Loop 结构

- 初始示例代码，for 循环，两种输出模式
- for 型循环，运行时指定循环次数（使用 context.set_shape_input，其实不太常用）
- while 型循环，两种输出模式
- while 型循环，引发错误的一种写法
- iterator 迭代层

初始示例代码，for 型循环，两种输出

```
import numpy as np
from cuda import cudart
import tensorrt as trt

nIn, cIn, hIn, wIn = 1, 3, 4, 5  # 输入张量 NCHW
t = np.array([6], dtype=np.int32)  # 循环次数
data = np.ones([nIn, cIn, hIn, wIn], dtype=np.float32)  # 输入数据

np.set_printoptions(precision=8, linewidth=200, suppress=True)
cudart.cudaDeviceSynchronize()

logger = trt.Logger(trt.Logger.ERROR)
builder = trt.Builder(logger)
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
config = builder.create_builder_config()
config.max_workspace_size = 1 << 30
inputT0 = network.add_input('inputT0', trt.DataType.FLOAT, (nIn, cIn, hIn, wIn))
#---------------------------------------------------------- --------------------# 替换部分
loop = network.add_loop()  # 添加 Loop 结构

limit = network.add_constant((), np.array([t], dtype=np.int32))  # build-time 常数型迭代次数
loop.add_trip_limit(limit.get_output(0), trt.TripLimit.COUNT)  # 指定 COUNT 型循环（ 类似 for 循环）

rLayer = loop.add_recurrence(inputT0)  # 循环入口
_H0 = network.add_elementwise(rLayer.get_output(0), rLayer.get_output(0), trt.ElementWiseOperation.SUM)
 # 循环体
#rLayer.set_input(0,inputT0)                                                                    #
rLayer 的第 0 输入是循环入口张量，这里可以不用再赋值
rLayer.set_input(1, _H0.get_output(0))  # rLayer 的第 1 输入时循环计算子图的输出张量

loopOutput0 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)  # 第一种循环输出，
只保留最终结果，index 参数被忽略
loopOutput1 = loop.add_loop_output(_H0.get_output(0), trt.LoopOutput.CONCATENATE, 0)  # 第二种循环输出，保
留所有中间结果，传入 _H0 则保留“第 1 到第 t 次迭代的结果”，传入 rLayer 则保留“第 0 到第 t-1 次迭代的结果”
loopOutput1.set_input(1, limit.get_output(0))  # 指定需要保留的长度，若这里传入张量的值 v <= t，则结果保留前 v 次
迭代，若 v > t，则多出部分用 0 填充
#---------------------------------------------------------- --------------------# 替换部分
network.mark_output(loopOutput0.get_output(0))
network.mark_output(loopOutput1.get_output(0))
engineString = builder.build_serialized_network(network, config)
engine = trt.Runtime(logger).deserialize_cuda_engine(engineString)
context = engine.create_execution_context()
_, stream = cudart.cudaStreamCreate()
```

```
inputH0 = np.ascontiguousarray(data.reshape(-1))
outputH0 = np.empty(context.get_binding_shape(1), dtype=trt.nptype(engine.get_binding_dtype(1)))
outputH1 = np.empty(context.get_binding_shape(2), dtype=trt.nptype(engine.get_binding_dtype(2)))
_, inputD0 = cudart.cudaMallocAsync(inputH0.nbytes, stream)
_, outputD0 = cudart.cudaMallocAsync(outputH0.nbytes, stream)
_, outputD1 = cudart.cudaMallocAsync(outputH1.nbytes, stream)

cudart.cudaMemcpyAsync(inputD0, inputH0.ctypes.data, inputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
context.execute_async_v2([int(inputD0), int(outputD0), int(outputD1)], stream)
cudart.cudaMemcpyAsync(outputH0.ctypes.data, outputD0, outputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaMemcpyAsync(outputH1.ctypes.data, outputD1, outputH1.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaStreamSynchronize(stream)

print("inputH0 :", data.shape)
print(data)
print("outputH0:", outputH0.shape)
print(outputH0)
print("outputH1:", outputH1.shape)
print(outputH1)

cudart.cudaStreamDestroy(stream)
cudart.cudaFree(inputD0)
cudart.cudaFree(outputD0)
cudart.cudaFree(outputD1)
```

- 输入张量形状 (1,3,4,5)

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\end{bmatrix} \begin{bmatrix}1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\end{bmatrix} \begin{bmatrix}1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

- 输出张量 0（loopOutput0）形状 (1,3,4,5)，循环最终的结果

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\end{bmatrix} \begin{bmatrix}64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\end{bmatrix} \begin{bmatrix}64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

- 输出张量 1（loopOutput1）形状 (6,1,3,4,5)，在输入张量的最前面增加一维，将每次迭代的输出依次放入

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix} \begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix} \begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix}\end{bmatrix}\end{bmatrix} \\ \begin{bmatrix}\begin{bmatrix}\begin{bmatrix}4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\end{bmatrix} \begin{bmatrix}4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\end{bmatrix} \begin{bmatrix}4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\end{bmatrix}\end{bmatrix}\end{bmatrix} \\ \cdots \\ \begin{bmatrix}\begin{bmatrix}\begin{bmatrix}64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\end{bmatrix} \begin{bmatrix}64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\end{bmatrix} \begin{bmatrix}64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\end{bmatrix}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

- 计算过程，类似以下代码

```
temp = inputT0
loopOutput0 = inputT0
loopOutput1 = np.zeros([t]+inputT0.shape)
for i in range(t):
    loopOutput0 += loopOutput0
    loopOutput1[t] = loopOutput0
return loopOutput0, loopOutput1
```

- LAST_VALUE 和 CONCATENATE 两种输出，可以只使用其中一个或两者同时使用（需要标记为两个不同的 loopOutput 层）
- 无 iterator 的循环不能将结果 CONCATENATE 到其他维度，也不能使用 REVERSE 输出，否则报错：

```
[TRT] [E] 10: [optimizer.cpp::computeCosts::2011] Error Code 10: Internal Error (Could not find any
implementation for node {ForeignNode[(Unnamed Layer* 1) [Recurrence]...(Unnamed Layer* 4)
[LoopOutput]]}.)
```

- 使用 LAST_VALUE 输出时，add_loop_output 的输入张量只能是 rLayer.get_output(0)，如果使用 _H0.get_output(0) 则会报错：

```
[TRT] [E] 4: [scopedOpValidation.cpp::reportIllegalLastValue::89] Error Code 4: Internal Error ((Unnamed
Layer* 4) [LoopOutput]: input of LoopOutputLayer with LoopOutput::kLAST_VALUE must be output from an
IRecurrenceLayer)
```

- 如果无限循环，则会收到报错：

```
terminate called after throwing an instance of 'nvinfer1::CudaRuntimeError'
  what():  an illegal memory access was encountered
Aborted (core dumped)
```

---

for 型循环，运行时指定循环次数（使用 context.set_shape_input，其实不太常用）

```
import numpy as np
from cuda import cudart
import tensorrt as trt

nIn, cIn, hIn, wIn = 1, 3, 4, 5  # 输入张量 NCHW
data = np.ones([nIn, cIn, hIn, wIn], dtype=np.float32)  # 输入数据
t = np.array([6], dtype=np.int32)  # 循环次数

np.set_printoptions(precision=8, linewidth=200, suppress=True)
cudart.cudaDeviceSynchronize()

logger = trt.Logger(trt.Logger.ERROR)
builder = trt.Builder(logger)
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
profile = builder.create_optimization_profile()
config = builder.create_builder_config()
config.max_workspace_size = 1 << 30
inputT0 = network.add_input('inputT0', trt.DataType.FLOAT, (nIn, cIn, hIn, wIn))
inputT1 = network.add_input('inputT1', trt.DataType.INT32, ())  # 循环次数作为输入张量在 runtime 指定
profile.set_shape_input(inputT1.name, (1, ), (6, ), (10, ))  # 这里设置的不是 shape input 的形状而是值，范围覆盖住之后需要的值就好
config.add_optimization_profile(profile)
#-------------------------------------------------------- --------------------# 替换部分
loop = network.add_loop()
loop.add_trip_limit(inputT1, trt.TripLimit.COUNT)
```

```
rLayer = loop.add_recurrence(inputT0)
_H0 = network.add_elementwise(rLayer.get_output(0), rLayer.get_output(0), trt.ElementWiseOperation.SUM)
rLayer.set_input(1, _H0.get_output(0))

loopOutput0 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)
loopOutput1 = loop.add_loop_output(_H0.get_output(0), trt.LoopOutput.CONCATENATE, 0)
loopOutput1.set_input(1, inputT1)
#---------------------------------------------------------- --------------------# 替换部分
network.mark_output(loopOutput0.get_output(0))
network.mark_output(loopOutput1.get_output(0))
engineString = builder.build_serialized_network(network, config)
engine = trt.Runtime(logger).deserialize_cuda_engine(engineString)
context = engine.create_execution_context()
context.set_shape_input(1, t)  # 运行时绑定真实形状张量值
_, stream = cudart.cudaStreamCreate()

inputH0 = np.ascontiguousarray(data.reshape(-1))
inputH1 = np.ascontiguousarray(np.zeros([1], dtype=np.int32).reshape(-1))  # 传形状张量数据可用垃圾值
outputH0 = np.empty(context.get_binding_shape(2), dtype=trt.nptype(engine.get_binding_dtype(2)))
outputH1 = np.empty(context.get_binding_shape(3), dtype=trt.nptype(engine.get_binding_dtype(3)))
_, inputD0 = cudart.cudaMallocAsync(inputH0.nbytes, stream)
_, inputD1 = cudart.cudaMallocAsync(inputH1.nbytes, stream)
_, outputD0 = cudart.cudaMallocAsync(outputH0.nbytes, stream)
_, outputD1 = cudart.cudaMallocAsync(outputH1.nbytes, stream)

cudart.cudaMemcpyAsync(inputD0, inputH0.ctypes.data, inputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
cudart.cudaMemcpyAsync(inputD1, inputH1.ctypes.data, inputH1.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
context.execute_async_v2([int(inputD0), int(inputD1), int(outputD0), int(outputD1)], stream)
cudart.cudaMemcpyAsync(outputH0.ctypes.data, outputD0, outputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaMemcpyAsync(outputH1.ctypes.data, outputD1, outputH1.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaStreamSynchronize(stream)

print("inputH0 :", data.shape)
print(data)
print("inputH1 :", t.shape)
print(t)
print("outputH0:", outputH0.shape)
print(outputH0)
print("outputH1:", outputH1.shape)
print(outputH1)

cudart.cudaStreamDestroy(stream)
cudart.cudaFree(inputD0)
cudart.cudaFree(outputD0)
cudart.cudaFree(outputD1)
```

- 输入张量和输出张量与初始示例代码相同

---

while 型循环，两种输出模式

```
import numpy as np
from cuda import cudart
import tensorrt as trt
```

```
nIn, cIn, hIn, wIn = 1, 3, 4, 5
data = np.ones([nIn, cIn, hIn, wIn], dtype=np.float32)
length = 7

np.set_printoptions(precision=8, linewidth=200, suppress=True)
cudart.cudaDeviceSynchronize()

logger = trt.Logger(trt.Logger.ERROR)
builder = trt.Builder(logger)
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
config = builder.create_builder_config()
config.max_workspace_size = 1 << 30
inputT0 = network.add_input('inputT0', trt.DataType.FLOAT, (nIn, cIn, hIn, wIn))
#------------------------------------------------------------------------------
loop = network.add_loop()  # 添加 Loop 结构
rLayer = loop.add_recurrence(inputT0)  # 循环入口

_H1 = network.add_reduce(rLayer.get_output(0), trt.ReduceOperation.MAX, (1 << 0) + (1 << 1) + (1 << 2) +
(1 << 3), False)  # 取循环体张量的第一个元素，判断其是否小于 6
_H2 = network.add_constant((), np.array([6], dtype=np.float32))
_H3 = network.add_elementwise(_H2.get_output(0), _H1.get_output(0), trt.ElementWiseOperation.SUB)
_H4 = network.add_activation(_H3.get_output(0), trt.ActivationType.RELU)
_H5 = network.add_identity(_H4.get_output(0))
_H5.get_output(0).dtype = trt.DataType.BOOL
loop.add_trip_limit(_H5.get_output(0), trt.TripLimit.WHILE)  # 判断结果转为 BOOL 类型，交给 TripLimit

_H0 = network.add_scale(rLayer.get_output(0), trt.ScaleMode.UNIFORM, np.array([1], dtype=np.float32),
np.array([1], dtype=np.float32), np.array([1], dtype=np.float32))  # 循环体，给输入元素加 1
rLayer.set_input(1, _H0.get_output(0))

loopOutput0 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)  # 第一种循环输出，
只保留最终结果，index 参数被忽略
loopOutput1 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.CONCATENATE, 0)  # 第二种循环输出，
保留所有中间结果，传入 rLayer 则保留“第 0 到第 t-1 次迭代的结果”（类比 while 循环），传入 _H0 则保留“第 1 到第 t 次迭
代的结果”（类比 do-while 循环，不推荐使用，可能有错误）
lengthLayer = network.add_constant((), np.array([length], dtype=np.int32))
loopOutput1.set_input(1, lengthLayer.get_output(0))  # 指定需要保留的长度，若这里传入张量的值 v <= t，则结果保留
前 v 次迭代，若 v > t，则多出部分用 0 填充
#------------------------------------------------------------------------------
network.mark_output(loopOutput0.get_output(0))
network.mark_output(loopOutput1.get_output(0))
engineString = builder.build_serialized_network(network, config)
engine = trt.Runtime(logger).deserialize_cuda_engine(engineString)
context = engine.create_execution_context()
_, stream = cudart.cudaStreamCreate()

inputH0 = np.ascontiguousarray(data.reshape(-1))
outputH0 = np.empty(context.get_binding_shape(1), dtype=trt.nptype(engine.get_binding_dtype(1)))
outputH1 = np.empty(context.get_binding_shape(2), dtype=trt.nptype(engine.get_binding_dtype(2)))
_, inputD0 = cudart.cudaMallocAsync(inputH0.nbytes, stream)
_, outputD0 = cudart.cudaMallocAsync(outputH0.nbytes, stream)
_, outputD1 = cudart.cudaMallocAsync(outputH1.nbytes, stream)

cudart.cudaMemcpyAsync(inputD0, inputH0.ctypes.data, inputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
context.execute_async_v2([int(inputD0), int(outputD0), int(outputD1)], stream)
cudart.cudaMemcpyAsync(outputH0.ctypes.data, outputD0, outputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
```

```
cudart.cudaMemcpyAsync(outputH1.ctypes.data, outputD1, outputH1.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaStreamSynchronize(stream)

print("inputH0 :", data.shape)
print(data)
print("outputH0:", outputH0.shape)
print(outputH0)
print("outputH1:", outputH1.shape)
print(outputH1)

cudart.cudaStreamDestroy(stream)
cudart.cudaFree(inputD0)
cudart.cudaFree(outputD0)
cudart.cudaFree(outputD1)
```

- 输入张量形状 (1,3,4,5)，结果与初始示例代码相同
- 输出张量 0（loopOutput0）形状 (1,3,4,5)，循环最终的结果

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\end{bmatrix} \begin{bmatrix}6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\end{bmatrix} \begin{bmatrix}6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

- 在 loopOutput1 中传入 rLayer，输出张量 1（loopOutput1）形状 (7,1,3,4,5)，保留“第 0 到第 4 次迭代的结果”

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\end{bmatrix} \begin{bmatrix}1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\end{bmatrix} \begin{bmatrix}1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\end{bmatrix}\end{bmatrix}\end{bmatrix}\\ \begin{bmatrix}\begin{bmatrix}\begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix} \begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix} \begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix}\end{bmatrix}\end{bmatrix}\\ \cdots\\ \begin{bmatrix}\begin{bmatrix}\begin{bmatrix}5. & 5. & 5. & 5. & 5.\\ 5. & 5. & 5. & 5. & 5.\\ 5. & 5. & 5. & 5. & 5.\\ 5. & 5. & 5. & 5. & 5.\end{bmatrix} \begin{bmatrix}5. & 5. & 5. & 5. & 5.\\ 5. & 5. & 5. & 5. & 5.\\ 5. & 5. & 5. & 5. & 5.\\ 5. & 5. & 5. & 5. & 5.\end{bmatrix} \begin{bmatrix}5. & 5. & 5. & 5. & 5.\\ 5. & 5. & 5. & 5. & 5.\\ 5. & 5. & 5. & 5. & 5.\\ 5. & 5. & 5. & 5. & 5.\end{bmatrix}\end{bmatrix}\end{bmatrix}\\ \begin{bmatrix}\begin{bmatrix}\begin{bmatrix}0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\end{bmatrix} \begin{bmatrix}0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\end{bmatrix} \begin{bmatrix}0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\end{bmatrix}\end{bmatrix}\end{bmatrix}\\ \begin{bmatrix}\begin{bmatrix}\begin{bmatrix}0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\end{bmatrix} \begin{bmatrix}0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\end{bmatrix} \begin{bmatrix}0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0. & 0.\end{bmatrix}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

- 在 loopOutput1 中传入 _H0，输出张量 1（loopOutput1）形状 (7,1,3,4,5)，保留“第 1 到第 5 次迭代的结果”
- **不推荐使用**，在循环判断他条件依赖循环体张量的时候可能有错误（见下一个示例）

$$
\left[\begin{array}{c}
\left[\left[\begin{bmatrix}2.&2.&2.&2.&2.\\2.&2.&2.&2.&2.\\2.&2.&2.&2.&2.\\2.&2.&2.&2.&2.\end{bmatrix}\begin{bmatrix}2.&2.&2.&2.&2.\\2.&2.&2.&2.&2.\\2.&2.&2.&2.&2.\\2.&2.&2.&2.&2.\end{bmatrix}\begin{bmatrix}2.&2.&2.&2.&2.\\2.&2.&2.&2.&2.\\2.&2.&2.&2.&2.\\2.&2.&2.&2.&2.\end{bmatrix}\right]\right]\\
\left[\left[\begin{bmatrix}3.&3.&3.&3.&3.\\3.&3.&3.&3.&3.\\3.&3.&3.&3.&3.\\3.&3.&3.&3.&3.\end{bmatrix}\begin{bmatrix}3.&3.&3.&3.&3.\\3.&3.&3.&3.&3.\\3.&3.&3.&3.&3.\\3.&3.&3.&3.&3.\end{bmatrix}\begin{bmatrix}3.&3.&3.&3.&3.\\3.&3.&3.&3.&3.\\3.&3.&3.&3.&3.\\3.&3.&3.&3.&3.\end{bmatrix}\right]\right]\\
\cdots\\
\left[\left[\begin{bmatrix}6.&6.&6.&6.&6.\\6.&6.&6.&6.&6.\\6.&6.&6.&6.&6.\\6.&6.&6.&6.&6.\end{bmatrix}\begin{bmatrix}6.&6.&6.&6.&6.\\6.&6.&6.&6.&6.\\6.&6.&6.&6.&6.\\6.&6.&6.&6.&6.\end{bmatrix}\begin{bmatrix}6.&6.&6.&6.&6.\\6.&6.&6.&6.&6.\\6.&6.&6.&6.&6.\\6.&6.&6.&6.&6.\end{bmatrix}\right]\right]\\
\left[\left[\begin{bmatrix}0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\end{bmatrix}\begin{bmatrix}0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\end{bmatrix}\begin{bmatrix}0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\end{bmatrix}\right]\right]\\
\left[\left[\begin{bmatrix}0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\end{bmatrix}\begin{bmatrix}0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\end{bmatrix}\begin{bmatrix}0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\\0.&0.&0.&0.&0.\end{bmatrix}\right]\right]
\end{array}\right]
$$

- 计算过程，loopOutput1 传入 rLayer 时类似以下代码

```
temp = inputT0
loopOutput1 = np.zeros([t]+inputT0.shape)
while (temp.reshape(-1)[0] < 6)
    loopOutput1[t] = temp
    temp += 1
loopOutput0 = temp
return loopOutput0, loopOutput1
```

- 计算过程，loopOutput1 传入 _H0 时类似以下代码

```
temp = inputT0
loopOutput1 = np.zeros([t]+inputT0.shape)
do
    temp += 1
    loopOutput1[t] = temp
while (temp.reshape(-1)[0] < 6)
loopOutput0 = temp
return loopOutput0, loopOutput1
```

- 可用的循环类型

tensorrt.TripLimit 名	说明
COUNT	for 型循环，给定循环次数
WHILE	while 型循环

- 这段示例代码要求 TensorRT>=8，TensorRT7 中运行该段代码会收到报错：

```
[TensorRT] ERROR: ../builder/myelin/codeGenerator.cpp (114) - Myelin Error in addNodeToMyelinGraph: 0
((Unnamed Layer* 2) [Reduce] outside operation not supported within a loop body.)
```

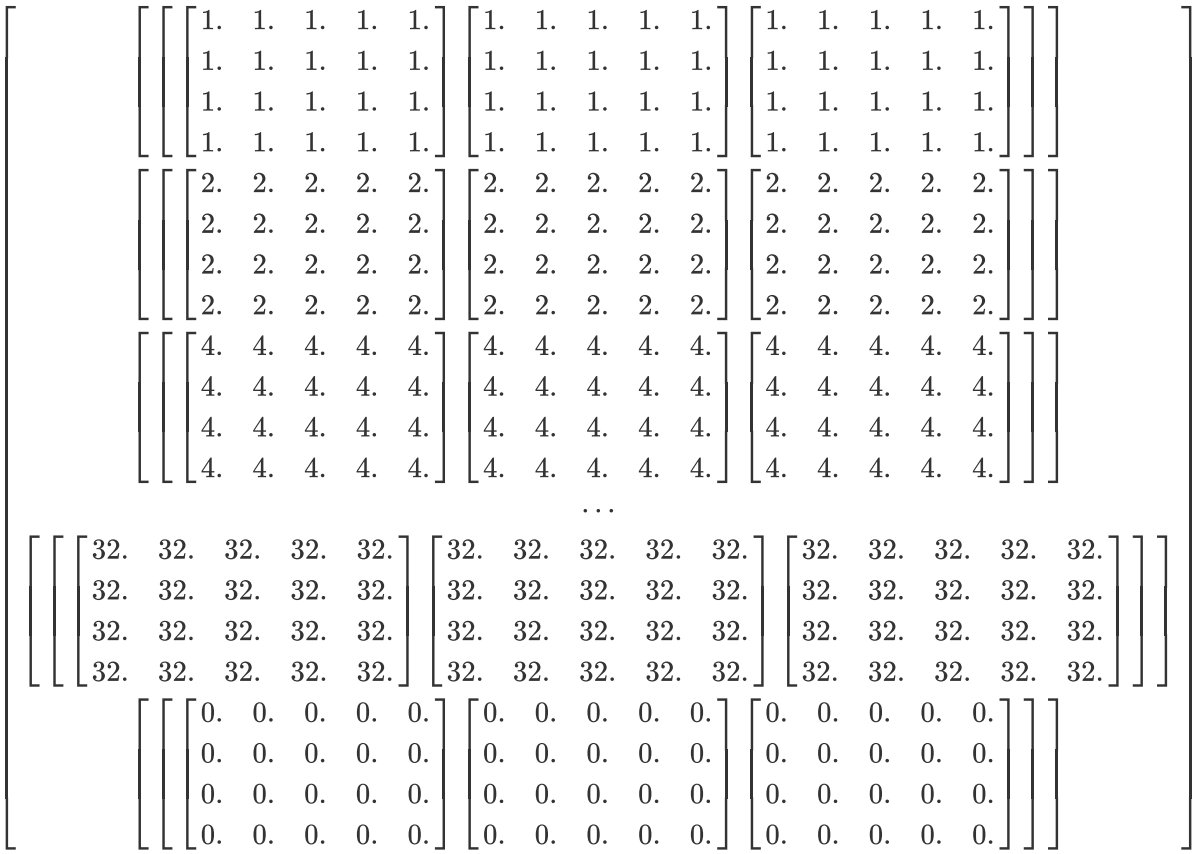
while 型循环，引发错误的一种写法

```
loop = network.add_loop()  # 替换上面 while 示例的 #---- 以内部分
rLayer = loop.add_recurrence(inputT0)
# Case 1:
_H1 = network.add_reduce(rLayer.get_output(0), trt.ReduceOperation.MAX, (1 << 0) + (1 << 1) + (1 << 2) +
(1 << 3), False)
_H2 = network.add_constant((), np.array([64], dtype=np.float32))
_H3 = network.add_elementwise(_H2.get_output(0), _H1.get_output(0), trt.ElementWiseOperation.SUB)
_H4 = network.add_activation(_H3.get_output(0), trt.ActivationType.RELU)
_H5 = network.add_identity(_H4.get_output(0))
_H5.get_output(0).dtype = trt.DataType.BOOL
_H6 = _H5
# Case 2:
'''
_H1 = network.add_slice(rLayer.get_output(0),[0,0,0,0],[1,1,1,1],[1,1,1,1])
_H2 = network.add_reduce(_H1.get_output(0),trt.ReduceOperation.MAX,(1<<0)+(1<<1)+(1<<2)+(1<<3),False)
_H3 = network.add_constant((),np.array([64],dtype=np.float32))
_H4 = network.add_elementwise(_H3.get_output(0),_H2.get_output(0),trt.ElementWiseOperation.SUB)
_H5 = network.add_activation(_H4.get_output(0),trt.ActivationType.RELU)
_H6 = network.add_identity(_H5.get_output(0))
_H6.get_output(0).dtype = trt.DataType.BOOL
'''
loop.add_trip_limit(_H6.get_output(0), trt.TripLimit.WHILE)

_H0 = network.add_elementwise(rLayer.get_output(0), rLayer.get_output(0), trt.ElementWiseOperation.SUM)
rLayer.set_input(1, _H0.get_output(0))

loopOutput0 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)
loopOutput1 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.CONCATENATE, 0)
lengthLayer = network.add_constant((), np.array([length], dtype=np.int32))
loopOutput1.set_input(1, lengthLayer.get_output(0))
```

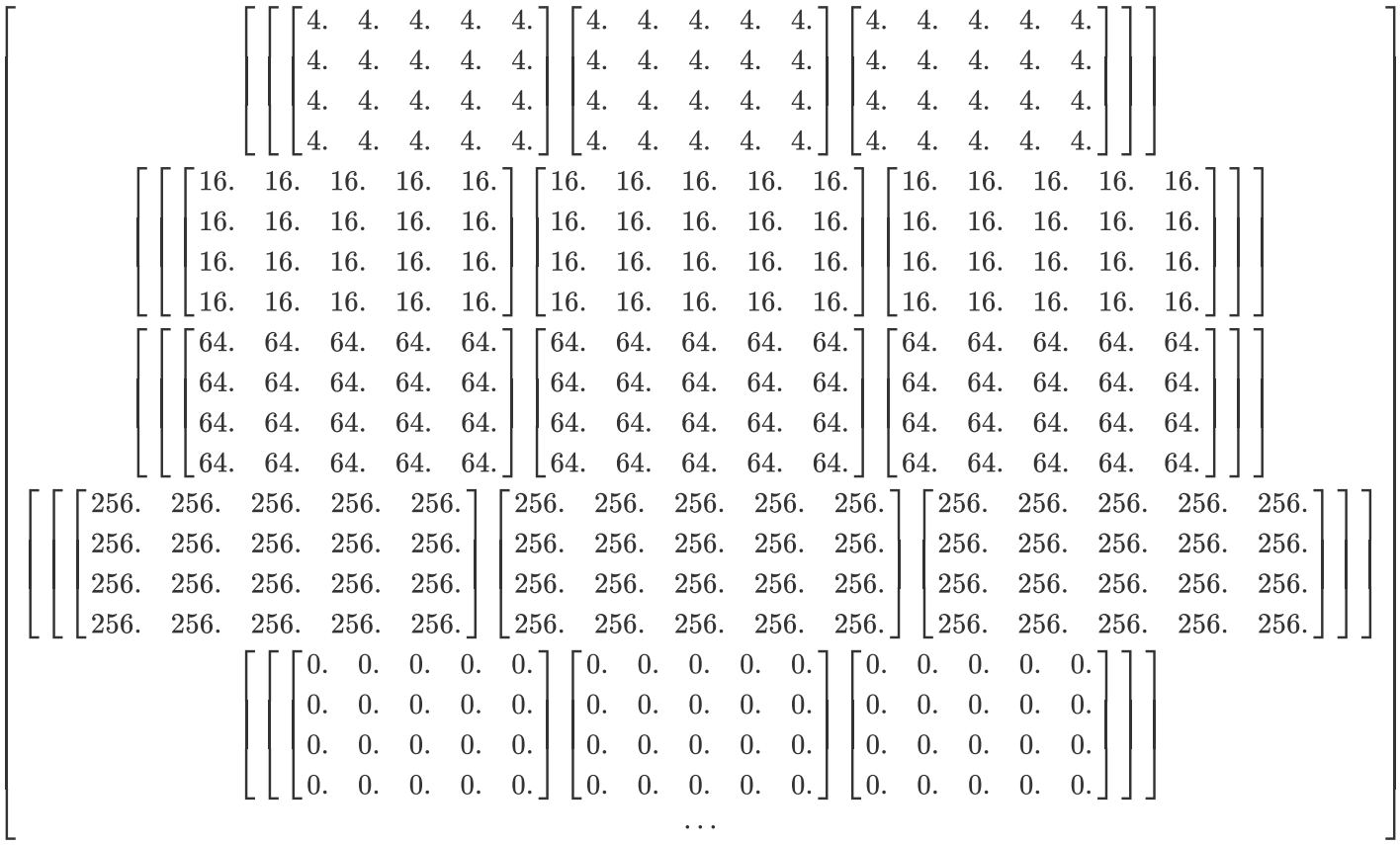
- 输出张量 0（loopOutput0）形状 (1,3,4,5)，循环最终的结果

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\\ 64. & 64. & 64. & 64. & 64.\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

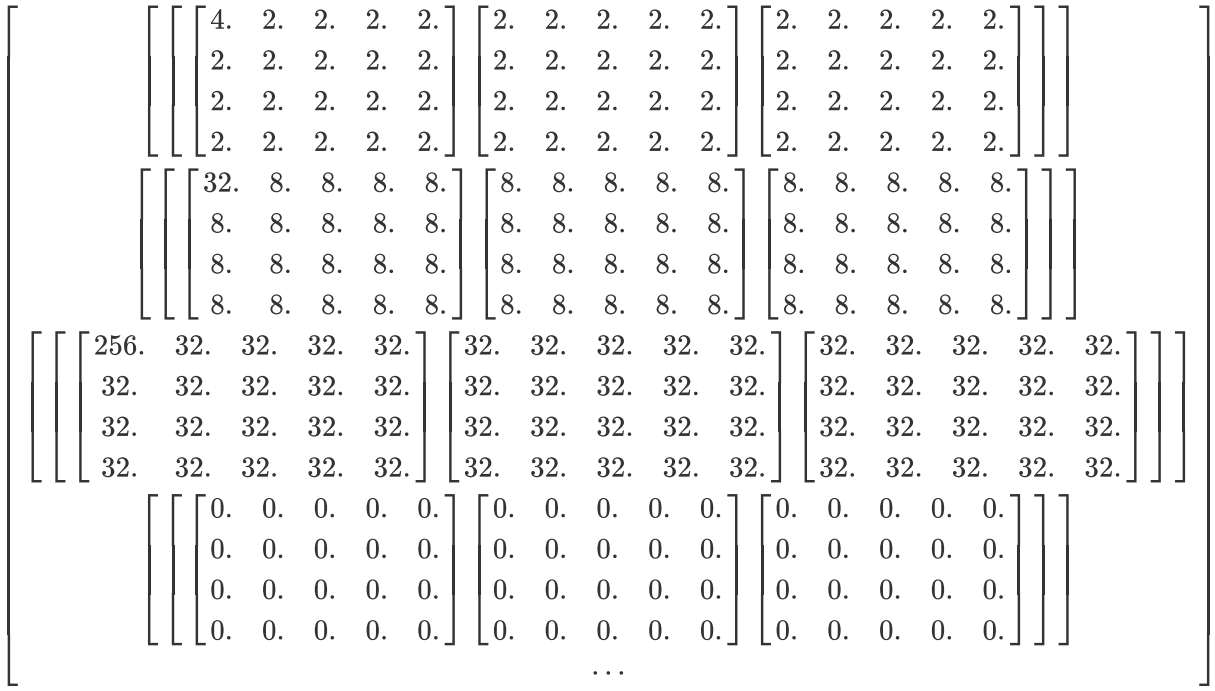
- Case 1 和 Case 2，在 loopOutput1 中传入 rLayer，输出张量 1（loopOutput1）形状 (7,1,3,4,5)，符合预期



- Case 1，在 loopOutput1 中传入 _H0，输出张量 1（loopOutput1）形状 (7,1,3,4,5)
- 结果是跳步保存的，原因是循环体是原地计算（in-place），在检验判断条件时会再把循环体再算一遍



- Case 2，在 loopOutput1 中传入 _H0，输出张量 1（loopOutput1）形状 (7,1,3,4,5)
- 结果是跳步保存的，且第一个元素参与了更多次运算，原因与 case 1 类似



iterator 迭代层

```
import numpy as np
from cuda import cudart
import tensorrt as trt

nIn, cIn, hIn, wIn = 1, 3, 4, 5
data = np.ones([nIn, cIn, hIn, wIn], dtype=np.float32) * np.arange(1, 1 + cIn,
dtype=np.float32).reshape(1, cIn, 1, 1)  # 输入数据

np.set_printoptions(precision=8, linewidth=200, suppress=True)
cudart.cudaDeviceSynchronize()

logger = trt.Logger(trt.Logger.ERROR)
builder = trt.Builder(logger)
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
config = builder.create_builder_config()
config.max_workspace_size = 1 << 30
inputT0 = network.add_input('inputT0', trt.DataType.FLOAT, (nIn, cIn, hIn, wIn))
#-------------------------------------------------------------------------------
loop = network.add_loop()
iteratorLayer = loop.add_iterator(inputT0, 1, False)  # 制造一个迭代器，在 C 维上每次正向抛出 1 层 (1,hIn,wIn)
iteratorLayer.axis = 1  # 重设抛出的轴号，最高维为 0，往地维递增
print(iteratorLayer.reverse)  # 是否反序抛出（见后面样例），仅用于输出不能修改，这里会在运行时输出 False

limit = network.add_constant((), np.array([cIn], dtype=np.int32))
loop.add_trip_limit(limit.get_output(0), trt.TripLimit.COUNT)

_H0 = network.add_constant([1, hIn, wIn], np.ones(hIn * wIn, dtype=np.float32))  # 首次循环前的循环体输入张
量，必须在循环外初始化好，这里相当于求和的初始值
rLayer = loop.add_recurrence(_H0.get_output(0))

_H1 = network.add_elementwise(rLayer.get_output(0), iteratorLayer.get_output(0),
trt.ElementWiseOperation.SUM)
rLayer.set_input(1, _H1.get_output(0))

loopOutput0 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)  # 只保留最后输出，
index 参数被忽略
```

```
loopOutput1 = loop.add_loop_output(_H1.get_output(0), trt.LoopOutput.CONCATENATE, 0)  # 保留所有中间输出，index 可以使用其他参数（例子见后面）
lengthLayer = network.add_constant((), np.array([cIn], dtype=np.int32))
loopOutput1.set_input(1, lengthLayer.get_output(0))
#--------------------------------------------------------------------------------
network.mark_output(loopOutput0.get_output(0))
network.mark_output(loopOutput1.get_output(0))
engineString = builder.build_serialized_network(network, config)
engine = trt.Runtime(logger).deserialize_cuda_engine(engineString)
context = engine.create_execution_context()
_, stream = cudart.cudaStreamCreate()

inputH0 = np.ascontiguousarray(data.reshape(-1))
outputH0 = np.empty(context.get_binding_shape(1), dtype=trt.nptype(engine.get_binding_dtype(1)))
outputH1 = np.empty(context.get_binding_shape(2), dtype=trt.nptype(engine.get_binding_dtype(2)))
_, inputD0 = cudart.cudaMallocAsync(inputH0.nbytes, stream)
_, outputD0 = cudart.cudaMallocAsync(outputH0.nbytes, stream)
_, outputD1 = cudart.cudaMallocAsync(outputH1.nbytes, stream)

cudart.cudaMemcpyAsync(inputD0, inputH0.ctypes.data, inputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
context.execute_async_v2([int(inputD0), int(outputD0), int(outputD1)], stream)
cudart.cudaMemcpyAsync(outputH0.ctypes.data, outputD0, outputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaMemcpyAsync(outputH1.ctypes.data, outputD1, outputH1.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaStreamSynchronize(stream)

print("inputH0 :", data.shape)
print(data)
print("outputH0:", outputH0.shape)
print(outputH0)
print("outputH1:", outputH1.shape)
print(outputH1)

cudart.cudaStreamDestroy(stream)
cudart.cudaFree(inputD0)
cudart.cudaFree(outputD0)
cudart.cudaFree(outputD1)
```

- 输入张量形状 (1,3,4,5)

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\\ 1. & 1. & 1. & 1. & 1.\end{bmatrix} & \begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix} & \begin{bmatrix}3. & 3. & 3. & 3. & 3.\\ 3. & 3. & 3. & 3. & 3.\\ 3. & 3. & 3. & 3. & 3.\\ 3. & 3. & 3. & 3. & 3.\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

- 输出张量 0（loopOutput0）形状 (1,4,5)，循环最终的结果

$$\begin{bmatrix}\begin{bmatrix}7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\end{bmatrix}\end{bmatrix}$$

- 输出张量 1（loopOutput1） 形状 (3,1,4,5)，在初始值 1 的基础上先加 1 再加 2 再加 3

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix}\end{bmatrix} & \begin{bmatrix}\begin{bmatrix}4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\end{bmatrix}\end{bmatrix} & \begin{bmatrix}\begin{bmatrix}7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

- 使用 REVERSE 模式（将 CONCATENATE 换成 REVERSE）输出张量 1（loopOutput1） 形状 (3,1,4,5)，相当于将 CONCATENATE 的结果在最高维上倒序

$$\left[\left[\begin{bmatrix}7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\end{bmatrix}\right]\left[\begin{bmatrix}4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\end{bmatrix}\right]\left[\begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix}\right]\right]$$

- 注意，可用的输出类型

tensorrt.LoopOutput 名	说明
LAST_VALUE	仅保留最后一个输出
CONCATENATE	保留指定长度的中间输出（从第一次循环向后）
REVERSE	保留执行长度的中间输出（从最后一次循环向前）

```
loop = network.add_loop()
iteratorLayer = loop.add_iterator(inputT0, 2, False)                                                                #
index 改成 2

limit = network.add_constant((),np.array([hIn],dtype=np.int32))                                                     # 循
环次数变为 hIn
loop.add_trip_limit(limit.get_output(0),trt.TripLimit.COUNT)

_H0 = network.add_constant([1,cIn,wIn],np.ones(cIn*wIn,dtype=np.float32))                                            # 循
环体输入张量，尺寸变为 [1,cIn,wIn]
rLayer = loop.add_recurrence(_H0.get_output(0))

_H1 =
network.add_elementwise(rLayer.get_output(0),iteratorLayer.get_output(0),trt.ElementWiseOperation.SUM)
rLayer.set_input(1, _H1.get_output(0))

loopOutput0 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)

loopOutput1 = loop.add_loop_output(_H1.get_output(0), trt.LoopOutput.CONCATENATE, 0)
lengthLayer = network.add_constant((),np.array([hIn],dtype=np.int32))                                               # 保
存长度变为 hIn
loopOutput1.set_input(1,lengthLayer.get_output(0))
```

- 输出张量 0（loopOutput0）形状 (1,3,5)，循环最终的结果

$$\left[\left[\begin{bmatrix}5. & 5. & 5. & 5. & 5.\\ 9. & 9. & 9. & 9. & 9.\\ 13. & 13. & 13. & 13. & 13.\end{bmatrix}\right]\right]$$

- 输出张量 1（loopOutput1） 形状 (4,1,3,5)，在初始值 1 的基础上分别依次加 1 或者加 2 或者加 3

$$\left[\begin{array}{c} \left[\left[\begin{array}{ccccc} 2. & 2. & 2. & 2. & 2. \\ 3. & 3. & 3. & 3. & 3. \\ 4. & 4. & 4. & 4. & 4. \end{array}\right]\right] \\ \left[\left[\begin{array}{ccccc} 3. & 3. & 3. & 3. & 3. \\ 5. & 5. & 5. & 5. & 5. \\ 7. & 7. & 7. & 7. & 7. \end{array}\right]\right] \\ \left[\left[\begin{array}{ccccc} 4. & 4. & 4. & 4. & 4. \\ 7. & 7. & 7. & 7. & 7. \\ 10. & 10. & 10. & 10. & 10. \end{array}\right]\right] \\ \left[\left[\begin{array}{ccccc} 5. & 5. & 5. & 5. & 5. \\ 9. & 9. & 9. & 9. & 9. \\ 13. & 13. & 13. & 13. & 13. \end{array}\right]\right] \end{array}\right]$$

```
loop = network.add_loop()
iteratorLayer = loop.add_iterator(inputT0, 3, False)  # index 改成 3

limit = network.add_constant((), np.array([wIn], dtype=np.int32))  # 循环次数变为 wIn
loop.add_trip_limit(limit.get_output(0), trt.TripLimit.COUNT)

_H0 = network.add_constant([1, cIn, hIn], np.ones(cIn * hIn, dtype=np.float32))  # 循环体输入张量，尺寸变为[1,cIn,wIn]
rLayer = loop.add_recurrence(_H0.get_output(0))

_H1 = network.add_elementwise(rLayer.get_output(0), iteratorLayer.get_output(0), trt.ElementWiseOperation.SUM)
rLayer.set_input(1, _H1.get_output(0))

loopOutput0 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)

loopOutput1 = loop.add_loop_output(_H1.get_output(0), trt.LoopOutput.CONCATENATE, 0)
lengthLayer = network.add_constant((), np.array([wIn], dtype=np.int32))  # 保存长度变为 wIn
loopOutput1.set_input(1, lengthLayer.get_output(0))
```

- 输出张量 0（loopOutput0）形状 (1,3,4)，循环最终的结果

$$\left[\left[\begin{array}{cccc} 6. & 6. & 6. & 6. \\ 11. & 11. & 11. & 11. \\ 16. & 16. & 16. & 16. \end{array}\right]\right]$$

- 输出张量 1（loopOutput1） 形状 (5,1,3,4)，在初始值 1 的基础上分别依次加 1 或者加 2 或者加 3

$$\left[\begin{array}{c} \left[\left[\begin{array}{cccc} 2. & 2. & 2. & 2. \\ 3. & 3. & 3. & 3. \\ 4. & 4. & 4. & 4. \end{array}\right]\right] \\ \left[\left[\begin{array}{cccc} 3. & 3. & 3. & 3. \\ 5. & 5. & 5. & 5. \\ 7. & 7. & 7. & 7. \end{array}\right]\right] \\ \left[\left[\begin{array}{cccc} 4. & 4. & 4. & 4. \\ 7. & 7. & 7. & 7. \\ 10. & 10. & 10. & 10. \end{array}\right]\right] \\ \left[\left[\begin{array}{cccc} 6. & 6. & 6. & 6. \\ 11. & 11. & 11. & 11. \\ 16. & 16. & 16. & 16. \end{array}\right]\right] \end{array}\right]$$

```
loop = network.add_loop()
iteratorLayer = loop.add_iterator(inputT0, 1, True)  # 在 C 维上使用反抛迭代器
```

```
limit = network.add_constant((), np.array([cIn], dtype=np.int32))
loop.add_trip_limit(limit.get_output(0), trt.TripLimit.COUNT)

_H0 = network.add_constant([1, hIn, wIn], np.ones(hIn * wIn, dtype=np.float32))
rLayer = loop.add_recurrence(_H0.get_output(0))

_H1 = network.add_elementwise(rLayer.get_output(0), iteratorLayer.get_output(0),
trt.ElementWiseOperation.SUM)
rLayer.set_input(1, _H1.get_output(0))

loopOutput0 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)
loopOutput1 = loop.add_loop_output(_H1.get_output(0), trt.LoopOutput.CONCATENATE, 0)
lengthLayer = network.add_constant((), np.array([cIn], dtype=np.int32))
loopOutput1.set_input(1, lengthLayer.get_output(0))
```

- 输出张量 0（loopOutput0）形状 (1,4,5)，循环最终的结果

$$\begin{bmatrix}\begin{bmatrix}7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\end{bmatrix}\end{bmatrix}$$

- 输出张量 1（loopOutput1） 形状 (3,1,4,5)，在初始值 1 的基础上先加 3 再加 2 再加 1，REVERSE 输出不再展示

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\end{bmatrix}\end{bmatrix} \begin{bmatrix}\begin{bmatrix}6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\\ 6. & 6. & 6. & 6. & 6.\end{bmatrix}\end{bmatrix} \begin{bmatrix}\begin{bmatrix}7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

```
loop = network.add_loop()
iteratorLayer = loop.add_iterator(inputT0, 1, False)

limit = network.add_constant((), np.array([cIn], dtype=np.int32))
loop.add_trip_limit(limit.get_output(0), trt.TripLimit.COUNT)

_H0 = network.add_constant([1, hIn, wIn], np.ones(hIn * wIn, dtype=np.float32))
rLayer = loop.add_recurrence(_H0.get_output(0))

_H1 = network.add_elementwise(rLayer.get_output(0), iteratorLayer.get_output(0),
trt.ElementWiseOperation.SUM)
rLayer.set_input(1, _H1.get_output(0))

lengthLayer = network.add_constant((), np.array([cIn], dtype=np.int32))
loopOutput0 = loop.add_loop_output(_H1.get_output(0), trt.LoopOutput.CONCATENATE, 1)  # 修改 index 参数，仅展示 CONTENATE 和 REVERSE 模式的结果，因为 LAST_VALUE 模式中该参数被忽略
loopOutput0.set_input(1, lengthLayer.get_output(0))
loopOutput1 = loop.add_loop_output(_H1.get_output(0), trt.LoopOutput.REVERSE, 1)
loopOutput1.set_input(1, lengthLayer.get_output(0))
```

- 输出张量 0（loopOutput0）形状 (1,3,4,5)，结果同 iterator 迭代层初始示例代码，但是结果在次高维上进行连接

$$\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\\ 2. & 2. & 2. & 2. & 2.\end{bmatrix} \begin{bmatrix}4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\\ 4. & 4. & 4. & 4. & 4.\end{bmatrix} \begin{bmatrix}7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\\ 7. & 7. & 7. & 7. & 7.\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

- 输出张量 1（loopOutput1） 形状 (1,3,4,5)，结果为输出张量 0 的倒序

$$\left[\left[\left[\begin{bmatrix} 7. & 7. & 7. & 7. & 7. \\ 7. & 7. & 7. & 7. & 7. \\ 7. & 7. & 7. & 7. & 7. \\ 7. & 7. & 7. & 7. & 7. \end{bmatrix} \begin{bmatrix} 4. & 4. & 4. & 4. & 4. \\ 4. & 4. & 4. & 4. & 4. \\ 4. & 4. & 4. & 4. & 4. \\ 4. & 4. & 4. & 4. & 4. \end{bmatrix} \begin{bmatrix} 2. & 2. & 2. & 2. & 2. \\ 2. & 2. & 2. & 2. & 2. \\ 2. & 2. & 2. & 2. & 2. \\ 2. & 2. & 2. & 2. & 2. \end{bmatrix}\right]\right]\right]$$