6.TensorRT 进阶用法

6.1. The Timing Cache

为了减少构建器时间，TensorRT 创建了一个层时序缓存，以在构建器阶段保存层分析信息。它包含的信息特定于目标构建器设备、CUDA 和 TensorRT 版本，以及可以更改层实现的 `BuilderConfig` 参数，例如`BuilderFlag::kTF32`或`BuilderFlag::kREFIT`。

如果有其他层具有相同的输入/输出张量配置和层参数，则 TensorRT 构建器会跳过分析并重用重复层的缓存结果。如果缓存中的计时查询未命中，则构建器会对该层计时并更新缓存。

时序缓存可以被序列化和反序列化。您可以通过`IBuilderConfig::createTimingCache`从缓冲区加载序列化缓存：

```
ITimingCache* cache =
 config->createTimingCache(cacheFile.data(), cacheFile.size());
```

将缓冲区大小设置为0会创建一个新的空时序缓存。

然后，在构建之前将缓存附加到构建器配置。

```
config->setTimingCache(*cache, false);
```

在构建期间，由于缓存未命中，时序缓存可以增加更多信息。在构建之后，它可以被序列化以与另一个构建器一起使用。

```
IHostMemory* serializedCache = cache->serialize();
```

如果构建器没有附加时间缓存，构建器会创建自己的临时本地缓存并在完成时将其销毁。

缓存与算法选择不兼容（请参阅算法选择和可重现构建部分）。可以通过设置`BuilderFlag`来禁用它。

6.2. Refitting An Engine

TensorRT 可以用新的权重改装引擎而无需重建它，但是，在构建时必须指定这样做的选项：

```
...
config->setFlag(BuilderFlag::kREFIT)
builder->buildSerializedNetwork(network, config);
```

稍后，您可以创建一个`Refitter`对象：

```
ICudaEngine* engine = ...;
IRefitter* refitter = createInferRefitter(*engine,gLogger)
```

然后更新权重。例如，要更新名为“`MyLayer`”的卷积层的内核权重：

```
Weights newWeights = ...;
refitter->setWeights("MyLayer",WeightsRole::kKERNEL,
                     newWeights);
```

新的权重应该与用于构建引擎的原始权重具有相同的计数。如果出现问题，例如错误的层名称或角色或权重计数发生变化，`setWeights` 返回 `false`。

由于引擎优化的方式，如果您更改一些权重，您可能还必须提供一些其他权重。该界面可以告诉您需要提供哪些额外的权重。

您可以使用 `INetworkDefinition::setWeightsName()` 在构建时命名权重 - ONNX 解析器使用此 API 将权重与 ONNX 模型中使用的名称相关联。然后，您可以使用 `setNamedWeights` 更新权重：

```
Weights newWeights = ...;
refitter->setNamedWeights("MyWeights", newWeights);
```

`setNamedWeights` 和 `setWeights` 可以同时使用，即，您可以通过 `setNamedWeights` 更新具有名称的权重，并通过 `setWeights` 更新那些未命名的权重。

这通常需要两次调用 `IRefitter::getMissing`，首先获取必须提供的权重对象的数量，然后获取它们的层和角色。

```
const int32_t n = refitter->getMissing(0, nullptr, nullptr);
std::vector<const char*> layerNames(n);
std::vector<WeightsRole> weightsRoles(n);
refitter->getMissing(n, layerNames.data(),
                        weightsRoles.data());
```

或者，要获取所有缺失权重的名称，请运行：

```
const int32_t n = refitter->getMissingWeights(0, nullptr);
std::vector<const char*> weightsNames(n);
refitter->getMissingWeights(n, weightsNames.data());
```

您可以按任何顺序提供缺失的权重：

```
for (int32_t i = 0; i < n; ++i)
    refitter->setWeights(layerNames[i], weightsRoles[i],
                         Weights{...});
```

返回的缺失权重集是完整的，从某种意义上说，仅提供缺失的权重不会产生对任何更多权重的需求。

提供所有权重后，您可以更新引擎：

```
bool success = refitter->refitCudaEngine();
assert(success);
```

如果 `refit` 返回 `false`，请检查日志以获取诊断信息，可能是关于仍然丢失的权重。
然后，您可以删除 `refitter`：

```
delete refitter;
```

更新后的引擎的行为就像它是从使用新权重更新的网络构建的一样。

要查看引擎中的所有可改装权重，请使用 `refitter->getAll(...)` 或 `refitter->getAllWeights(...)` ；类似于上面使用 `getMissing` 和 `getMissingWeights` 的方式。

6.3. Algorithm Selection and Reproducible Builds

TensorRT 优化器的默认行为是选择全局最小化引擎执行时间的算法。它通过定时每个实现来做到这一点，有时，当实现具有相似的时间时，系统噪声可能会决定在构建器的任何特定运行中将选择哪个。不同的实现通常会使用不同的浮点值累加顺序，两种实现可能使用不同的算法，甚至以不同的精度运行。因此，构建器的不同调用通常不会导致引擎返回位相同的结果。

有时，确定性构建或重新创建早期构建的算法选择很重要。通过提供 `IAlgorithmSelector` 接口的实现并使用 `setAlgorithmSelector` 将其附加到构建器配置，您可以手动指导算法选择。

方法 `IAlgorithmSelector::selectAlgorithms` 接收一个 `AlgorithmContext` ，其中包含有关层算法要求的信息，以及一组满足这些要求的算法选择。它返回 TensorRT 应该为层考虑的算法集。

构建器将从这些算法中选择一种可以最小化网络全局运行时间的算法。如果未返回任何选项并且 `BuilderFlag::kREJECT_EMPTY_ALGORITHMS` 未设置，则 TensorRT 将其解释为意味着任何算法都可以用于该层。要覆盖此行为并在返回空列表时生成错误，请设置 `BuilderFlag::kREJECT_EMPTY_ALGORITHMSS` 标志。

在 TensorRT 完成对给定配置文件的网络优化后，它会调用 `reportAlgorithms` ，它可用于记录为每一层做出的最终选择。

`selectAlgorithms` 返回一个选项。要重播早期构建中的选择，请使用 `reportAlgorithms` 记录该构建中的选择，并在 `selectAlgorithms` 中返回它们。

`sampleAlgorithmSelector` 演示了如何使用算法选择器在构建器中实现确定性和可重复性。

注意：

- 算法选择中的“层”概念与 `INetworkDefinition` 中的 `ILayer` 不同。由于融合优化，前者中的“层”可以等同于多个网络层的集合。
- `selectAlgorithms` 中选择最快的算法可能不会为整个网络产生最佳性能，因为它可能会增加重新格式化的开销。
- 如果 TensorRT 发现该层是空操作，则 `IAlgorithm` 的时间在 `selectAlgorithms` 中为0 。
- `reportAlgorithms` 不提供提供给 `selectAlgorithms` 的 `IAlgorithm` 的时间和工作空间信息。

6.4. Creating A Network Definition From Scratch

除了使用解析器，您还可以通过网络定义 API 将网络直接定义到 TensorRT。此场景假设每层权重已在主机内存中准备好在网络创建期间传递给 TensorRT。

以下示例创建了一个简单的网络，其中包含 `Input` 、 `Convolution` 、 `Pooling` 、 `MatrixMultiply` 、 `Shuffle` 、 `Activation` 和 `Softmax` 层。

6.4.1. C++

本节对应的代码可以在 `sampleMNISTAPI` 中找到。在此示例中，权重被加载到以下代码中使用的 `weightMap` 数据结构中。

首先创建构建器和网络对象。请注意，在以下示例中，记录器通过所有 C++ 示例通用的 `logger.cpp` 文件进行初始化。 C++ 示例帮助程序类和函数可以在 `common.h` 头文件中找到。

```
    auto builder = SampleUniquePtr<nvinfer1::IBuilder>
(nvinfer1::createInferBuilder(sample::gLogger.getTRTLogger()));
    const auto explicitBatchFlag = 1U << static_cast<uint32_t>
(nvinfer1::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
    auto network = SampleUniquePtr<nvinfer1::INetworkDefinition>(builder-
>createNetworkV2(explicitBatchFlag));
```

`kEXPLICIT_BATCH` 标志的更多信息，请参阅显式与隐式批处理部分。

通过指定输入张量的名称、数据类型和完整维度，将输入层添加到网络。一个网络可以有多个输入，尽管在这个示例中只有一个：

```
auto data = network->addInput(INPUT_BLOB_NAME, datatype, Dims4{1, 1, INPUT_H,
INPUT_W});
```

添加带有隐藏层输入节点、步幅和权重的卷积层，用于过滤器和偏差。

```
auto conv1 = network->addConvolution(
*data->getOutput(0), 20, DimsHW{5, 5}, weightMap["conv1filter"],
weightMap["conv1bias"]);
conv1->setStride(DimsHW{1, 1});
```

注意：传递给 TensorRT 层的权重在主机内存中。

添加池化层；请注意，前一层的输出作为输入传递。

```
auto pool1 = network->addPooling(*conv1->getOutput(0), PoolingType::kMAX,
DimsHW{2, 2});
pool1->setStride(DimsHW{2, 2});
```

添加一个 Shuffle 层来重塑输入，为矩阵乘法做准备：

```
int32_t const batch = input->getDimensions().d[0];
int32_t const mmInputs = input.getDimensions().d[1] * input.getDimensions().d[2]
* input.getDimensions().d[3];
auto inputReshape = network->addShuffle(*input);
inputReshape->setReshapeDimensions(Dims{2, {batch, mmInputs}});
```

现在，添加一个 `MatrixMultiply` 层。在这里，模型导出器提供了转置权重，因此为这些权重指定了 `kTRANSPOSE` 选项。

```
IConstantLayer* filterConst = network->addConstant(Dims{2, {nbOutputs,
mmInputs}}, mWeightMap["ip1filter"]);
auto mm = network->addMatrixMultiply(*inputReshape->getOutput(0),
MatrixOperation::kNONE, *filterConst->getOutput(0),
MatrixOperation::kTRANSPOSE);
```

添加偏差，它将在批次维度上广播。

```
auto biasConst = network->addConstant(Dims{2, {1, nbOutputs}},
mWeightMap["ip1bias"]);
auto biasAdd = network->addElementWise(*mm->getOutput(0), *biasConst-
>getOutput(0), ElementWiseOperation::kSUM);
```

添加 ReLU 激活层：

```
auto relu1 = network->addActivation(*ip1->getOutput(0), ActivationType::kRELU);
```

添加 SoftMax 层以计算最终概率：

```
auto prob = network->addSoftMax(*relu1->getOutput(0));
```

为 SoftMax 层的输出添加一个名称，以便在推理时可以将张量绑定到内存缓冲区：

```
prob->getOutput(0)->setName(OUTPUT_BLOB_NAME);
```

将其标记为整个网络的输出：

```
network->markOutput(*prob->getOutput(0));
```

代表 MNIST 模型的网络现已完全构建。请参阅构建引擎和反序列化文件部分，了解如何构建引擎并使用此网络运行推理。

6.4.2. Python

此部分对应的代码可以在network_api_pytorch_mnist中找到。

这个例子使用一个帮助类来保存一些关于模型的元数据：

```
class ModelData(object):
    INPUT_NAME = "data"
    INPUT_SHAPE = (1, 1, 28, 28)
    OUTPUT_NAME = "prob"
    OUTPUT_SIZE = 10
    DTYPE = trt.float32
```

在此示例中，权重是从 Pytorch MNIST 模型导入的。

```
weights = mnist_model.get_weights()
```

创建记录器、构建器和网络类。

```
TRT_LOGGER = trt.Logger(trt.Logger.ERROR)
builder = trt.Builder(TRT_LOGGER)
EXPLICIT_BATCH = 1 << (int)(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
network = builder.create_network(common.EXPLICIT_BATCH)
```

`kEXPLICIT_BATCH` 标志的更多信息，请参阅显式与隐式批处理部分。

接下来，为网络创建输入张量，指定张量的名称、数据类型和形状。

```
input_tensor = network.add_input(name=ModelData.INPUT_NAME,
dtype=ModelData.DTYPE, shape=ModelData.INPUT_SHAPE)
```

添加一个卷积层，指定输入、输出图的数量、内核形状、权重、偏差和步幅：

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

添加一个池化层，指定输入（前一个卷积层的输出）、池化类型、窗口大小和步幅：

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

添加下一对卷积和池化层：

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

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

添加一个 Shuffle 层来重塑输入，为矩阵乘法做准备：

```
batch = input.shape[0]
mm_inputs = np.prod(input.shape[1:])
input_reshape = net.add_shuffle(input)
input_reshape.reshape_dims = trt.Dims2(batch, mm_inputs)
```

现在，添加一个 `MatrixMultiply` 层。在这里，模型导出器提供了转置权重，因此为这些权重指定了 `kTRANSPOSE` 选项。

```
filter_const = net.add_constant(trt.Dims2(nbOutputs, k),
weights["fc1.weight"].numpy())
mm = net.add_matrix_multiply(input_reshape.get_output(0),
trt.MatrixOperation.NONE, filter_const.get_output(0),
trt.MatrixOperation.TRANSPOSE);
```

添加将在批次维度广播的偏差添加：

```
bias_const = net.add_constant(trt.Dims2(1, nbOutputs),
weights["fc1.bias"].numpy())
bias_add = net.add_elementwise(mm.get_output(0), bias_const.get_output(0),
trt.ElementWiseOperation.SUM)
```

添加 Relu 激活层：

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

添加最后的全连接层，并将该层的输出标记为整个网络的输出：

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

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

代表 MNIST 模型的网络现已完全构建。请参阅构建引擎和执行推理部分，了解如何构建引擎并使用此网络运行推理。

6.5. Reduced Precision

6.5.1. Network-level Control of Precision

默认情况下，TensorRT 以 32 位精度工作，但也可以使用 16 位浮点和 8 位量化浮点执行操作。使用较低的精度需要更少的内存并实现更快的计算。

降低精度支持取决于您的硬件（请参阅NVIDIA TensorRT 支持矩阵中的硬件和精度部分）。您可以查询构建器以检查平台上支持的精度支持：
C++

```
if (builder->platformHasFastFp16()) { … };
```

Python

```
if builder.platform_has_fp16:
```

在构建器配置中设置标志会通知 TensorRT 它可能会选择较低精度的实现：
C++

```
config->setFlag(BuilderFlag::kFP16);
```

Python

```
config.set_flag(trt.BuilderFlag.FP16)
```

共有三个精度标志：`FP16`、`INT8` 和 `TF32`，它们可以独立启用。请注意，如果 TensorRT 导致整体运行时间较短，或者不存在低精度实现，TensorRT 仍将选择更高精度的内核。

当 TensorRT 为层选择精度时，它会根据需要自动转换权重以运行层。

sampleGoogleNet和sampleMNIST提供了使用这些标志的示例。

虽然使用 `FP16` 和 `TF32` 精度相对简单，但使用 `INT8` 时会有额外的复杂性。有关详细信息，请参阅使用 INT8章节。

6.5.2. Layer-level Control of Precision

`builder-flags` 提供了允许的、粗粒度的控制。然而，有时网络的一部分需要更高的动态范围或对数值精度敏感。您可以限制每层的输入和输出类型：

C++

```
layer->setPrecision(DataType::kFP16)
```

Python

```
layer.precision = trt.fp16
```

这为输入和输出提供了首选类型（此处为D`ataType::kFP16`）。

您可以进一步设置图层输出的首选类型：

C++

```
layer->setOutputType(out_tensor_index, DataType::kFLOAT)
```

Python

```
layer.set_output_type(out_tensor_index, trt.fp16)
```

计算将使用与输入首选相同的浮点类型。大多数 TensorRT 实现具有相同的输入和输出浮点类型；但是，`Convolution`、`Deconvolution` 和 `FullyConnected` 可以支持量化的 `INT8` 输入和未量化的 `FP16` 或 `FP32` 输出，因为有时需要使用来自量化输入的更高精度输出来保持准确性。

设置精度约束向 TensorRT 提示它应该选择一个输入和输出与首选类型匹配的层实现，如果前一层的输出和下一层的输入与请求的类型不匹配，则插入重新格式化操作。请注意，只有通过构建器配置中的标志启用了这些类型，TensorRT 才能选择具有这些类型的实现。

默认情况下，TensorRT 只有在产生更高性能的网络时才会选择这样的实现。如果另一个实现更快，TensorRT 会使用它并发出警告。您可以通过首选构建器配置中的类型约束来覆盖此行为。

C++

```
config->setFlag(BuilderFlag::kPREFER_PRECISION_CONSTRAINTS)
```

Python

```
config.set_flag(trt.BuilderFlag.PREFER_PRECISION_CONSTRAINTS)
```

如果约束是首选的，TensorRT 会服从它们，除非没有具有首选精度约束的实现，在这种情况下，它会发出警告并使用最快的可用实现。

要将警告更改为错误，请使用`OBEY`而不是`PREFER`：

C++

```
config->setFlag(BuilderFlag::kOBEY_PRECISION_CONSTRAINTS);
```

Python

```
config.set_flag(trt.BuilderFlag.OBEY_PRECISION_CONSTRAINTS);
```

sampleINT8API说明了使用这些 API 降低精度。

精度约束是可选的 - 您可以查询以确定是否已使用C++ 中的`layer->precisionIsSet()`或 Python 中的`layer.precision_is_set`设置了约束。如果没有设置精度约束，那么从 C++ 中的`layer->getPrecision()`返回的结果，或者在 Python 中读取精度属性，是没有意义的。输出类型约束同样是可选的。

`layer->getOutput(i)->setType()`和`layer->setOutputType()`之间存在区别——前者是一种可选类型，它限制了 TensorRT 将为层选择的实现。后者是强制性的（默认为 FP32）并指定网络输出的类型。如果它们不同，TensorRT 将插入一个强制转换以确保两个规范都得到尊重。因此，如果您为产生网络输出的层调用`setOutputType()`，通常还应该将相应的网络输出配置为具有相同的类型。

6.5.3. Enabling TF32 Inference Using C++

TensorRT 默认允许使用 TF32 Tensor Cores。在计算内积时，例如在卷积或矩阵乘法期间，TF32 执行执行以下操作：

- 将 FP32 被乘数舍入到 FP16 精度，但保持 FP32 动态范围。
- 计算四舍五入的被乘数的精确乘积。
- 将乘积累加到 FP32 总和中。

TF32 Tensor Cores 可以使用 FP32 加速网络，通常不会损失准确性。对于需要高动态范围的权重或激活的模型，它比 FP16 更强大。

不能保证 TF32 Tensor Cores 会被实际使用，也没有办法强制实现使用它们 - TensorRT 可以随时回退到 FP32，如果平台不支持 TF32，则总是回退。但是，您可以通过清除 TF32 builder 标志来禁用它们。

C++

config->clearFlag(BuilderFlag::kTF32);

Python

config.clear_flag(trt.BuilderFlag.TF32)

尽管设置了`BuilderFlag::kTF32`，但在构建引擎时设置环境变量`NVIDIA_TF32_OVERRIDE=0`会禁用`TF32`。此环境变量在设置为0时会覆盖 NVIDIA 库的任何默认值或编程配置，因此它们永远不会使用 **TF32 Tensor Cores** 加速 FP32 计算。这只是一个调试工具，NVIDIA 库之外的任何代码都不应更改基于此环境变量的行为。除0以外的任何其他设置都保留供将来使用。

警告：在引擎运行时将环境变量`NVIDIA_TF32_OVERRIDE`设置为不同的值可能会导致无法预测的精度/性能影响。引擎运转时最好不要设置。

注意：除非您的应用程序需要 TF32 提供的更高动态范围，否则 `FP16` 将是更好的解决方案，因为它几乎总能产生更快的性能。

6.6. I/O Formats

TensorRT 使用许多不同的数据格式优化网络。为了允许在 TensorRT 和客户端应用程序之间有效传递数据，这些底层数据格式在网络 I/O 边界处公开，即用于标记为网络输入或输出的张量，以及在将数据传入和传出插件时。对于其他张量，TensorRT 选择导致最快整体执行的格式，并可能插入重新格式化以提高性能。

您可以通过分析可用的 I/O 格式以及对 TensorRT 之前和之后的操作最有效的格式来组装最佳数据管道。

要指定 I/O 格式，请以位掩码的形式指定一种或多种格式。
以下示例将输入张量格式设置为`TensorFormat::kHWC8`。请注意，此格式仅适用于`DataType::kHALF`，因此必须相应地设置数据类型。

C++

```
auto formats = 1U << TensorFormat::kHWC8;
network->getInput(0)->setAllowedFormats(formats);
network->getInput(0)->setType(DataType::kHALF);
```

Python

```
formats = 1 << int(tensorrt.TensorFormat.HWC8)
network.get_input(0).allowed_formats = formats
network.get_input(0).dtype = tensorrt.DataType.HALF
```

通过设置构建器配置标志`DIRECT_IO`，可以使 TensorRT 避免在网络边界插入重新格式化。这个标志通常会适得其反，原因有两个：

- 如果允许 TensorRT 插入重新格式化，则生成的引擎可能会更慢。重新格式化可能听起来像是浪费工作，但它可以允许最有效的内核耦合。
- 如果 TensorRT 在不引入此类重新格式化的情况下无法构建引擎，则构建将失败。故障可能仅发生在某些目标平台上，因为这些平台的内核支持哪些格式。

该标志的存在是为了希望完全控制重新格式化是否发生在 I/O 边界的用户，例如构建仅在 DLA 上运行而不回退到 GPU 进行重新格式化的引擎。

sampleIOFormats说明了如何使用 C++ 指定 IO 格式。

下表显示了支持的格式。

Table 1. Supported I/O formats

Format	kINT32	kFLOAT	kHALF	kINT8
kLINEAR	Only for GPU	Supported	Supported	Supported
kCHW2	N/A	N/A	Only for GPU	N/A
kCHW4	N/A	N/A	Supported	Supported
kHWC8	N/A	N/A	Only for GPU	N/A
kCHW16	N/A	N/A	Supported	N/A
kCHW32	N/A	Only for GPU	Only for GPU	Supported
kDHWC8	N/A	N/A	Only for GPU	N/A
kCDHW32	N/A	N/A	Only for GPU	Only for GPU
kHWC	N/A	Only for GPU	N/A	N/A
kDLA_LINEAR	N/A	N/A	Only for DLA	Only for DLA
kDLA_HWC4	N/A	N/A	Only for DLA	Only for DLA
kHWC16	N/A	N/A	Only for NVIDIA Ampere GPUs and later	N/A

请注意，对于矢量化格式，通道维度必须补零到矢量大小的倍数。例如，如果输入绑定的维度为[16,3,224,224]、`kHALF`数据类型和`kHWC8`格式，则绑定缓冲区的实际所需大小将为1`6* 8 *224*224*sizeof(half)`字节，甚至尽管`engine->getBindingDimension()` API 将张量维度返回为`[16,3,224,224]`。填充部分中的值（即本例中的C=3,4,...,7）必须用零填充。

有关这些格式的数据在内存中的实际布局方式，请参阅数据格式说明。

6.7.Compatibility of Serialized Engines

仅当与用于序列化引擎的相同操作系统、CPU 架构、GPU 模型和 TensorRT 版本一起使用时，序列化引擎才能保证正常工作。

TensorRT 检查引擎的以下属性，如果它们与引擎被序列化的环境不匹配，将无法反序列化：

- TensorRT 的主要、次要、补丁和构建版本
- 计算能力（主要和次要版本）

这确保了在构建阶段选择的内核存在并且可以运行。此外，TensorRT 用于从 cuDNN 和 cuBLAS 中选择和配置内核的 API 不支持跨设备兼容性，因此在构建器配置中禁用这些策略源的使用。
TensorRT 还会检查以下属性，如果它们不匹配，则会发出警告：

- 全局内存总线带宽
- 二级缓存大小
- 每个块和每个多处理器的最大共享内存
- 纹理对齐要求
- 多处理器数量

- GPU 设备是集成的还是分立的

如果引擎序列化和运行时系统之间的 GPU 时钟速度不同，则从序列化系统中选择的策略对于运行时系统可能不是最佳的，并且可能会导致一些性能下降。

如果反序列化过程中可用的设备内存小于序列化过程中的数量，反序列化可能会由于内存分配失败而失败。

在大型设备上构建小型模型时，TensorRT 可能会选择效率较低但可在可用资源上更好地扩展的内核。因此，如果优化单个TensorRT 引擎以在同一架构中的多个设备上使用，最好的方法是在最小的设备上运行构建器。或者，您可以在计算资源有限的大型设备上构建引擎（请参阅限制计算资源部分）。

6.8. Explicit vs Implicit Batch

TensorRT 支持两种指定网络的模式：显式批处理和隐式批处理。

在隐式批处理模式下，每个张量都有一个隐式批处理维度，所有其他维度必须具有恒定长度。此模式由 TensoRT 的早期版本使用，现在已弃用，但继续支持以实现向后兼容性。
在显式批处理模式下，所有维度都是显式的并且可以是动态的，即它们的长度可以在执行时改变。许多新功能（例如动态形状和循环）仅在此模式下可用。 ONNX 解析器也需要它。

例如，考虑一个处理 NCHW 格式的具有 3 个通道的大小为 HxW 的 N 个图像的网络。在运行时，输入张量的维度为 [N,3,H,W]。这两种模式在`INetworkDefinition`指定张量维度的方式上有所不同：

- 在显式批处理模式下，网络指定 [N,3,H,W]。
- 在隐式批处理模式下，网络仅指定 [3,H,W]。批次维度 N 是隐式的。

“跨批次对话”的操作无法在隐式批次模式下表达，因为无法在网络中指定批次维度。隐式批处理模式下无法表达的操作示例：

- 减少整个批次维度
- 重塑批次维度
- 用另一个维度转置批次维度

例外是张量可以在整个批次中广播，通过方法`ITensor::setBroadcastAcrossBatch`用于网络输入，并通过隐式广播用于其他张量。

显式批处理模式消除了限制 - 批处理轴是轴 0。显式批处理的更准确术语是“`batch oblivious`”，因为在这种模式下，TensorRT 对引导轴没有特殊的语义含义，除非特定操作需要. 实际上，在显式批处理模式下，甚至可能没有批处理维度（例如仅处理单个图像的网络），或者可能存在多个长度不相关的批处理维度（例如比较从两个批处理中提取的所有可能对）。

`INetworkDefinition`时，必须通过标志指定显式与隐式批处理的选择。这是显式批处理模式的 C++ 代码：

```
IBuilder* builder = ...;
INetworkDefinition* network = builder->createNetworkV2(1U <<
static_cast<uint32_t>(NetworkDefinitionCreationFlag::kEXPLICIT_BATCH)));
```

对于隐式批处理，使用`createNetwork`或将 0 传递给`createNetworkV2`。

这是显式批处理模式的 Python 代码：

```
builder = trt.Builder(...)
builder.create_network(1 <<
int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
```

对于隐式批处理，省略参数或传递 0。

6.9. Sparsity

NVIDIA 安培架构 GPU 支持结构化稀疏性。为了利用该特性获得更高的推理性能，卷积核权重和全连接权重必须满足以下要求：

对于每个输出通道和内核权重中的每个空间像素，每 4 个输入通道必须至少有 2 个零。换句话说，假设内核权重的形状为[K, C, R, S]和C % 4 == 0 ，那么要求是：

```
for k in K:
    for r in R:
        for s in S:
            for c_packed in range(0, C // 4):
                num_zeros(weights[k, c_packed*4:(c_packed+1)*4, r, s]) >= 2
```

要启用稀疏特性，请在构建器配置中设置`kSPARSE_WEIGHTS`标志，并确保启用 `kFP16` 或 `kINT8` 模式。例如

C++

config->setFlag(BuilderFlag::kSPARSE_WEIGHTS);

Python

config.set_flag(trt.BuilderFlag.SPARSE_WEIGHTS)

在构建 TensorRT 引擎时，在 TensorRT 日志的末尾，TensorRT 会报告哪些层包含满足结构稀疏性要求的权重，以及 TensorRT 在哪些层中选择了利用结构化稀疏性的策略。在某些情况下，具有结构化稀疏性的策略可能比正常策略慢，TensorRT 在这些情况下会选择正常策略。以下输出显示了一个显示有关稀疏性信息的 TensorRT 日志示例：

```
[03/23/2021-00:14:05] [I] [TRT] (Sparsity) Layers eligible for sparse math:
conv1, conv2, conv3
[03/23/2021-00:14:05] [I] [TRT] (Sparsity) TRT inference plan picked sparse
implementation for layers: conv2, conv3
```

强制内核权重具有结构化的稀疏模式可能会导致准确性损失。要通过进一步微调恢复丢失的准确性，请参阅PyTorch 中的 Automatic SParsity 工具。

要使用`trtexec`测量结构化稀疏性的推理性能，请参阅trtexec部分。

6.10. Empty Tensors

TensorRT 支持空张量。如果张量具有一个或多个长度为零的维度，则它是一个空张量。零长度尺寸通常不会得到特殊处理。如果一条规则适用于长度为 L 的任意正值 L 的维度，它通常也适用于 L=0。

例如，当沿最后一个轴连接两个维度为 `[x,y,z]` 和 `[x,y,w]` 的张量时，结果的维度为 `[x,y,z+w]`，无论 `x,y, z,` 或者 `w` 为零。

隐式广播规则保持不变，因为只有单位长度维度对广播是特殊的。例如，给定两个维度为 `[1,y,z]` 和 `[x,1,z]` 的张量，它们由`IElementWiseLayer`计算的总和具有维度`[x,y,z]`，无论 `x、y 或 z` 是否为零.

如果一个引擎绑定是一个空的张量，它仍然需要一个非空的内存地址，并且不同的张量应该有不同的地址。这与C++中每个对象都有唯一地址的规则是一致的，例如`new float[0]`返回一个非空指针。如果使用可能返回零字节空指针的内存分配器，请改为请求至少一个字节。

有关空张量的任何每层特殊处理，请参阅TensorRT 层。

6.11. Reusing Input Buffers

TensorRT 还包括一个可选的 CUDA 事件作为`enqueue`方法的参数，一旦输入缓冲区可以重用，就会发出信号。这允许应用程序在完成当前推理的同时立即开始重新填充输入缓冲区以进行下一次推理。例如：

C++

context->enqueueV2(&buffers[0], stream, &inputReady);

Python

context.execute_async_v2(buffers, stream_ptr, inputReady)

6.12. Engine Inspector

TensorRT 提供`IEngineInspector` API 来检查 TensorRT 引擎内部的信息。从反序列化的引擎中调用`createEngineInspector()`创建引擎`inspector`，然后调用`getLayerInformation()`或`getEngineInformation() inspector` API分别获取引擎中特定层或整个引擎的信息。可以打印出给定引擎的第一层信息，以及引擎的整体信息，如下：

C++

```
auto inspector = std::unique_ptr<IEngineInspector>(engine-
>createEngineInspector());
inspector->setExecutionContext(context); // OPTIONAL
std::cout << inspector->getLayerInformation(0, LayerInformationFormat::kJSON);
// Print the information of the first layer in the engine.
std::cout << inspector->getEngineInformation(LayerInformationFormat::kJSON); //
Print the information of the entire engine.
```

Python

```

inspector = engine.create_engine_inspector();
inspector.execution_context = context; # OPTIONAL
print(inspector.get_layer_information(0, LayerInformationFormat.JSON); # Print
the information of the first layer in the engine.
print(inspector.get_engine_information(LayerInformationFormat.JSON); # Print the
information of the entire engine.
```

请注意，引擎/层信息中的详细程度取决于构建引擎时的`ProfilingVerbosity`构建器配置设置。默认情况下，`ProfilingVerbosity`设置为`kLAYER_NAMES_ONLY`，因此只会打印层名称。如果`ProfilingVerbosity`设置为`kNONE`，则不会打印任何信息；如果设置为`kDETAILED`，则会打印详细信息。

`getLayerInformation()` API 根据`ProfilingVerbosity`设置打印的层信息的一些示例：

kLAYER_NAMES_ONLY

```
node_of_gpu_0/res4_0_branch2a_1 + node_of_gpu_0/res4_0_branch2a_bn_1 +
node_of_gpu_0/res4_0_branch2a_bn_2
```

kDETAILED


```
{
  "Name": "node_of_gpu_0/res4_0_branch2a_1 + node_of_gpu_0/res4_0_branch2a_bn_1
+ node_of_gpu_0/res4_0_branch2a_bn_2",
  "LayerType": "CaskConvolution",
  "Inputs": [
  {
    "Name": "gpu_0/res3_3_branch2c_bn_3",
    "Dimensions": [16,512,28,28],
    "Format/Datatype": "Thirty-two wide channel vectorized row major Int8
format."
  }],
  "Outputs": [
  {
    "Name": "gpu_0/res4_0_branch2a_bn_2",
    "Dimensions": [16,256,28,28],
    "Format/Datatype": "Thirty-two wide channel vectorized row major Int8
format."
  }],
  "ParameterType": "Convolution",
  "Kernel": [1,1],
  "PaddingMode": "kEXPLICIT_ROUND_DOWN",
  "PrePadding": [0,0],
  "PostPadding": [0,0],
  "Stride": [1,1],
  "Dilation": [1,1],
  "OutMaps": 256,
  "Groups": 1,
  "Weights": {"Type": "Int8", "Count": 131072},
  "Bias": {"Type": "Float", "Count": 256},
  "AllowSparse": 0,
  "Activation": "RELU",
  "HasBias": 1,
  "HasReLU": 1,
  "TacticName":
"sm80_xmma_fprop_implicit_gemm_interleaved_i8i8_i8i32_f32_nchw_vect_c_32kcrs_vec
t_c_32_nchw_vect_c_32_tilesize256x128x64_stage4_warpsize4x2x1_g1_tensor16x8x32_s
imple_t1r1s1_epifadd",
  "TacticValue": "0x11bde0e1d9f2f35d"
}
```


另外，当引擎使用动态形状构建时，引擎信息中的动态维度将显示为-1，并且不会显示张量格式信息，因为这些字段取决于推理阶段的实际形状。要获取特定推理形状的引擎信息，请创建一个`IExecutionContext`，将所有输入尺寸设置为所需的形状，然后调用`inspector->setExecutionContext(context)`。设置上下文后，检查器将打印上下文中设置的特定形状的引擎信息。

trtexec工具提供了`--profilingVerbosity`、`--dumpLayerInfo`和`--exportLayerInfo`标志，可用于获取给定引擎的引擎信息。有关详细信息，请参阅trtexec部分。

目前，引擎信息中只包含绑定信息和层信息，包括中间张量的维度、精度、格式、策略指标、层类型和层参数。在未来的 TensorRT 版本中，更多信息可能会作为输出 JSON 对象中的新键添加到引擎检查器输出中。还将提供有关检查器输出中的键和字段的更多规范。