TensorRT的最佳性能实践

13.1. Measuring Performance

在开始使用 TensorRT 进行任何优化工作之前，必须确定应该测量什么。没有衡量标准，就不可能取得可靠的进展或衡量是否取得了成功

Latency

网络推理的性能度量是从输入呈现给网络到输出可用所经过的时间。这是单个推理的网络**延迟**。较低的延迟更好。在某些应用中，低延迟是一项关键的安全要求。在其他应用程序中，延迟作为服务质量问题对用户来说是直接可见的。对于批量处理，**延迟可能根本不重要**。

Throughput

另一个性能测量是在固定的时间单位内可以完成多少推理。这是网络的**吞吐量**。吞吐量越高越好。更高的吞吐量表明更有效地利用固定计算资源。对于批量处理，所花费的总时间将由网络的吞吐量决定。

查看延迟和吞吐量的另一种方法是确定最大延迟并在该延迟下测量吞吐量。像这样的服务质量测量可以是用户体验和系统效率之间的合理折衷。

在测量延迟和吞吐量之前，您需要选择开始和停止计时的确切点。根据网络和应用程序，选择不同的点可能是有意义的。

在很多应用中，都有一个处理流水线，整个系统的性能可以通过整个处理流水线的延迟和吞吐量来衡量。由于预处理和后处理步骤在很大程度上取决于特定应用程序，因此本节仅考虑网络推理的延迟和吞吐量。

13.1.1. Wall-clock Timing

经过时间（计算开始和结束之间经过的时间）可用于测量应用程序的整体吞吐量和延迟，以及将推理时间置于更大系统的上下文中。 C++11 在`<chrono>`标准库中提供了高精度计时器。例如，`std::chrono::system_clock`表示系统范围的经过时间，而`std::chrono::high_resolution_clock`以可用的最高精度测量时间。

以下示例代码片段显示了测量网络推理主机时间：

```
#include <chrono>

auto startTime = std::chrono::high_resolution_clock::now();
context->enqueueV2(&buffers[0], stream, nullptr);
cudaStreamSynchronize(stream);
auto endTime = std::chrono::high_resolution_clock::now();
float totalTime = std::chrono::duration<float, std::milli>
(endTime - startTime).count();
```

如果设备上一次只发生一个推理，那么这可能是一种简单的方法来分析各种操作所花费的时间。推理通常是异步的，因此请确保添加显式 CUDA 流或设备同步以等待结果可用。

13.1.2. CUDA Events

仅在主机上计时的一个问题是它需要主机/设备同步。优化的应用程序可能会在设备上并行运行许多推理，并具有重叠的数据移动。此外，同步本身给定时测量增加了一些噪声。
为了帮助解决这些问题，CUDA 提供了一个事件 API 。此 API 允许您将事件放入 CUDA 流中，这些事件将在遇到事件时由 GPU 打上时间戳。然后，时间戳的差异可以告诉您不同操作花费了多长时间。

以下示例代码片段显示了计算两个 CUDA 事件之间的时间：

```
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);

cudaEventRecord(start, stream);
context->enqueueV2(&buffers[0], stream, nullptr);
cudaEventRecord(end, stream);

cudaEventSynchronize(end);
float totalTime;
cudaEventElapsedTime(&totalTime, start, end);
```

13.1.3. Built-In TensorRT Profiling

深入挖掘推理性能需要在优化网络中进行更细粒度的时序测量。
TensorRT 有一个**Profiler** (C++ , Python) 接口，您可以实现该接口以便让 TensorRT 将分析信息传递给您的应用程序。调用时，网络将以分析模式运行。完成推理后，将调用您的类的分析器对象以报告网络中每一层的时间。这些时序可用于定位瓶颈、比较序列化引擎的不同版本以及调试性能问题。

分析信息可以从常规推理`enqueueV2()`启动或 CUDA 图启动中收集。有关详细信息，请参阅`IExecutionContext::setProfiler()`和`IExecutionContext::reportToProfiler()` （ C++ 、Python ） 。

循环内的层编译为单个单片层，因此，这些层的单独时序不可用。

公共示例代码 (`common.h`) 中提供了一个展示如何使用`IProfiler`接口的示例，然后在位于 GitHub 存储库中的sampleNMT中使用。

您还可以使用`trtexec`在给定输入网络或计划文件的情况下使用 TensorRT 分析网络。有关详细信息，请参阅trtexec部分。

13.1.4. CUDA Profiling Tools

推荐的 CUDA 分析器是NVIDIA Nsight™ Systems 。一些 CUDA 开发人员可能更熟悉 nvprof 和 nvvp，但是，这些已被弃用。在任何情况下，这些分析器都可以用于任何 CUDA 程序，以报告有关在执行期间启动的内核、主机和设备之间的数据移动以及使用的 CUDA API 调用的时序信息。

Nsight Systems 可以通过多种方式配置，以仅报告程序执行的一部分的时序信息，或者也可以将传统的 CPU 采样配置文件信息与 GPU 信息一起报告。

仅分析推理阶段

分析 TensorRT 应用程序时，您应该仅在构建引擎后启用分析。在构建阶段，所有可能的策略都被尝试和计时。分析这部分执行将不会显示任何有意义的性能测量，并将包括所有可能的内核，而不是实际选择用于推理的内核。限制分析范围的一种方法是：

- 第一阶段：构建应用程序，然后在一个阶段序列化引擎。

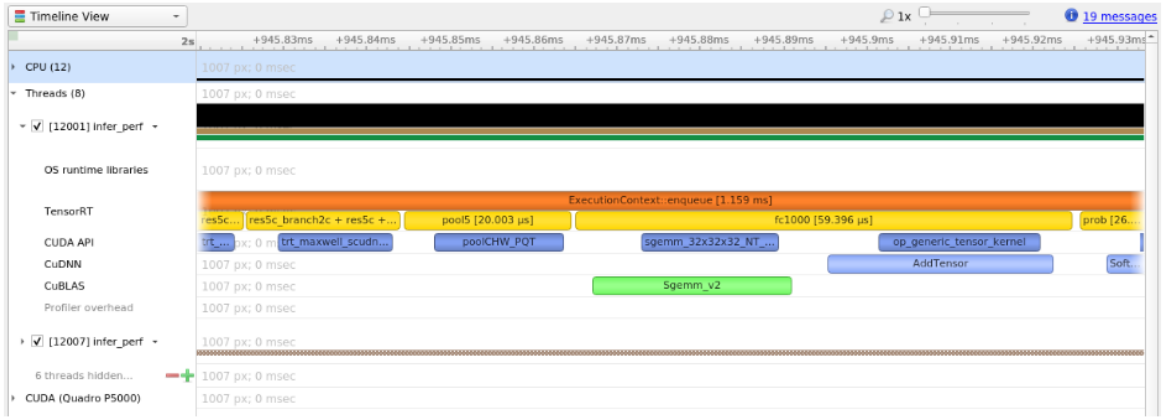
- 第二阶段：加载序列化引擎并在第二阶段运行推理并仅对第二阶段进行分析。
- 如果应用程序无法序列化引擎，或者应用程序必须连续运行两个阶段，您还可以在第二阶段周围添加 `cudaProfilerStart() / cudaProfilerStop()` CUDA API，并在 Nsight Systems 命令中添加 `-c cudaProfilerApi` 标志以仅配置文件 `cudaProfilerStart()`和`cudaProfilerStop()` 之间的部分。

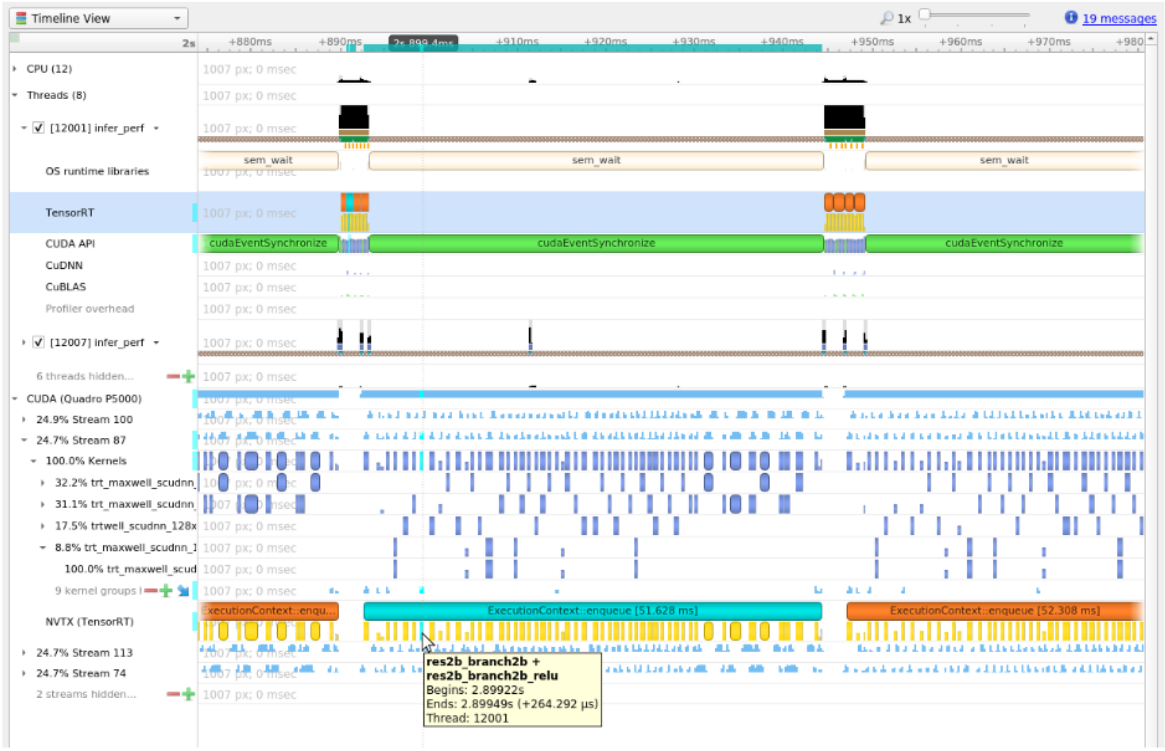
在 Nsight Systems 中使用 `NVTX` 跟踪
启用NVIDIA 工具扩展 SDK (NVTX)跟踪允许 Nsight Compute 和 Nsight Systems 收集由 TensorRT 应用程序生成的数据。 NVTX 是一个基于 C 的 API，用于标记应用程序中的事件和范围。

将内核名称解码回原始网络中的层可能很复杂。因此，TensorRT 使用 NVTX 为每一层标记一个范围，然后允许 CUDA 分析器将每一层与调用来实现它的内核相关联。在 TensorRT 中，NVTX 有助于将运行时引擎层的执行与 CUDA内核调用相关联。 Nsight Systems 支持在时间轴上收集和可视化这些事件和范围。 Nsight Compute 还支持在应用程序挂起时收集和显示给定线程中所有活动 NVTX 域和范围的状态。

在 TensorRT 中，每一层都可以启动一个或多个内核来执行其操作。启动的确切内核取决于优化的网络和存在的硬件。根据构建器的选择，可能会有多个额外的操作对穿插在层计算中的数据进行重新排序；这些重新格式化操作可以作为设备到设备的内存副本或自定义内核来实现。

例如，以下屏幕截图来自 Nsight Systems。





控制 NVTX 跟踪中的详细程度

默认情况下，TensorRT 仅在 NVTX 标记中显示层名称，而用户可以在构建引擎时通过设置 `IBuilderConfig` 中的 `ProfilingVerbosity` 来控制细节级别。例如，要禁用 NVTX 跟踪，请将 `ProfilingVerbosity` 设置为 `kNONE`：

C++

```
builderConfig->setProfilingVerbosity(ProfilingVerbosity::kNONE);
```

Python

```
builder_config.profiling_verbosity = trt.ProfilingVerbosity.NONE
```

另一方面，您可以通过将 `ProfilingVerbosity` 设置为 `kDETAILED` 来选择允许 TensorRT 在 NVTX 标记中打印更详细的层信息，包括输入和输出尺寸、操作、参数、顺序编号等：

C++

```
builderConfig->setProfilingVerbosity(ProfilingVerbosity::kDETAILED);
```

Python

```
builder_config.profiling_verbosity = trt.ProfilingVerbosity.DETAILED
```

trtexec运行 Nsight 系统
以下是使用trtexec工具收集 Nsight Systems 配置文件的命令示例：

```
trtexec --onnx=foo.onnx --profilingVerbosity=detailed --saveEngine=foo.plan
```

```
nsys profile -o foo_profile trtexec --loadEngine=foo.plan --warmUp=0 --duration=0 --iterations=50
```

第一个命令构建引擎并将其序列化为 `foo.plan` ，第二个命令使用 `foo.plan` 运行推理并生成一个 `foo_profile.qdrep` 文件，然后可以在 Nsight Systems GUI 界面中打开该文件以进行可视化。

`--profilingVerbosity=detailed` 标志允许 TensorRT 在 NVTX 标记中显示更详细的层信息，而 `--warmUp =0` `--duration=0` `--iterations=50` 标志允许您控制要运行的推理迭代次数。默认情况下，trtexec运行推理三秒钟，这可能会导致输出 `qdrep` 文件非常大。

13.1.5. Tracking Memory

跟踪内存使用情况与执行性能一样重要。通常，设备上的内存比主机上的内存更受限制。为了跟踪设备内存，推荐的机制是创建一个简单的自定义 GPU 分配器，它在内部保留一些统计信息，然后使用常规 CUDA 内存分配函数 `cudaMalloc` 和 `cudaFree` 。

可以为构建器 `IBuilder` 设置自定义 GPU 分配器以进行网络优化，并在使用 `IGpuAllocator` API反序列化引擎时为 `IRuntime` 设置。自定义分配器的一个想法是跟踪当前分配的内存量，并将带有时间戳和其他信息的分配事件推送到分配事件的全局列表中。查看分配事件列表可以分析一段时间内的内存使用情况。

在移动平台上，GPU 内存和 CPU 内存共享系统内存。在内存大小非常有限的设备上，如 Nano，系统内存可能会因大型网络而耗尽；甚至所需的 GPU 内存也小于系统内存。在这种情况下，增加系统交换大小可以解决一些问题。一个示例脚本是：

```
echo "######alloc swap######"
if [ ! -e /swapfile ];then
    sudo fallocate -l 4G /swapfile
    sudo chmod 600 /swapfile
    sudo mkswap /swapfile
    sudo /bin/sh -c 'echo  "/swapfile \t none \t swap \t defaults \t 0 \t 0" >>
/etc/fstab'
    sudo swapon -a
fi
```

13.2. Optimizing TensorRT Performance

以下部分重点介绍 GPU 上的一般推理流程和一些提高性能的一般策略。这些想法适用于大多数 CUDA 程序员，但对于来自其他背景的开发人员可能并不那么明显。

13.2.1. Batching

最重要的优化是使用批处理并行计算尽可能多的结果。在 TensorRT 中，批次是可以统一处理的输入的集合。批次中的每个实例都具有相同的形状，并以完全相同的方式流经网络。因此，每个实例都可以简单地并行计算。

网络的每一层都有计算前向推理所需的一定数量的开销和同步。通过并行计算更多结果，这种开销可以更有效地得到回报。此外，许多层的性能受到输入中最小维度的限制。如果批量大小为 1 或较小，则此大小通常可能是性能限制维度。例如，具有V个输入和K个输出的完全连接层可以针对一个批次实例实现为1xV矩阵与VxK权重矩阵的矩阵乘法。如果对N个实例进行批处理，则这将变为NxV乘以VxK矩阵。向量矩阵乘法器变成矩阵矩阵乘法器，效率更高。

更大的批量大小几乎总是在 GPU 上更有效。非常大的批次，例如N > 2^16，有时可能需要扩展索引计算，因此应尽可能避免。但通常，增加批量大小会提高总吞吐量。此外，当网络包含 `MatrixMultiply` 层或完全连接层时，如果硬件支持，由于使用了 Tensor Cores，32 的倍数的批大小往往对 FP16 和 INT8 推理具有最佳性能。

由于应用程序的组织，有时无法进行批处理推理工作。在一些常见的应用程序中，例如根据请求进行推理的服务器，可以实现机会批处理。对于每个传入的请求，等待时间T 。如果在此期间有其他请求进来，请将它们一起批处理。否则，继续进行单实例推理。这种类型的策略为每个请求增加了固定的延迟，但可以将系统的最大吞吐量提高几个数量级。

使用批处理

如果在创建网络时使用显式批处理模式，则批处理维度是张量维度的一部分，您可以通过添加优化配置文件来指定批处理大小和批处理大小的范围以优化引擎。有关更多详细信息，请参阅使用动态形状部分。

如果在创建网络时使用隐式批处理模式，则 `IExecutionContext::execute`（Python 中的 `IExecutionContext.execute`）和 `IExecutionContext::enqueue`（Python 中的 `IExecutionContext.execute_async`）方法采用批处理大小参数。在使用 `IBuilder::setMaxBatchSize`（Python中的 `Builder.max_batch_size`）构建优化网络时，还应该为构建器设置最大批量大小。当调用 `IExecutionContext::execute` 或 `enqueue` 时，作为绑定参数传递的绑定是按张量组织的，而不是按实例组织的。换句话说，一个输入实例的数据没有组合到一个连续的内存区域中。相反，每个张量绑定都是该张量的实例数据数组。

另一个考虑因素是构建优化的网络会针对给定的最大批量大小进行优化。最终结果将针对最大批量大小进行调整，但对于任何较小的批量大小仍然可以正常工作。可以运行多个构建操作来为不同的批量大小创建多个优化引擎，然后在运行时根据实际批量大小选择要使用的引擎。

13.2.2. Streaming

一般来说，CUDA 编程流是一种组织异步工作的方式。放入流中的异步命令保证按顺序运行，但相对于其他流可能会乱序执行。特别是，两个流中的异步命令可以被调度为同时运行（受硬件限制）。

在 TensorRT 和推理的上下文中，优化的最终网络的每一层都需要在 GPU 上工作。但是，并非所有层都能够充分利用硬件的计算能力。在单独的流中安排请求允许在硬件可用时立即安排工作，而无需进行不必要的同步。即使只有一些层可以重叠，整体性能也会提高。

使用流式传输

1. 识别独立的推理批次。
2. 为网络创建一个引擎。
3. `cudaStreamCreate`为每个独立批次创建一个 CUDA 流，并为每个独立批次创建一个`IExecutionContext`。
4. `IExecutionContext::enqueue`从适当的`IExecutionContext`请求异步结果并传入适当的流来启动推理工作。
5. 在所有工作启动后，与所有流同步以等待结果。执行上下文和流可以重用于以后的独立工作批次。

多个流
运行多个并发流通常会导致多个流同时共享计算资源的情况。这意味着与优化 TensorRT 引擎时相比，推理期间网络可用的计算资源可能更少。这种资源可用性的差异可能会导致 TensorRT 选择一个对于实际运行时条件不是最佳的内核。为了减轻这种影响，您可以在引擎创建期间限制可用计算资源的数量，使其更接近实际运行时条件。这种方法通常以延迟为代价来提高吞吐量。有关更多信息，请参阅限制计算资源。

也可以将多个主机线程与流一起使用。一种常见的模式是将传入的请求分派到等待工作线程池中。在这种情况下，工作线程池将每个都有一个执行上下文和 CUDA 流。当工作变得可用时，每个线程将在自己的流中请求工作。每个线程将与其流同步以等待结果，而不会阻塞其他工作线程。

13.2.3. CUDA Graphs

CUDA 图是一种表示内核序列（或更一般地是图）的方式，其调度方式允许由 CUDA 优化。当您的应用程序性能对将内核排入队列所花费的 CPU 时间敏感时，这可能特别有用。
TensorRT 的`enqueuev2()`方法支持对不需要 CPU 交互的模型进行 CUDA 图捕获。例如：

C++

```
// Capture a CUDA graph instance
cudaGraph_t graph;
cudaGraphExec_t instance;
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
context->enqueueV2(buffers, stream, nullptr);
cudaStreamEndCapture(stream, &graph);
cudaGraphInstantiate(&instance, graph, NULL, NULL, 0);

// To run inferences:
cudaGraphLaunch(instance, stream);
cudaStreamSynchronize(stream);
```

不支持图的模型包括带有循环或条件的模型。在这种情况下，`cudaStreamEndCapture()`将返回`cudaErrorStreamCapture*`错误，表示图捕获失败，但上下文可以继续用于没有 CUDA 图的正常推理。

捕获图时，重要的是要考虑在存在动态形状时使用的两阶段执行策略。

1. 更新模型的内部状态以考虑输入大小的任何变化
2. 将工作流式传输到 GPU

对于在构建时输入大小固定的模型，第一阶段不需要每次调用工作。否则，如果自上次调用以来输入大小发生了变化，则可能需要进行一些工作来更新派生属性。

第一阶段的工作不是为捕获而设计的，即使捕获成功也可能会增加模型执行时间。因此，在更改输入的形状或形状张量的值后，调用`enqueueV2()`一次以在捕获图形之前刷新延迟更新。

使用 TensorRT 捕获的图特定于捕获它们的输入大小，以及执行上下文的状态。修改捕获图表的上下文将导致执行图表时未定义的行为 - 特别是，如果应用程序通过`createExecutionContextWithoutDeviceMemory()`为激活提供自己的内存，则内存地址也会作为图表的一部分被捕获。绑定位置也被捕获为图表的一部分。

`trtexec`允许您检查构建的 TensorRT 引擎是否与 CUDA 图形捕获兼容。有关详细信息，请参阅 trtexec部分。

13.2.4. Enabling Fusion

13.2.4.1. Layer Fusion

TensorRT 尝试在构建阶段在网络中执行许多不同类型的优化。在第一阶段，尽可能将层融合在一起。融合将网络转换为更简单的形式，但保持相同的整体行为。在内部，许多层实现具有在创建网络时无法直接访问的额外参数和选项。相反，融合优化步骤检测支持的操作模式，并将多个层融合到一个具有内部选项集的层中。

考虑卷积后跟`ReLU`激活的常见情况。要创建具有这些操作的网络，需要使用`addConvolution`添加卷积层，然后使用`addActivation`和`kRELU`的`ActivationType`添加激活层。未优化的图将包含用于卷积和激活的单独层。卷积的内部实现支持直接从卷积核一步计算输出上的 ReLU 函数，而无需第二次内核调用。融合优化步骤将检测 ReLU 之后的卷积，验证实现是否支持这些操作，然后将它们融合到一层。

为了调查哪些融合已经发生或没有发生，构建器将其操作记录到构建期间提供的记录器对象。优化步骤在`kINFO`日志级别。要查看这些消息，请确保将它们记录在`ILogger`回调中。

融合通常通过创建一个新层来处理，该层的名称包含被融合的两个层的名称。例如，在 MNIST 中，名为`ip1`的全连接层（`InnerProduct`）与名为`relu1`的`ReLU`激活层融合，以创建名为`ip1 + relu1`的新层。

13.2.4.2. Types Of Fusions

以下列表描述了支持的融合类型。

支持的层融合

ReLU ReLU Activation

执行 ReLU 的激活层，然后执行 ReLU 的激活将被单个激活层替换。

Convolution and ReLU Activation
卷积层可以是任何类型，并且对值没有限制。激活层必须是 ReLU 类型。

Convolution and GELU Activation

输入输出精度要一致；它们都是`FP16`或`INT8`。激活层必须是`GELU`类型。 TensorRT 应该在具有 CUDA 10.0 或更高版本的 Turing 或更高版本的设备上运行。

Convolution and Clip Activation

卷积层可以是任何类型，并且对值没有限制。激活层必须是Clip类型。

Scale and Activation

Scale 层后跟一个 Activation 层可以融合成一个 Activation 层。

Convolution And ElementWise Operation

卷积层后跟 ElementWise 层中的简单求和、最小值或最大值可以融合到卷积层中。总和不得使用广播，除非广播跨越批量大小。

Padding and Convolution/Deconvolution

如果所有填充大小都是非负的，则可以将后跟卷积或反卷积的填充融合到单个卷积/反卷积层中。

Shuffle and Reduce

一个没有 reshape 的 Shuffle 层，然后是一个 Reduce 层，可以融合成一个 Reduce 层。 Shuffle 层可以执行排列，但不能执行任何重塑操作。 Reduce 层必须有一组keepDimensions维度。

Shuffle and Shuffle

每个 Shuffle 层由转置、重塑和第二个转置组成。一个 Shuffle 层后跟另一个 Shuffle 层可以被单个 Shuffle 替换（或什么都没有）。如果两个 Shuffle 层都执行 reshape 操作，则只有当第一个 shuffle 的第二个转置是第二个 shuffle 的第一个转置的逆时，才允许这种融合。

Scale
可以擦除添加0 、乘以1或计算 1 的幂的Scale 层。

Convolution and Scale

卷积层后跟kUNIFORM或kCHANNEL的 Scale 层融合为单个卷积。如果秤具有非恒定功率参数，则禁用此融合。

Reduce

执行平均池化的 Reduce 层将被 Pooling 层取代。 Reduce 层必须有一个keepDimensions集，使用 kAVG操作在批处理之前从CHW输入格式减少H和W维度。

Convolution and Pooling
卷积层和池化层必须具有相同的精度。卷积层可能已经具有来自先前融合的融合激活操作。

Depthwise Separable Convolution
带有激活的深度卷积，然后是带有激活的卷积，有时可能会融合到单个优化的 DepSepConvolution 层中。两个卷积的精度必须为 INT8，并且设备的计算能力必须为 7.2 或更高版本。

SoftMax and Log

如果 SoftMax 尚未与先前的日志操作融合，则可以将其融合为单个 Softmax 层。

SoftMax 和 TopK

可以融合成单层。 SoftMax 可能包含也可能不包含 Log 操作。

FullyConnected

FullyConnected 层将被转换为 Convolution 层，所有用于卷积的融合都会生效。

Supported Reduction Operation Fusions

GELU

一组表示以下方程的 `Unary` 层和 `ElementWise` 层可以融合到单个 GELU 归约操作中。

$0.5x \times (1 + tanh(2/\pi(x + 0.044715x^3)))$

或替代表示：
$0.5x \times (1 + erf(x/\sqrt{2}))$

L1Norm

一个一元层kABS操作和一个 Reduce 层kSUM操作可以融合成一个 L1Norm 归约操作。

Sum of Squares

具有相同输入（平方运算）的乘积 ElementWise 层后跟kSUM 约简可以融合为单个平方和约简运算。

L2Norm

kSQRT UnaryOperation之后的平方和运算可以融合到单个 L2Norm 归约运算中。

LogSum

一个缩减层kSUM后跟一个kLOG UnaryOperation 可以融合成一个单一的 LogSum 缩减操作。

LogSumExp

一个一元kEXP ElementWise操作后跟一个 LogSum 融合可以融合成一个单一的 LogSumExp 约简。

13.2.4.3. PointWise Fusion

多个相邻的 `PointWise` 层可以融合到一个 `PointWise` 层中，以提高性能。

支持以下类型的 `PointWise` 层，但有一些限制：

Activation

每个ActivationType 。

Constant

仅具有单个值的常量（大小 == 1）。

ElementWise

每个ElementWiseOperation 。

PointWise

PointWise本身也是一个 PointWise 层。

Scale

仅支持 `ScaleMode::kUNIFORM` 。

Unary

每个UnaryOperation 。

融合的 PointWise 层的大小不是无限的，因此，某些 `PointWise` 层可能无法融合。

Fusion 创建一个新层，其名称由融合的两个层组成。例如，名为add1的 ElementWise 层与名为relu1的 ReLU 激活层融合，新层名称为： `fusedPointwiseNode(add1, relu1)` 。

13.2.4.4. Q/DQ Fusion

从 QAT 工具（如NVIDIA 的 PyTorch 量化工具包）生成的量化 INT8 图由具有比例和零点的 `onnx::QuantizeLinear` 和 `onnx::DequantizeLinear` 节点对 `(Q/DQ)` 组成。从 TensorRT 7.0 开始，要求 `zero_point` 为0 。

Q/DQ 节点帮助将 FP32 值转换为 INT8，反之亦然。这样的图在 FP32 精度上仍然会有权重和偏差。

权重之后是 `Q/DQ` 节点对，以便在需要时可以对它们进行量化/去量化。偏置量化是使用来自激活和权重的尺度执行的，因此偏置输入不需要额外的 `Q/DQ` 节点对。偏差量化的假设是 $S_weights * S_input = S_bias$ 。

与 Q/DQ 节点相关的融合包括量化/去量化权重，在不改变模型数学等价性的情况下对 Q/DQ 节点进行交换，以及擦除冗余 Q/DQ 节点。应用 Q/DQ 融合后，其余的构建器优化将应用于图。

Fuse Q/DQ with weighted node (Conv, FC, Deconv)

如果我们有一个

```
[DequantizeLinear (Activations), DequantizeLinear (weights)] > Node >
        QuantizeLinear
```

([DQ, DQ] > Node > Q) 序列，然后融合到量化节点 (QNode)。

支持权重的 Q/DQ 节点对需要加权节点支持多个输入。因此，我们支持添加第二个输入（用于权重张量）和第三个输入（用于偏置张量）。可以使用 `setInput(index, tensor)` API 为卷积、反卷积和全连接层设置其他输入，其中 index = 2 用于权重张量， index = 3 用于偏置张量。

在与加权节点融合期间，我们会将 FP32 权重量化为 INT8，并将其与相应的加权节点融合。类似地，FP32 偏差将被量化为 INT32 并融合。

使用非加权节点融合 Q/DQ

如果我们有一个 `DequantizeLinear > Node > QuantizeLinear ( DQ > Node > Q )` 序列，那么它将融合到量化节点 `( QNode )`。

Commutate Q/DQ nodes

`DequantizeLinear` commutation is allowed when $\Phi(DQ(x)) == DQ(\Phi(x))$.
`QuantizeLinear` commutation is allowed when $Q(\Phi(x)) == \Phi(Q(x))$.

此外，交换逻辑还考虑了可用的内核实现，从而保证了数学等价性。

Insert missing Q/DQ nodes

如果一个节点缺少一个 Q/DQ 节点对，并且$max(abs(\Phi(x))) == max(abs(x))$；（例如，MaxPool），将插入缺少的 Q/DQ 对以运行更多具有 INT8 精度的节点。

Erase redundant Q/DQ nodes

有可能在应用所有优化之后，该图仍然有 Q/DQ 节点对，它们本身就是一个空操作。 Q/DQ 节点擦除融合将删除此类冗余对。

13.2.5. Limiting Compute Resources

当减少的数量更好地代表运行时的预期条件时，限制在引擎创建期间可用于 TensorRT 的计算资源量是有益的。例如，当期望 GPU 与 TensorRT 引擎并行执行额外工作时，或者当期望引擎在资源较少的不同 GPU 上运行时（请注意，推荐的方法是在 GPU 上构建引擎，即将用于推理，但这可能并不总是可行的）。

您可以通过以下步骤限制可用计算资源的数量：

1. 启动 CUDA MPS 控制守护进程。

   `nvidia-cuda-mps-control -d`

2. `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` 环境变量一起使用的计算资源量。例如，导出 `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=50` 。

3. 构建网络引擎。

4. 停止 CUDA MPS 控制守护程序。

   `echo quit | nvidia-cuda-mps-control`

生成的引擎针对减少的计算核心数量（本例中为 50%）进行了优化，并在推理期间使用类似条件时提供更好的吞吐量。鼓励您尝试不同数量的流和不同的 MPS 值，以确定网络的最佳性能。

有关nvidia-cuda-mps-control 的更多详细信息，请参阅nvidia-cuda-mps-control文档和此处的相关 GPU 要求。

13.3. Optimizing Layer Performance

以下描述详细说明了如何优化列出的层。

Concatenation Layer

如果使用隐式批处理维度，连接层的主要考虑是如果多个输出连接在一起，它们不能跨批处理维度广播，必须显式复制。大多数层支持跨批次维度的广播以避免不必要地复制数据，但如果输出与其他张量连接，这将被禁用。

Gather Layer
请使用0轴。 Gather 层没有可用的融合。

Reduce Layer

要从 Reduce 层获得最大性能，请在最后一个维度上执行归约（尾部归约）。这允许最佳内存通过顺序内存位置读取/写入模式。如果进行常见的归约操作，请尽可能以将融合为单个操作的方式表达归约。

RNN Layer

如果可能，请选择使用较新的 RNNv2 接口而不是传统的 RNN 接口。较新的接口支持可变序列长度和可变批量大小，以及具有更一致的接口。为了获得最佳性能，更大的批量大小更好。通常，大小为 64 的倍数可获得最高性能。双向 RNN 模式由于增加了依赖性而阻止了波前传播，因此，它往往更慢。

此外，新引入的基于 `ILoop` 的 API 提供了一种更灵活的机制，可以在循环中使用通用层，而不受限于一小组预定义的 `RNNv2` 接口。 ILoop 循环实现了一组丰富的自动循环优化，包括循环融合、展开和循环不变的代码运动，仅举几例。例如，当同一 `MatrixMultiply` 或 `FullyConnected` 层的多个实例正确组合以在沿序列维度展开循环后最大化机器利用率时，通常会获得显着的性能提升。如果您可以避免 `MatrixMultiply` 或 `FullyConnected` 层沿序列维度具有循环数据依赖性，则此方法效果最佳。

Shuffle

如果输入张量仅用于 shuffle 层，并且该层的输入和输出张量不是网络的输入和输出张量，则省略相当于对基础数据的身份操作的 shuffle 操作。 TensorRT 不会为此类操作执行额外的内核或内存副本。

TopK

要从 TopK 层中获得最大性能，请使用较小的K值来减少数据的最后一维，以实现最佳的顺序内存访问。通过使用 Shuffle 层来重塑数据，然后适当地重新解释索引值，可以一次模拟沿多个维度的缩减。

有关层的更多信息，请参阅TensorRT 层。

13.4. Optimizing for Tensor Cores

Tensor Core 是在 NVIDIA GPU 上提供高性能推理的关键技术。在 TensorRT 中，所有计算密集型层（`MatrixMultiply`、`FullyConnected`、`Convolution` 和 `Deconvolution`）都支持 Tensor Core 操作。

如果输入/输出张量维度与某个最小粒度对齐，则张量核心层往往会获得更好的性能：

- 在卷积和反卷积层中，对齐要求是输入/输出通道维度
- 在 `MatrixMultiply` 和 `FullyConnected` 层中，对齐要求是在 `MatrixMultiply` 中的矩阵维度K和N上，即M x K乘以K x N

下表捕获了建议的张量维度对齐，以获得更好的张量核心性能。

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

在不满足这些要求的情况下使用 Tensor Core 实现时，TensorRT 会隐式地将张量填充到最接近的对齐倍数，将模型定义中的维度向上舍入，以在不增加计算或内存流量的情况下允许模型中的额外容量。

TensorRT 总是对层使用最快的实现，因此在某些情况下，即使可用，也可能不使用 Tensor Core 实现。

13.5. Optimizing Plugins

TensorRT 提供了一种注册执行层操作的自定义插件的机制。插件创建者注册后，您可以在序列化/反序列化过程中查找注册表找到创建者并将相应的插件对象添加到网络中。

加载插件库后，所有 TensorRT 插件都会自动注册。有关自定义插件的更多信息，请参阅使用自定义层扩展 TensorRT 。

插件的性能取决于执行插件操作的 CUDA 代码。适用标准CUDA 最佳实践。在开发插件时，从执行插件操作并验证正确性的简单独立 CUDA 应用程序开始会很有帮助。然后可以通过性能测量、更多单元测试和替代实现来扩展插件程序。代码运行并优化后，可以作为插件集成到 TensorRT 中。
为了尽可能获得最佳性能，在插件中支持尽可能多的格式非常重要。这消除了在网络执行期间对内部重新格式化操作的需要。有关示例，请参阅使用自定义层扩展 TensorRT部分。

13.6. Optimizing Python Performance

使用 Python API 时，大多数相同的性能注意事项都适用。在构建引擎时，构建器优化阶段通常会成为性能瓶颈；不是 API 调用来构建网络。 Python API 和 C++ API 的推理时间应该几乎相同。

在 Python API 中设置输入缓冲区涉及使用pycuda或其他 CUDA Python 库（如cupy ）将数据从主机传输到设备内存。其工作原理的详细信息将取决于主机数据的来源。在内部， pycuda支持允许有效访问内存区域的Python 缓冲区协议。这意味着，如果输入数据在numpy数组或其他也支持缓冲区协议的类型中以合适的格式可用，则可以有效地访问和传输到 GPU。为了获得更好的性能，请确保您使用pycuda分配一个页面锁定的缓冲区，并在那里写入最终的预处理输入。

有关使用 Python API 的更多信息，请参阅Python API 。

13.7. Improving Model Accuracy

TensorRT 可以根据构建器配置以 `FP32`、`FP16` 或 `INT8` 精度执行层。默认情况下，TensorRT 选择以可实现最佳性能的精度运行层。有时这可能会导致准确性下降。通常，以更高的精度运行层有助于提高准确性，但会影响一些性能。

我们可以采取几个步骤来提高模型的准确性：

1. 验证层输出：
    1. 使用Polygraphy转储层输出并验证没有 NaN 或 Inf。 `--validate`选项可以检查 NaN 和 Infs 。此外，我们可以将层输出与来自例如 ONNX 运行时的黄金值进行比较。
    2. 对于 FP16，模型可能需要重新训练以确保中间层输出可以以 FP16 精度表示，而不会出现上溢/下溢。
    3. 对于 INT8，考虑使用更具代表性的校准数据集重新校准。如果您的模型来自 PyTorch，除了 TensorRT 中的 PTQ，我们还提供了 NVIDIA 的 Quantization Toolkit for PyTorch for QAT 框架中的 QAT。您可以尝试这两种方法并选择更准确的方法。
2. 操纵层精度：
    1. 有时以特定精度运行层会导致输出不正确。这可能是由于固有的层约束（例如，LayerNorm 输出不应为 INT8）、模型约束（输出发散导致准确性差）或报告TensorRT 错误。
    2. 您可以控制层执行精度和输出精度。
    3. 一个实验性的调试精度工具可以帮助自动找到以高精度运行的层。
3. 使用算法选择和可重现构建来禁用不稳定的策略：
    1. 当构建+运行与构建+运行之间的准确性发生变化时，可能是由于为层选择了错误的策略。
    2. 使用算法选择器从好的和坏的运行中转储策略。将算法选择器配置为仅允许策略的子集（即仅允许来自良好运行的策略等）。
    c. 您可以使用Polygraphy自动执行此过程。

每次运行变化的准确性不应改变；一旦为特定 GPU 构建了引擎，它应该会在多次运行中产生位准确的输出。如果没有，请提交TensorRT 错误。