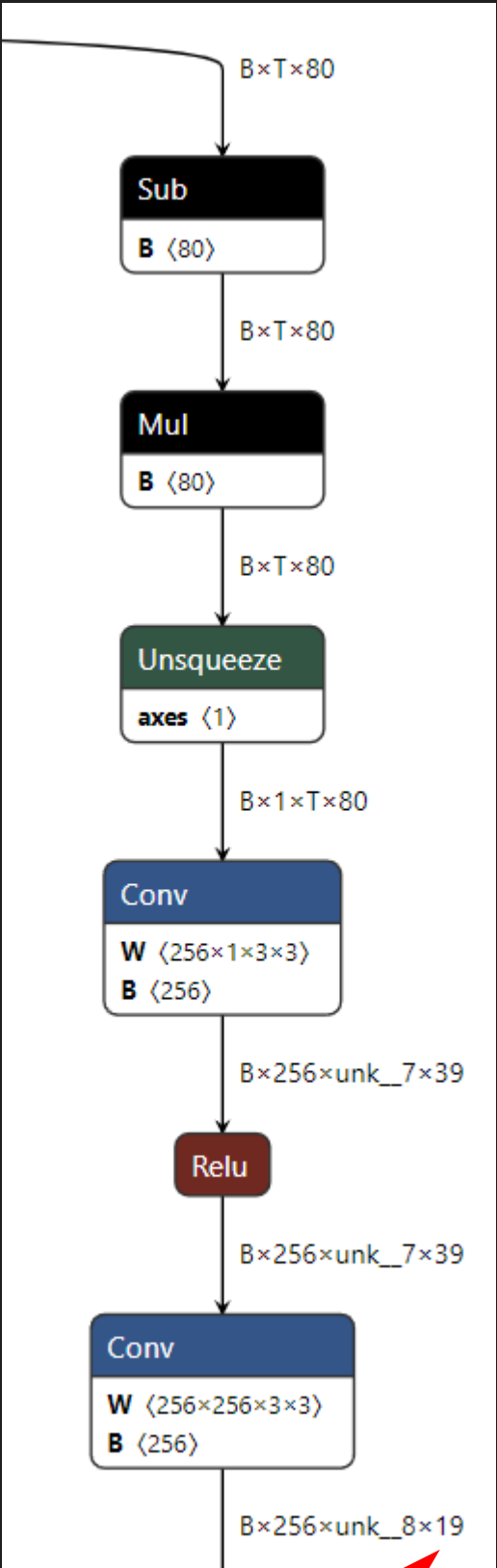


# • 优化技术

## ➤ 尺寸对齐

- 矩阵的边长对齐到某些倍数时性能更佳（与 Cache Line 宽度等有关）
  - <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-3-0>
- Tensor Core 对矩阵边长有约束，否则只能使用 Cuda Core 进行计算
- TensorRT 中推荐的尺寸对齐：
  - <https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#optimize-tensor-cores>

Table 4. Types of Tensor Cores	
Tensor Core Operation Type	Suggested Tensor Dimension Alignment in Elements
TF32	4
FP16	8 for dense math, 16 for sparse math
INT8	32



80经两次下采样  
变成 19



# • 优化技术

## ➤ 尺寸对齐

➤ 一个例子：某个 Self-Attention 模块在不同 BatchSize 和 Sequence Length 下的表现

➤ 另一个例子

➤ `cookbook/10-BestPractice/AlignSize`

➤ 构造网络反复做 MatMul+Add+ReLU

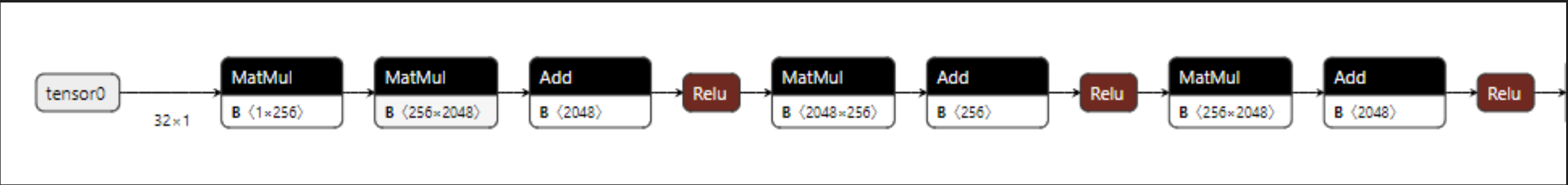
➤  $m=32, k=256, n=2048$ : 0.193232 ms

➤  $m=31, k=256, n=2048$ : 0.192441 ms

➤  $m=32, k=255, n=2048$ : 0.280797 ms

➤  $m=32, k=256, n=2047$ : 0.278274 ms

nBS	nSL	Time/ms
16	125	0.21299
16	128	0.19862
16	250	0.60175
16	256	0.55751
16	500	2.00085
16	512	1.71833
32	125	0.40129
32	128	0.37055
32	250	1.22323
32	256	1.10041
32	500	3.95247
32	512	3.40681



# • 优化技术

## ➤ 模型尺寸和精度问题

➤ 模型 dynamic shape 尺寸较大时对显存要求很高，本地显存不足时可以先尝试构建尺寸较小的模型

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

➤ TensorRT8.0 – 8.4EA 的 Myelin 显存需求大

➤ 尝试退回 TensorRT7 或使用将来的 TensorRT8.4GA

➤ 这个问题在 TensorRT8.4GA 后会得到缓解和最终解决

➤ Ampere GPU 默认开启 TF32 模式而非使用 FP32 模式

➤ trtexec 需要使用 `-noTF32` 选项关闭

➤ script 需要清除该 builderFlag 关闭

### tensorrt.BuilderFlag

Valid modes that the builder can enable when creating an engine from a network definition.

Members:

FP16 : Enable FP16 layer selection

INT8 : Enable Int8 layer selection

DEBUG : Enable debugging of layers via synchronizing after every layer

GPU\_FALLBACK : Enable layers marked to execute on GPU if layer cannot execute on DLA

STRICT\_TYPES : Deprecated: Enables strict type constraints. Equivalent to setting PREFER\_PRECISION\_CONSTRAINTS, DIRECT\_IO, and REJECT\_EMPTY\_ALGORITHMS.

REFIT : Enable building a refittable engine

DISABLE\_TIMING\_CACHE : Disable reuse of timing information across identical layers.

TF32 : Allow (but not require) computations on tensors of type DataType.FLOAT to use TF32. TF32 computes inner products by rounding the inputs to 10-bit mantissas before multiplying, but accumulates the sum using 23-bit mantissas. **Enabled by default.**

SPARSE\_WEIGHTS : Allow the builder to examine weights and use optimized functions when weights have suitable sparsity.

SAFETY\_SCOPE : Change the allowed parameters in the EngineCapability.STANDARD flow to match the restrictions that EngineCapability.SAFETY check against for DeviceType.GPU and EngineCapability.DLA\_STANDALONE check against the DeviceType.DLA case. This flag is forced to true if EngineCapability.SAFETY at build time if it is unset.

OBEY\_PRECISION\_CONSTRAINTS : Require that layers execute in specified precisions. Build fails otherwise.

PREFER\_PRECISION\_CONSTRAINTS : Prefer that layers execute in specified precisions. Fall back (with warning) to another precision if build would otherwise fail.

DIRECT\_IO : Require that no reformats be inserted between a layer and a network I/O tensor for which ITensor.allowed\_formats was set. Build fails if a reformat is required for functional correctness.

REJECT\_EMPTY\_ALGORITHMS : Fail if IAlgorithmSelector.select\_algorithms returns an empty set of algorithms.





# • 优化技术

## ➤ 模型尺寸和精度问题

➤ 模型 dynamic shape 尺寸较大时对显存要求很高，本地显存不足时可以先尝试构建尺寸较小的模型

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

➤ TensorRT8.0 – 8.4EA 的 Myelin 显存需求大

➤ 尝试退回 TensorRT7 或使用将来的 TensorRT8.4GA

➤ 这个问题在 TensorRT8.4GA 后会得到缓解和最终解决

➤ Ampere GPU 默认开启 TF32 模式而非使用 FP32 模式

➤ trtexec 需要使用 `-noTF32` 选项关闭

➤ script 需要清除该 builderFlag 关闭

### tensorrt.BuilderFlag

Valid modes that the builder can enable when creating an engine from a network definition.

Members:

FP16 : Enable FP16 layer selection

INT8 : Enable Int8 layer selection

DEBUG : Enable debugging of layers via synchronizing after every layer

GPU\_FALLBACK : Enable layers marked to execute on GPU if layer cannot execute on DLA

STRICT\_TYPES : Deprecated: Enables strict type constraints. Equivalent to setting PREFER\_PRECISION\_CONSTRAINTS, DIRECT\_IO, and REJECT\_EMPTY\_ALGORITHMS.

REFIT : Enable building a refittable engine

DISABLE\_TIMING\_CACHE : Disable reuse of timing information across identical layers.

TF32 : Allow (but not require) computations on tensors of type DataType.FLOAT to use TF32. TF32 computes inner products by rounding the inputs to 10-bit mantissas before multiplying, but accumulates the sum using 23-bit mantissas. **Enabled by default.**

SPARSE\_WEIGHTS : Allow the builder to examine weights and use optimized functions when weights have suitable sparsity.

SAFETY\_SCOPE : Change the allowed parameters in the EngineCapability.STANDARD flow to match the restrictions that EngineCapability.SAFETY check against for DeviceType.GPU and EngineCapability.DLA\_STANDALONE check against the DeviceType.DLA case. This flag is forced to true if EngineCapability.SAFETY at build time if it is unset.

OBEY\_PRECISION\_CONSTRAINTS : Require that layers execute in specified precisions. Build fails otherwise.

PREFER\_PRECISION\_CONSTRAINTS : Prefer that layers execute in specified precisions. Fall back (with warning) to another precision if build would otherwise fail.

DIRECT\_IO : Require that no reformats be inserted between a layer and a network I/O tensor for which ITensor.allowed\_formats was set. Build fails if a reformat is required for functional correctness.

REJECT\_EMPTY\_ALGORITHMS : Fail if IAlgorithmSelector.select\_algorithms returns an empty set of algorithms.



# • 比赛复盘

➤ 使用各项优化技术后的预计得分（不是评分标准）：

- 完成 onnx-graphsurgeon 模型调整：~800分
- 使用 FP32 模式正确运行模型 ~1100分
- 完成 Layer Normalization Plugin：~1400分
- 完成其他图优化：~1600分
- 正确使用 FP16 / INT8（含精度控制）：~1900分
- 使用除了 Attention Plugin 以外所有优化手段：~2200分
- 使用高效 Attention Plugin：>2200分



# • 彩蛋部分

➤ 初赛可用的其他优化方法：

➤ 权重裁剪

➤ 输出张量裁剪

➤ 形状裁剪

➤ 打表法（.npz 的读取和使用）



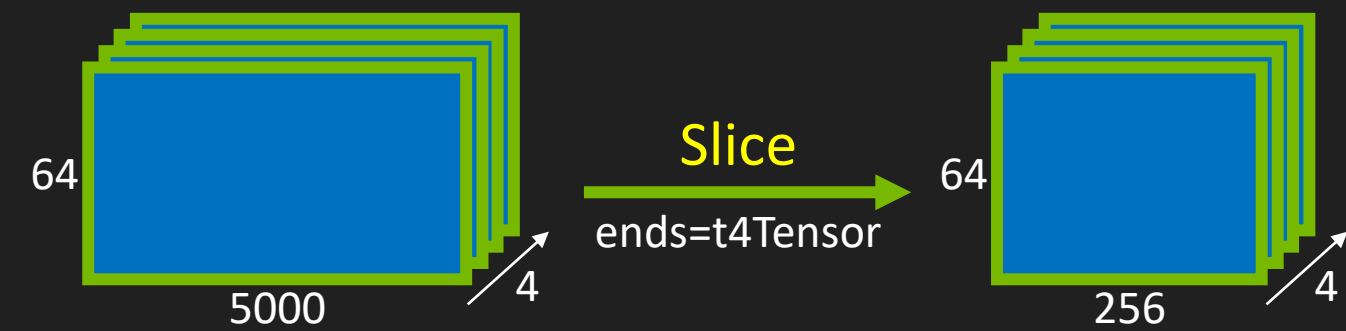
# • 优化技术

## ➤ 权重裁剪

➤ 已知输入数据动态范围时，裁剪权重以减小 .plan 体积和数据拷贝时间

## ➤ 举例

➤ encoder/decoder 的 position embedding 表



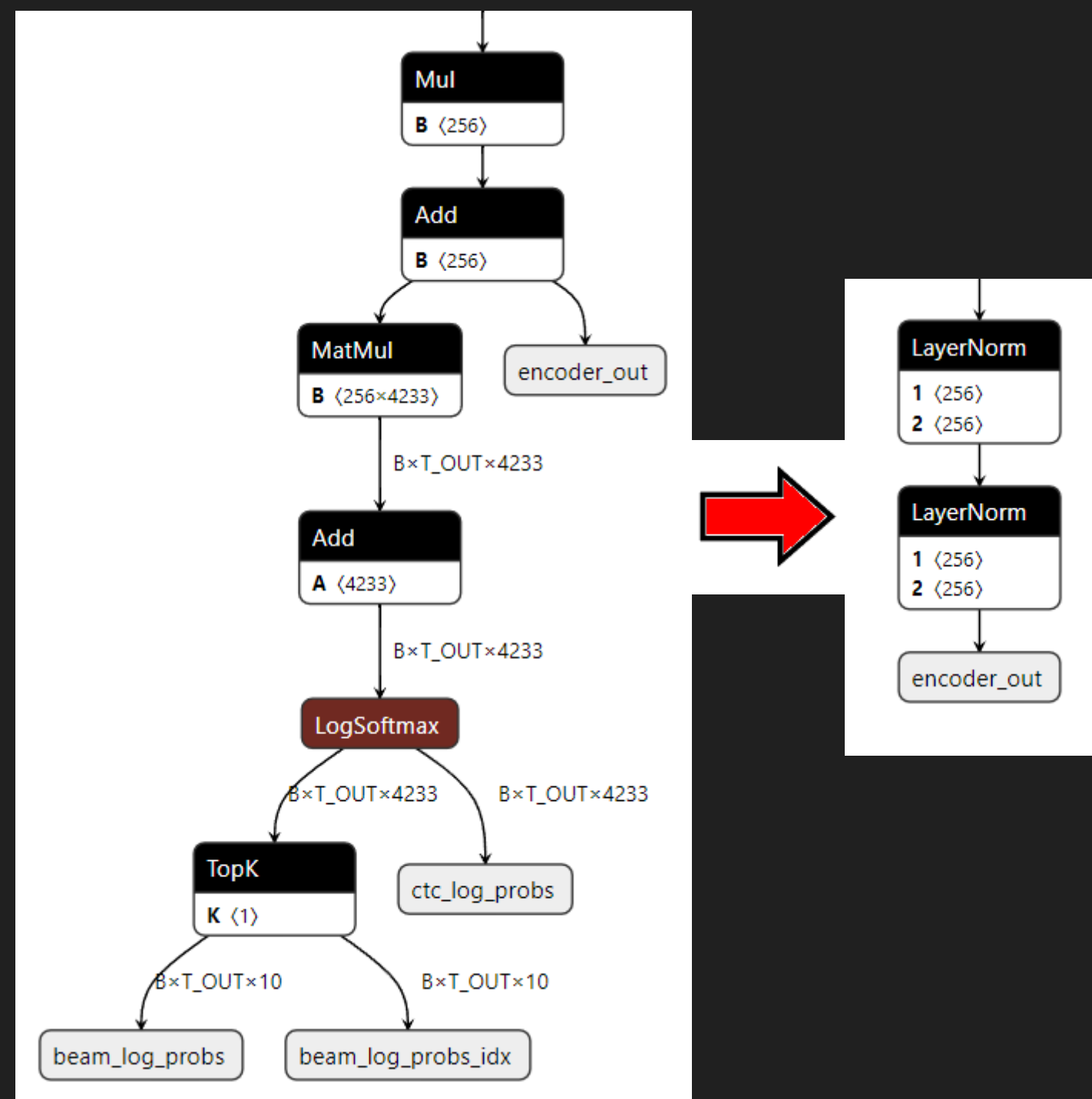
# • 优化技术

## ➤ 输出张量裁剪

➤ 比赛中 encoder 只用到了 encoder\_out 和 encoder\_out\_lens 参与评分，剩余的可以删掉

➤ 节省了 1 个矩阵乘法、1 个 LogSoftmax、1 个 TopK

➤ 可以在 .onnx 中删掉，也可以在 TensorRT 中 unmark\_output 掉





# • 优化技术

## ➤ 形状裁剪

➤ [W] Myelin graph with multiple dynamic values may have poor performance if they differ.

Dynamic values are:

➤ [W] (# 0 (SHAPE encoder\_out))

➤ [W] (# 0 (SHAPE encoder\_out\_len))

➤ [W] [TRT] Myelin graph with multiple dynamic values may have poor performance if they differ. Dynamic values are:

➤ [W] [TRT] (# 1 (RESHAPE (# 0 (SHAPE speech\_lengths)) E0 256 19 | (# 0 (SHAPE speech\_lengths)) E0 4864 zerosPlaceholder)) where E0=(+ (CEIL\_DIV (+ (# 1 (SHAPE speech)) -10) 4) 1)

➤ [W] [TRT] (- 0 (CEIL\_DIV (- -1 (MIN (- 0 E0) (MAX 0 (- -2 E0)))) 2)) where E0=(CEIL\_DIV (- -1 (MIN (# 1 (SHAPE speech)) (MAX 0 (+ (# 1 (SHAPE speech)) -2)))) 2)

➤ [W] [TRT] (# 0 (SHAPE speech\_lengths))

INPUTS		encoder	INPUTS		decoder
speech	name:	speech	encoder_out	name:	encoder_out
	type:	float32[B,T,80]		type:	float32[B,T,256]
speech_lengths	name:	speech_lengths	encoder_out_lens	name:	encoder_out_lens
	type:	int32[B]		type:	int32[B]
OUTPUTS			OUTPUTS		
encoder_out	name:	encoder_out	hyps_pad_sos_eos	name:	hyps_pad_sos_eos
	type:	float32[B,T_OUT,Addencoder_out_dim_2]		type:	int64[B,10,64]
encoder_out_lens	name:	encoder_out_lens	hyps_lens_sos	name:	hyps_lens_sos
	type:	int32[B]		type:	int32[B,10]
ctc_log_probs	name:	ctc_log_probs	ctc_score	name:	ctc_score
	type:	float32[B,T_OUT,4233]		type:	float32[B,10]
beam_log_probs	name:	beam_log_probs	decoder_out	name:	decoder_out
	type:	float32[B,T_OUT,10]		type:	float32
beam_log_probs_idx	name:	beam_log_probs_idx	best_index	name:	best_index
	type:	int64[B,T_OUT,10]		type:	int64[B]



# • 优化技术

## ➤ 打表法

➤ 使用 Plugin 直接保存输出数据，不计算而是直接抛出结果

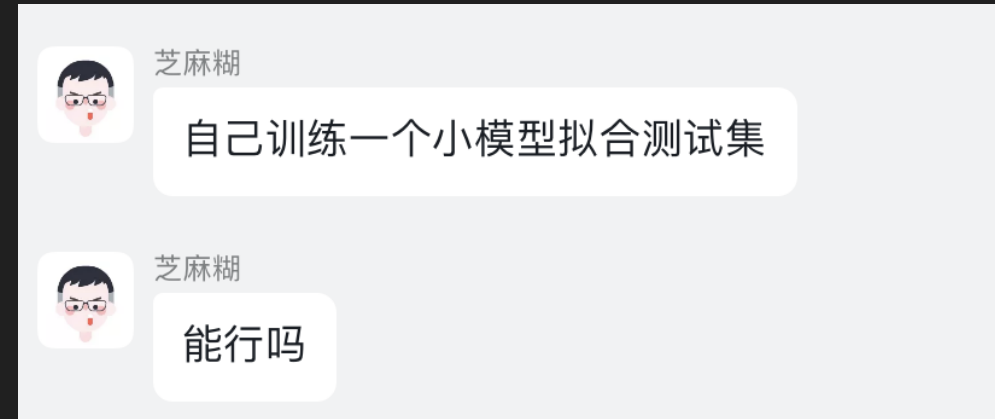
➤ 参考 Cookbook/05-Plugin/loadNpz

➤ **这个范例的正经用途**：在 Plugin 中读写 .npz

➤ 核心代码：<https://github.com/rogersce/cnpy>

➤ 结果：将性能提高三个数量级：)

➤ encoder 得分：1192096.5971，decoder 得分：589651.4553，总分：1011363.0546



```
1  int32_t LoadNpzPlugin::enqueue(...) noexcept
2  {
3      int nElement = 1;
4      for (int i = 0; i < inputDesc[0].dims.nbDims; ++i)
5      {
6          nElement *= 4;
7      }
8      cudaMemcpyAsync(outputs[0], pGPU_, sizeof(float) * nElement, cudaMemcpyDeviceToDevice, stream);
9      return 0;
10 }
```



- **优化技术**

- 更多优化技术和经验

- 参考 cookbook 10-BestPractice (持续更新中)





